

NodeJS Internals and Architecture

Understand the inner working of Node to build
efficient apps

Introduction

Introduction

- Welcome
- Who this course is for?
- Course Outline

Node Architecture

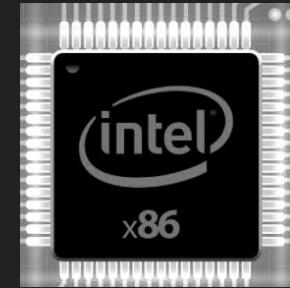
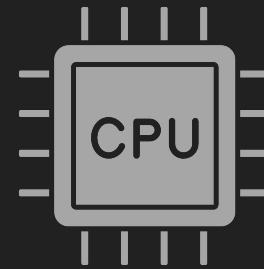
Understanding Node Execution

Interpreted Language and role of V8

Interpreted vs Compiled Languages

Machine Code

- Programs run on machine code
- Specific to the CPU
- Each CPU has different instructions set
- RISC vs CISC



Assembly

- Closest to the machine code
- Still sometimes CPU specific
- Easier to write
- Not easy enough though

```
1 mov r0, #1
2 mov r1, #3
3 add r3, r0, r1
4 str r3, #0xffeeddcc
5
6
7 mov r0, #1
8 mov r1, #3
9 add r3, r0, r1
10 str r3, #0xffeeddcc
```

High level languages

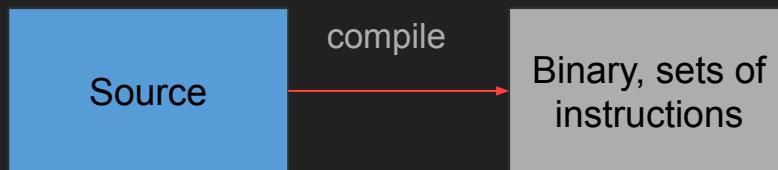
- HLL are more convenient
- Abstractions to hide complexity*
- Need to compile for a CPU
- Compile turns code to machine code
- Linking creates executable file

```
1 #include <stdio.h>
2
3 int main ()
4 {
5     int a = 1;
6     int b = 3;
7     int c = a + b;
8     printf("a + b = %d", c);
9
10    return 0;
11 }
```

*That is good and bad

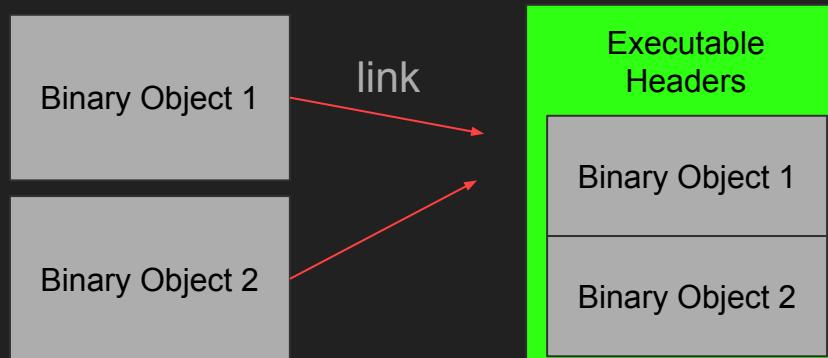
Compiling

- Compile produces machine code in form of object files
- Each object file may represent a source file
- Object files are not ready to be run
- They need to be linked and create an executable
- E.g. gcc, clang, rustc



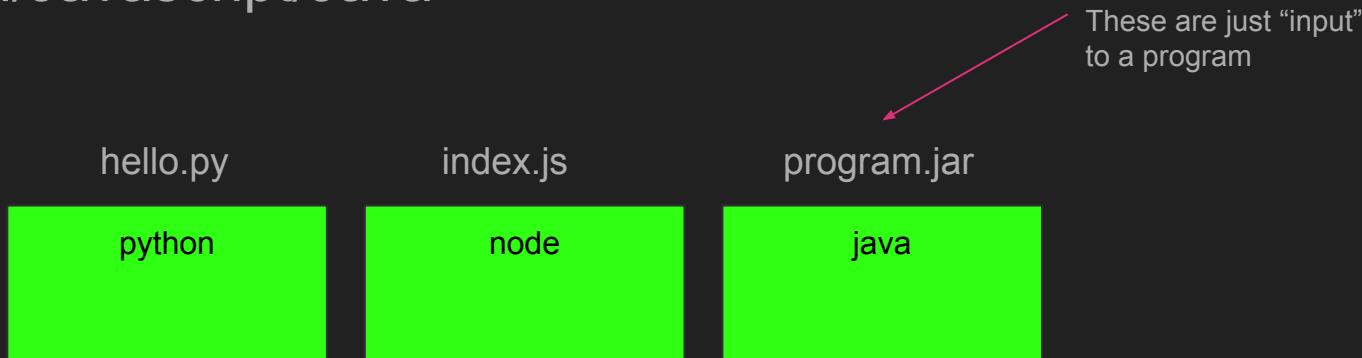
Linking

- Linkers creates an executable file
- Finds and links all object files required and create one file
- The file is an “executable”
- Executable files have types
- E.g. ld,gold linker,lld, mold



Interpreted Languages

- A compiled program doesn't work everywhere
- Must match the CPU/OS
- Can I write my code once and run it everywhere?
- Interpreted languages
- Python/Javascript/Java



Interpreted Languages

- Must have a runtime, node.exe
- Node.exe is a compiled program for every OS/CPU
- Your code index.js runs everywhere
 - Windows node.exe index.js
 - Linux ./node index.js
- Same for Python

Interpreted Languages

- The trick is each line Interpreted
- If you see “+” do “this”, if you see “-” do “this”
- Slower
- Byte code (not string code)
- The Job of V8 engine in Node

Just in Time Compilation (JIT)

- I'm interpreting this code a lot
- Let me compile it directly to machine code
- Put it on the heap
- Mark memory as executable
- Point the CPU program counter to it

Garbage Collection

- Memory management is tricky
- Some languages manages it for you
 - Go, Python, Java
- Some languages you have to do it
 - C, C++
- Some implement it as part of code
 - Node (v8)
- Garbage collection is part of the runtime
- Tags every object and tracks it
- Can cause slow downs

V8 Engine

- Just in Time Compilation
- JavaScript -> Bytecode (faster)
- Mark and Sweep Garbage collection
- Efficient heap memory management
 - Remember JavaScript is dynamically typed
 - We don't know how large objects
 - Inline caching

V8 Engine

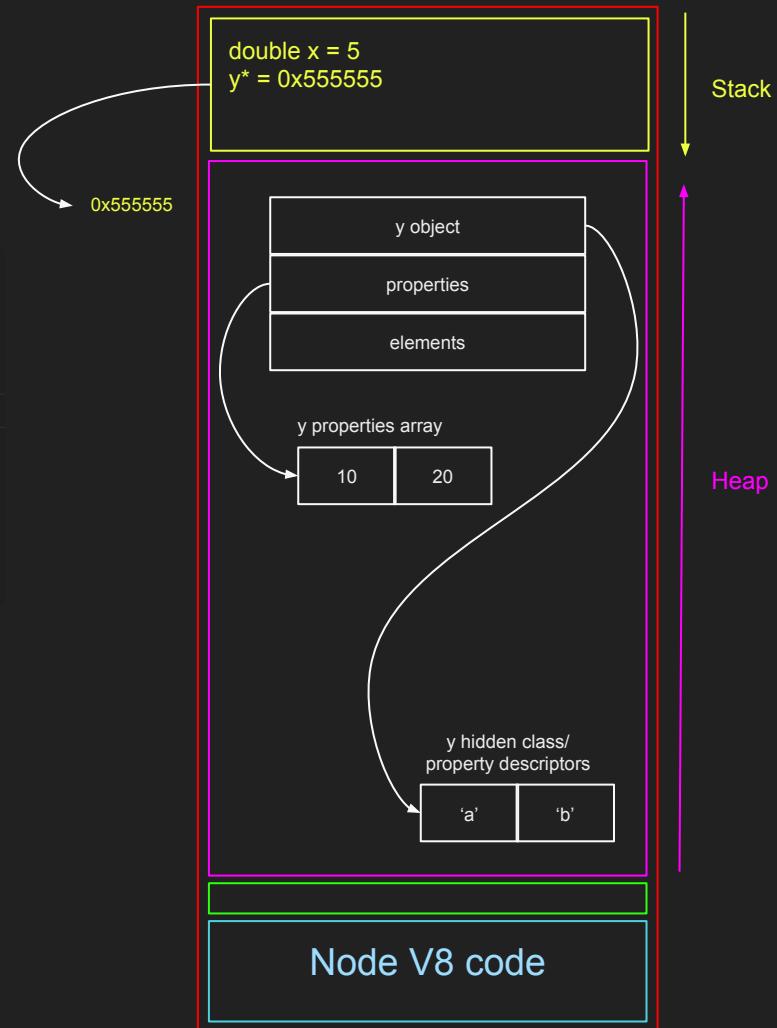
- V8 stores primitive data type in the stack
 - `let x = 5;`
- Objects and Arrays are stored in the heap
- Both are treated as objects
- The Object has two arrays:
 - Named properties
 - `{ "x": 90 } → [90]`
 - `{ "a": 10, "b": 20, 0: 30 } → [10, 20]`
 - Elements (index based)
 - `[8, 9, 10] → [8, 9, 10]`
 - `{ "a": 10, "b": 20, 0: 30 } → [30]`
 - 8 byte size
- Properties “keys” themselves are stored in a hidden class



V8 Engine Object

```
3 /*  
4 This is a primitive value, v8 detects that and stores it in the stack  
5 */  
6 let x = 5;  
7 |  
8 /*  
9 This is an object, y is a pointer that is allocated in the stack  
10 the object itself is allocated in the heap  
11 */  
12 let y = {"a": 10, "b": 20};
```

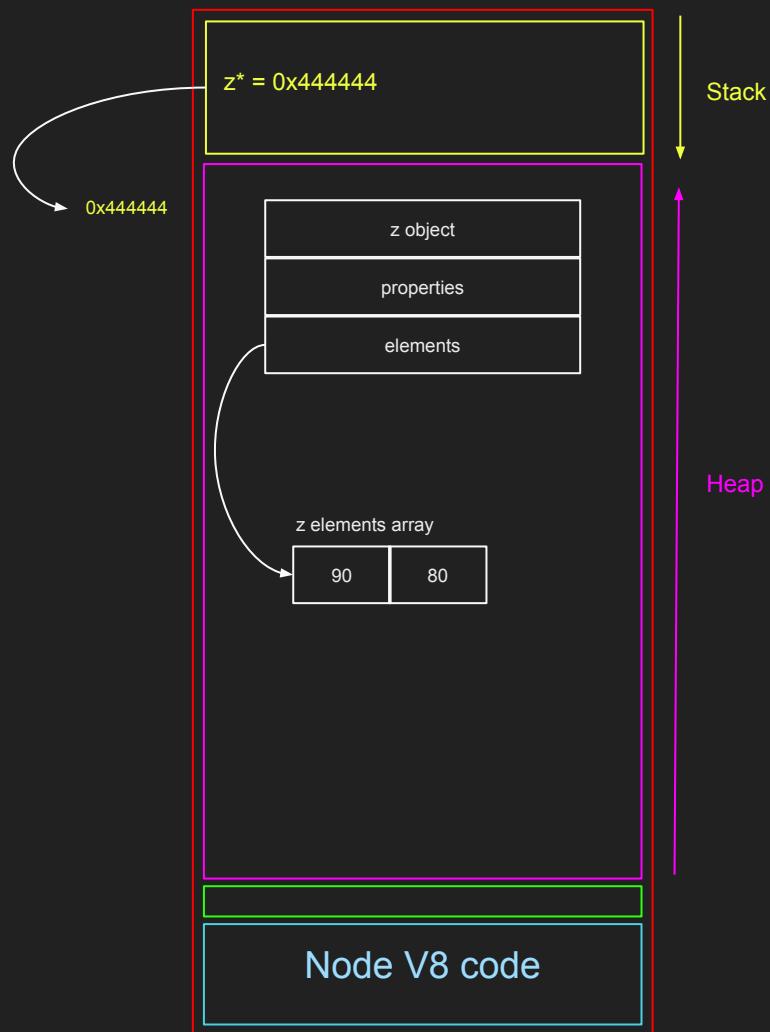
- x is stored in stack as 5
- y pointer stored in the stack
- y js object stored in the heap
- Property values stored separately as an array
- Property names stored separately as an array
 - Hidden class / descriptors
- Properties array points to the properties array



V8 Engine Arrays

```
14 /*  
15 Array is also an object, z is a pointer that is allocated in the stack  
16 the object itself is allocated in the heap  
17 */  
18 let z = [90, 80]
```

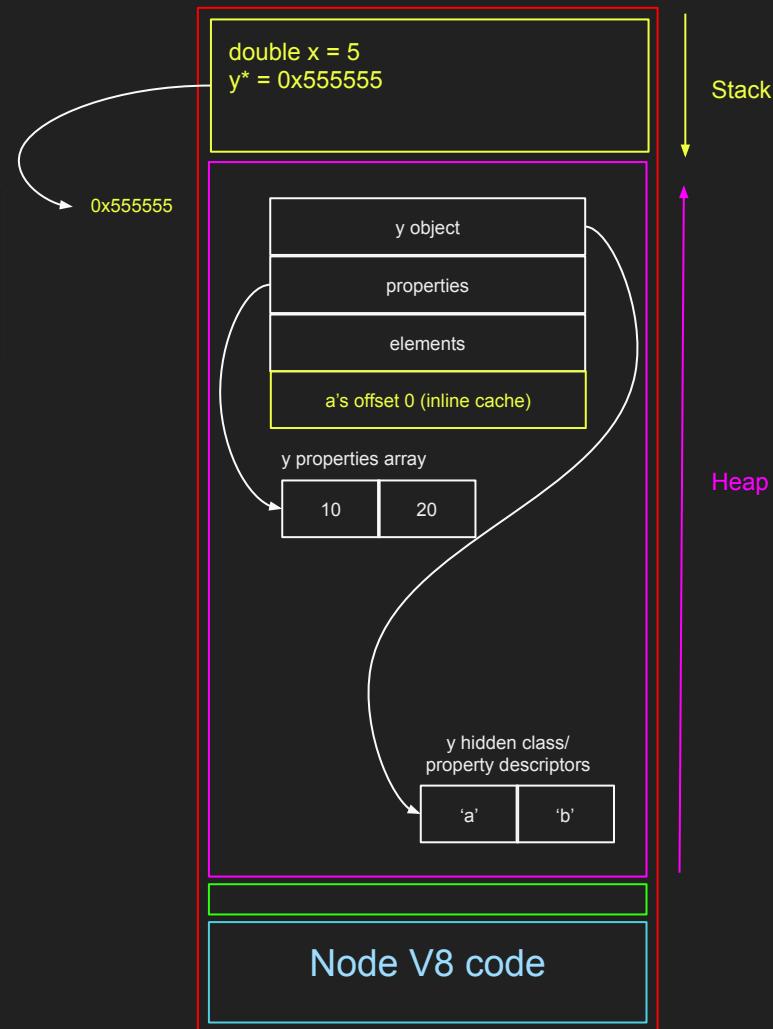
- z pointer stored in the stack
- z js object stored in the heap
- Array is just elements (index based)
- Elements pointer to the elements



V8 Engine inline caching

```
8  /*
9  This is an object, y is a pointer that is allocated in the stack
10 the object itself is allocated in the heap
11 */
12 let y = {"a": 10, "b": 20};
```

- `y.a` is accessed
- V8 needs to know the offset of `a` in the properties array
- V8 looks up hidden class property descriptor array
- Finds '`a`' in offset 0
- Uses 0 to access the properties array gets 10!
- This is slow...
- So v8 remember this and cache it in the object



Summary

- Compiling vs Linking
- Compiled Languages
- Interpreted Languages
- Garbage collection
- V8

The Event Loop

Node's loop

Node Execution

- Node is a C++ program that takes input
- The first input is a JavaScript file
- Node reads the file and interpret the JavaScript
- JS Code is executed line by line serially until it is done

How about these situations?

- Timers
 - Execute this code after X ms
- File I/O
 - Read this file and execute this code when done
- Network I/O
 - Listen on a port and execute this code when a connection is accepted
 - Connect to a server and check for incoming data on a connection
- Serial execution doesn't work

Meet the Event Loop

- Single Thread*
- Asynchronous Non-blocking I/O
- A loop with phases
- Each phase has queue of callbacks
- Terminates when there is no callbacks left

Event Loop (Simplified)

- Initial code execution
- Optionally register callback
- Execute callback
- Callback may register more callbacks
- Run until the queues are empty

Execute

Callbacks
Queues



Execute initial code

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y; ←
5 function timer1Callback() { console.log("timeout elapsed 1ms")}
6 function timer2Callback() {fs.readFile ("c.txt", readFileCCallback)}
7 function readFileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000);
12 }
13 function readFileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback);
17 }
18 fs.readFile ("a.txt", readFileACallback);
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
21
```

Execution

x=1
y=2
z = x + y

Callbacks

Register readFile A callback

Execution

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y;
5 function timer1Callback() { console.log("timeout elapsed 1ms")}
6 function timer2Callback() {fs.readFile ("c.txt", readFileCCallback)}
7 function readFileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000);
12 }
13 function readFileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback);
17 }
18 fs.readFile ("a.txt", readFileACallback); ←
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
21
```

Callbacks

readFileACallback

Register setTimeout 1ms timer1Callback

Execution

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y;
5 function timer1Callback() { console.log("timeout elapsed 1ms") }
6 function timer2Callback() {fs.readFile ("c.txt", readFileCCallback)}
7 function readFileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000);
12 }
13 function readFileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback);
17 }
18 fs.readFile ("a.txt", readFileACallback);
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
```

Callbacks

readFileACallback
timer1Callback

Pop readFileACallback callback

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y;
5 function timer1Callback() { console.log("timeout elapsed 1ms") }
6 function timer2Callback() {fs.readFile ("c.txt", readFileCCallback)}
7 function readFileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000);
12 }
13 function readFileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback);
17 }
18 fs.readFile ("a.txt", readFileACallback);
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
21
```

Execution

readFileACallback



Callbacks

— readFileACallback
timer1Callback

Register writeFile b callback

Execution

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y;
5 function timer1Callback() { console.log("timeout elapsed 1ms") }
6 function timer2Callback() {fs.readFile ("c.txt", readFileCCallback)}
7 function readFileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000);
12 }
13 function readFileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback); ←
17 }
18 fs.readFile ("a.txt", readFileACallback);
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
21
```

readFileACallback
console.log

Callbacks

— **readFileACallback**
— **timer1Callback**
— **writeFileBCallback**

ReadFileACallback done, pop timer1CB

Execution

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y;
5 function timer1Callback() { console.log("timeout elapsed 1ms") } ←
6 function timer2Callback() {fs.readFile ("c.txt", readfileCCallback)}
7 function readfileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000);
12 }
13 function readfileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback);
17 }
18 fs.readFile ("a.txt", readfileACallback);
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
21
```

timer1Callback
console.log

Callbacks

timer1Callback
writeFileBCallback

Pop writeFile b.txt callback

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y;
5 function timer1Callback() { console.log("timeout elapsed 1ms") }
6 function timer2Callback() {fs.readFile ("c.txt", readFileCCallback)}
7 function readFileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000);
12 }
13 function readFileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback);
17 }
18 fs.readFile ("a.txt", readFileACallback);
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
21
```

Execution

writeFileBCallback
console.log



Callbacks

— writeFileBCallback

Register timer2Callback

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y;
5 function timer1Callback() { console.log("timeout elapsed 1ms") }
6 function timer2Callback() {fs.readFile ("c.txt", readFileCCallback)}
7 function readFileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000); ←
12 }
13 function readFileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback);
17 }
18 fs.readFile ("a.txt", readFileACallback);
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
21
```

Execution

writeFileBCallback

Callbacks

— writeFileBCallback
timer2Callback

writeFile b done, popup Timeout 1s callback

Execution

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y;
5 function timer1Callback() { console.log("timeout elapsed 1ms")}
6 function timer2Callback() {fs.readFile ("c.txt", readfileCCallback)} ←
7 function readfileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000);
12 }
13 function readfileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback);
17 }
18 fs.readFile ("a.txt", readfileACallback);
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
21
```

timer2Callback

Callbacks

timer2Callback

1s elapsed, register readFile c.txt CB

Execution

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y;
5 function timer1Callback() { console.log("timeout elapsed 1ms") }
6 function timer2Callback() {fs.readFile ("c.txt", readFileCCallback)} ←
7 function readFileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000);
12 }
13 function readFileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback);
17 }
18 fs.readFile ("a.txt", readFileACallback);
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
21
```

Callbacks

timer2Callback

readFileCCallback

pop readFile c.txt

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y;
5 function timer1Callback() { console.log("timeout elapsed 1ms") }
6 function timer2Callback() {fs.readFile ("c.txt", readFileCCallback)}
7 function readFileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000);
12 }
13 function readFileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback);
17 }
18 fs.readFile ("a.txt", readFileACallback);
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
21
```

Execution

readFileCCallback
console.log



Callbacks

—readFileCCallback

Done. Queues empty, Node exit

Execution

```
1 import fs from "node:fs"
2 const x = 1;
3 const y = 2;
4 const z = x + y;
5 function timer1Callback() { console.log("timeout elapsed 1ms")}
6 function timer2Callback() {fs.readFile ("c.txt", readFileCCallback)}
7 function readFileCCallback() {console.log("read c after a second")}
8 function writeFileBCallback() {
9     //setTimeout 1000ms
10    console.log("write b.txt");
11    setTimeout( timer2Callback,1000);
12 }
13 function readFileACallback() {
14     //writeFile b
15     console.log("read a.txt");
16     fs.writeFile("b.txt", "test", writeFileBCallback);
17 }
18 fs.readFile ("a.txt", readFileACallback);
19 //setTimeout 1ms
20 setTimeout(timer1Callback, 1);
21
```

Callbacks

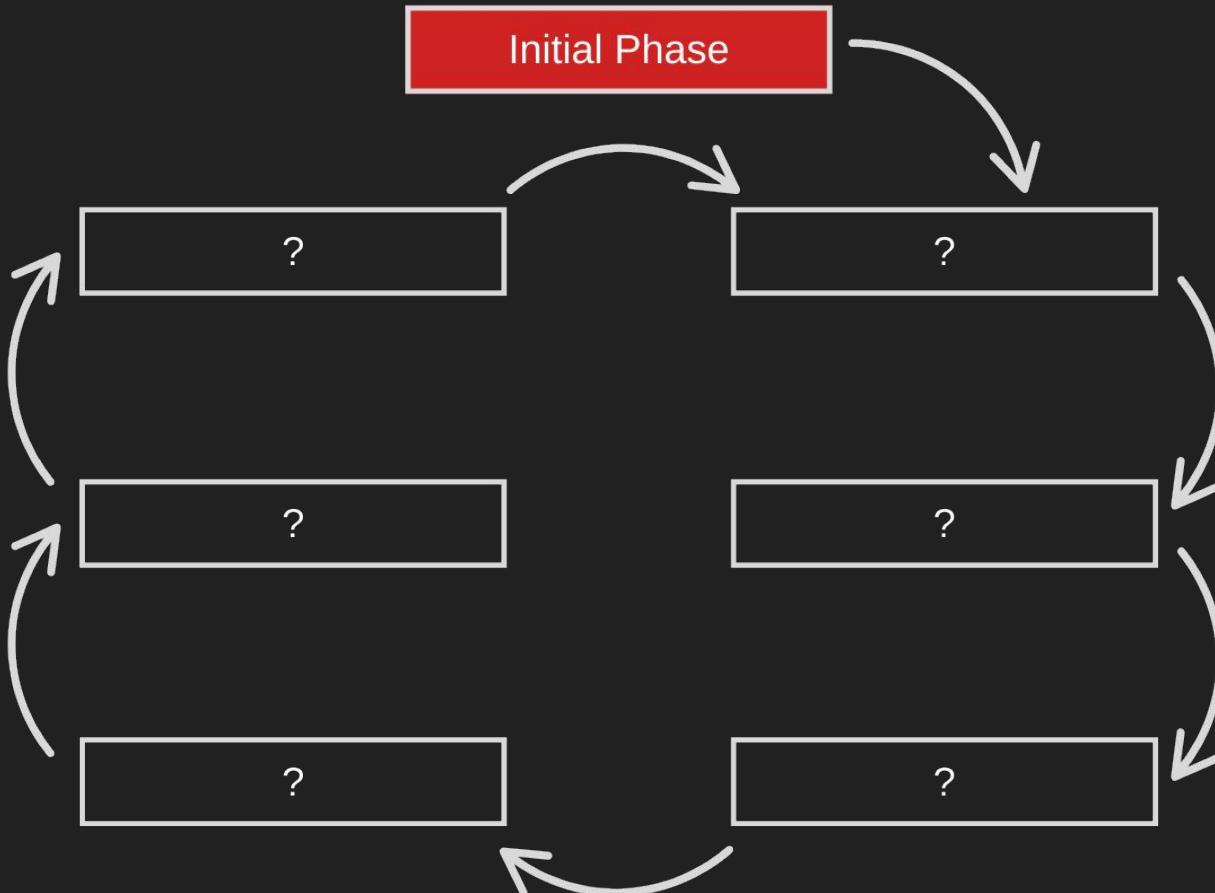
read a.txt
timeout elapsed 1ms
write b.txt
read c after a second

Summary & demo

- Node is a program takes input
- The first input is a JavaScript file
- Node reads the file and execute the JavaScript
- After the execution Node starts a loop
- Execution can register callbacks
- Callbacks can register more callbacks
- Code exercise →

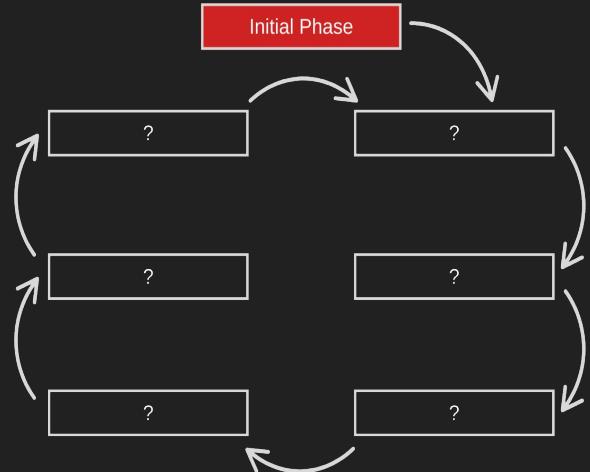
The Main Module

The Initial phase



Main Module

- All main code execute synchronously
- No callbacks can get executed
- Main loop not yet initialized
- Also called initial phase
- Runs once!



What is the output?

```
1 const x = 1;
2 const y = x + 1;
3 setTimeout( ()=> console.log("Should run in 1ms") , 1);
4
5 for (let i =0; i < 1000000000;i ++);
6 console.log("Will this be printed first?");
7
```

What is the output?

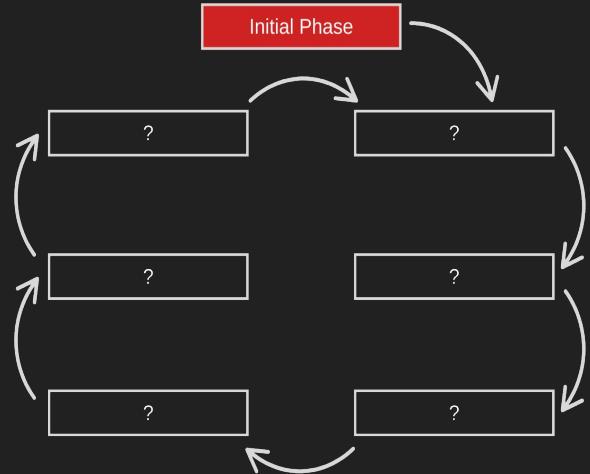
```
1 const x = 1;
2 const y = x + 1;
3 setTimeout( ()=> console.log("Should run in 1ms"), 1);
4
5 for (let i =0; i < 1000000000;i ++);
6 console.log("Will this be printed first?");
```

7

- HusseinMac:node HusseinNasser\$ node test.js
Will this be printed first?
Should run in 1ms
- HusseinMac:node HusseinNasser\$ █

Modules Loads before main module

- Modules are resolved first
- Code executed first
- All require/import modules are loaded
- Before running any of the main module
- Modules loading other modules



Keep initial phase short!

- This is executes first
- Keeping it short is great for performance
- Spins up node faster
- Know your modules

What is the output?

JS loadmodule.mjs ×

JS loadmodule.mjs > ...

```
1 import "/Users/HusseinNasser/Projects/node/a.mjs";
2 console.log("Before main module")
3 const x = 1;
4 const y = x + 1;
5
6 for (let i =0; i < 1000000000;i ++);
7 console.log("End of Main module");
```

JS a.mjs ×

JS a.mjs > [⊕] i

```
1 console.log("about to load a.mjs")
2 for ([let i =0; i < 1000000000;i ++]);
3 console.log("just loaded a.mjs")
```

What is the output?

JS loadmodule.mjs ×

JS loadmodule.mjs > ...

```
1 import "/Users/HusseinNasser/Projects/node/a.mjs";
2 console.log("Before main module")
3 const x = 1;
4 const y = x + 1;
5
6 for (let i =0; i < 1000000000;i ++);
7 console.log("End of Main module");
```

about to load a.mjs
just loaded a.mjs
Before main module
End of Main module

○ HusseinMac:node HusseinNasser\$ █

JS a.mjs ×

JS a.mjs > [?] i

```
1 !console.log("about to load a.mjs")
2 for (let i =0; i < 1000000000;i ++);
3 console.log("just loaded a.mjs")
```

Keep initial phase short!

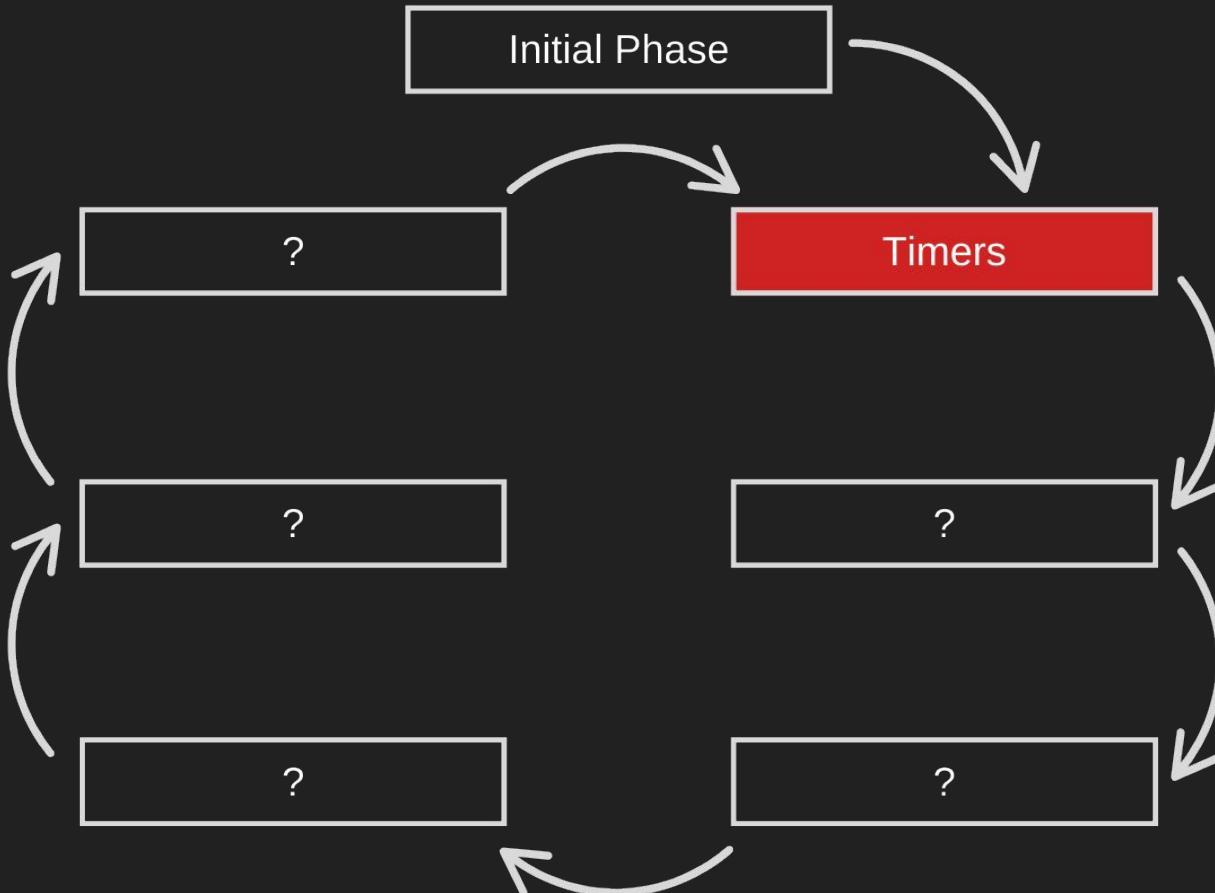
- Because this executes first, keeping it short is great for performance
- Spins up node faster
- Know your modules

Summary & Demo

- Initial phase executes all code synchronously
- Modules are loaded first
- Once all modules are loaded and executed the main executes
- Only then the main loop gets initialized
- Code exercise →

The Timer Phase

Where timers get executed

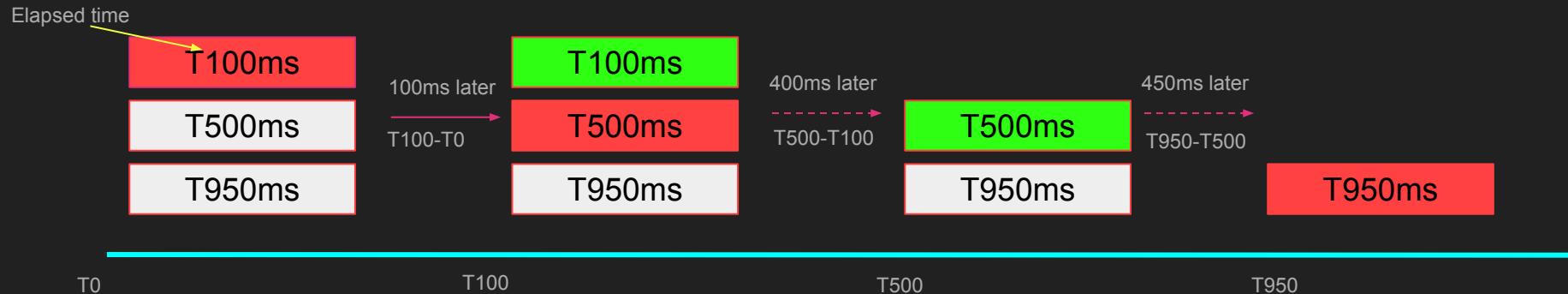


Timers

- After initial phase
- Event main loop gets initialized
- first phase in the loop is timers
- Done by the underlying library (libuv)
- Timer callbacks get scheduled, sorted by duration
- Not accurate
 - Can be slowed down by other phases

Timer implementation

- E.g. 3 timers get scheduled at T0, for 100, 500 and 950 ms
- Timers persist the end time
- Ask the OS to sleep on libuv thread for smallest (100 ms)
- Main loop keeps running normally
- OS wakes up the thread after 100 ms and marks the timer as “ready”
- Main thread enters timer phase, picks up the ready callbacks and execute
- OS sleeps again for the smallest remaining timer (500-T100) (400ms)



What is the output?

```
2 const timerCallback = (a,b) => console.log(`Timer callback ${a} delayed by ${Date.now() - start - b}`);
3
4 const start = Date.now();
5 setTimeout(timerCallback, 100, '100 ms',100);
6 setTimeout(timerCallback, 0, '0 ms',0);
7 setTimeout(timerCallback, 1, '1 ms', 1);
8 setTimeout(timerCallback, 300, '300 ms', 300);
9 setTimeout(timerCallback, 6000, '6000 ms', 6000);
10
11 //the initial phase sync execute this code (which takes 380 ms), waits for it to finish
12 //then goes to the timer phase and noticed all timers are ready except one, so it executes the 4 callbacks
13 // and then waits until the next timer is ready (6 second one) then picks it up
14 for (let i = 1; i <= 1000000000; i++);
```

What is the output?

```
2 const timerCallback = (a,b) => console.log(`Timer callback ${a} delayed by ${Date.now() - start - b}`);
3
4 const start = Date.now();
5 setTimeout(timerCallback, 100, '100 ms',100);
6 setTimeout(timerCallback, 0, '0 ms',0);
7 setTimeout(timerCallback, 1, '1 ms', 1);
8 setTimeout(timerCallback, 300, '300 ms', 300);
9 setTimeout(timerCallback, 6000, '6000 ms', 6000);
10
11 //the initial phase sync execute this code (which takes 380 ms), waits for it to finish
12 //then goes to the timer phase and noticed all timers are ready except one, so it executes the 4 callbacks
13 // and then waits until the next timer is ready (6 second one) then picks it up
14 for (let i = 1; i <= 1000000000; i++);
```

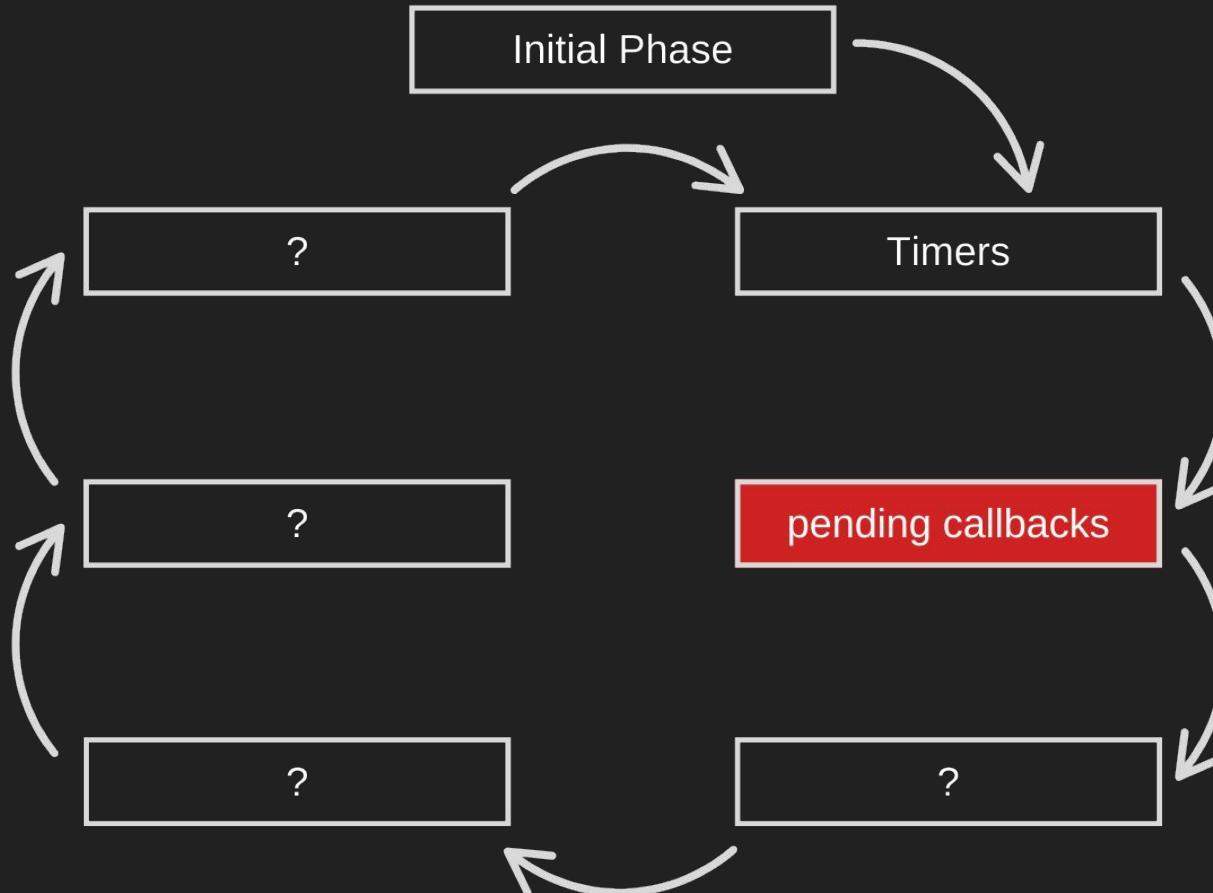
```
 MacBook-Pro:node-course-content HusseinNasser$ node 02-timer/021-timer-one-long.js
Timer callback 0 ms delayed by 385
Timer callback 1 ms delayed by 387
Timer callback 100 ms delayed by 289
Timer callback 300 ms delayed by 89
Timer callback 6000 ms delayed by 2
MacBook-Pro:node-course-content HusseinNasser$
```

Summary & Demo

- Runs after the initial phase
- Done by the underlying library (libuv)
- Timer callbacks get scheduled, sorted by duration
- Other phases can slow it down
- Code exercise →
 - Timers

Pending callbacks

Delayed callbacks (TCP errors)



Pending callbacks

- Phase for callbacks with low priority
- Make room for more important callbacks
- Examples are tcp errors, and network delays
- Deferring it so try later may fix it, avoid unnecessary retries
- Not always though

Example

```
38 // Create a client
39 const client1 = new net.Socket();
40 const client2 = new net.Socket();
41
42 // Connect to the server that exists (this is example.com)
43 ~client2.connect(80, '93.184.215.14', () => {
44   console.log(`Connected to server at ${'93.184.215.14'}:${8080}`);
45 });
46 // Connect to the server (fails)
47 ~client1.connect(9999, '192.168.4.21', () => {
48   console.log(`Connected to server at ${'192.168.4.21'}:${9999}`);
49 });
50
51 // Handle errors
52 ~client1.on('error', (err) => {
53   console.error(`Connection error: ${err.message}`);
54 });
55 // Handle errors
56 ~client2.on('error', (err) => {
57   console.error(`Connection error: ${err.message}`);
58 });
```

Port 80 is open, this succeeds, callback added to poll phase

Port 9999 isn't open so this failed, error event is raised

This callback will be scheduled in pending callbacks

Example

```
38 // Create a client
39 const client1 = new net.Socket();
40 const client2 = new net.Socket();
41
42 // Connect to the server that exists (this is example.com)
43 ~client2.connect(80, '93.184.215.14', () => {
44 |   console.log(`Connected to server at ${'93.184.215.14'}:${8080}`);
45 });
46 // Connect to the server (fails)
47 ~client1.connect(9999,'192.168.4.21', () => {
48 |   console.log(`Connected to server at ${'192.168.4.21'}:${9999}`);
49 });
50
51 // Handle errors
52 ~client1.on('error', (err) => {
53 |   console.error(`Connection error: ${err.message}`);
54 });
55 // Handle errors
56 ~client2.on('error', (err) => {
57 |   console.error(`Connection error: ${err.me
58 });
```

Connected to server at 93.184.215.14:80
Connection error: connect ECONNREFUSED 192.168.4.21:9999
Connection closed

Switch connect order, same output

```
28 // Create a client
29 const clientFail = new net.Socket();
30 const clientSuccess = new net.Socket();
31
32 // Connect to the server (fails)
33 clientFail.connect(9999, '192.168.4.21', () => {
34   console.log(`Connected to server at ${'192.168.4.21'}:${9999}`);
35 });
36 // Connect to the server that exists (this is example.com)
37 clientSuccess.connect(80, '93.184.215.14', () => {
38   console.log(`Connected to server at ${'93.184.215.14'}:${80}`);
39 });
40
41 // Handle errors
42 clientFail.on('error', (err) => {
43   console.error(`Connection error: ${err.message}`);
44 });
45 // Handle errors
46 clientSuccess.on('error', (err) => {
47   console.error(`Connection error: ${err.message}`);
48 });
49
```

Port 9999 isn't open so this failed

Port 80 is open this succeeds, callback added to poll phase

```
--  
Connected to server at 93.184.215.14:80  
Connection error: connect ECONNREFUSED 192.168.4.21:9999  
Connection closed
```

Not always!

- Node makes exceptions
- Loopback gets priority as per my testing

```
32  // Connect to the server (fails)
33 ~ clientFail.connect(9999, '127.0.0.1', () => {
34 |   console.log(`Connected to server at ${'192.168.4.21'}:${{9999}}`);
35 );
36 // Connect to the server that exists (this is example.com)
37 ~ clientSuccess.connect(80, '93.184.215.14', () => {
38 |   console.log(`Connected to server at ${'93.184.215.14'}:${{80}}`);
39 );
40
41 // Handle errors
42 ~ clientFail.on('error', (err) => {
43 |   console.error(`Connection error: ${err.message}`);
44 );
45 // Handle errors
46 ~ clientSuccess.on('error', (err) => {
47 |   console.error(`Connection error: ${err.message}`);
48 );
49
50
```

TERMINAL PORTS PROBLEMS OUTPUT DEBUG CONSOLE

node + □

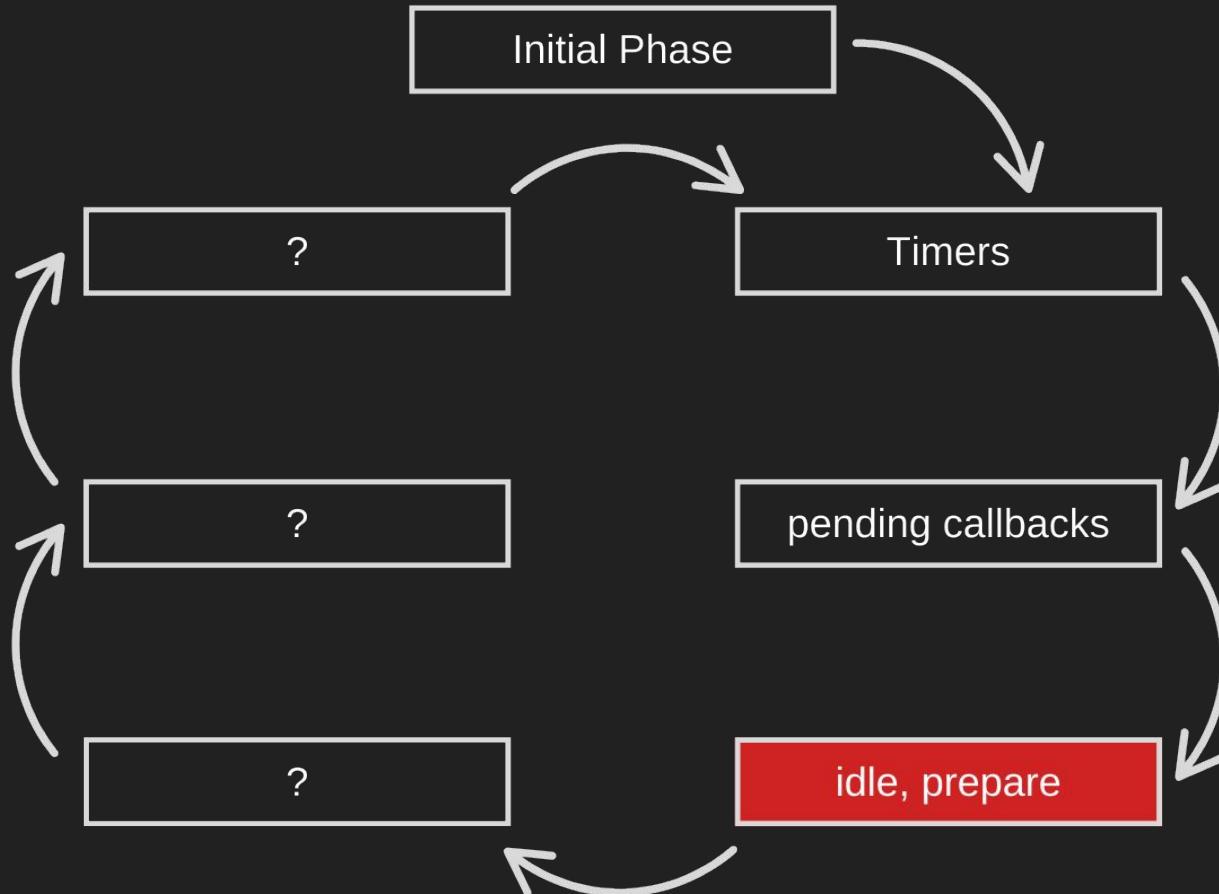
```
HusseinMac:node-course-content HusseinNasser$ node 03-pending-callbacks/031-tcp-error-loopback.js
START
END
Connection error: connect ECONNREFUSED 127.0.0.1:9999
Connected to server at 93.184.215.14:80
```

Summary & Demo

- Pending callbacks host TCP errors
- Callbacks from TCP errors are scheduled in this phase
- The next loop iteration will pick them and execute them
- Users can add retry logic
- **Code exercise →**
 - TCP errors

Idle, prepare

Internal for node



Idle, prepare

- Two steps in one phase, used by lib_uv
- Node js map this phase to the lib_uv
- prepare, idle is not exposed to modules
- Can be overridden with C++ Node extensions

Idle

- Idle executes every iteration
- It still runs in every iteration
- Internal tasks

prepare

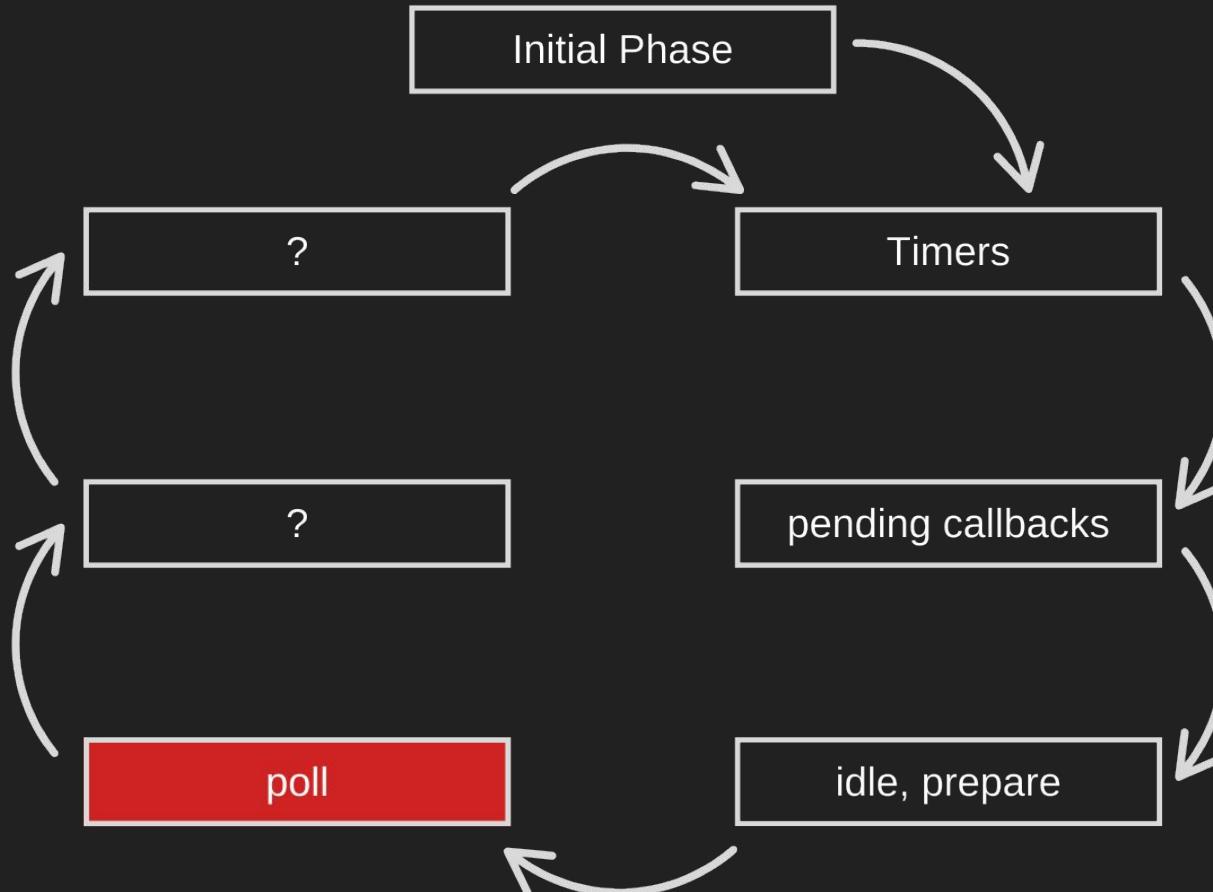
- Prepare executes every iteration before poll
- Used to initialize internal structures to prepare for io
- E.g. as we receive new connections we want to add it to epoll instance
 - `epoll_ctl(9, EPOLL_CTL_ADD, 10)`
 - `epoll_ctl(13, EPOLL_CTL_DEL, 21)`
- Follows a wait

Summary & Demo

- Idle, prepare runs in every iteration
- Not exposed by default
- Can be overridden with C++ extensions,
- Added here for completion
- Code exercise →
 - Show the prepare phase (server setups)

poll

Most important phase



Poll phase

- Two things happen in this phase
- IO is being polled
 - Listen on sockets
 - Read/write connections
 - Accept connections
 - Read/write files
- Callbacks are executed
 - OnRead -> if a file finished reading do this
 - onConnect → if a connection is established do this
 - onListen → if a socket is listening do this
- Dynamic imports are also here

Blocking

- Node will block in poll phase
 - Only when there isn't anything going on
- E.g. epoll_wait for connections
- Except when there are timers!
 - epoll_wait with a timeout

How does IO work in Node

- IO in Node is asynchronous
- Enabled by the lib_uv library
- Network IO is implemented differently from file io
- Depends on kernel
- We will discuss this in detail in future sections

Example - readFile file io

```
1 //example where the initial phase is long and there is a readfile
2 const fs = require(`fs`);
3
4 console.log(`START`);  

5
6 const readFileCallback = (err, data) => { ←
7   console.log(`readFileCallback ${data}`);
8 };
9 const f = 'test.txt'
10 //in the intial phase this file will only be opened (not read)
11 //if it fails we immediately know, we add a callback in the poll queue with
12 fs.readFile(f, readFileCallback);
13
14 //slow down the initial phase
15 for (let i = 1; i <= 10000000000; i++);
16 //after the loop ends and initial phase is done, we enter the poll phase and
17 console.log(`END`); → Once initial phase is done, read is
                                scheduled in the poll
```

Open is called in initial phase , read is called on the poll phase , once read is ready, callback is added and executed.

Example - readFile file io

```
1 //example where the initial phase is long and there is a readfile
2 const fs = require(`fs`);
3
4 console.log(`START`);
5
6 const readFileCallback = (err, data) => {
7   console.log(`readFileCallback ${data}`);
8 };
9 const f = 'test.txt'
10 //in the intial phase this file will only be opened (not read)
11 //if it fails we immediately know, we add a callback in the poll queue with
```

```
● MacBook-Pro:05-poll HusseinNasser$ node 050-poll.js
1 START
1 END
1 readFileCallback hello world
1 ○ MacBook-Pro:05-poll HusseinNasser$ ◻
```

Example - Socket connect

```
1 //example where pending callbacks are scheduled in a diff
2 const net = require('net');
3 console.log(`START`);
4 // Create a client
5 const clientSuccess = new net.Socket();
6 // Connect to the server that exists (this is example.com)
7 clientSuccess.connect(80, '93.184.215.14', () => {
8     console.log(`Connected to server at ${'93.184.215.14'}
9 });
10 // Handle errors
11 clientSuccess.on('error', (err) => {
12     console.error(`Connection error: ${err.message}`);
13 });
```

A socket is created,
and a SYN is sent
This happens during
the poll phase

Example - Socket connect

```
1 //example where pending callbacks are scheduled in a diff
2 const net = require('net');
3 console.log(`START`);
4 // Create a client
5 const clientSuccess = new net.Socket();
6 // Connect to the server that exists (this is example.com)
7 clientSuccess.connect(80, '93.184.215.14', () => {
8   console.log(`Connected to server at ${clientSuccess.address().address}:${clientSuccess.address().port}`);
9 });
10 clientSuccess.on('data', (data) => {
11   console.log(`Data received: ${data}`);
12 });
13 clientSuccess.on('end', () => {
14   console.log(`Connection closed`);
15 });
16 clientSuccess.on('error', (err) => {
17   console.error(`Error: ${err.message}`);
18 });
19 clientSuccess.end();
```

```
⑥ HusseinMac:node-course-content HusseinNasser$ node 05-poll/053-connect.js
START
END
Connected to server at 93.184.215.14:80
^C
⑦ HusseinMac:node-course-content HusseinNasser$ █
```

Keeps running until
we control c or close
all connections
(why?)

Why does Node keep running after connect?

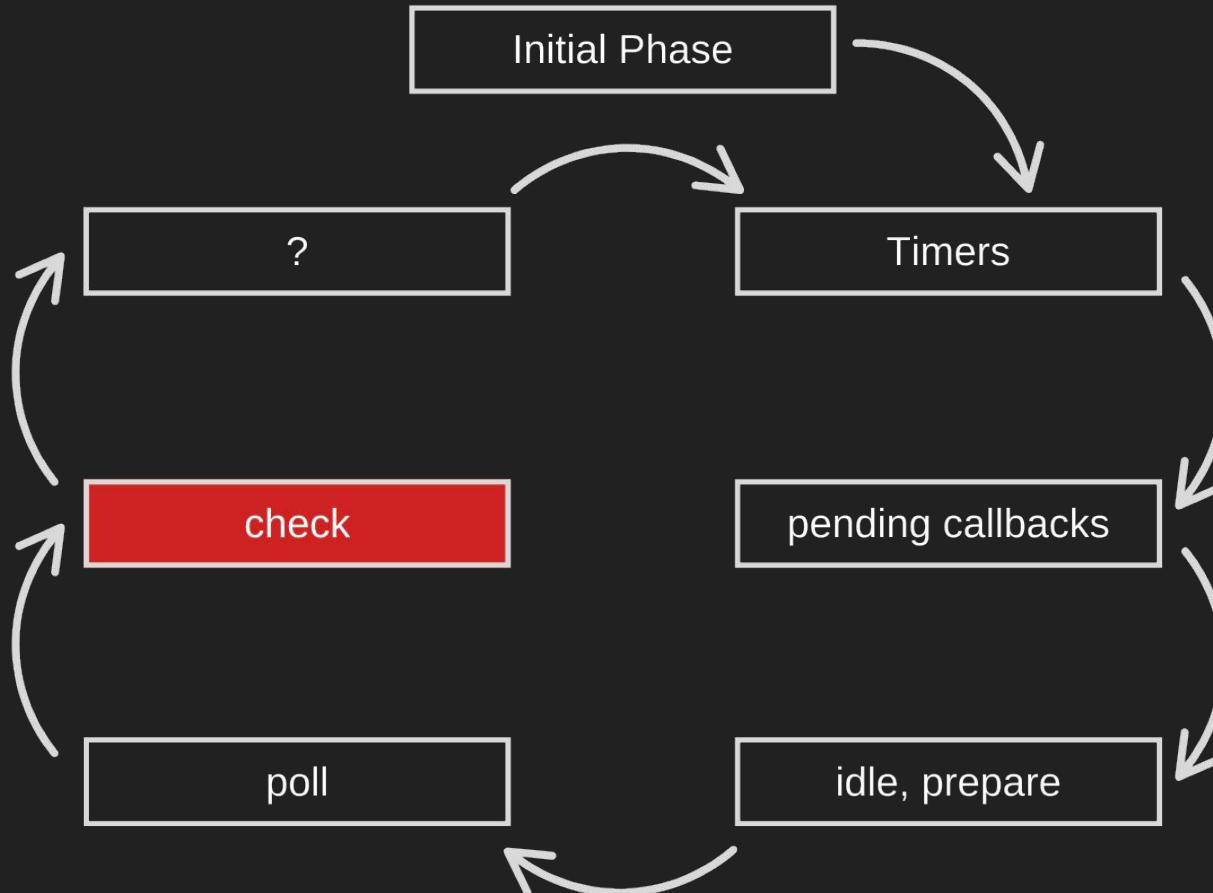
- If there is at least client connection, Node loops and keeps reading
- Until all connections closed
- Same for listening socket

Summary & Demo

- Poll phase executes io events and callbacks
- IO Callbacks can be scheduled by any phase
- IO can be blocking and non-blocking
- Node tries to not block if there are other work
- Code exercise →
 - Readfile
 - Readfile with timer
 - Failed readFile
 - Server listen

check

Phase to execute things while waiting



Check

- Execute code right after poll phase
- Or if the poll phase is busy waiting
- `setImmediate` is the function to schedule check callbacks
- Use it for deterministic outcome
 - Always executed after poll and before timer

Example

```
6 const readFileCallback = (err, data) => {
7   console.log(`readFileCallback ${data}`);
8 };
9 const f = 'test.txt'
10 //in the intial phase this file will only be opened (not
11 //if it fails we immediately know, we add a callback in 1
12 fs.readFile(f, readFileCallback);
13 setImmediate( ()=> console.log("setImmediate called"));
```

Read file is
scheduled poll phase

setImmediate
callback in the check
phase

Example (file exists)

```
6 const readFileCallback = (err, data) => {
7   console.log(`readFileCallback ${data}`);
8 };
9 const f = 'test.txt'
10 //in the intial phase this file will only be opened (not
11 //if it fails we immediately know, we add a callback in 1
12 fs.readFile(f, readFileCallback);
```

- HusseinMac:06-check HusseinNasser\$ node 060-setImmediate.js

START

END

setImmediate called ←
readFileCallback

Output when file
exists..

- HusseinMac:06-check HusseinNasser\$ █

Example (File doesn't exists)

```
6 const readFileCallback = (err, data) => {
7   console.log(`readFileCallback ${data}`);
8 };
9 const f = 'test.txt'
10 //in the intial phase this file will only be opened (not
11 //if it fails we immediately know, we add a callback in
12 fs.readFile(f, readFileCallback);
13 setImmediate( ()=> console.log("setImmediate called"));
```

Example (File doesn't exists)

```
6 const readFileCallback = (err, data) => {
7   console.log(`readFileCallback ${data}`);
8 };
9 const f = 'test.txt'
10 //in the intial phase this file will only be opened (not
11 //if it fails we immediately know, we add a callback in 1
12 fs.readFile(f, readFileCallback);
```

● HusseinMac:06-check HusseinNasser\$ node 060-setImmediate.js

START

END

readFileCallback undefined

setImmediate called

Output when file
doesn't exist (why?)

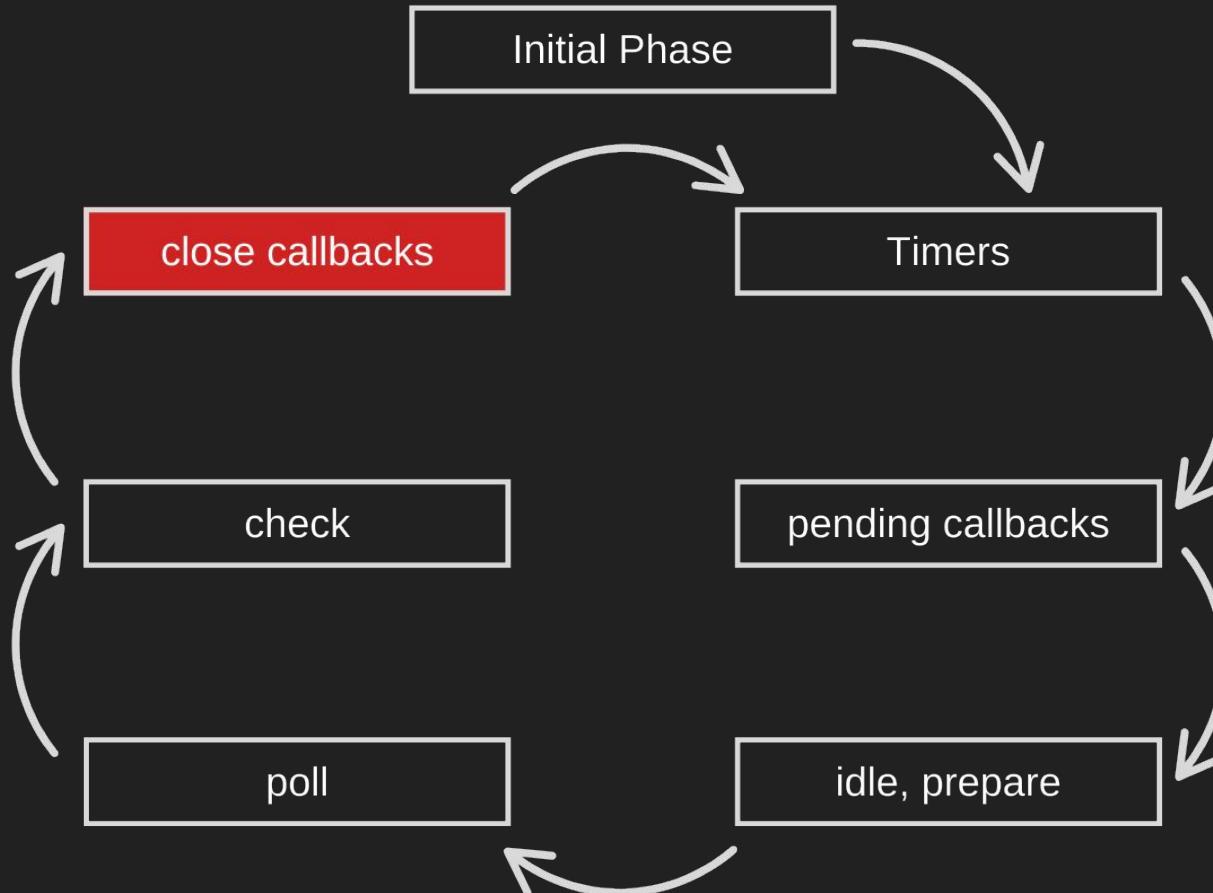
○ HusseinMac:06-check HusseinNasser\$

Summary & Demo

- Check phase is in lock step with poll phase
- Execute code as soon as the poll phase ends/idles
- Difference setTimeout vs setImmediate
- Code exercise →
 - setImmediate vs timer

close

Where close callbacks are scheduled



close

- Close events happen here
- Clean up phase
- E.g. a connection being terminated
- Connection terminates is still an IO (poll)
- Wait for poll and fire the close to “clean up”
- E.g. socket.close event (TCP close handshake)

Example

```
5 const clientSuccess = new net.Socket();
6 // Connect to the server that exists (this is example.com
7 ^clientSuccess.connect(80, '93.184.215.14',
8 |   console.log(`Connected to server at ${`'${'`});
9 |});
```

10 //destroy the connection after 5 seconds

```
11 setTimeout( ()=> clientSuccess.destroy(), 5000);
12 // Handle close events
13 ^clientSuccess.on('close', () => {
14 |   console.error(`Connection closed.`);
15 |});
```

16

Destroy the socket after 5 seconds, the “destroy” is an IO which is scheduled in the poll phase, once it's done, the close callback event is scheduled on the close phase

This will run in the close phase (last), after everything in that iteration is done

Example

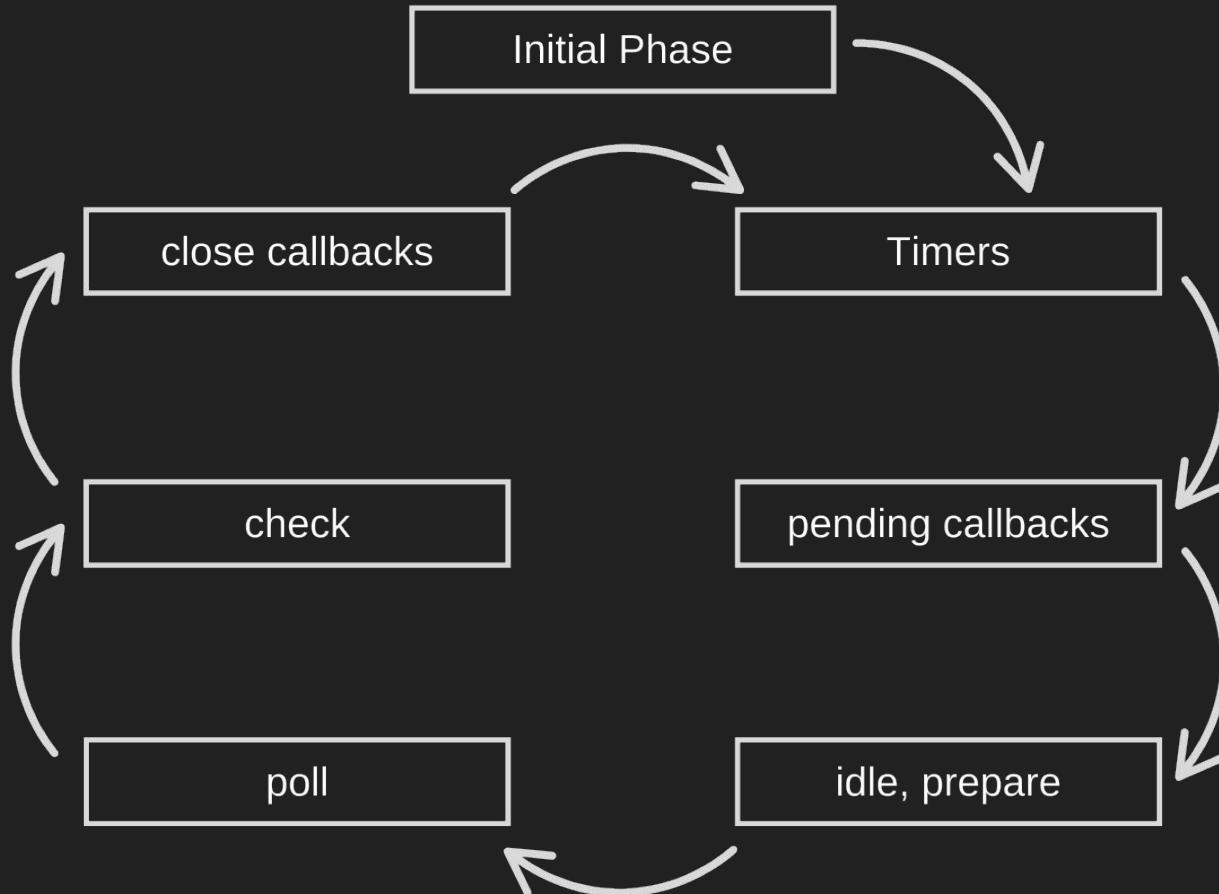
```
5 const clientSuccess = new net.Socket();
6 // Connect to the server that exists (this is example.com)
7 clientSuccess.connect(80, '93.184.215.14', () => {
8   console.log(`Connected to server at ${'93.184.215.14'}
9 });
10 //destroy the connection after 5 seconds
11 setTimeout( ()=> clientSuccess.destroy(), 5000);
12 // Handle close events
13 clientSuccess.on('close', () => {
14   console.error(`Connection closed.`);
    Connected to server at 93.184.215.14:80
    Connection closed.          ←

```

This will trigger after 5
seconds

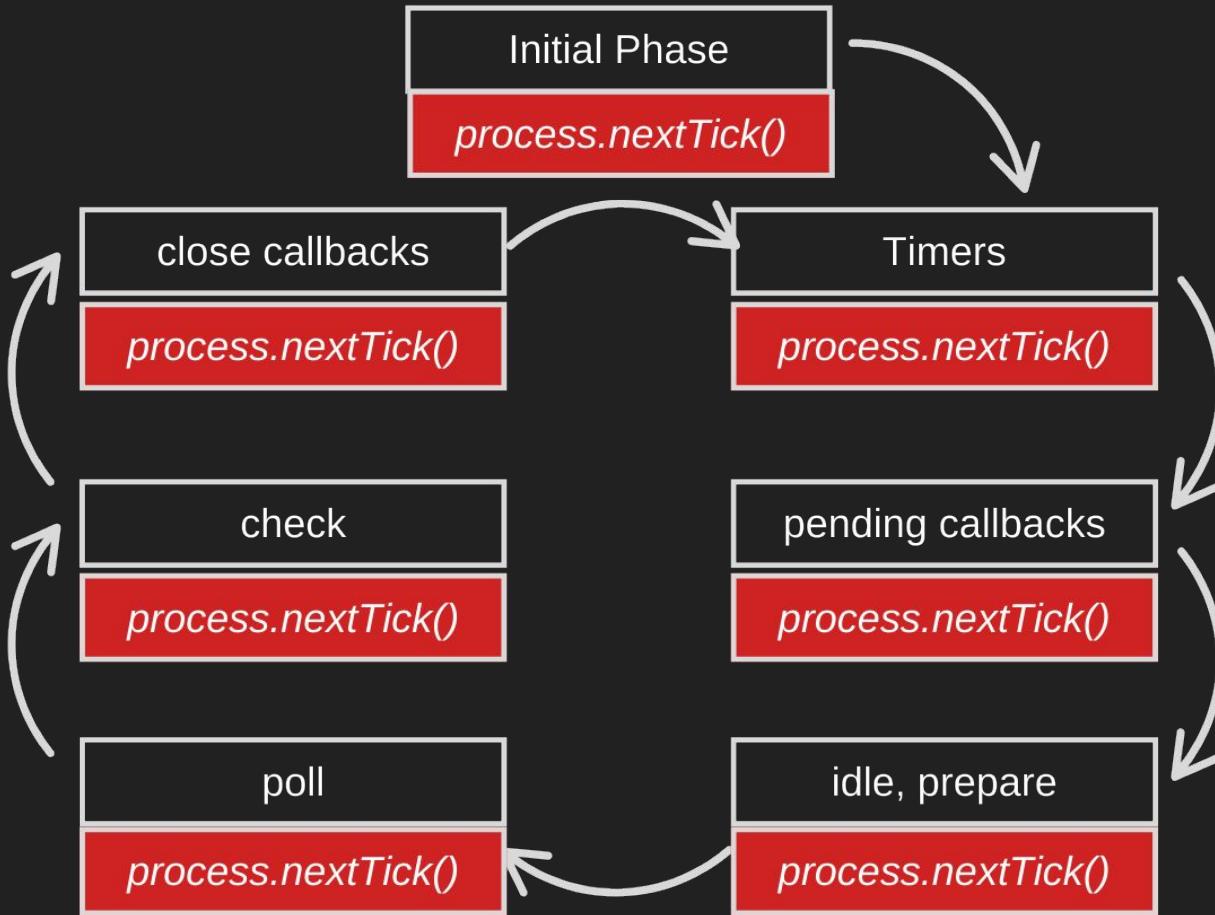
Summary & Demo

- close phase executes after the check phase
- For clean up purposes
- Code exercise →
 - TCP Connection close



process.nextTick()

Hidden phase



process.nextTick()

- Node runs C++
- When jumping from C++ to JavaScript we call it a ‘tick’
- process.nextTick executes after each Tick
- Roughly after each phase
- Priority code (even before setImmediate)
- Promises are scheduled here too!

Why not setTimeout (0)?

- setTimeout is delayed until the timer phase
- nextTick executes as soon as the current phase is finished

NextTick tells you when initial phase ends

- The first nextTick is right after initial phase
- Good indicator to know when did the initial phase really end
- And when did the event loop about to start

Example

```
1 console.log("start");
2 for (let i =0; i< 100000000; i++);
3 console.log ("end");
4 setTimeout(()=> console.log("timer"), 0);
5 process.nextTick( ()=>console.log("nextTick()"))
6
```

Schedules nextTick after initial phase and before timer

Example

```
1 console.log("start");
2 for (let i =0; i< 100000000; i++);
3 console.log ("end");
4 setTimeout(()=> console.log("timer"), 0);
5 process.nextTick( ()=>console.log("nextTick()"))
6
```

- HusseinMac:08-nextTick HusseinNasser\$ node 080-process.nextTick.js
start
end
nextTick()
timer

Another example

08-nextTick > JS 083-nextTickInit.js > test

```
1 let val;  
2 function test () {  
3     console.log(val)  
4 }  
5 test();  
6 val = 1;  
7
```

The output here is “undefined” because the initial phase executed synchronously and test is called before val is initialized

TERMINAL PORTS PROBLEMS OUTPUT DEBUG CONSOLE

```
● HusseinMac:08-nextTick HusseinNasser$ node 083-nextTickInit.js  
undefined  
○ HusseinMac:08-nextTick HusseinNasser$
```

Another example

```
1 let val;  
2 function test () {  
3     console.log(val)  
4 }  
5 //if we call test() directly  
6 //the output is undefined  
7 //if we call it through nextTick  
8 //the output is 1  
9 process.nextTick( test)  
10 val = 1;  
11
```

By scheduling `process.nextTick` we let the main phase initialize, and the output is now 1 (because `val` is now initialized)

Powerful for modules too.

TERMINAL PORTS PROBLEMS OUTPUT DEBUG CONSOLE

- HusseinMac:08~nextTick HusseinNasser\$ node 083-nextTickInit.js
undefined
- HusseinMac:08~nextTick HusseinNasser\$ node 083-nextTickInit.js
1
- HusseinMac:08~nextTick HusseinNasser\$

Danger!

- Don't schedule nextTick in nextTick!
- Recursive will starve other phase (like poll)
- It won't go into nextTick, it will go into the current "Tick"

```
4 ^ process.nextTick( ()=>{  
5   console.log("nextTick after initial phase")  
6 ^   //danger! if you keep doing this it will  
7   //starve other phases  
8   process.nextTick (() =>  
9     console.log ("another nextTick "))  
10 }  
11 )
```

Summary & Demo

- `process.nextTick` can be used for priority workload
- Waiting for the current phase to finish
- Executes after each phase
- Watch out scheduling `nextTick` inside `nextTick`
- Code exercise →
 - `nextTick` examples

How Promises Work

What is behind promises

Promises

- Callbacks are hard to read
- Promises are syntax sugar on top of callbacks
- Async await made it better
- It looks synchronous but its asynchronous
- Promises callbacks are scheduled in the process.nextTick

Example

```
1 console.log("Start Initial phase")
2 function promiseWork () {
3     return new Promise( (resolve, reject) => {
4         setTimeout(() => {
5             //done work
6             console.log("--Processing...\"");
7             const output = "done!"
8             //resolve
9             resolve(output)
10        }, 4000)
11    })
12 }
13 console.log("Before running")
14 const x = promiseWork();
15 console.log("After running")
16
17 x.then(a => console.log("resolved " + a))
18 console.log("End Initial phase")
19 setTimeout( () => console.log("timer"), 0);
```

Async, just powers through..

“Then” is called when resolve is called
catch is called when reject is called

Then is called when it is resolved

Example

```
1 console.log("Start Initial phase")
2 function promiseWork () {
3     return new Promise( (resolve, reject) => {
4         setTimeout(() => {
5             //done work
6             console.log("--Processing..."); 
7             const output = "done!"
8             //resolve
9             resolve(output)
10        }, 4000)
11    } )
12 }
13 console.log("Before running")
14 const x = promiseWork();
15 console.log("After running")
16
17 x.then(a => console.log("resolved " + a))
18 console.log("End Initial phase")
19 setTimeout( () => console.log("timer") , 0);
```

- MacBook-Pro:node-course-content HusseinNasser\$
Start Initial phase
Before running
After running
End Initial phase
timer
--Processing...
resolved done!
- MacBook-Pro:node-course-content HusseinNasser\$

Implement with nextTick

```
1 console.log("Start Initial phase")
2
3 function promiseWork (callback) {
4     //return new Promise( (resolve, reject) => resolve(output) )
5     //do work
6     console.log("Processing...");
7     const output = "done!"
8     //done work, resolve
9     process.nextTick( callback, output ) ← This schedules
10 }                                              the callback in
11 promiseWork( result => {                      the nextTick
12     |     console.log("resolved " + result)
13     | );
14 console.log("End Initial phase")
15 setTimeout( () => console.log("timer") , 0);
```

Async wait?

- Simulate synchronous execution
- Still callbacks
- Anything after the “await” gets scheduled as a callback
- More like a pointer, go “here” after resolving the promise

Example

```
1 console.log("Start Initial phase")
2 function promiseWork () {
3     return new Promise( (resolve, reject) => {
4         //schedule to simulate slow process.
5         setTimeout(() => {
6             //done work
7             console.log("--Processing...");
8             const output = "done!"
9             //resolve
10            resolve(output)
11        }, 4000)
12    }
13 }
14 async function run () {
15     console.log("Before running")
16     const x = await promiseWork();
17     console.log("After running")
18     console.log("resolved " + x)
19 }
20 console.log("End Initial phase")
21 run();
```

The callback is resolved when resolve is called.

The callback is this line, callback points to this line after promise is resolved

Example async await

```
1 console.log("Start Initial phase")
2 function promiseWork () {
3     return new Promise( (resolve, reject) => {
4         //schedule to simulate slow process.
5         setTimeout(() => {
6             //done work
7             console.log("--Processing...");
8             const output = "done!"
9             //resolve
10            resolve(output)
11        }, 4000)
12    })
13 }
14 async function run () {
15     console.log("Before running") →
16     const x = await promiseWork(); →
17     console.log("After running") →
18     console.log("resolved " + x)
19 }
20 console.log("End Initial phase")
21 run();
```

- MacBook-Pro:node-course-content HusseinNasser\$
Start Initial phase
End Initial phase
- Before running
→ --Processing...
→ After running
resolved done!
- MacBook-Pro:node-course-content HusseinNasser\$

Summary & Demo

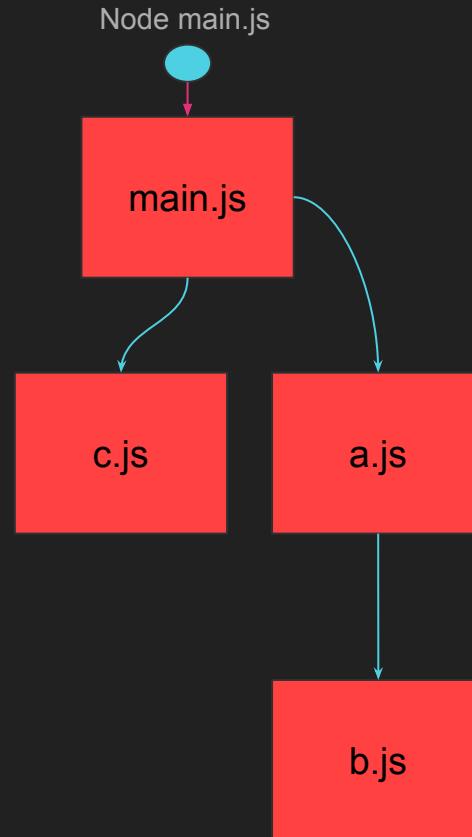
- Promises are implemented as callbacks
- Run on nextTick
- Its asynchronous
- Async await make it “look” synchronous
- Code exercise →
 - Promises vs callbacks examples
 - Make readFile into a promise

require vs import

Importing modules

Importing module

- Importing module is useful
- Libraries, functions
- Reads file from disk + parse it + execute
- Load work relative to file
- But when do they load?



require

- CommonJS syntax
- Synchronously loading modules
- Mostly done in the initial phase
- Supports conditionals
- Can also be called during event loop
 - Timer, Poll etc.
- Can be used in .js only

require example

```
1 console.log("Start Initial phase")
2 const fs = require('fs'); ← Loads in initial
3 require("./098-moduleC.js") phase
4 const f = 100;
5 if (f == 110)
6 {
7     //this won't execute unless
8     //condition is met
9     console.log("Before requiring module A") ← Module A won't load
10    require("./099-moduleA.js") because it doesn't fit
11    console.log("After requiring module A") the condition
12 }
13 fs.readFile(__filename, () => { ← Module D will load
14     console.log("Before requiring module D") ← during poll phase
15     require("./096-moduleD.js") (after initial phase)
16     console.log("After requiring module D")
17 });
18 console.log("End initial phase ")
```

require example

```
1 console.log("Start Initial phase")
2 const fs = require('fs');
3 require ("./098-moduleC.js")
4 const f = 100;
5 if (f == 110)
6 {
7     //this won't execute unless
8     //condition is met
9     console.log("Before requiring module A")
10    require ("./099-moduleA.js")
11    console.log("After requiring module A")
12 }
13 fs.readFile(__filename, () => {
14     console.log("Before requiring module D")
15     require ("./096-moduleD.js")
16     console.log("After requiring module D")
17 });
18 console.log("End initial phase ")
```

● MacBook-Pro:node-course-content HusseinNasser\$
Start Initial phase
--Begin module C
--End Module C
End initial phase
read file done
Before requiring module D
--Begin module D
--End Module D
After requiring module D
○ MacBook-Pro:node-course-content HusseinNasser\$

Import

- Asynchronous
- Executes in the poll phase
- Returns a promise
- Gets scheduled just like any other io
- Can be used in .js and .mjs
- Can be awaited to give synchronous feel
- *Import x from* is different.

Example

```
1 console.log("Start Initial phase")
2 const fs = require('fs');
3 import("./099-moduleA.js") ← Get scheduled in
4 function loadModule() {
5     console.log("before importing module B")
6     import("./097-moduleB.mjs") ← poll phase
7     console.log("after importing module B")
8 }
9 console.log("Before calling B")
10 loadModule();
11 console.log("After calling B")
12
13 setTimeout( ()=>console.log("timer"),0)
14
15 fs.readFile(__filename, () => {
16     console.log("read file done") ← Adding those for
17     setImmediate(() => console.log('immediate'));
18 });
19 console.log("End initial phase ")
```

Example

```
1 console.log("Start Initial phase")
2 const fs = require('fs');
3 import("./099-moduleA.js")
4 function loadModule() {
5     console.log("before importing module B")
6     import("./097-moduleB.mjs")
7     console.log("after importing module B")
8 }
9 console.log("Before calling B")
10 loadModule();
11 console.log("After calling B")
12
13 setTimeout( ()=>console.log("timer"),0)
14
15 fs.readFile(__filename, () => {
16     console.log("read file done")
17     setImmediate(() => console.log('immediate'))
18 });
19 console.log("End initial phase ")
```

● MacBook-Pro:node-course-content HusseinNasser\$
Start Initial phase
Before calling B
before importing module B
after importing module B
After calling B
End initial phase
timer
read file done
--Begin module A
--End Module A
immediate
--Begin module B
--End Module B
○ MacBook-Pro:node-course-content HusseinNasser\$

Import x from is the exception

- Loads immediately before initial phase

```
1 console.log("start")
2 //returns a promise
3 const x = import("./a.mjs")
4 console.log(x)
5 //resolves the promise and get x (returns a module)
6 const y = await import("./b.mjs")
7 console.log(y)
8 //resolves the promise ( like await)
9 //but returns the value
10 import z from "./c.mjs"
11 console.log(z)
12 process.nextTick(()=>console.log("Initial phas
13 console.log("end")
```

```
--Begin module c
--End Module c
start
Promise { <pending> }
--Begin module a
--End Module a
--Begin module b
--End Module b
[Module: null prototype] { default: { test: 'testb' } }
{ test: 'testc' }
end
Initial phase end.
```

Import x from is the exception

- Must be at the top
- Can't be done in conditionals blocks

```
7 if (1==2) {  
8   import xx from "./c.mjs"  
9   console.log(xx)  
0 }  
  
An import declaration can only be used at the top level  
of a module.ts(1473)  
View Problem (CF8) No quick fixes available
```

Summary & Demo

- Require loads modules synchronously often in initial phase
- Import loads modules asynchronously in the poll phase
- Code exercise →
 - Require vs import
 - Async await import

The Anatomy of Node Packages

Breaking it down

Packages

- You can work with individual files with Node
- But dependencies become challenges
- meet packages

package.json

- Manifest of the current package and all its dependencies
- semantic versioning
 - "fetch": "^1.1.0",
 - Means install version 1.1.0 or later but don't go beyond major release 1.,
 - ie anything after 1.1.0 and under < 2.0.0
 - "crepe": "~7.6.0",
 - Means install version 7.6.0 or later but don't go beyond minor 6
 - ie anything after 7.6.0 and under < 7.7.0
- Only has the direct dependencies for this package

package.json

```
1  {
2    "name": "test",
3    "version": "1.0.0",
4    "main": "110-index.js",
5    ▷ Debug
6    "scripts": {
7      "test": "echo \\\"Error: no test specified\\\" && exit 1"
8    },
9    "author": "",
10   "license": "ISC",
11   "description": "",
12   "dependencies": {
13     "fetch": "^1.1.0",
14     "lodash": "^4.17.21"
15   }
```

package-lock.json

- Package.json describes the latest possible
- But we need to know each package version and their dependencies
- Meet package-**lock**.json
- Snapshot of what the dev installed exactly
- Used for consistency
- Has ALL dependencies of packages

package-lock.json

11-package.json > {} package-lock.json > {} packages

```
1 1 < {
2   "name": "test",
3   "version": "1.0.0",
4   "lockfileVersion": 3,
5   "requires": true,
6   "packages": [
7     "": { ...
8     },
9     "node_modules/biskviit": { ...
10    },
11    "node_modules/encoding": { ...
12    },
13    "node_modules/fetch": {
14      "version": "1.1.0",
15      "resolved": "https://registry.npmjs.org/fetch/-/fetch-1.1.0.tgz",
16      "integrity": "sha512-508TwrGzoNblBG/jtK4NFuZwNCkZX6s5GfRN0aGtm+QG
17      "license": "MIT",
18      "dependencies": {
19        "biskviit": "1.0.1",
20        "encoding": "0.1.12"
21      }
22    },
23    "node_modules/iconv-lite": { ...
24    },
25    "node_modules/lodash": { ...
26  }
```

node_modules

- Downloaded packages go to node_modules folder
- Loading modules/packages by looking up this folder
- Contains all downloaded packages
- Based on package-lock.json

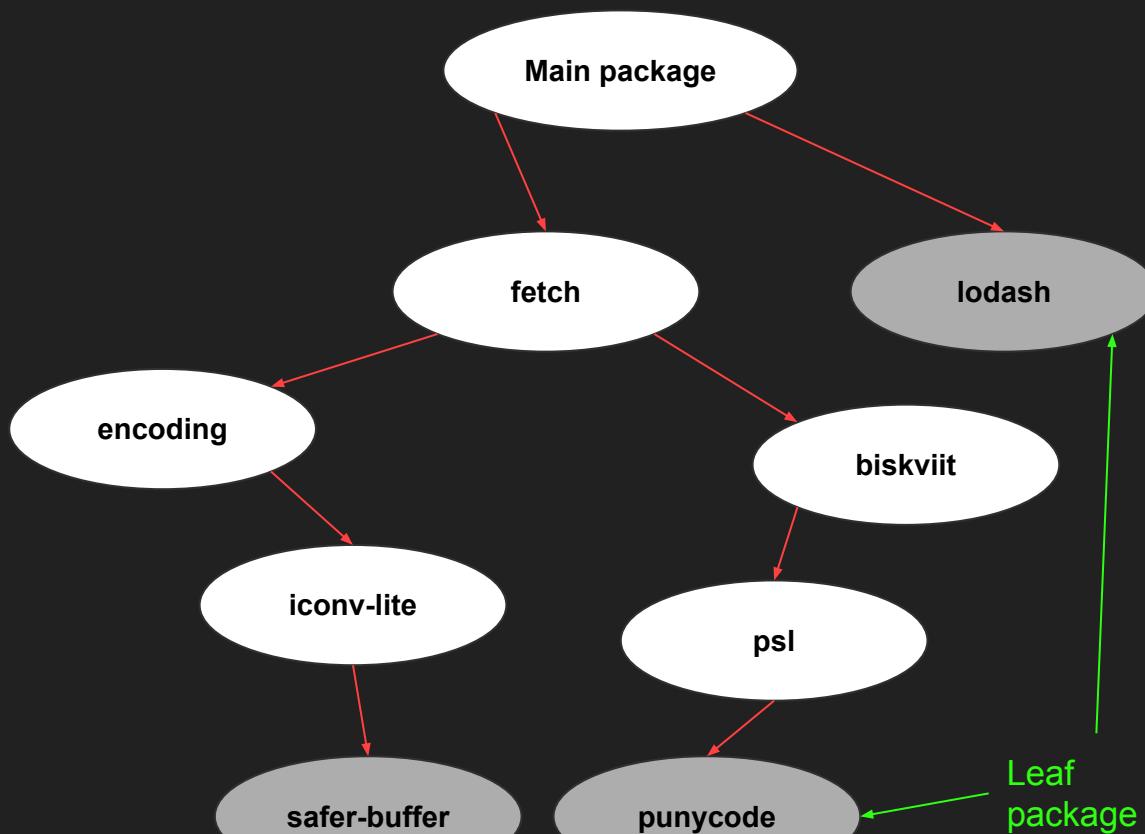
A screenshot of a file explorer interface, likely from a code editor like VS Code. The left pane shows a tree view of a 'node_modules' directory. The 'punycode' package is selected, indicated by a blue border around its folder icon. The right pane displays the detailed contents of the 'punycode' folder, including files like LICENSE-MIT.txt, package.json, punycode.es6.js, punycode.js, README.md, safer-buffer, and .package-lock.json. There are also several small green circular icons on the far right.

```
11-package.json
  node_modules
    biskviit
    encoding
    fetch
    iconv-lite
    lodash
    psl
  punycode
    LICENSE-MIT.txt
    package.json
    punycode.es6.js
    punycode.js
    README.md
    safer-buffer
    .package-lock.json
```

Package manager

- You can do this yourself (if you want)
- OR use a package manager that
 - hosts and manages packages
 - Download and install packages
 - Updates package.json and package-lock.json
- Updates the package.json package-lock.json
- Downloads to node_modules
- example npm , yarn
- npm init creates a package.json

```
1 {  
2   "lockfileVersion": 3,  
3   "requires": true,  
4   "packages": [  
5     "test": {  
6       "dependencies": {  
7         "fetch": "^1.1.0",  
8         "lodash": "^4.17.21"  
9       }  
10     },  
11     "node_modules/biskviit": { ... },  
12     "node_modules/encoding": { ... },  
13     "node_modules/fetch": { ... },  
14     "node_modules/iconv-lite": { ... },  
15     "node_modules/lodash": { ... },  
16     "node_modules/psl": { ... },  
17     "node_modules/punycode": { ... },  
18     "node_modules/safer-buffer": { ... },  
19   ]  
20 }
```



Considerations

- You don't push `node_modules` to git
 - It's massive, and you can always generate it from `package-lock.json`
- You push `package.json` and `package-lock.json`
 - `package.json` is the source
 - `Package-lock.json` is the snapshot
- This way all clients have repeatability

Summary & Demo

- Packages makes dependencies easier
- Managed by package manager
- 3 files package.json, package-json, node_modules
- Code exercise →
 - Create package json
 - We want reference node-fetch
 - View the latest using npm view node-fetch
 - E.g. Latest version 3.3.2, we want a specific version 3.3.1

When does Node terminate

When does Node terminate?

- When there is nothing to do
- But why does Node sometimes doesn't terminate?
- When we call a socket listen there are things to do
- They are things to do they just not visible to you
- Another reason to understand the fundamentals

Example , Connection

```
2 const net = require('net');
3 console.log(`START`);
4 // Create a client
5 const clientSuccess = new net.Socket();
6 // Connect to the server that exists (this is example.com)
7 clientSuccess.connect(80, '93.184.215.14', () => {
8     console.log(`Connected to server at ${'93.184.215.14'}:${80}`);
9 });
10 //destroy the connection after 5 seconds
11 setTimeout( ()=> clientSuccess.destroy(), 5000);
12 // Handle close events
13 clientSuccess.on('close', () => {
14     console.error(`Connection closed.`);
15 });
16
17 //slow down the initial phase
18 for (let i = 1; i <= 500000000; i++);
19 //after the loop ends and initial phase is done, we enter the poll ph
20 console.log(`END`);
```

Once a connection is created,
node will keep reading from it
until it is closed

Example , Server

```
1 const net = require('net');
2
3 console.log("start")
4 //when listening on a socket, accept is being called
5 //to accept new connections, this happens while the socket
6 //is alive
7 const server = net.createServer(() => {}).listen(8080);
8 setTimeout( ()=> server.close(), 4000);
9
10 server.on('listening', () => console.log("listening.."));
11
12 console.log("end");
13
```

Ones a socket is listening
Node will keep calling accept
on it until it is closed.

Summary & Demo

- Node looks like it is not doing anything but it is
- Connections, sockets
- Code exercise →
 - Client connection
 - Server listener

Node Internals

Diving Deeper

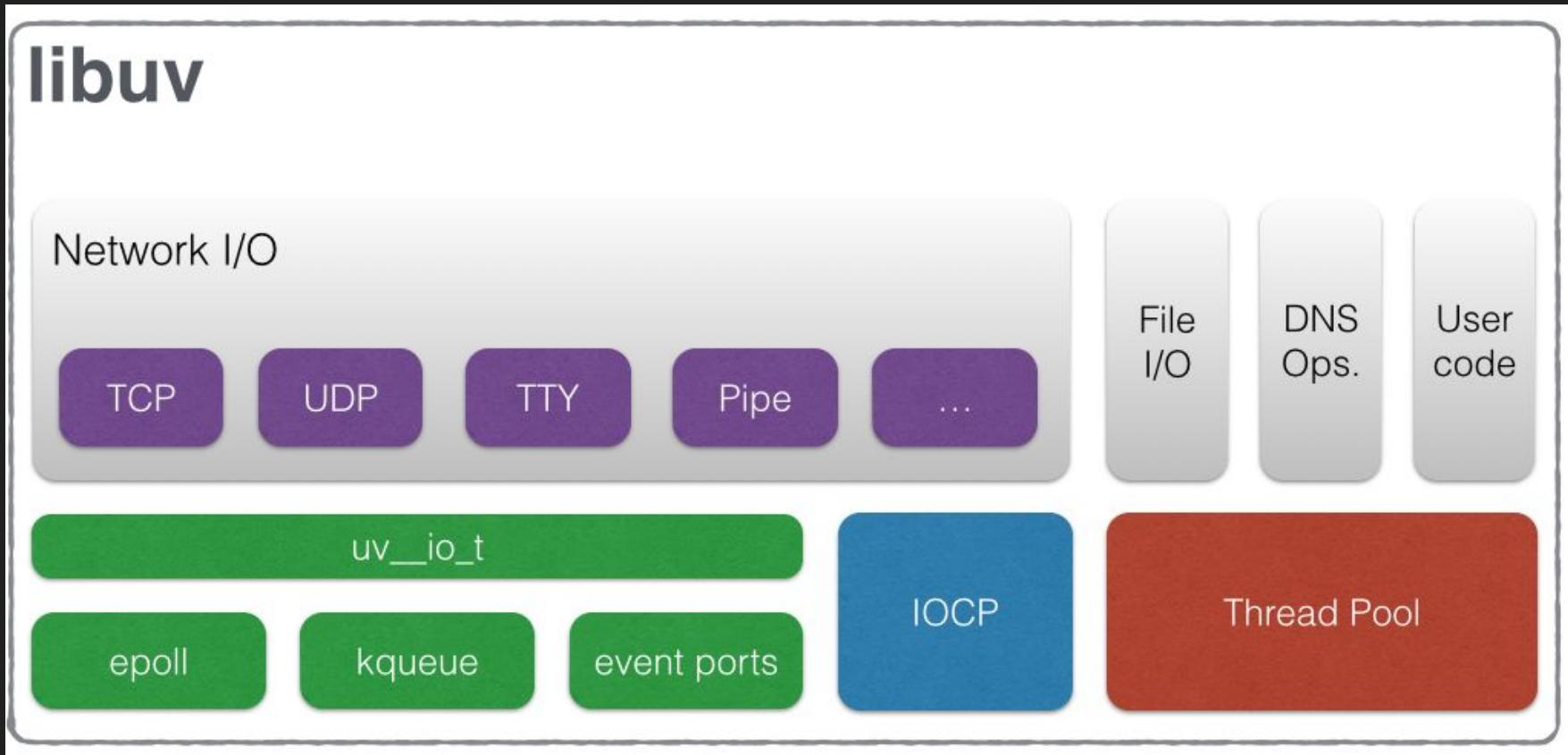
Libuv Overview

Libuv design

libuv

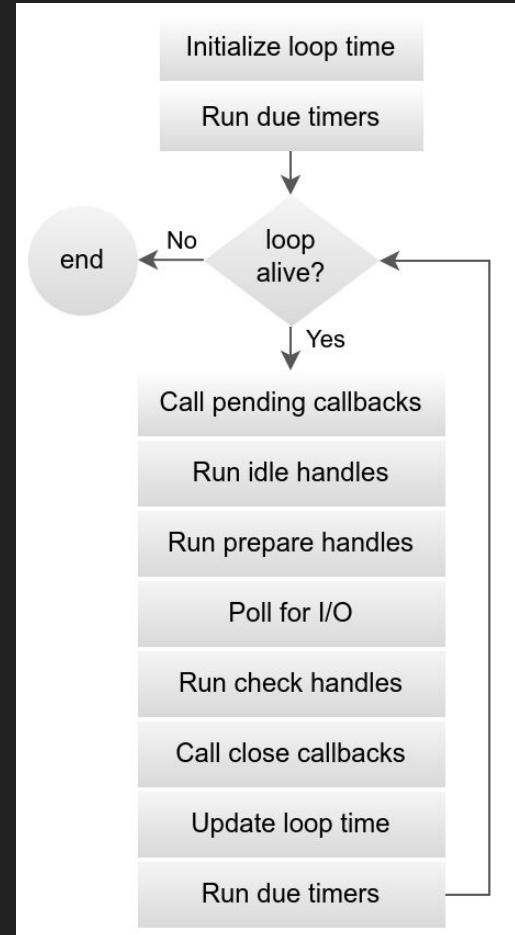
- The IO library behind NodeJS
- Does event loop, polling, networking and file operations
- Cross platform

Libuv components



IO loop

- Almost 1-1 with Node JS loop

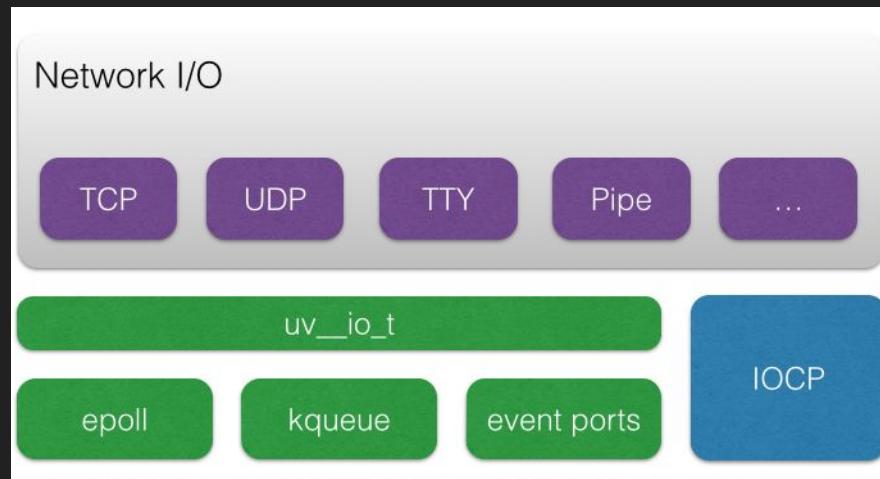


Libuv thread pool

- A pool of threads dedicated for certain tasks
- Can be configured with `UV_THREADPOOL_SIZE`
- Default 4
- Only used by
 - File IO (async)
 - DNS
 - Some user code and libraries (crypto)

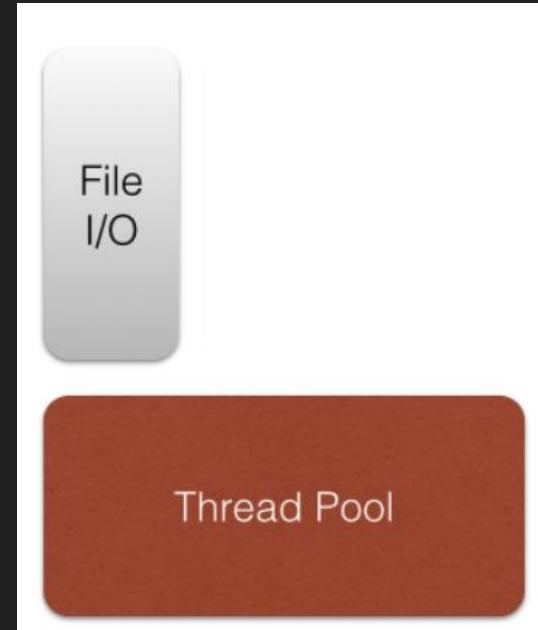
Network IO

- All network IOs are done asynchronously
- Depending on the OS
 - epoll ->Linux
 - Kqueue ->BSD (mac)
 - Event ports (solaris)
 - IOCP (Windows)



File IO

- File cannot be done asynchronous (iouring is the exception)
- So libuv does a blocking read on a thread
- Keeps the main thread unblock
- Simulates asynchronous



Summary

- Node uses Libuv
- For IO loop
- For asynchronous IO
- File/network IO is different, more in next lecture

Asynchronous IO

Non blocking reads and writes

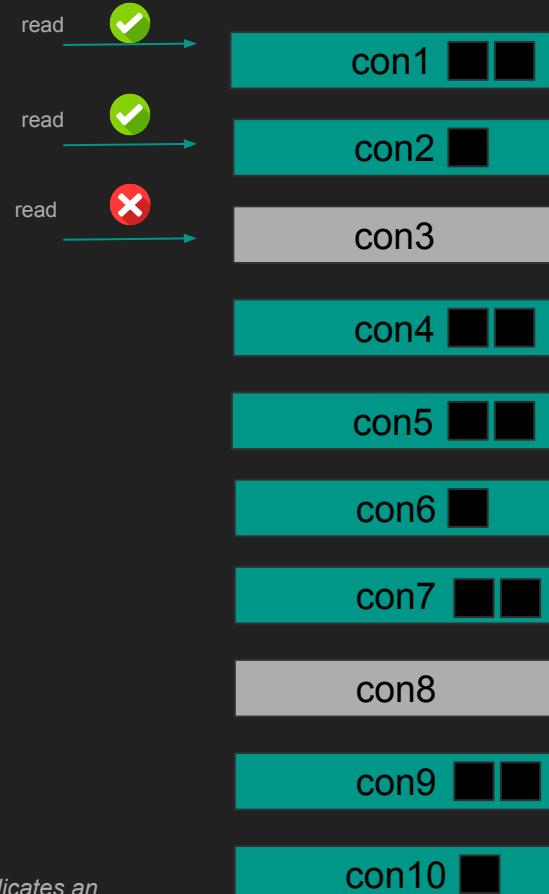
Blocking operations

- Read, write and accept are blocking
- The process cannot move their Program counter
- Read blocks when no data
- accept blocks when no connections
- Leads to context switches and slow down



Blocking operations

```
2 int main ()
3 {
4     for (int i = 0; i < 10; i++)
5     {
6         //read from connection
7         read(connections[i], &buf)
8     }
9     return 0;
10 }
11 }
12 }
```



This is just a pseudo code for simplicity, green indicates the receive buffer for the connection has data to be read, gray indicates an empty receive buffer (client hasn't sent anything). In this example the process gets blocked on reading connection 3 until some data arrives there, while the other connections are starved.

Asynchronous I/O

- Read blocks when no data in receive buffer
- Accept blocks when no connections in accept queue
- Ready approach
 - Ask the OS to tell us if a file is ready
 - When it is ready, we call it without blocking
 - Select,epoll, kqueue
- Completion approach
 - Ask the OS (or worker thread to do the blocking io)
 - When completed notify
 - IOCP, io_uring
- Doesn't work with storage files

Select (polling)

- Select takes a collection of file descriptors for kernel to monitor
- Select is blocking (with a timeout)
- When any is ready, select returns
- Process checks which one is ready (loop w/`FD_ISSET`)
- Process calls read/write/accept etc. on the file descriptor

```
#include <sys/select.h>

typedef /* ... */ fd_set;

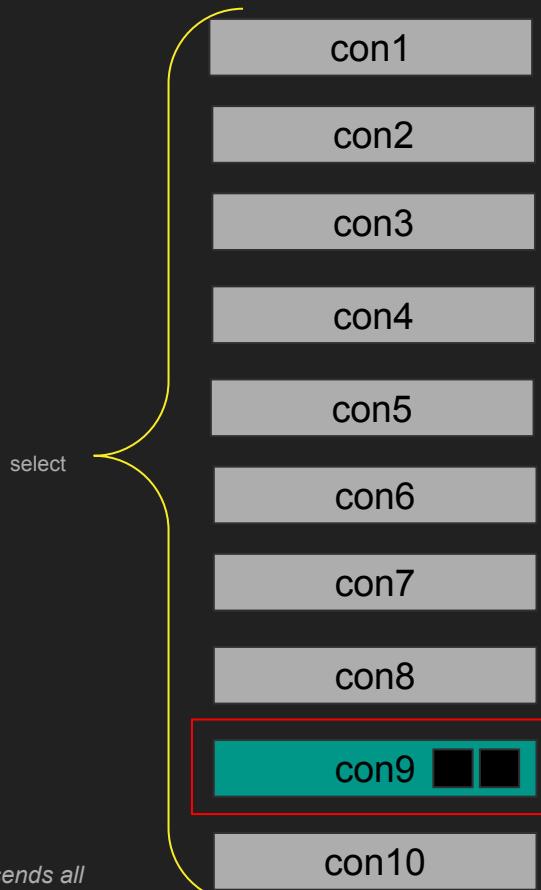
int select(int nfds, fd_set *_Nullable restrict readfds,
           fd_set *_Nullable restrict writefds,
           fd_set *_Nullable restrict exceptfds,
           struct timeval *_Nullable restrict timeout);

void FD_CLR(int fd, fd_set *set);
int FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);

int pselect(int nfds, fd_set *_Nullable restrict readfds,
            fd_set *_Nullable restrict writefds,
            fd_set *_Nullable restrict exceptfds,
            const struct timespec *_Nullable restrict timeout,
            const sigset_t *_Nullable restrict sigmask);
```

select

```
17 int main ()  
18 {  
19     //select an array of collections  
20     //blocks until at least one connection has something to read  
21     select(connections); ←  
22     //if we are here there is something to read but we need to check all  
23     for (int i = 0; i < 10; i++) {  
24         if (FD_ISSET(connections[i])) //check if its ready  
25             //read from connection  
26             read(connections[i], &buf)  
27     }  
28  
29     return 0;  
30 }
```



In this example, only connection 9 has data in the receive buffer, but client is monitoring all 10 connections. The select call sends all 10 file descriptors, kernel checks one by one (all 10) discovers the connection 9 had data, unblocks and return

select

```
17 int main ()  
18 {  
19     //select an array of collections  
20     //blocks until at least one connection has something to read  
21     select(connections);  
22     //if we are here there is something to read but we need to check all  
23     for (int i = 0; i < 10; i++) { ←  
24         if (FD_ISSET(connections[i])) //check if its ready  
25             //read from connection  
26             read(connections[i], &buf) ←  
27     }  
28  
29     return 0;  
30 }
```

con1

con2

con3

con4

con5

con6

con7

con8

con9



con10

The client then gets its program counter incremented, loops through all connections and check which one is ready $O(n)$. It finds that connection 9 is ready and issues the read on it with no blocking.

Select pros and cons

- Pros
 - Avoid reading (unready) resources
 - `async`
- Cons
 - But slow we have to loop through all of them $O(n)$
 - Lots of copying from kernel/user space
 - Supports fixed size of file descriptors

epoll (eventing)

- Register an interest list of the fds (once) in the kernel
- Kernel keeps working and updates the ready list
- User calls epoll_wait, kernel builds an events array of ready list

```
#include <sys/epoll.h>
```

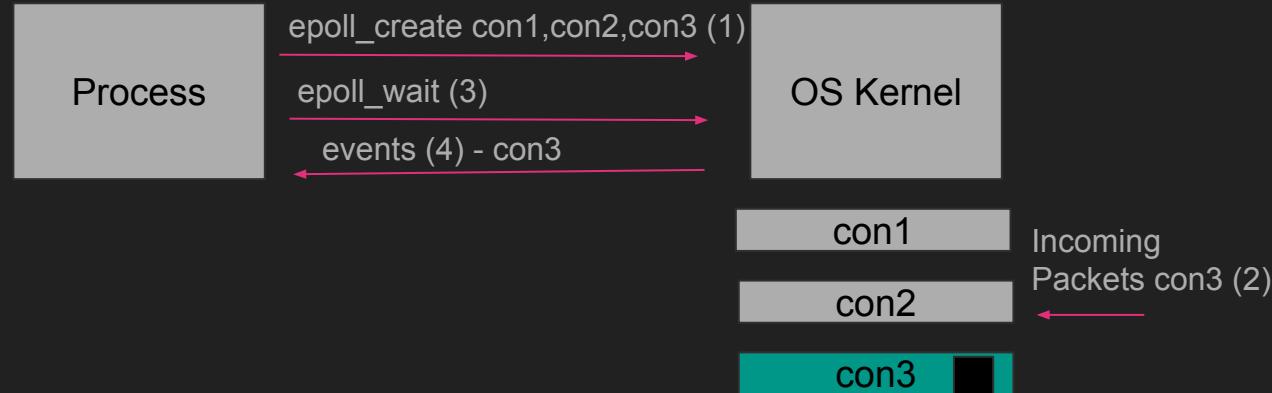
DESCRIPTION [top](#)

The **epoll** API performs a similar task to **poll(2)**: monitoring multiple file descriptors to see if I/O is possible on any of them. The **epoll** API can be used either as an edge-triggered or a level-triggered interface and scales well to large numbers of watched file descriptors.

The central concept of the **epoll** API is the **epoll instance**, an in-kernel data structure which, from a user-space perspective, can be considered as a container for two lists:

- The *interest list* (sometimes also called the **epoll set**): the set of file descriptors that the process has registered an interest in monitoring.
- The *ready list*: the set of file descriptors that are "ready" for I/O. The ready list is a subset of (or, more precisely, a set of references to) the file descriptors in the interest list. The ready list is dynamically populated by the kernel as a result of I/O activity on those file descriptors.

The following system calls are provided to create and manage an epoll instance:



epoll

```
17 int main ()  
18 {  
19     //create an interest list in the kernel  
20     //get back a file descriptor for the epoll instance  
21     e = epoll_create(connections); ←  
22     //add file desc  
23     epoll_ctl(add,connections); ←  
24     //meanwhile kernel is receiving data  
25     while(true) {  
26         //wait on the epoll,  
27         epoll_wait(e, &events, &count)  
28         //we ONLY get the ready stuff  
29         for (int i =0; i < count; i++)  
30             read(events[i].fd.read(), &buf)  
31     }  
32  
33     return 0;  
34 }
```

Again this is pseudo code, we first create the epoll_instance, which lives in the kernel memory. Then we add the connections/file descriptors

epoll e (kernel)

con1

con2

con3

con4

con5

con6

con7

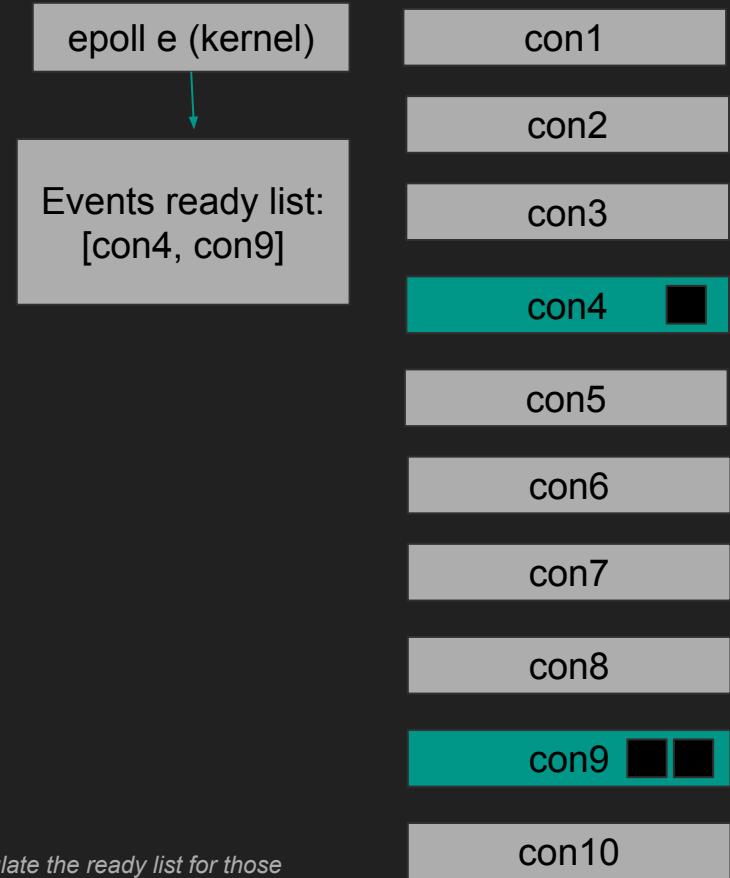
con8

con9

con10

epoll

```
17 int main ()  
18 {  
19     //create an interest list in the kernel  
20     //get back a file descriptor for the epoll instance  
21     e = epoll_create(connections);  
22     //add file desc  
23     epoll_ctl(add,connections);  
24     //meanwhile kernel is receiving data  
25     while(true) {  
26         //wait on the epoll,  
27         epoll_wait(e, &events, &count)  
28         //we ONLY get the ready stuff  
29         for (int i =0; i < count; i++)  
30             read(events[i].fd.read(), &buf)  
31     }  
32  
33     return 0;  
34 }
```



The code process moves on, meanwhile the kernel keeps receiving packets and as it does so, it will populate the ready list for those connections with data. In this case we got data for con9 and con4

epoll

```
17 int main ()  
18 {  
19     //create an interest list in the kernel  
20     //get back a file descriptor for the epoll instance  
21     e = epoll_create(connections);  
22     //add file desc  
23     epoll_ctl(add,connections);  
24     //meanwhile kernel is receiving data  
25     while(true) {  
26         //wait on the epoll,  
27         epoll_wait(e, &events, &count) ←  
28         //we ONLY get the ready stuff  
29         for (int i =0; i < count; i++)  
30             read(events[i].fd.read(), &buf)  
31     }  
32  
33     return 0;  
34 }
```

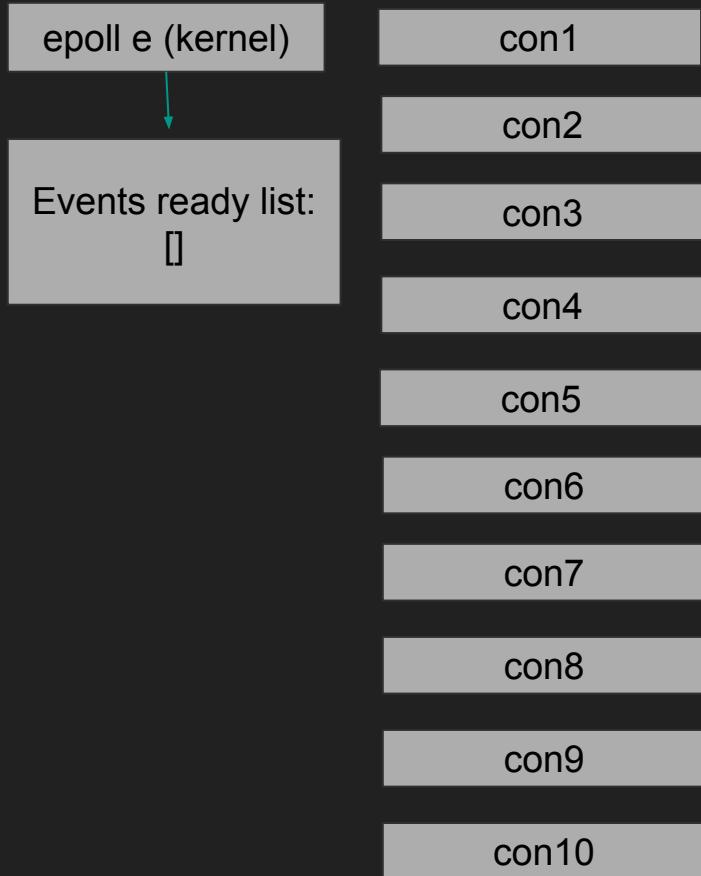


At some point, code calls `epoll_wait`, if the epoll instance is empty wait is blocked, else we immediately send back the array of events to the user space. Only what is ready. Events count is now 2 elements.

epoll

```
17 int main ()  
18 {  
19     //create an interest list in the kernel  
20     //get back a file descriptor for the epoll instance  
21     e = epoll_create(connections);  
22     //add file desc  
23     epoll_ctl(add,connections);  
24     //meanwhile kernel is receiving data  
25     while(true) {  
26         //wait on the epoll,  
27         epoll_wait(e, &events, &count)  
28         //we ONLY get the ready stuff  
29         for (int i =0; i < count; i++)  
30             read(events[i].fd.read(), &buf) ←  
31     }  
32  
33     return 0;  
34 }
```

the user calls read as non-blocking on connection 9 and connection 4 emptying the receive buffers.



epoll drawbacks

- Complex
 - Level-triggered vs edge-triggered
- Only in linux
- Too many syscalls
- Doesn't work on files

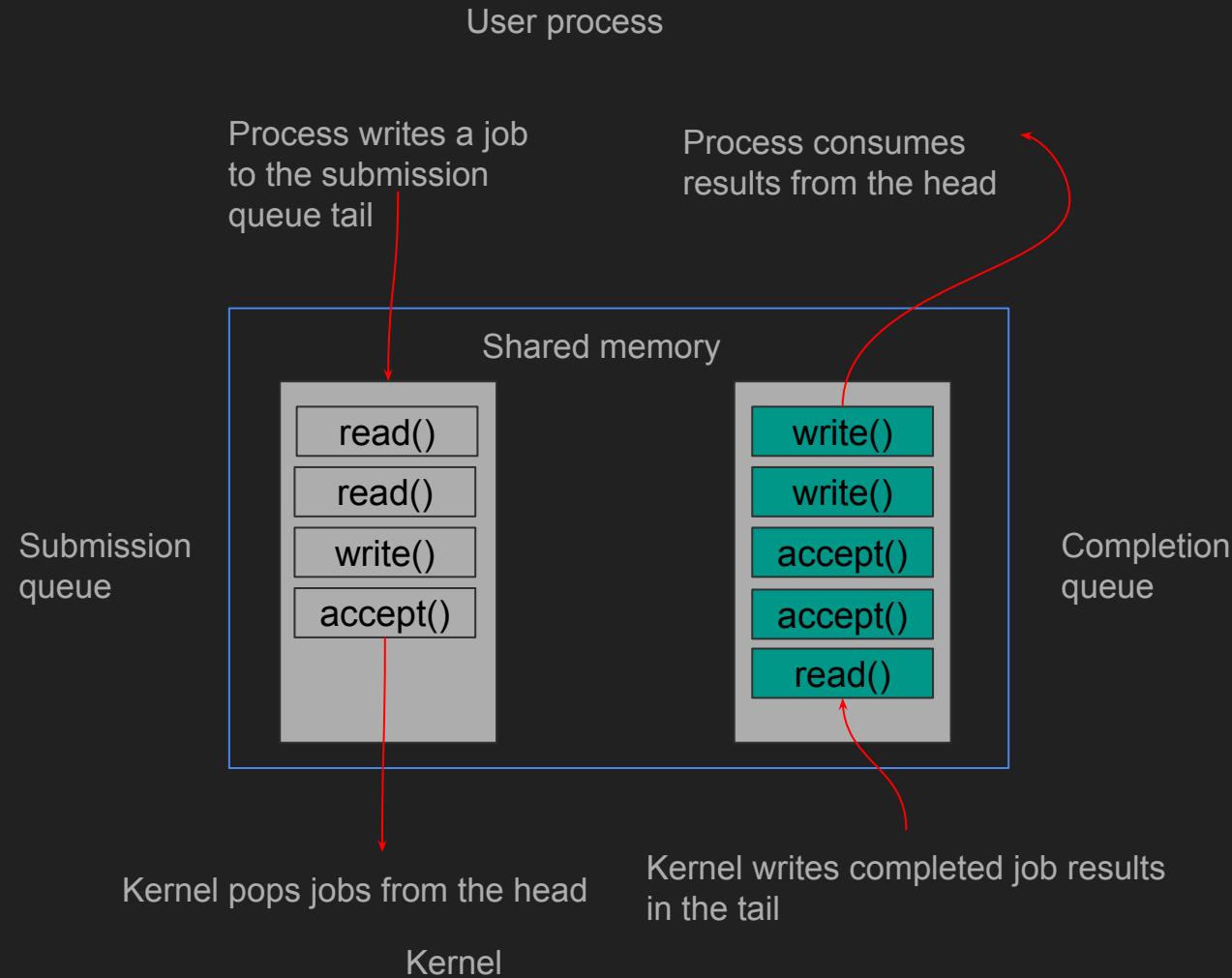
io_uring

- Based on completion
- Kernel does the work
- Shared memory, user puts “job”
- kernel does the work and writes results

io_uring

- Based on completion
- Kernel does the work
- Shared memory, user puts “job”
- kernel does the work and writes results

io_uring

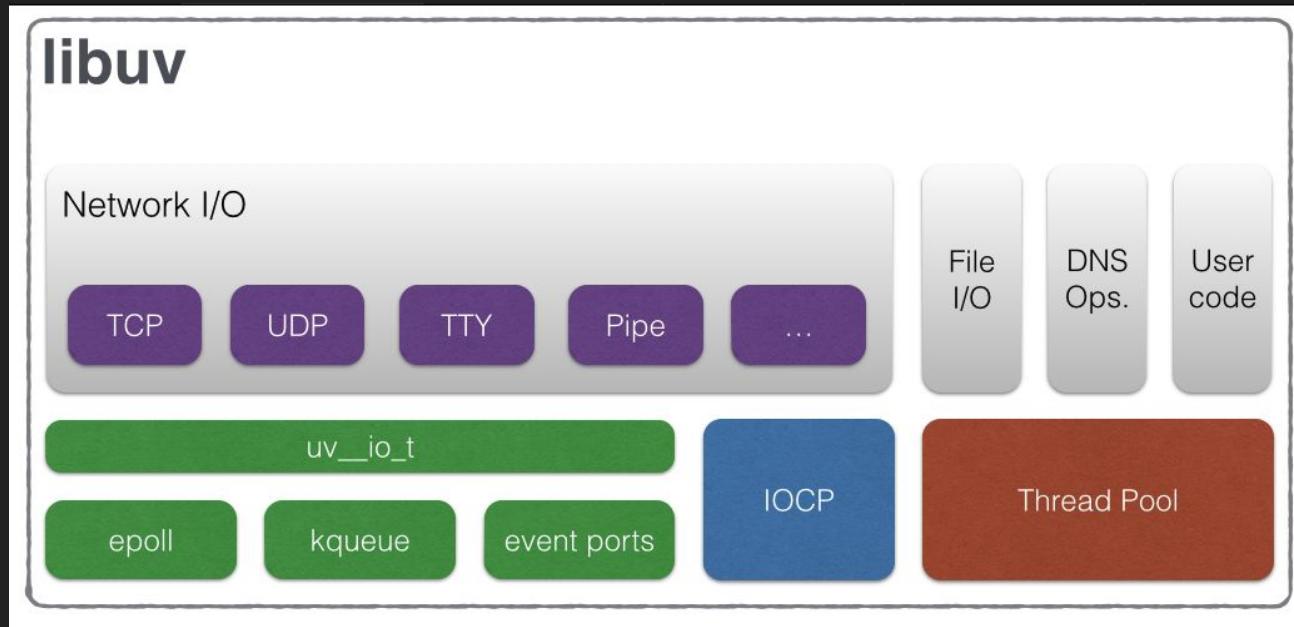


io_uring

- Fast, non-blocking
- Security is a big issue (shared memory)
- Google disabled it for now

Cross platform

- Node (through lib_uv) supports all platforms async io



Summary

- Some kernel syscalls are blocking
- Process put to sleep
- Asynchronous io is a way around it
- Ready based or completion based

Node Network IO

Network operations

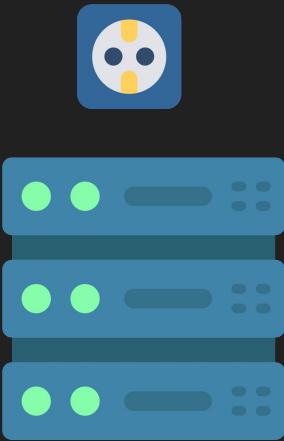
Network IO

- **Connect to a listening server**
 - Client side: Create connection on the client
- **Accept connections from listening Sockets**
 - Server side: Create connections on the server
- **Read data from a connection**
 - Client and server: Reading from connections
- **Write data to a connection**
 - Client and server: Writing to a connections





10.0.0.3



10.0.0.2

10.0.0.3 wants to connect to a 10.0.0.2 on an open port 8080



10.0.0.3

Socket listener
(10.0.0.2: 8080)
Fd:19
on.listening-> fired



10.0.0.2

10.0.0.2 listens on 8080 socket, gets fd 19 listening socket. Fires on listening event

Attempt to
connect to
10.0.0.2:8080



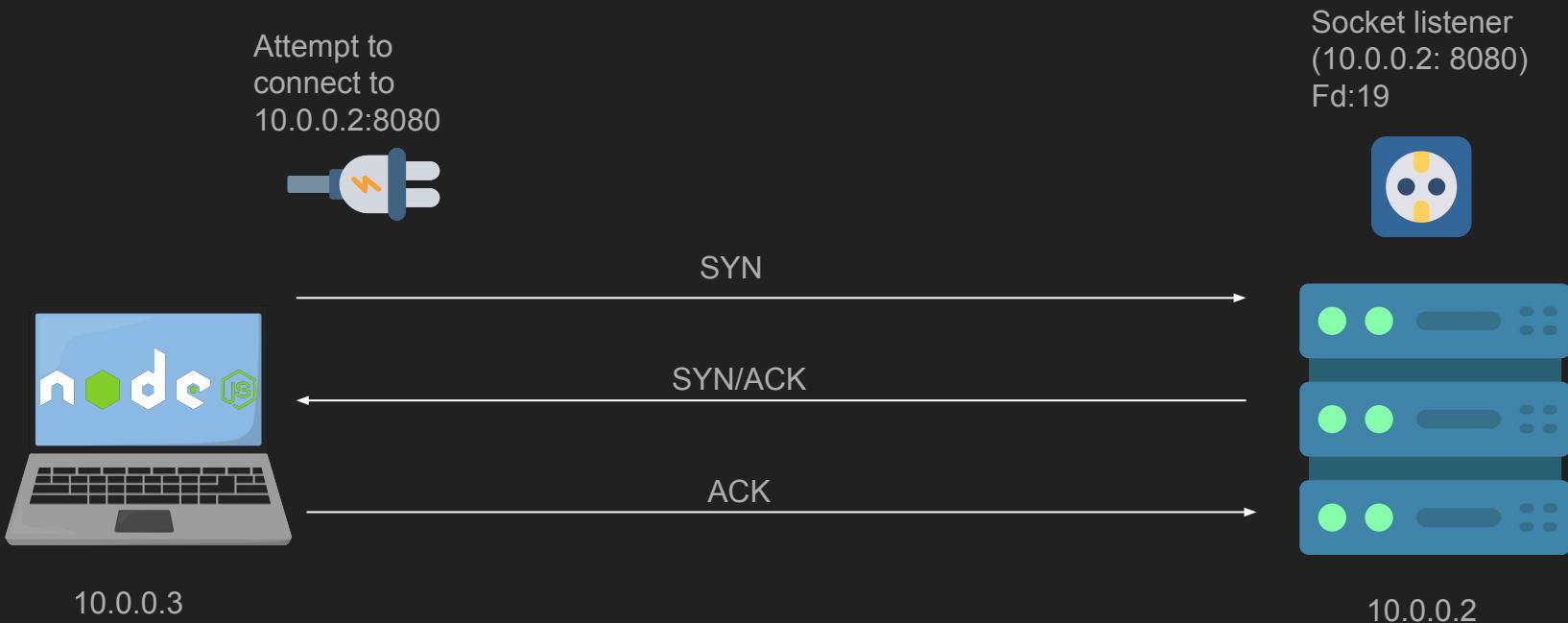
10.0.0.3

Socket listener
(10.0.0.2: 8080)
Fd:19



10.0.0.2

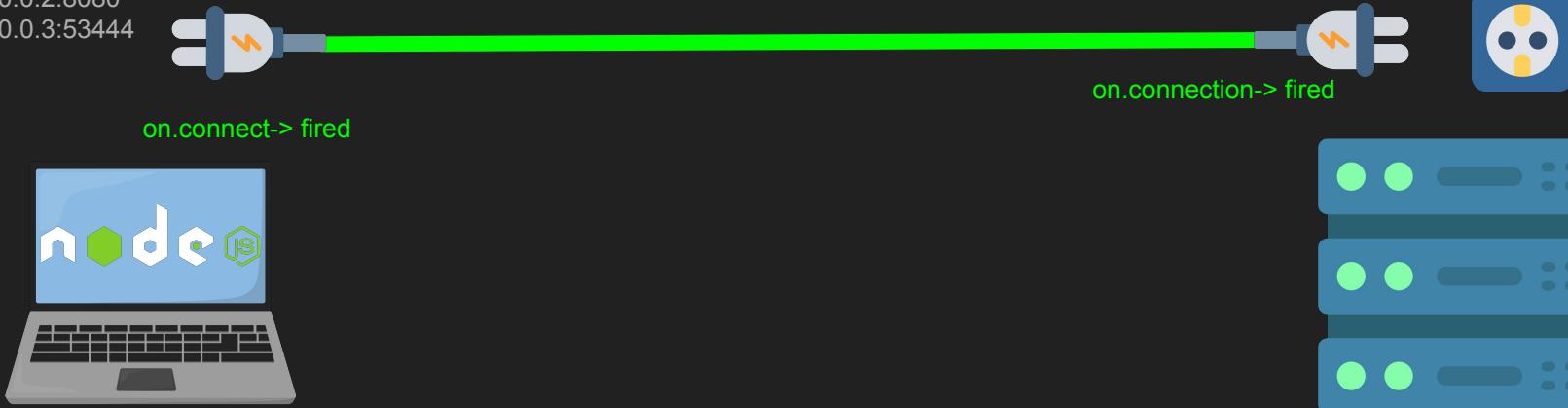
10.0.0.3 attempts to connect to 10.0.0.2:8080



Three way handshake happens

Connected to
10.0.0.2:8080

Connection
created,
Fd 18:
10.0.0.2:8080
10.0.0.3:53444



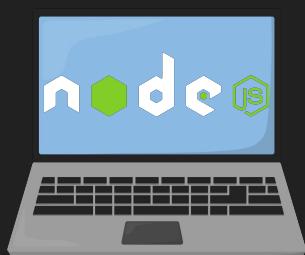
Connection
accepted!,
Fd 20:
10.0.0.2:8080
10.0.0.3:53444

Socket listener
(10.0.0.2: 8080)
Fd:19

on.connection-> fired

Successfully connected, server accepts the connection gets a new fd 20, fires on connection event. client also gets a connection fd 18. Fires on connect event.

Connection
created,
Fd 18:
10.0.0.2:8080
10.0.0.3:53444



10.0.0.3

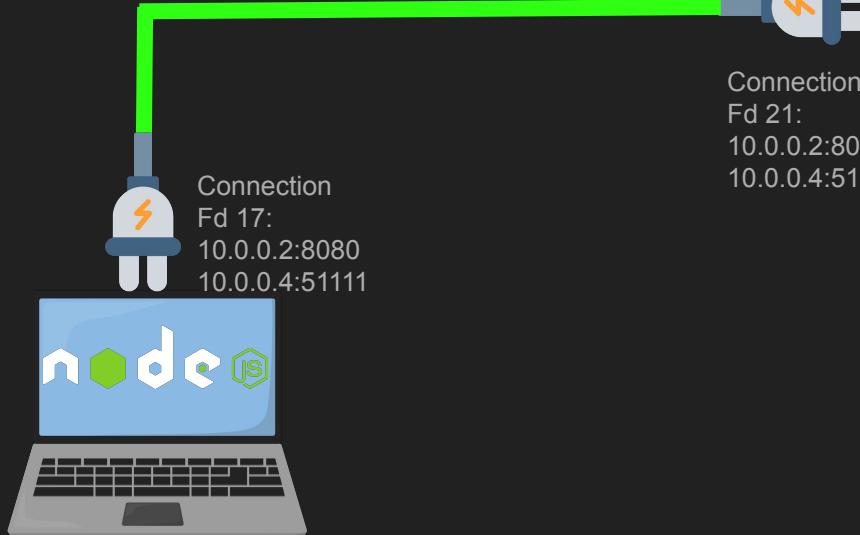


Connection
created,
Fd 20:
10.0.0.2:8080
10.0.0.3:53444

Socket listener
(10.0.0.2: 8080)
Fd:19



Connection
Fd 21:
10.0.0.2:8080
10.0.0.4:51111



10.0.0.4

*Another client connects,
Server accepts the connection fd 21.*

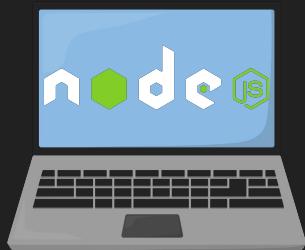
node.js
10.0.0.2

10.0.0.3 Write “hello”

Fd 18:
8080:53444

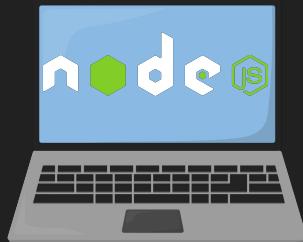
write(18, hello)

hello



10.0.0.3

Fd 17:
8080:51111

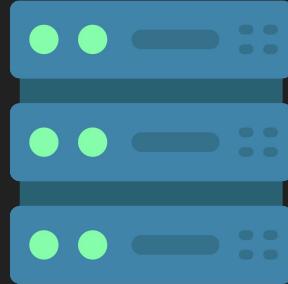


10.0.0.4

Fd 20:
8080:53444



Fd 21:
8080:51111

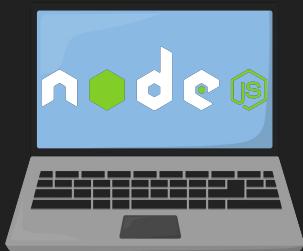


node.js
10.0.0.2

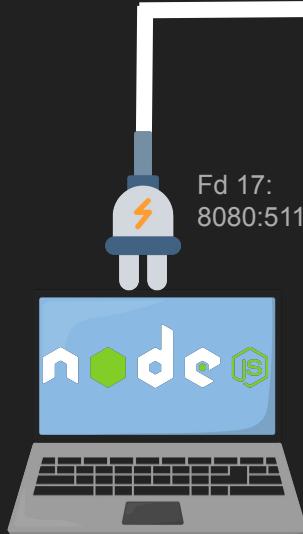
Socket listener
(10.0.0.2: 8080)
Fd:19

10.0.0.2 receives “hello”

Fd 18:
8080:53444



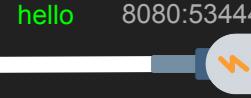
10.0.0.3



10.0.0.4

read*(20)-->on.data fired

Fd 20:
8080:53444



Fd 21:
8080:51111

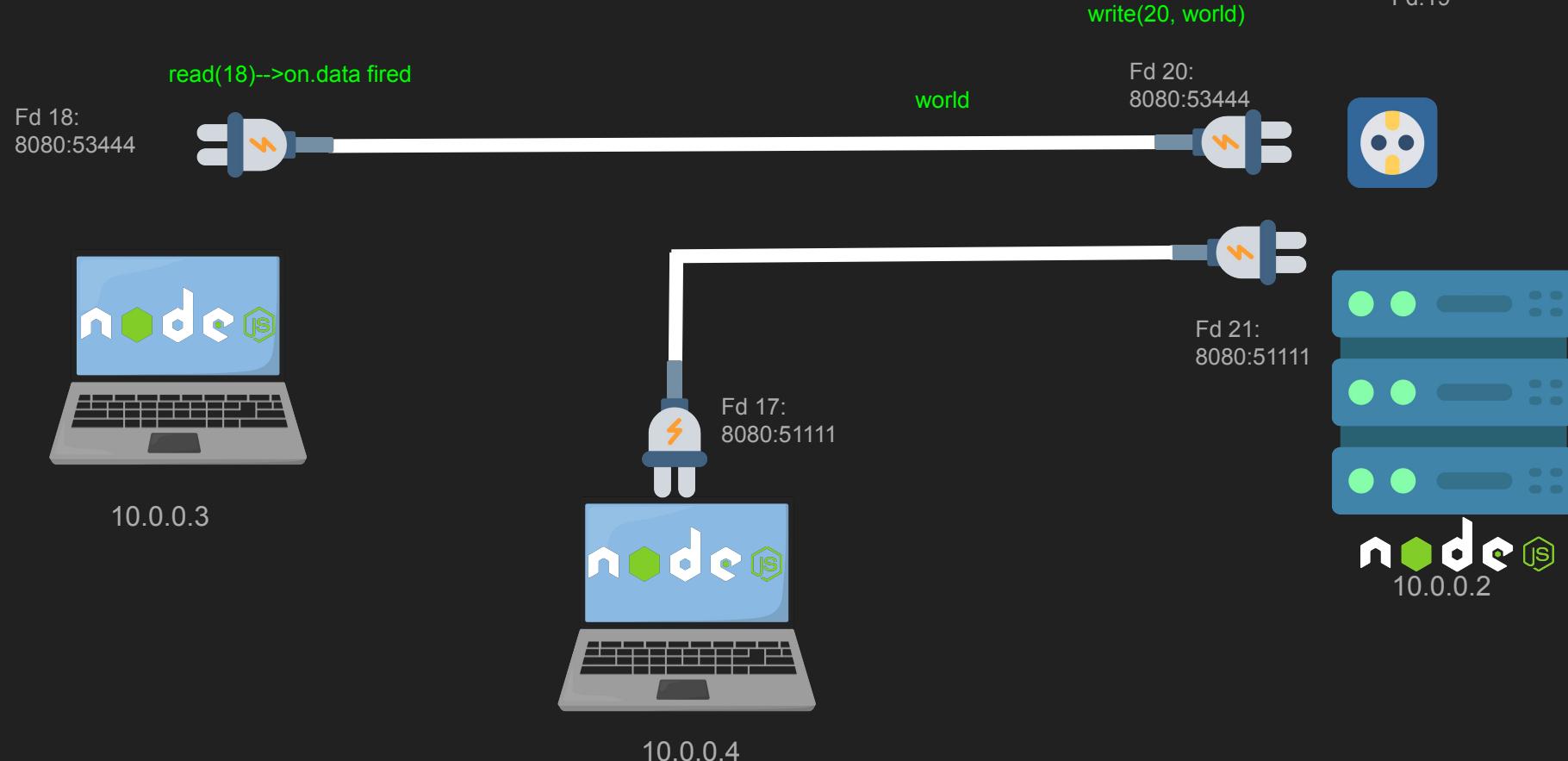


node.js
10.0.0.2

*Node doesn't expose read, it is implicit called.

Socket listener
(10.0.0.2: 8080)
Fd:19

10.0.0.2 writes “world”



Example - Connection

```
13-async-io > js 130-network-read.js > ⌂ close
1  const net = require('net');
2  const client = new net.Socket(); // Create a client socket
3  //connect to a server that exists (this is example.com)
4  //connect, write and close, 10 seconds apart each
5  setTimeout(connect, 0);
6  setTimeout(write, 10000);
7  setTimeout(close, 20000);
8
9  function connect(){
10    //once a connection is created, epoll will be waited on IO (reads and writes)
11    client.connect(80, '93.184.215.14', () => console.log(`Connected to server at ${'93.184.215.14'}:${{80}}`));
12    client.on("data", (data) => console.log( "got some data: " + data.toString()));
13  }
14  function write(){
15    //before this happens we need to check
16    //if the socket is ready to be written to
17    console.log("write initiated")
18    client.write("something...");
19  }
20  function close(){
21    console.log("close event...")
22    client.destroy();
23 }
```

Once a connection is established, a socket file descriptor is created (client side)

When data comes to the connection this is called. Done via read syscall (response)

Issue a write syscall on the connection socket

Example - Listener Socket

```
13-async-io > JS 131-network-listen.js > ⌂ listen
```

```
1  const net = require('net');
2  const client = new net.Socket(); // Create a client
3  let server;
4  //connect to the server that exists (this is example.com)
5  //connect, listen, write, close 10 seconds apart each
6  setTimeout(listen, 0);
7  setTimeout(connect,10000 );
8  setTimeout(write, 20000);
9  setTimeout(close, 30000);
10
11 function listen(){
12   //a listening socket is created
13   server = net.createServer(() => {}).listen(8080);
14   server.on("listening", ()=> console.log("server created."))
15   server.on("connection", (connection) => console.log(`new connection! ${connection.remotePort}`))
16   //when a connection is accepted a new connection is created (the server side connection)
17 }
18
19 function connect(){
20   //a socket connection is created (client)
21   client.connect(8080, '127.0.0.1', () => console.log(`Connected to server at ${'127.0.0.1':$8080}`))
22   client.on("data", (data) => console.log( "got some data" + data.toString()));
23 }
24
25 function close(){
26   console.log("close initiated...")
27   client.destroy();
28   //destroy server, otherwise, will keep polling for connections
29   server.close();
30 }
31
32 function write(){
33   console.log("write initiated")
34   client.write("something..");
```

This creates a listening socket file descriptor

Connection is accepted, connection file descriptor created (server side)

Client establishes a connection , gets a file descriptor

Summary & Demo

- Node polls for IO
- Accept, read, write
- Code exercise →
 - strace node and epoll

Node File IO

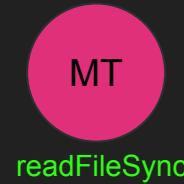
File operations

readFile

- Opens, reads the entire file and closes the file
- May keeps the file open
- Cost of opening /closing/path resolving
- Done asynchronously

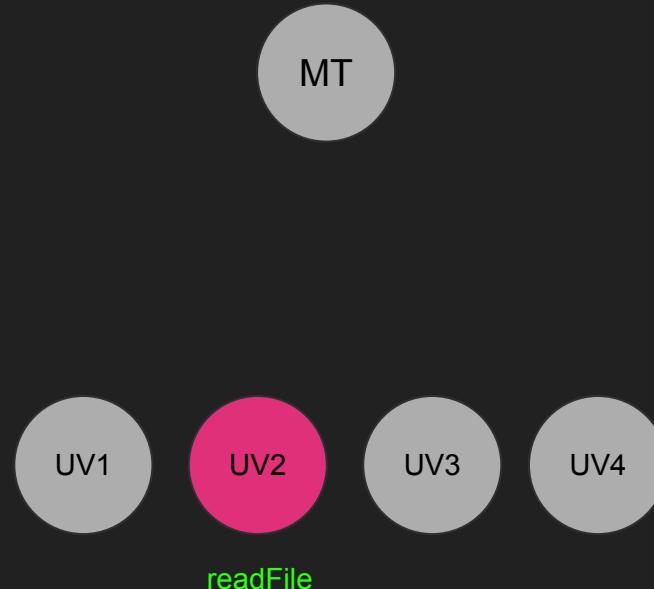
readFileSync

- Executes right there and then (blocks)
- On the main thread
- Whatever phase we happen to be at
 - Including initial phase (watch out)
- Once read is done, data is returned



readFile (asynchronous)

- Schedules execution to the poll phase
- Executes on a lib_uv thread (blocks)
- Once read is done, callback is scheduled
- Main thread is not blocked
- UV_THREADPOOL_SIZE controls it



Example

```
1 const fs = require("fs");
2 //done in the initial phase blocks main thread
3 console.log("Start initial phase")
4 console.log("--About to read file sync")
5 console.log("--Result of sync read " + fs.readFileSync(__filename, 'utf-8').length);
6 console.log("--Finished reading file sync")
7 console.log("--About to read file async")
8 fs.readFile(__filename, 'utf-8', (err,data)=>
9     console.log("--result of async read " + data.length));
10 console.log("--Finished reading file async")
11 //schedule to timer phase
12 setTimeout( ()=> {
13     console.log("----From timer About to read file sync")
14     console.log("----Result of sync read " + fs.readFileSync(__filename, 'utf-8').length);
15     console.log("----From timer Finished reading file sync")
16     console.log("----From timer About to read file async")
17     fs.readFile(__filename, 'utf-8', (err,data)=>
18         console.log("----From timer result of async read " + data.length));
19     console.log("----From timer Finished reading file async")
20 } , 1);
21 for (let i = 0; i < 1000000; i++);
22 console.log(["End initial phase"])
```

Example

```
1 const fs = require("fs");
2 //done in the initial phase blocks main thread
3 console.log("Start initial phase")
4 console.log("--About to read file sync")
5 console.log("--Result of sync read " + fs.readFileSync(__filename, 'utf-8').length)
6 console.log("--Finished reading file sync")
7 console.log("--About to read file async")
8 fs.readFile(__filename, 'utf-8', (err,data)=>
9     console.log("--result of async read " + data.length)
10    console.log("--Finished reading file async")
11 //schedule to timer phase
12 setTimeout( ()=> {
13     console.log("----From timer About to read file sync")
14     console.log("----Result of sync read " + fs.readFileSync(__filename, 'utf-8').length)
15     console.log("----From timer Finished reading file sync")
16     console.log("----From timer About to read file async")
17     fs.readFile(__filename, 'utf-8', (err,data)=>
18         console.log("----From timer result of async read " + data.length);
19         console.log("----From timer Finished reading file async")
20     } , 1);
21 for (let i = 0; i < 1000000; i++);
22 console.log(["End initial phase"])
```

```
MacBook-Pro:node-course-content HusseinNasser$ Start initial phase
--About to read file sync
--Result of sync read 1049
--Finished reading file sync
--About to read file async
--Finished reading file async
End initial phase
----From timer About to read file sync
----Result of sync read 1049
----From timer Finished reading file sync
----From timer About to read file async
----From timer Finished reading file async
--result of async read 1049
----From timer result of async read 1049
```

```
MacBook-Pro:node-course-content HusseinNasser$
```

Working with file descriptors

- To avoid open/close you can open once get a file number
- Issue a read with the file descriptor
- Much faster, avoid path lookups.
- Use open, close
 - Those has sync and async versions
- Read(offset, buffer)

File system flags

[Node.js v23.6.0](#) | ▶ [Table of contents](#) | ▶ [Index](#) | ▶ [Other versions](#) | ▶ [Options](#)

File system flags

The following flags are available wherever the `flag` option takes a string.

- `'a'` : Open file for appending. The file is created if it does not exist.
- `'ax'` : Like `'a'` but fails if the path exists.
- `'a+'` : Open file for reading and appending. The file is created if it does not exist.
- `'ax+'` : Like `'a+'` but fails if the path exists.
- `'as'` : Open file for appending in synchronous mode. The file is created if it does not exist.
- `'as+'` : Open file for reading and appending in synchronous mode. The file is created if it does not exist.
- `'r'` : Open file for reading. An exception occurs if the file does not exist.
- `'rs'` : Open file for reading in synchronous mode. An exception occurs if the file does not exist.
- `'r+'` : Open file for reading and writing. An exception occurs if the file does not exist.
- `'rs+'` : Open file for reading and writing in synchronous mode. Instructs the operating system to bypass the local file system cache.

This is primarily useful for opening files on NFS mounts as it allows skipping the potentially stale local cache. It has a very real impact on I/O performance so using this flag is not recommended unless it is needed.

This doesn't turn `fs.open()` or `fsPromises.open()` into a synchronous blocking call. If synchronous operation is desired, something like `fs.openSync()` should be used.

- `'w'` : Open file for writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx'` : Like `'w'` but fails if the path exists.
- `'w+'` : Open file for reading and writing. The file is created (if it does not exist) or truncated (if it exists).
- `'wx+'` : Like `'w+'` but fails if the path exists.

File streams

- `readFile` loads everything in memory
- Can't read large files
- [ERR_FS_FILE_TOO_LARGE]: File size (7322357281) is greater than 2 GiB
- Use streams
- Will have dedicated lecture on streams

File streams

```
1 const fs = require("fs");
2
3 const p = '/Users/HusseinNasser/Desktop/udemy/backendcourse/Outline.mp4'
4
5 const stream = fs.createReadStream(p);
6
7 stream.on('data', (chunk) => {
8     console.log('Chunk:', chunk.length); // Process each chunk
9 });
10
11 stream.on('end', () => {
12     console.log('File reading completed.');
13 });
14
15 stream.on('error', (err) => {
16     console.error('Error reading file:', err);
17 });
18
```

Issues multiple
async file reads with
fixed buffer size

Gotchas

- The callback of `readFile` indicates the time the callback was executed
- Not when the actual read has occurred/done
- It possible the read has finished a while back but poll phase didn't picked it up

Summary & Demo

- Node file io
- Sync vs async
- Threads
- Streams
- Code exercise →
 - strace node file io
 - File read sync vs async
 - File read with file descriptors
 - File read with streams

Node HTTP

HTTP module

HTTP

- Hypertext transfer protocol
- Request / Response
- Node implements HTTP client and server
- Supports HTTP/1.1 and HTTP/2
- On top of TCP

Client / Server

- (Client) Browser, python or javascript app, or any app that makes HTTP request
- (Server) HTTP Web Server, e.g. IIS, Apache Tomcat, NodeJS, Python Tornado

HTTP Request



HTTP Request

```
curl -v http://husseinnasser.com/about
```

```
> GET /about HTTP/1.1
> Host: husseinnasser.com
> User-Agent: curl/7.79.1
> Accept: */*
```

HTTP Response

Protocol

Code

Code Text

Headers

Body

HTTP Response

```
< HTTP/2 301
< location: https://www.husseinnasser.com/about
< date: Wed, 26 Oct 2022 17:10:59 GMT
< content-type: text/html; charset=UTF-8
< server: ghs
< content-length: 232
< x-xss-protection: 0
< x-frame-options: SAMEORIGIN
<
<HTML><HEAD><meta http-equiv="content-type" content
<TITLE>301 Moved</TITLE></HEAD><BODY>
```

HTTP

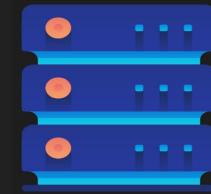
open



GET /

Headers+
index.html
<html>...

80



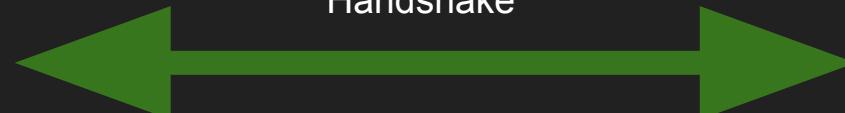
close

HTTPS

open



Handshake

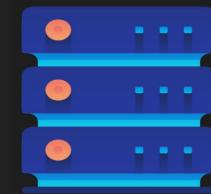


GET /



Headers+
index.html
<html>...

443



close



HTTP 1.1

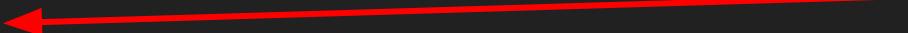
open



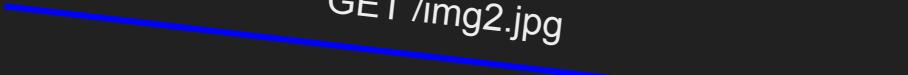
GET /index.html



GET /img1.jpg



GET /img2.jpg



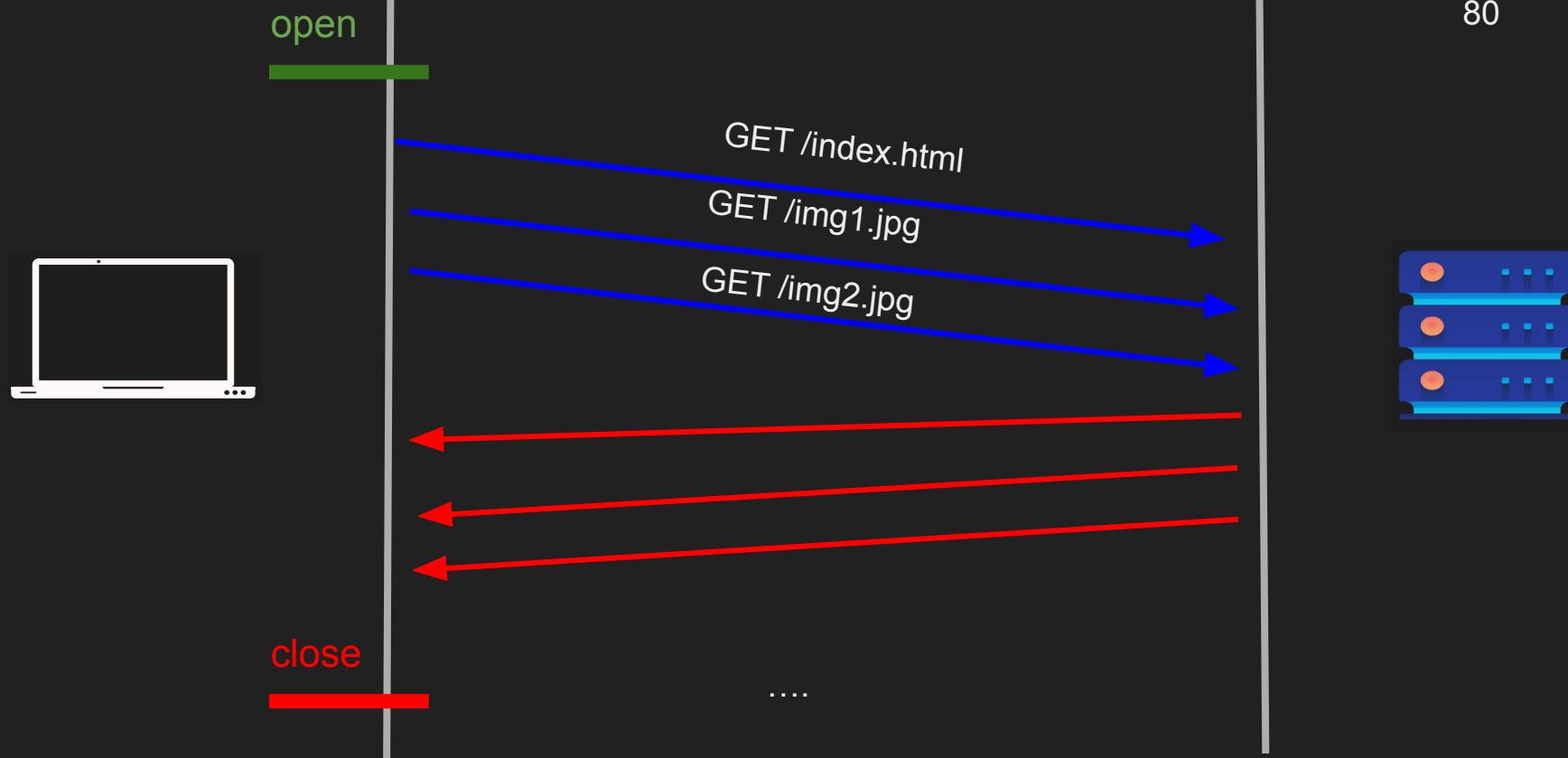
close

80



...

HTTP 1.1 Pipelining



HTTP 1.1

- Persisted TCP Connection
 - keepalive
- Low Latency & Low CPU Usage
- Fixed length response
- Streaming with Chunked transfer
- Proxying

HTTP Client

- Client connects to a server
- Client gets a connection
- Client writes a request to the connection
- Client gets back a response



10.0.0.3



Connection
created,
Fd 18:
10.0.0.2:8080
10.0.0.3:53444

Connection
accepted!,
Fd 20:
10.0.0.2:8080
10.0.0.3:53444
Socket listener
(10.0.0.2: 8080)
Fd:19

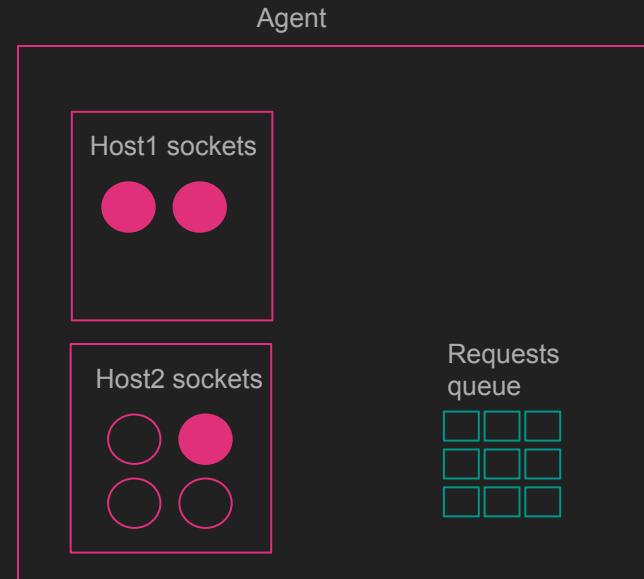


10.0.0.2



Node HTTP agent (`http.Agent`)

- Client side
- Supports pool of connections (per host)
- Queue of requests
- Keep socket alive to keep the connection alive
- There is always an agent, a global one is default

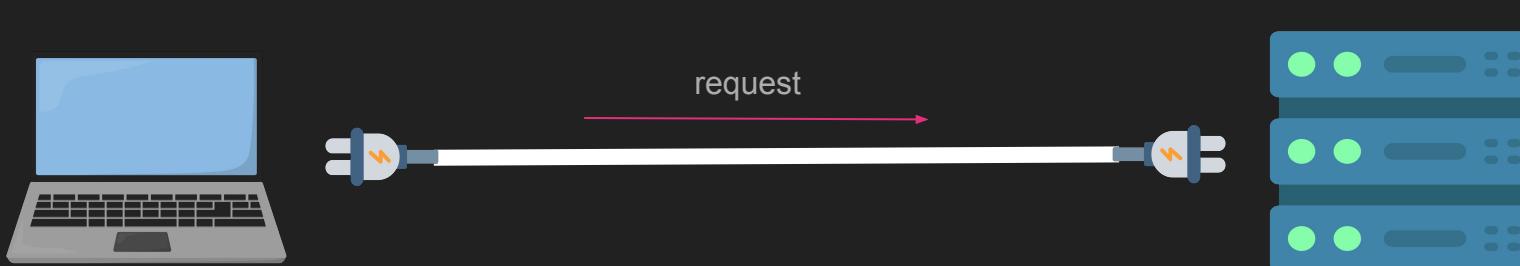


Node Request (`http.clientRequest`)

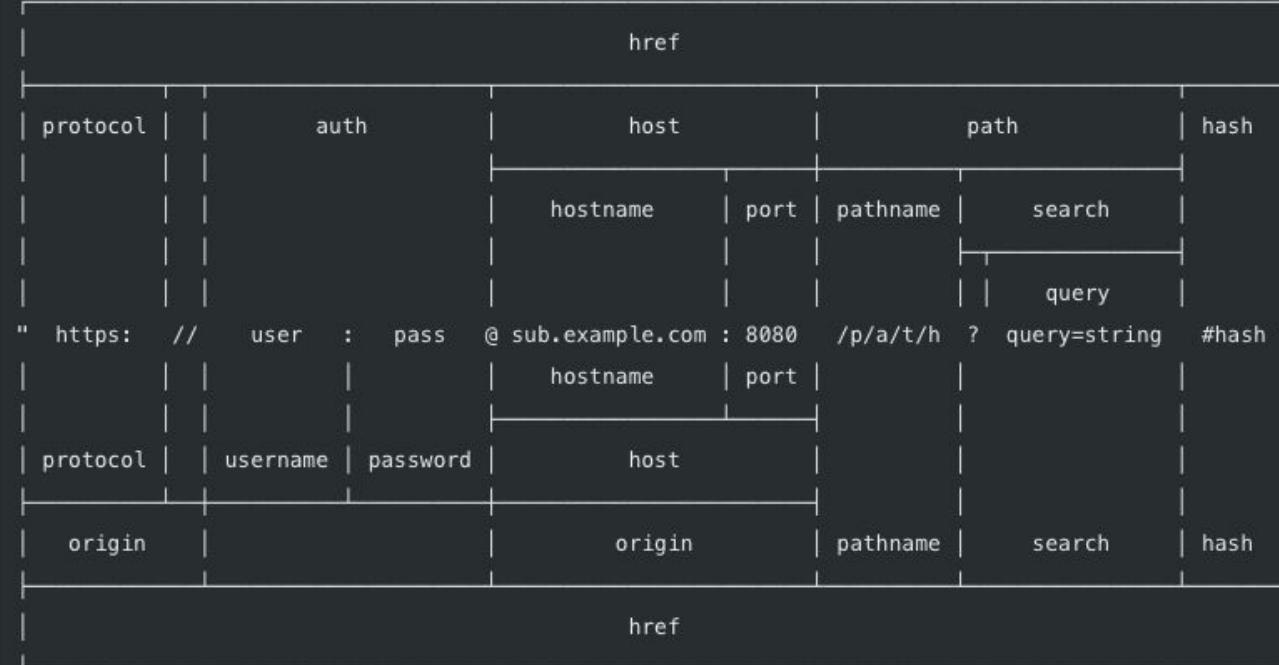
- Using an agent, send a request
- If no agent is provided, use the global one
- Pick a free connection and writes the request to it
- If no free connection is available create one
 - Unless `maxSockets` is reached

Node Request (`http.clientRequest`)

- Takes a URL
 - String or URL object (why?)
 - Parsing URLs isn't cheap, you can parse once and pass it.
- And options
 - Method, path, headers, agent
 - Many more!



A look at URL anatomy

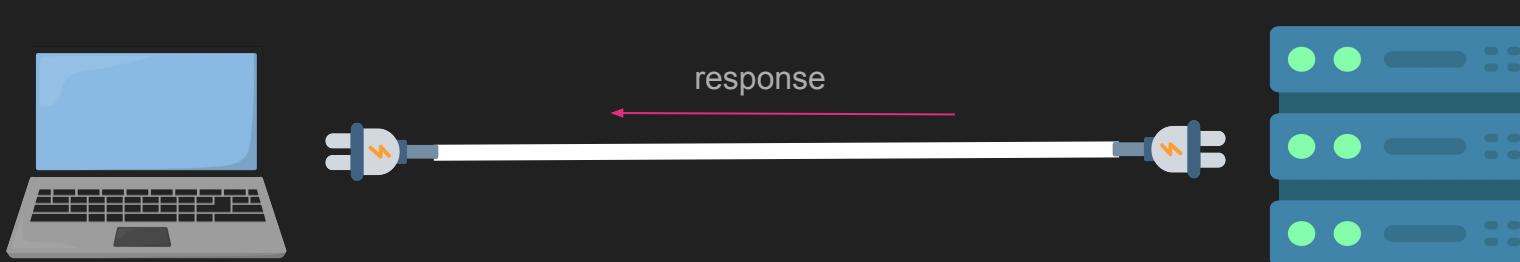


(All spaces in the "" line should be ignored. They are purely for formatting.)

<https://nodejs.org/api/url.html>

Client on Response

- Response event
- We get a Readable stream
- Backend writes headers, status then body on the socket
- It only contains the headers, status code
- Read can happen by assigning “data” event
 - Read it from the kernel to user space
 - Parse it
 - Encode it



When the response is ready

- The on response callback is fired when a response is received
- What does that mean?
- Node reads from the connection
- Parses the data until it gets the end of a response
- Packages the response as object and puts it on the callback

Response start → < HTTP/1.1 200 OK
< Content-Type: text/html
< Date: Thu, 16 Jan 2025 00:04:36 GMT
< Content-Length: 30
< Connection: keep-alive

Response end → < <html><body>xxxx</body></html>

On response fires

Request options

- `agent <http.Agent> | <boolean>` Controls `Agent` behavior. Possible values:
 - `undefined`(**default**): use `http.globalAgent` for this host and port.
 - `Agent` Object: explicitly use the passed in `Agent`.
 - `false`: causes a new `Agent` with default values to be used.
- `auth <string>` Basic authentication ('user:password') to compute an Authorization header.
- `createConnection<Function>` A function that produces a socket/stream to use for the request when the `agent` option is not used. This can be used to avoid creating a custom `Agent` class just to override the default `createConnection` function. See `agent.createConnection()` for more details. Any `Duplex` stream is a valid return value.
- `defaultPort <number>` Default port for the protocol. **Default:** `agent.defaultPort` if an `Agent` is used, else `undefined`
- `family <number>` IP address family to use when resolving `host` or `hostname`. Valid values are `4` or `6`. When unspecified, both IP v4 and v6 will be used.
- `headers <Object>` An object containing request headers.
- `hints <number>` Optional `dns.lookup()` hints.
- `host <string>` A domain name or IP address of the server to issue the request to. **Default:** 'localhost'.
- `hostname <string>` Alias for `host`. To support `url.parse()`, `hostname` will be used if both `host` and `hostname` are specified.
- `insecureHTTPParser<boolean>` If set to `true`, it will use a HTTP parser with leniency flags enabled. Using the insecure parser should be avoided. See `--insecure-http-parser` for more information. **Default:** `false`
- `joinDuplicateHeaders<boolean>` It joins the field line values of multiple headers in a request with `,` instead of discarding the duplicates. See `message.headers` for more information. **Default:** `false`.
- `localAddress <string>` Local interface to bind for network connections.
- `localPort <number>` Local port to connect from.
- `lookup <Function>` Custom lookup function. **Default:** `dns.lookup()`.
- `maxHeaderSize <number>` Optionally overrides the value of `--max-http-header-size`(the maximum length of response headers in bytes) for responses received from the server. **Default:** 16384 (16 kB).
- `method <string>` A string specifying the HTTP request method. **Default:** 'GET'.
- `path <string>` Request path. Should include query string if any. E.G. '/index.html?page=12'. An exception is thrown when the request path contains illegal characters. Currently, only spaces are rejected but that may change in the future. **Default:** '/'.
- `port <number>` Port of remote server. **Default:** `defaultPort` if set, else 80.
- `protocol <string>` Protocol to use. **Default:** 'http:'.
- `setDefaultHeaders<boolean>`: Specifies whether or not to automatically add default headers such as `Connection` `Content-Length`, `Transfer-Encoding` and `Host`. If set to `false` then all necessary headers must be added manually. Defaults to `true`.
- `setHost <boolean>`: Specifies whether or not to automatically add the `Host` header. If provided, this overrides `setDefaultHeaders` Defaults to `true`.
- `signal <AbortSignal>`: An `AbortSignal` that may be used to abort an ongoing request.
- `socketPath <string>` Unix domain socket. Cannot be used if one of `host` or `port` is specified, as those specify a TCP Socket.
- `timeout <number>`: A number specifying the socket timeout in milliseconds. This will set the timeout before the socket is connected.
- `uniqueHeaders <Array>` A list of request headers that should be sent only once. If the header's value is an array, the items will be joined using `,`.

Some options*

- agent `<http.Agent>` : The agent connection pool (Default global pool)
- port `<number>` Port of remote server. **Default:** defaultPort if set, else 80.

Example

```
1 const http = require("node:http");
2 const req = http.request("http://example.com", {"method": "GET"});
3
4 req.on("response", res => {
5     console.log(res.headers)
6     console.log(res.statusCode);
7     //set the encoding
8     res.setEncoding('utf-8')
9     res.on("data", data => console.log("some data" + data))
10 })
11
12 console.log(req.getHeaders());
13 req.end(); // must call it to actually send the request
```

Host to connect to and
send a request to
(default agent)

GET request

Listen to the response
event (when a
response is received
call this)

Actually write the
request...

Receiving a response and actually executing the response callback are two different things..

Example

```
1 const http = require("node:http");
2 MacBook-Pro:node-course-content HusseinNasser$ node 15-http/150-raw-request.js
3 [Object: null prototype] { host: 'example.com' }
4 {
5   'content-type': 'text/html',
6   etag: '"84238dfc8092e5d9c0dac8ef93371a07:1736799080.121134"',
7   'last-modified': 'Mon, 13 Jan 2025 20:11:20 GMT',
8   'cache-control': 'max-age=1881',
9   date: 'Fri, 17 Jan 2025 14:25:25 GMT',
10  'content-length': '1256',
11  connection: 'keep-alive'
12 }
13 200
14 some data<!doctype html>
15 <html>;,
16 |
17 |
18 console.log(req.getHeaders());
19 req.end();// must call it to actually send the request
20
```

Aborting a request

- Usually means removing it from the Agent queue
- Once the request is sent it is VERY difficult to abort.
- Requires surgical architecture across both frontend, proxies and backend
- Can be done with AbortSignal

Fetch

- You can also use fetch since its built-in to Node
- Works on browsers and Node
- There are other request libraries
- Will do a performance showdown at the end of course

HTTP Server

- Node can be used to listen and act as a server
- Listens on a port, address
- Accepts a connection
- Receives a request from HTTP clients (Node or others)
- Prepares and writes a response



10.0.0.3

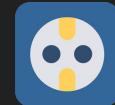
Connection
created,
Fd 18:
10.0.0.2:8080
10.0.0.3:53444



Connection
accepted!,
Fd 20:
10.0.0.2:8080
10.0.0.3:53444
Socket listener
(10.0.0.2: 8080)
Fd:19



10.0.0.2



Server on Request event

- The on request callback is fired when a request is received
- What does that mean?
- Node reads from the socket connections
- Parses the data until it gets the end of a request
- Packages the request as object and puts it on the callback

Request start → > GET / HTTP/1.1
> Host: example.com
> User-Agent: curl/8.11.1
> Accept: */*

Request end → >

On request fires * Request completely sent off

Server Options

- `connectionsCheckingInterval`: Sets the interval value in milliseconds to check for request and headers timeout in incomplete requests. **Default:** 30000.
- `headersTimeout`: Sets the timeout value in milliseconds for receiving the complete HTTP headers from the client. See [server.headersTimeout](#) for more information. **Default:** 60000.
- `highWaterMark <number>`: Optionally overrides all `sockets`' `readableHighWaterMark` and `writableHighWaterMark`. This affects `highWaterMark` property of both `IncomingMessage` and `ServerResponse`. **Default:** See [stream.getDefaultHighWaterMark\(\)](#).
- `insecureHTTPParser <boolean>`: If set to `true`, it will use a HTTP parser with leniency flags enabled. Using the insecure parser should be avoided. See [--insecure-http-parser](#) for more information. **Default:** `false`.
- `IncomingMessage <http.IncomingMessage>`: Specifies the `IncomingMessage` class to be used. Useful for extending the original `IncomingMessage`. **Default:** `IncomingMessage`.
- `joinDuplicateHeaders <boolean>`: If set to `true`, this option allows joining the field line values of multiple headers in a request with a comma (,) instead of discarding the duplicates. For more information, refer to [message.headers](#). **Default:** `false`.
- `keepAlive <boolean>`: If set to `true`, it enables keep-alive functionality on the socket immediately after a new incoming connection is received, similarly on what is done in `[socket.setKeepAlive([enable]), initialDelay][socket.setKeepAlive(enable, initialDelay)]`. **Default:** `false`.
- `keepAliveInitialDelay <number>`: If set to a positive number, it sets the initial delay before the first keepalive probe is sent on an idle socket. **Default:** 0.
- `keepAliveTimeout`: The number of milliseconds of inactivity a server needs to wait for additional incoming data, after it has finished writing the last response, before a socket will be destroyed. See [server.keepAliveTimeout](#) for more information. **Default:** 5000.
- `maxHeaderSize <number>`: Optionally overrides the value of `--max-http-header-size` for requests received by this server, i.e. the maximum length of request headers in bytes. **Default:** 16384 (16 KiB).
- `noDelay <boolean>`: If set to `true`, it disables the use of Nagle's algorithm immediately after a new incoming connection is received. **Default:** `true`.
- `requestTimeout`: Sets the timeout value in milliseconds for receiving the entire request from the client. See [server.requestTimeout](#) for more information. **Default:** 300000.
- `requireHostHeader <boolean>`: If set to `true`, it forces the server to respond with a 400 (Bad Request) status code to any HTTP/1.1 request message that lacks a Host header (as mandated by the specification). **Default:** `true`.
- `ServerResponse <http.ServerResponse>`: Specifies the `ServerResponse` class to be used. Useful for extending the original `ServerResponse`. **Default:** `ServerResponse`.
- `uniqueHeaders <Array>`: A list of response headers that should be sent only once. If the header's value is an array, the items will be joined using ; .
- `rejectNonStandardBodyWrites <boolean>`: If set to `true`, an error is thrown when writing to an HTTP response which does not have a body. **Default:** `false`.

Some options

- noDelay `<boolean>` - Disable Nagle's algorithm on new created connections (default true)
- requestTimeout `<ms>` : - How long to wait for a client request to fully received (300,000)
- headersTimeout `<ms>` : How long to wait for a client HTTP headers to fully received (60,000)

Example

```
1 const http = require("node:http");                                Create server object
2
3 const server = http.createServer();                                ← wait on a request event
4
5 server.on("request", (req, res) => {
6     res.statusCode= 200;
7     res.setHeader("content-type", "text/plain")
8     res.write("hello world" + req.url)
9     res.end();           ← Write the response
10 }
11
12 console.log("Before listening")                                     Poll phase listening (socket
13 server.listen(8080, ()=> console.log("Actualy listening.."));      bound , epoll_wait is called)
14 console.log("After listening")
```

^ Receiving a request and actually executing the request callback are two different things..

Example

```
1 const http = require("nod
2
3 const server = http.creat
4
5 server.on("request", (req
6   res.statusCode= 200;
7   res.setHeader("conten
8   res.write("hello worl
9   res.end();
10 })
11
12 console.log("Before listening")
13 server.listen(8080, ()=> console.log("Actualy listening.."));
14 console.log("After listening")
```

```
* Connection #0 to host 127.0.0.1 left intact
● hello world/MacBook-Pro:node-course-content HusseinNasser$ curl http://127.0.0.1:8080 -v
*   Trying 127.0.0.1:8080...
* Connected to 127.0.0.1 (127.0.0.1) port 8080
* using HTTP/1.x
> GET / HTTP/1.1
> Host: 127.0.0.1:8080
> User-Agent: curl/8.11.1
> Accept: */*
>
* Request completely sent off
< HTTP/1.1 200 OK
< content-type: text/plain
< Date: Fri, 17 Jan 2025 14:17:08 GMT
< Connection: keep-alive
< Keep-Alive: timeout=5
< Transfer-Encoding: chunked
<
* Connection #0 to host 127.0.0.1 left intact
○ hello world/MacBook-Pro:node-course-content HusseinNasser$ █
```

```
MacBook-Pro:node-course-content HusseinNasser$ node 15-http/153-raw-httpserv
Before listening
After listening
Actualy listening..
```

How about TLS?

- HTTPS is a different beast
- You will appreciate the cost even more
- Next lecture!

How about Express module?

- They are all wrappers on top of basic node http/https
- They do more work (watch out), body parser
- If -> `request.path === '/api'` -> run this callback

Summary & Demo

- Node HTTP
- Client and Server
- Code exercise →
 - raw client request
 - Raw server

Node HTTPS

Understanding HTTPS/TLS

HTTPS

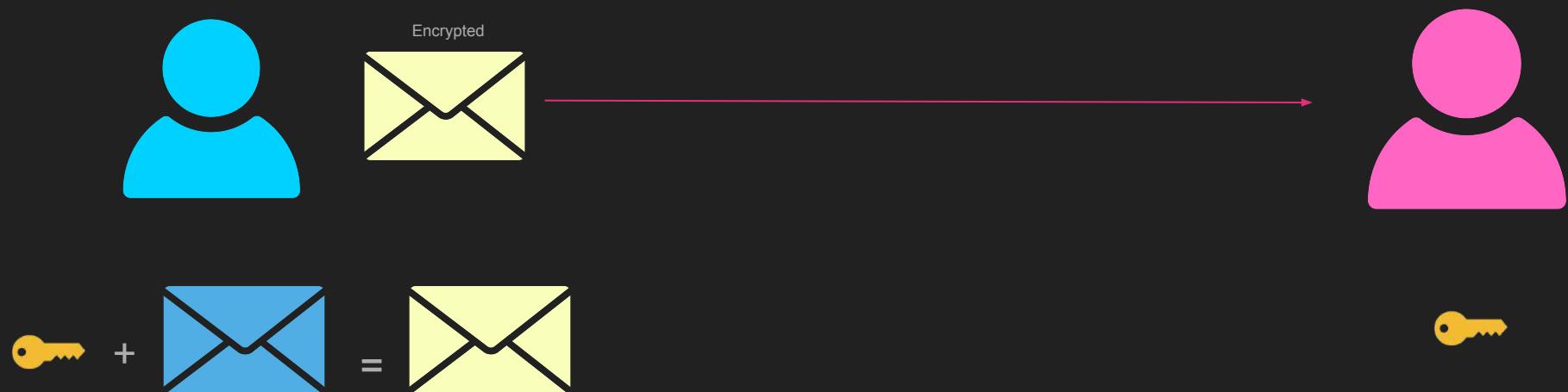
- Hypertext transfer protocol (Secure)
- Node implements HTTPS client and server
- On top of TLS (Transport Layer Security)

Encryption

- Encryptions are two types
- Symmetric -> You encrypt with key and decrypt with the same key
 - One key 
 - Fast but both client and server must have the same key
- Asymmetric -> You encrypt with a key and decrypt with another
 - Comes in pairs Two keys Private  and Public 
 - Slower but both client and server can have their own public keys
- We always want to encrypt with Symmetric encryptions
- Exchange the symmetric key with asymmetric encryption

Symmetric Encryption

- Assume both parties have the same key (The most difficult thing)
- User uses the key to encrypt message
- Send it



Symmetric Encryption

- Receiver gets the encrypted message
- Uses the same key to decrypt
- E.g. AES

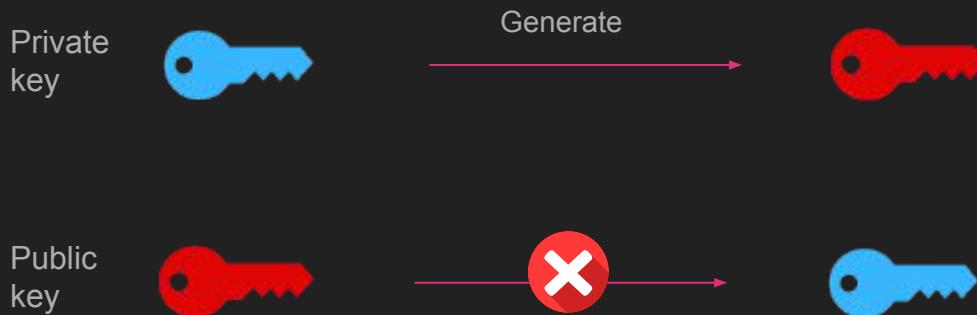


+



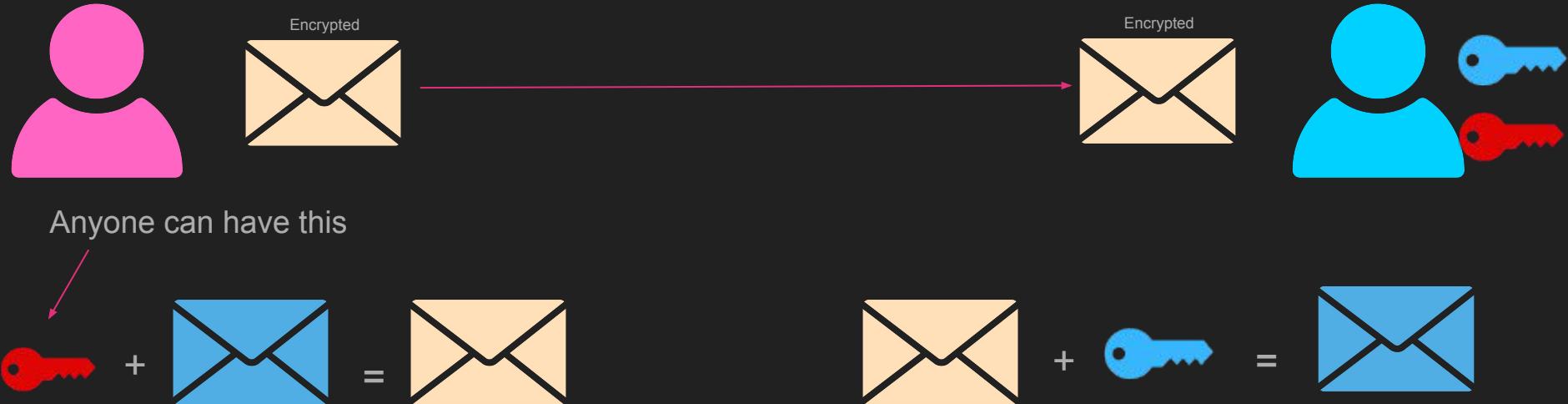
Public Key vs Private Key Rules

- Public key private keys are pairs (e.g. Red public, Blue Private)
- Given the Private Key you can generate the Public key
- Given the Public Key you cannot get the Private key



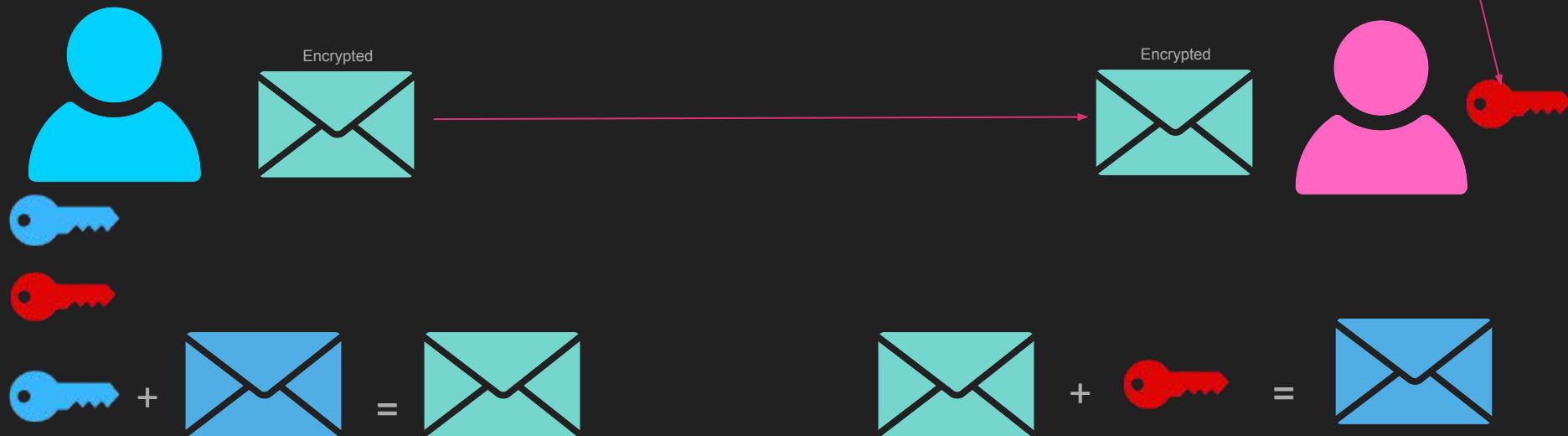
Encrypting with the Public Key

- You can encrypt a message with Public Key
- And only owner of Private key can decrypt it
- Proves Authenticity



Encrypting with Private Key

- You can encrypt a message with the Private key
 - and only the corresponding Public Key can decrypt it
 - Only owner of the private key could have signed this document
 - Protects confidentiality, nobody could have missed with it



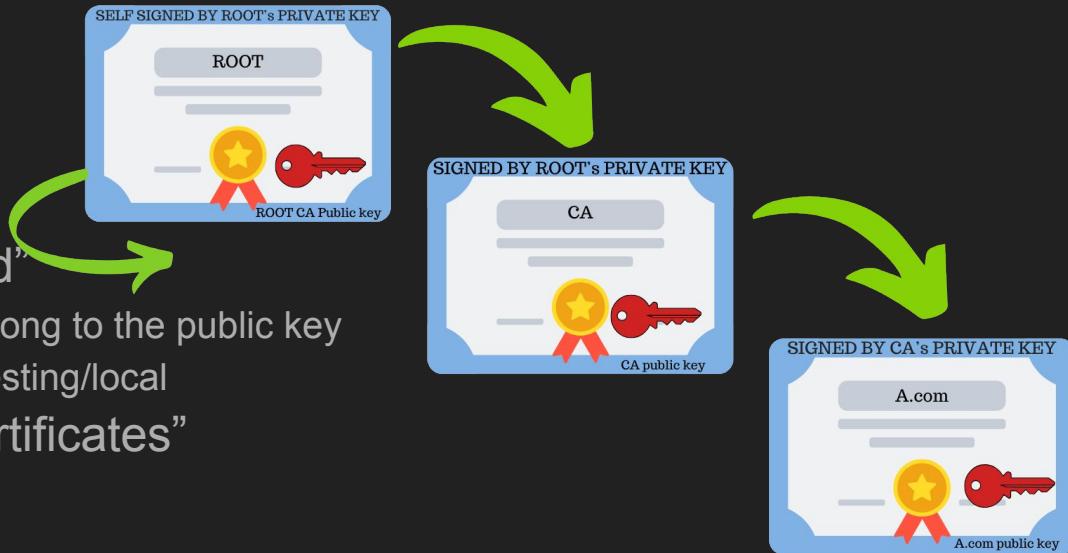
Certificates

- We need a way to proof authenticity
- Generate a pair of public/private key
- Put a public key in a certificate
- Put the website name in the certificate
- Sign the certificate with the private key
- Meet x509



Certificates

- Certificates can be “self signed”
 - Ie private key signing the cert belong to the public key
 - Usually untrusted and used for testing/local
- Certificates can sign “other certificates”
 - Creating a trust chain
 - Issuer name is who issued it
 - Lets encrypt
- Ultimately a ROOT cert is found
 - ROOT certs are always self signed
 - They are trusted by everyone
 - Installed with OS root (certificate store)



Certificate Verification

X509
Name: A.com
Issuer: CA
PublicKey: APB
Sign: CAPRK*

X509
Name: CA
Issuer: RootTrust
PublicKey: CAPB
Sign: RootTrustPRK*

X509
Name: RootTrust
Issuer: RootTrust
PublicKey: RootTrustPB
Sign: RootTrustPRK*

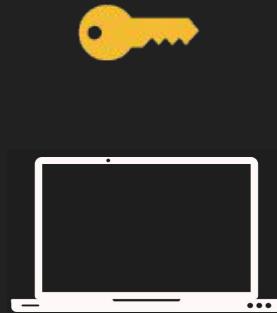
Client receives the full chain, wants to verify A.com cert signature which has been signed by CA public key issuer, so it gets the CA and gets the CAPUB to verify, but also it needs to trust the CA cert so it verifies that by getting the RootTrust public key and verifies it, but the RootTrust is self signed so it looks up its local cert store. If it is there it is trusted. Else rejected.

TLS

- Transport Layer security
- Encrypt using the same key on both client and server
- For that we need to exchange the key
- We use public key encryption to exchange key
- We share certificate for authentication

TLS 1.2 (RSA)

open



RSA Public key



RSA Private key



Client hello

Server hello (cert)

Change cipher, fin

Change cipher, fin



GET /



Headers+
index.html

<html>...

....

close

Problems with that approach

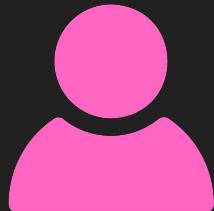
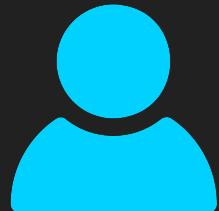
- Encrypting the symmetric key with public key is simple
- But its not perfectly forward
- Attacker can record all encrypted communications
- If the server private key is leaked (heart bleed)
- They can go back and decrypt everything
- We need ephemeral keys! Meet Diffie Hellman

Diffie Hellman

- Let us not share the symmetric key at all
- Let us only share parameters enough to generate it
- Each party generate the same key
- Party one generates **X** number (private)
 - Also generates **g** and **n** (public, random and prime)
- Party two generates **Y** number (private)

Public **g,n**

Private X



Private Y

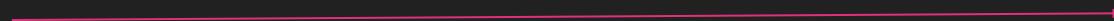
Diffie Hellman

- Party 1 sends g and n to Party 2
- Anyone can sniff those values fine.
- Now both has g and n

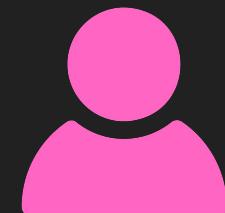
Public g,n
Private X



Public g,n

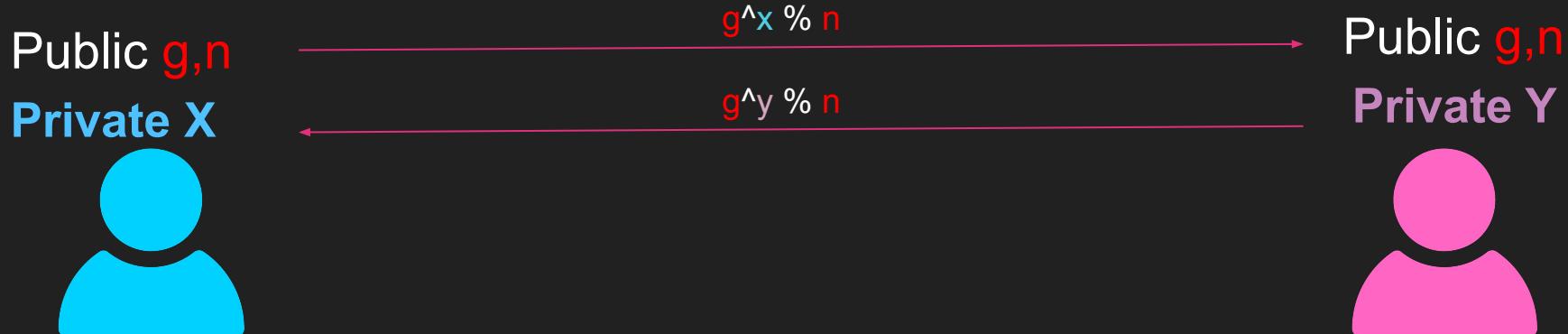


Private Y



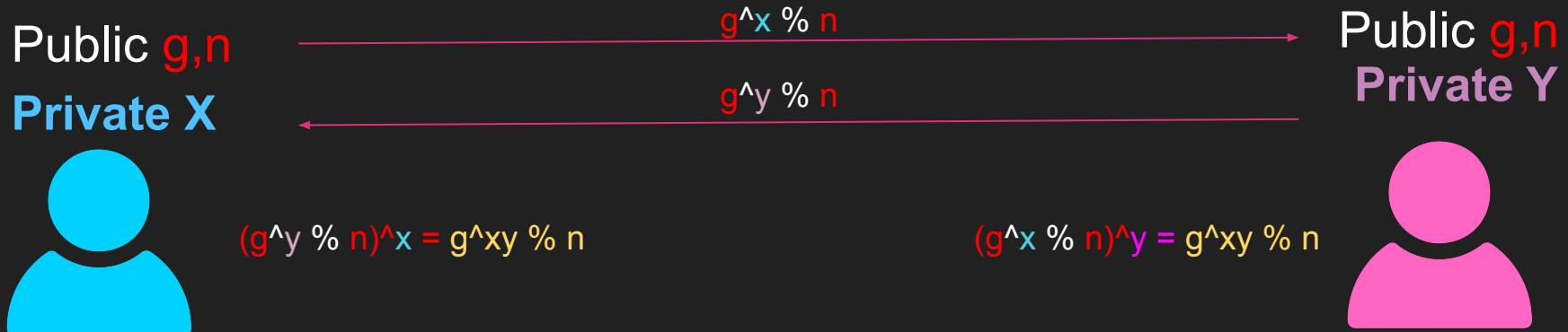
Diffie Hellman

- Party 1 takes g to the power of $X \% n$
 - $g^X \% n$ is now a public value
 - Cannot be broken to get X !
- Party 2 does the same with Y
 - $g^Y \% n$ is now a public value
 - Cannot be broken to get Y
- Both parties share the new values



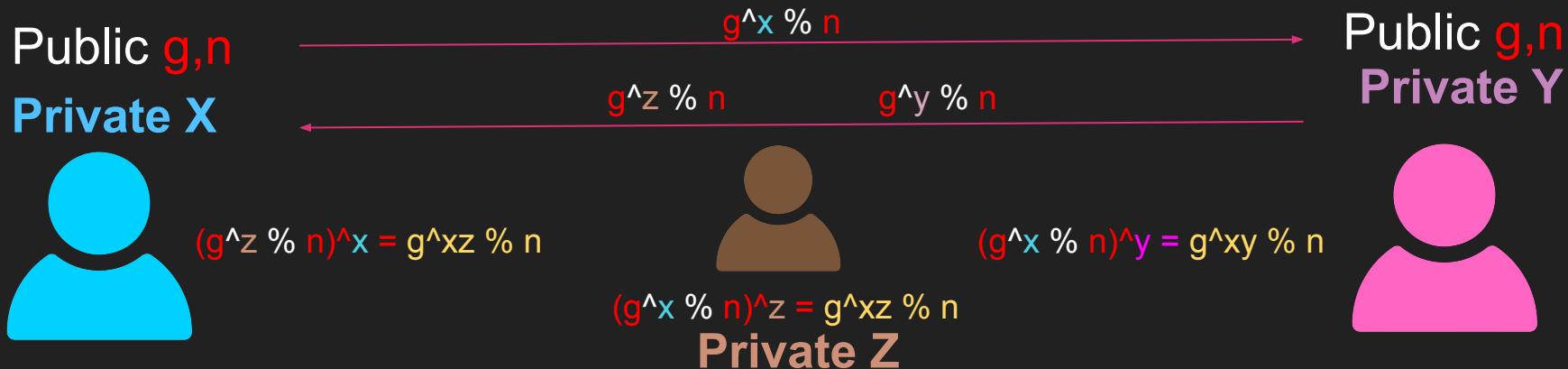
Diffie Hellman

- Party 1 takes Y's value and raise it to X
 - $(g^y \% n)^x = g^{xy} \% n$
- Party 2 takes X's value and raise it to Y
 - $(g^x \% n)^y = g^{xy} \% n$
- Both now has the same value $g^{xy} \% n$
- This is the used as a seed for the key $g^{xy} \% n$



More problems! MITM

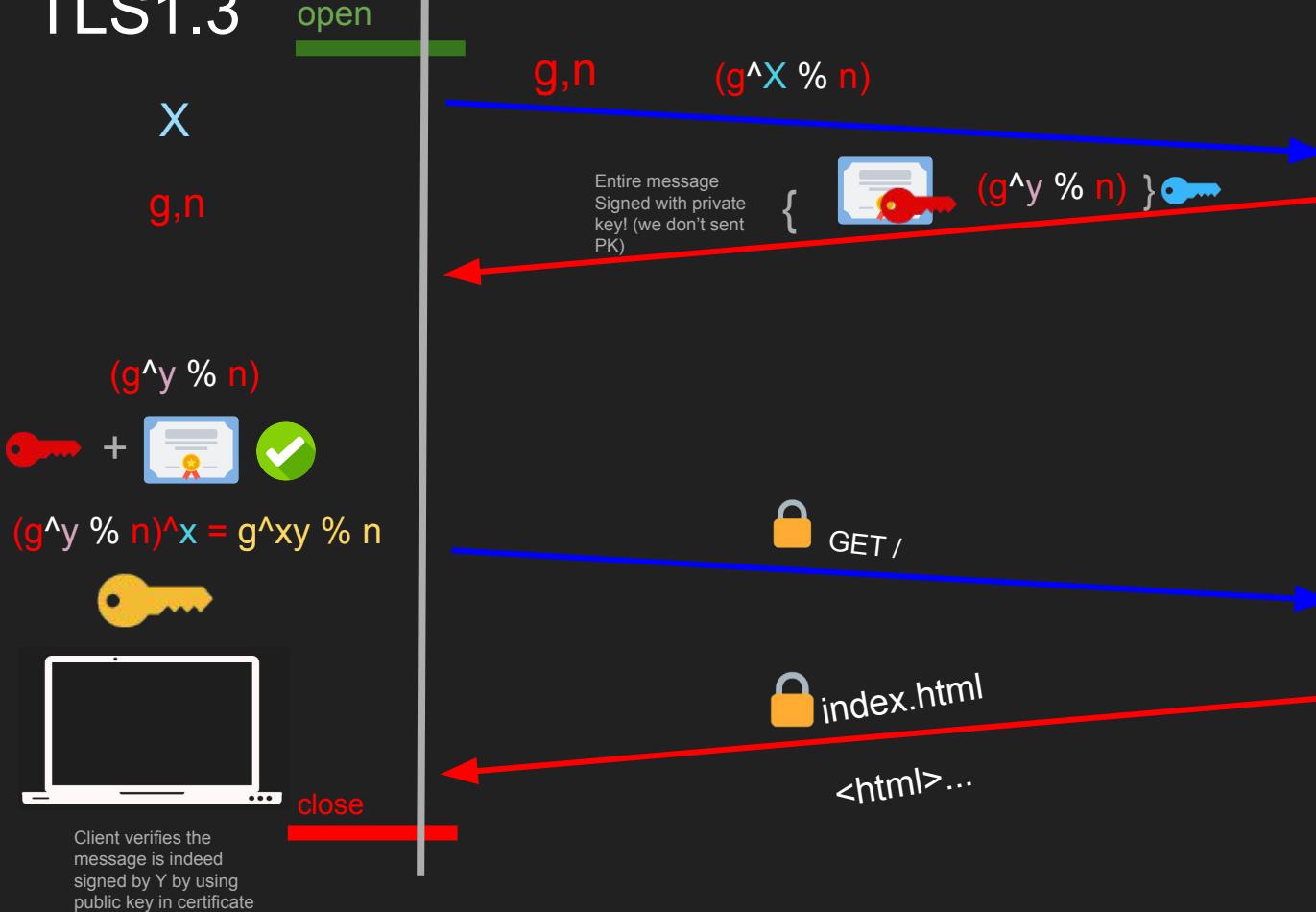
- This solves perfect secrecy
- But what if someone intercepts and put their own DH keys
- MITM replace Y's parameter with their own
- X doesn't know that happened (it's just numbers)



Solved with signing

- We bring back public key encryption
- But only to sign the entire DH message
- With certificates

TLS1.3



Public key



Private key

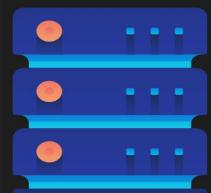


Y

$(g^x \% n)$



$(g^x \% n)^y = g^{xy} \% n$



There is more to TLS

- More stuff is sent in the TLS handshake
- TLS extensions
 - ALPN
 - SNI
- Cipher algorithms
- Key generation algorithms
- Key size
- Digital signature algorithms
- Client side certificates

Node HTTPS

- Node HTTPS Server requires a certificate and private key
- The rest is the same
- More work though!
- Requests gets hit with additional cost
- Responses gets hit with additional cost

Generate Private key and Certificate with OpenSSL

- OpenSSL is a library for cryptographic operations
- Generate private key
 - `openssl genrsa -out private-key.pem 2048`
- Generate Certificate x509 (which contains public key)
 - `openssl req -new -x509 -key private-key.pem -out certificate.pem -days 365`
 - Answer questions to fill the the 509 fields
 - Most important is common name , subject alternative which is the website

Example Server

16-https > JS 161-raw-https-server.js > ...

```
• 1 const https = require('node:https');
  2 const fs = require('fs');
  3
  4 // Define certificate and private key
  5 const options = {
  6   key: fs.readFileSync('private-key.pem'),
  7   cert: fs.readFileSync('certificate.pem')
  8 };
  9
 10 // Create an HTTPS server
 11 const server = https.createServer(options, (req, res) => {
 12   res.writeHead(200, { 'Content-Type': 'text/plain' });
 13   res.end('Hello , on HTTPS!\n');
 14 });
 15
 16 // Start the server
 17 server.listen(8443, () => {
 18   console.log(`HTTPS server is running on https://localhost:8443`);
 19 });
 20
 21 //consume like curl https://localhost:8443 --insecure
 22
```

We pass in our private keys
(secret don't leak it!) 

We pass the certificate 

Request is decrypted before we
get here

Response is encrypted right after
this

Example Client

16-https > JS 160-raw-https.js > ...

```
1 const http = require("node:https");           New module that knows how to do https
2 const req = http.request("https://example.com", { "method": "GET"});
3
4 req.on("response", res => {
5     console.log(res.headers)                  https scheme
6     console.log(res.statusCode);
7     //set the encoding
8     res.setEncoding('utf-8')                  Response is decrypted before we
9     res.on("data", data => console.log("some data" + data))      get here
10    }
11    Request is encrypted right after
12    this
13    req.end(); // must call it to actually send the request
14    // (end the stream //we will discuss this more on the stream lecture)
15    let x = req.getHeaders();
16    console.log(x)
```

Summary & Demo

- Encryptions
- TLS
- Certificates
- Node HTTPS
- Code exercise →
 - raw https request
 - Create a certificate/private/public key
 - Raw https server

Node DNS

When does Node do DNS

Why DNS

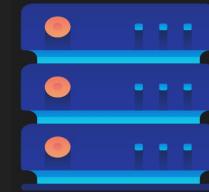
- People can't remember IPs
- A domain is a text points to an IP or a collection of IPs
- Additional layer of abstraction is good
- IP can change while the domain remain
- We can serve the closest IP to a client
- Load balancing

```
MacBook-Pro:~ HusseinNasser$ nslookup example.com
Server:      192.168.4.1
Address:     192.168.4.1#53

Non-authoritative answer:
Name:   example.com
Address: 23.192.228.84
Name:   example.com
Address: 23.192.228.80
Name:   example.com
Address: 96.7.128.198
```

DNS

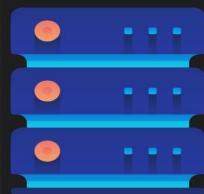
- A new addressing system means we need a mapping. Meet DNS
- If you have an IP and you need the MAC, we use ARP
- If you have the name and you need the IP, we use DNS
- Built on top of UDP
- Port 53
- Many records (A, AAAA, MX, TXT, CNAME)



Google.com
(142.251.40.46)

How DNS works

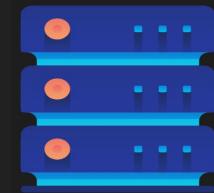
- DNS resolver - frontend and cache
- ROOT Server - Hosts IPs of TLDs
- Top level domain server - Hosts IPs of the ANS
- Authoritative Name server - Hosts the IP of the target server



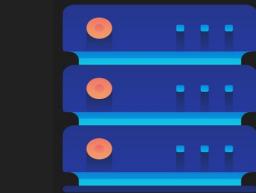
server



Resolver



ANS

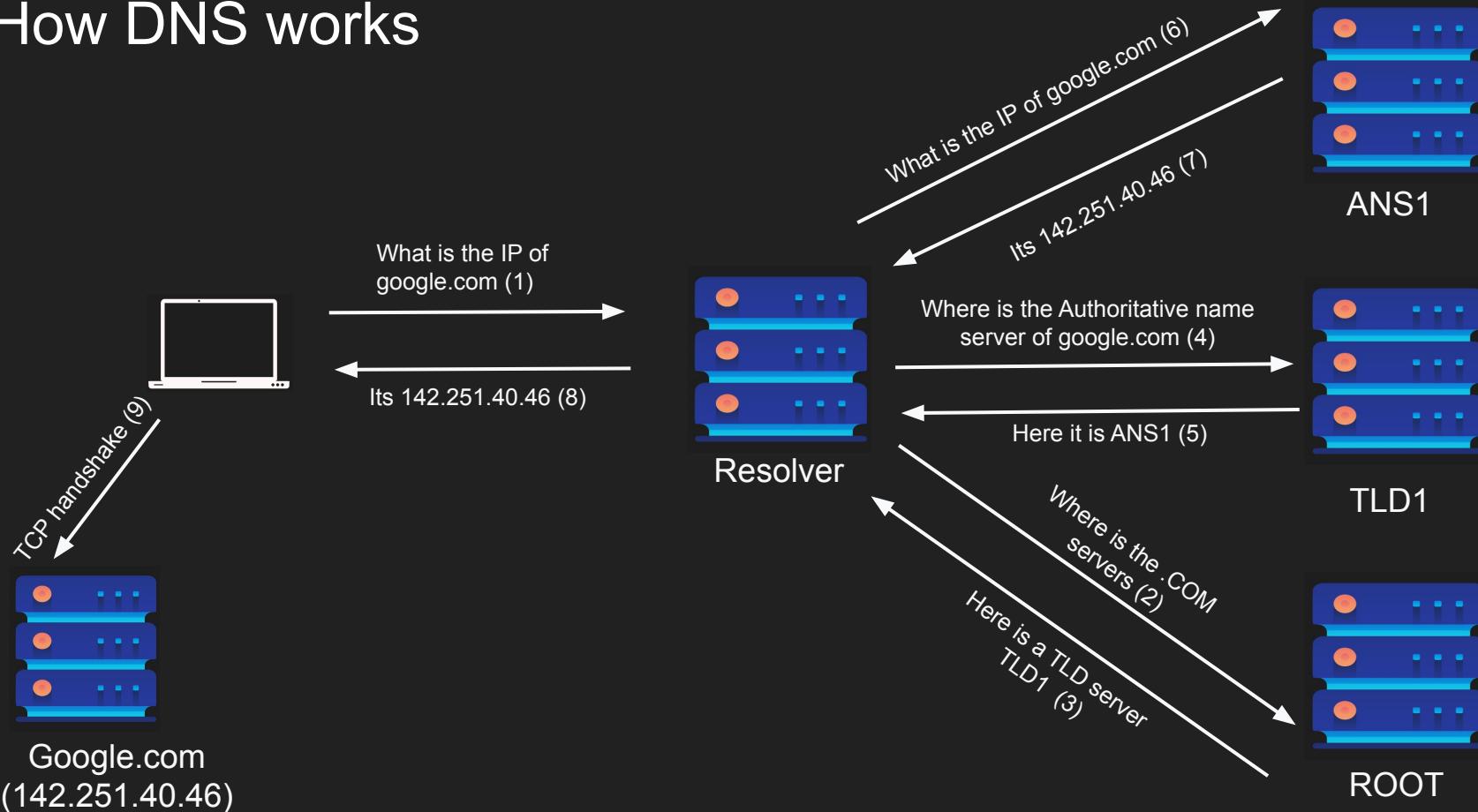


TLD
(top level domain)

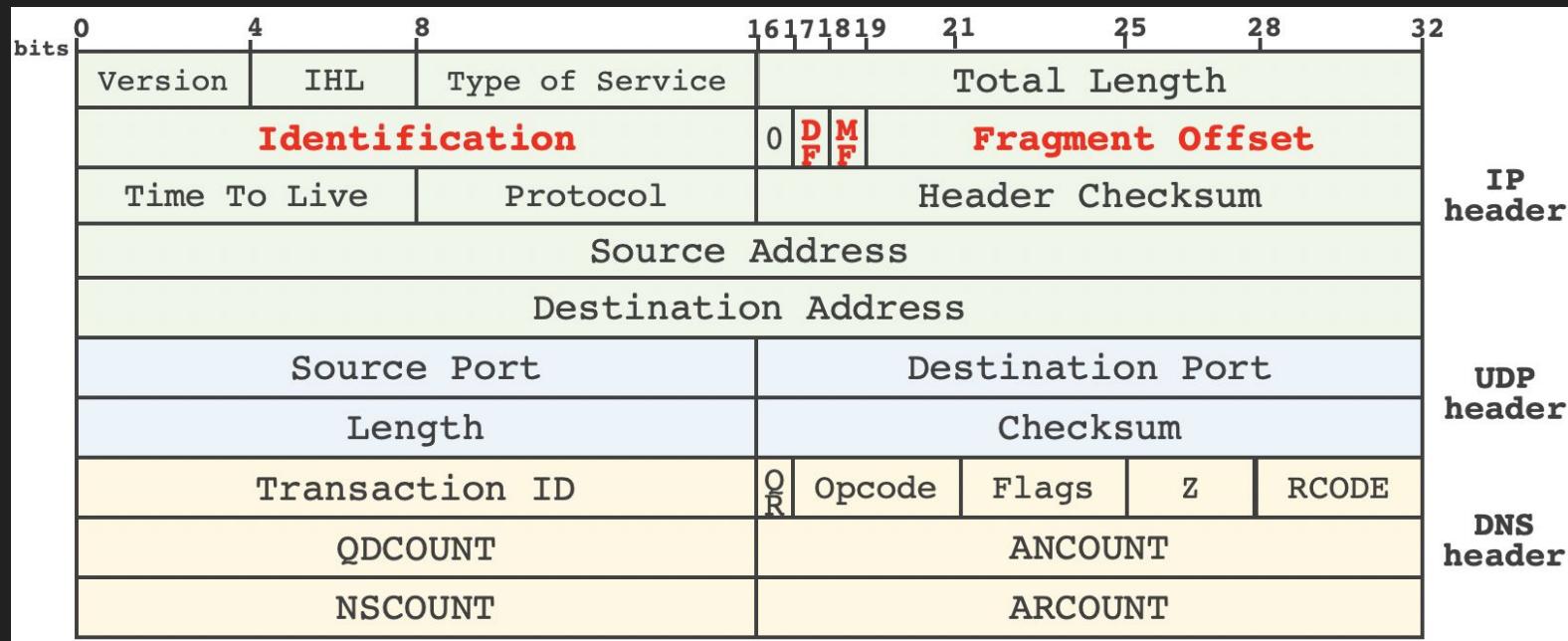


ROOT

How DNS works



DNS Packet



Source: <https://www.usenix.org/system/files/sec20-zheng.pdf>

RFC: <https://datatracker.ietf.org/doc/html/rfc1035>

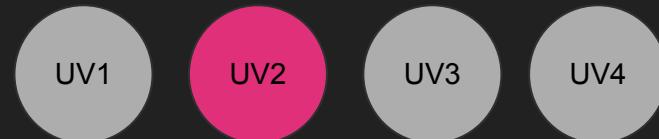
Notes about DNS

- Many layers for redundancy
- Works on local (hosts file)
 - Linux /etc/hosts
 - Windows C:\Windows\System32\drivers\etc
- DNS is not encrypted by default.
- Many attacks against DNS (DNS hijacking/DNS poisoning)
- DoT / DoH attempts to address this

 MT

Node DNS

- Node does DNS using the thread pool (libuv)
- Why though? Isn't DNS network -> UDP?
- Because we need to look locally for the cache
 - Hosts file and then network DNS mixed workload
 - Thus blocking
 - `getaddrinfo()` is the function
- Too much DNS queries may starve your pool



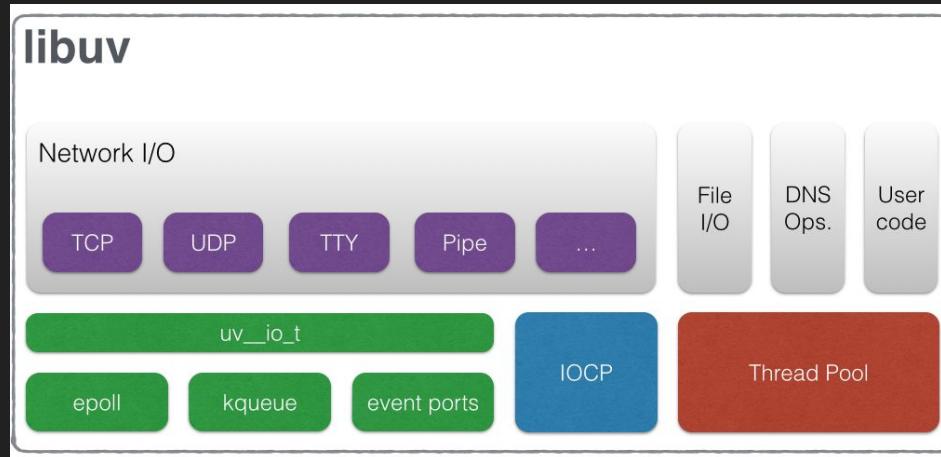
Needs DNS before
we do anything

```
2 const req = http.request("https://example.com", { "method": "GET"});
```

```
3
```

Node DNS

- Resolve vs Lookup
- Lookup checks hosts file first / cache and then network
 - Linux /etc/hosts
 - Windows C:\Windows\System32\drivers\etc
- Resolve goes to the network (asynchronous)



Example Implicit DNS lookup

17-DNS > **js** 170-SimpleDNS.js > ...

```
1 const http = require("node:http");
2 //this does a DNS
3 const req = http.request("http://example.com", { "method": "GET"});
4 req.end();
5 let x = req.getHeaders();
6 console.log(x)
7
8
```

This does DNS lookup (does file io (hosts file) and then optionally DNS query

Example DNS lookup

17-DNS > `js 171-DNSlookup.js > ...`

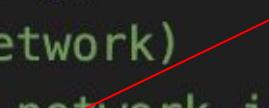
```
1 const dns = require('dns');
2 console.log("Before DNS lookup")
3 //this may do a DNS, it looks up hosts file and if nothing is there it may do DNS
4 //it is still async but goes onto the UVthread
5 dns.lookup('example.org', (err, address, family) => {
6   console.log('address: %j family: IPv%s', address, family);
7
8   console.log("After DNS lookup")
9 }
```

Explicit DNS lookup (does
file io (hosts file) and then
optionally DNS query

Example DNS Resolve

17-DNS > **JS** 172-DNSResolve.js > ...

```
1  const dns = require('dns');
2  //this always go DNS (network)
3  //this is just a normal network io (epoll)
4  dns.resolve4('example.org', (err, address) => {
5    console.log('addresses: %j', address);});
```



Server bind gotaches

- Listening on a host port require DNS
- Which requires the event loop (uv threads)
- `server.listen(8080); //this binds in initial phase`
Need DNS to resolve this
- `server.listen(8080, "localhost"); //this is scheduled until poll phase`
- Watch out for long running initial phases

Summary & Demo

- DNS
- Lookup vs Resolve
- Node DNS
- Code exercise →
 - Request resolve
 - DNS lookup
 - DNS resolve
 - Raw https server with host resolution

Node TCP

Beyond Node TCP

TCP

- Stands for Transmission Control Protocol
- Stream based Layer 4 protocol
- Ability to address processes in a host using ports
- “Controls” the transmission unlike UDP which is a firehose
- Connection based (Requires handshake)
- All protocols are built on TCP (or UDP)

TCP Use cases

- Reliable communication
- Remote shell
- Database connections
- Web communications
- Any bidirectional communication



TCP Connection

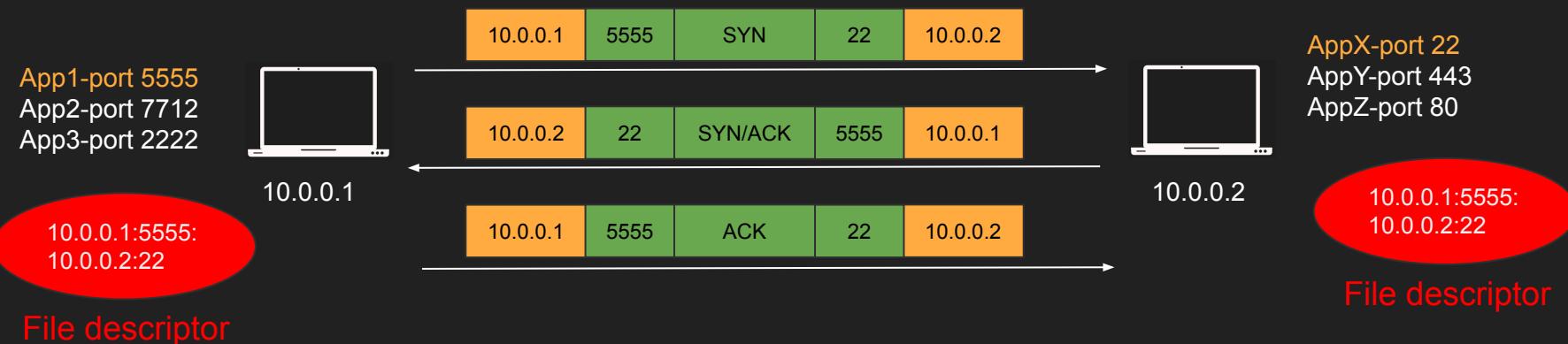
- Must create a connection to send data
- Connection is identified by 4 properties
 - SourceIP-SourcePort
 - DestinationIP-DestinationPort
- Represented by a socket on both client and server
- Requires a 3-way TCP handshake

TCP Composition

- Stream of segments
- Segments are sequenced and ordered
- Segments are acknowledged
- Lost segments are retransmitted
- Segments != Messages

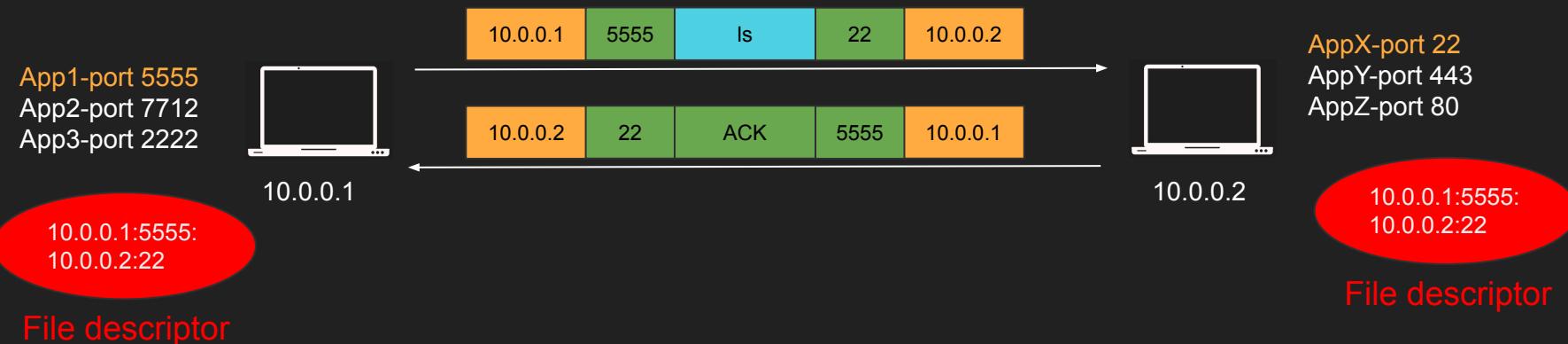
Connection Establishment

- App1 on 10.0.0.1 want to send data to AppX on 10.0.0.2
- App1 sends SYN to AppX to synchronous sequence numbers
- AppX sends SYN/ACK to synchronous its sequence number
- App1 ACKs AppX SYN.
- Three way handshake



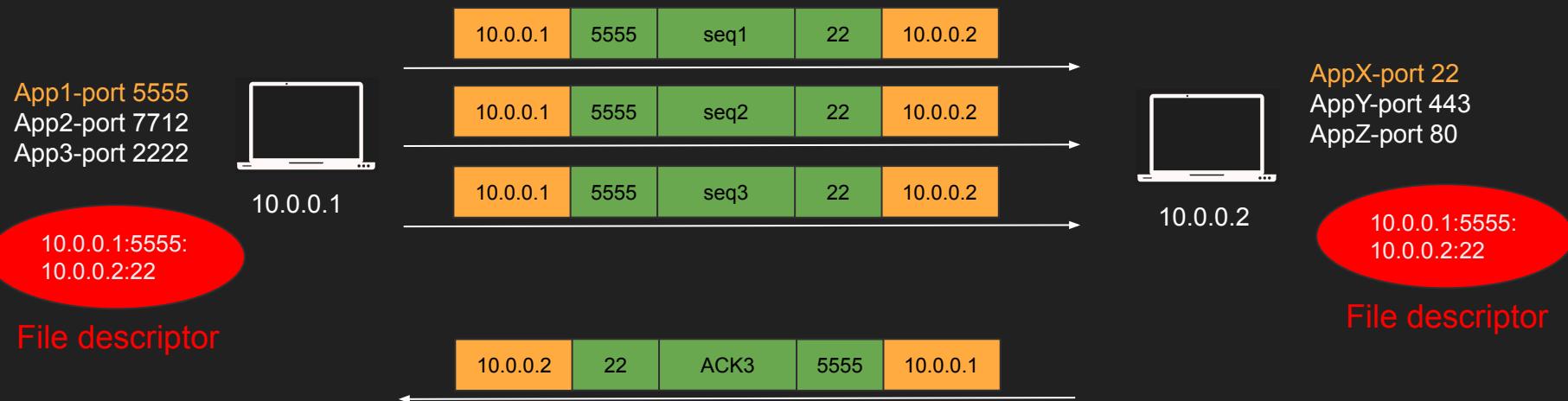
Sending data

- App1 sends data to AppX
- App1 encapsulate the data in a segment and send it
- AppX acknowledges the segment
- Hint: Can App1 send new segment before ack of old segment arrives?



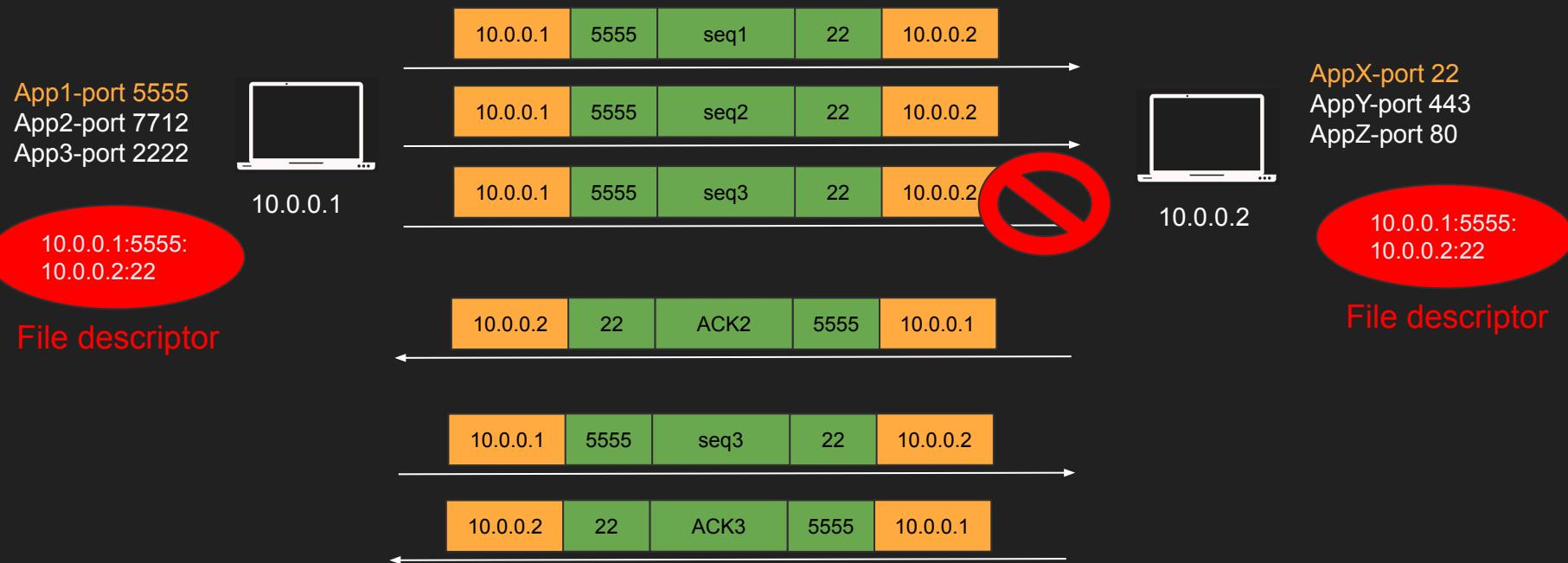
Acknowledgment

- App1 sends segment 1,2 and 3 to AppX
- AppX acknowledge all of them with a single ACK 3



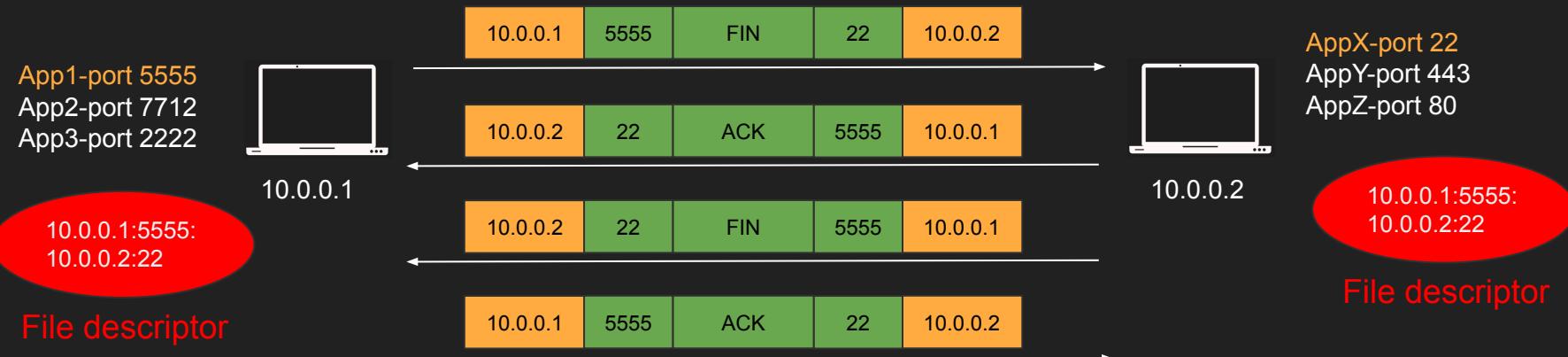
Lost data

- App1 sends segment 1,2 and 3 to AppX
- Seg 3 is lost, AppX acknowledge 3
- App1 resend Seq 3



Closing Connection

- App1 wants to close the connection
- App1 sends FIN, AppX ACK
- AppX sends FIN, App1 ACK
- Four way handshake



TCP Pros

- Guarantee delivery
- Flow Control and Congestion Control
- Ordered Packets no corruption or app level work

TCP Cons

- Large header overhead compared to UDP
- Considered high latency for certain workloads (Slow start/ congestion/ acks)
- Does too much at a low level (hence QUIC)
 - Single connection to send multiple streams of data (HTTP requests)
 - Stream 1 has nothing to do with Stream 2
 - Both Stream 1 and Stream 2 packets must arrive

Node TCP

- Node TCP is exposed via net module
- HTTP/DNS and other protocols use the net module
- Server creates a listening socket
- Client creates a connection socket

TCP Client Example

```
18-tcp > JS 180-tcp-client.js > ...
```

```
1 const net = require("net")
2 //create a tcp connection to example.com (this does DNS)
3 const connection = net.createConnection({"host": "example.com", "port": 80})
4 connection.on("connect", () => {
5   console.log("connected! 3 way handshake done we got a full fledged connection")
6   console.log(connection.localAddress + " " + connection.localPort);
7 })
```

socket

Create a
connection

Socket gets filled if 3
way handshake
completes here.

TCP Server Example

```
18-tcp > js 181-tcp-server.js > ...
```

```
1  const net = require("net")
2  console.log("Before server")
3  server = net.createServer(() => {})
4  console.log("Before Bind")
5  //this requires DNS which requires the event loop
6  server.listen(8080, "127.0.0.1");
7  server.on("listening", ()=> console.log("server created."))
8  server.on("connection", (connection) => {
9      console.log(`new connection! ${connection.remotePort}`)
10     //wire the data event
11     connection.on("data", (data) => console.log(`got new data` + data.toString() ))
12 }
13 //if you try to connect before the initial phase completes you will fail
14 console.log(["After initial phase"])
```

Listen on 8080 127.0.0.1
Can't connect with ::1 or any other ip
Got a connection, wire the event to read from it..

Summary & Demo

- TCP
- Node TCP
- Code exercise →
 - TCP client
 - TCP server

Node UDP

User Datagram Protocol

UDP

- Stands for User Datagram Protocol
- Message Based Layer 4 protocol
- Simple protocol to send and receive messages
- Prior communication not required (double edge sword)

UDP Use cases

- Video streaming
- VPN
- DNS
- WebRTC



Source and Destination Port

- App1 on 10.0.0.1 sends data to AppX on 10.0.0.2
- Destination Port = 53
- AppX responds back to App1
- We need Source Port so we know how to send back data
- Source Port = 5555



UDP Pros

- Simple protocol
- Header size is small so datagrams are small
- Uses less bandwidth
- Consumes less memory (no state stored in the server/client)
- Low latency - no handshake , order, retransmission or guaranteed delivery

UDP Cons

- No acknowledgement No guarantee delivery
- Connection-less - anyone can send data without prior knowledge
- No flow control
- No congestion control
- No ordered packets
- Security - can be easily spoofed

Node UDP

- Implemented in the dgram module
- Bind to set the source ip and port
- Send takes the destination ip and port and message



sIP:sPort



dIP:dPort



sIP:sPort



dIP:dPort

Node UDP Server

- Create a UDP socket
- Bind it to a port (3333)! Now you are listening
- Destination becomes undefined (server can receive data from anywhere)

10.0.0.2



10.0.0.1



0:0:0:0:3333  *.*

Node UDP Client

- Create a UDP socket
- Send a message 10.0.0.1: 3333
- This automatically binds to a random port
- Destination becomes undefined (ie client can receive data from anywhere)

10.0.0.2



10.0.0.1



0:0:0:0:5555



.

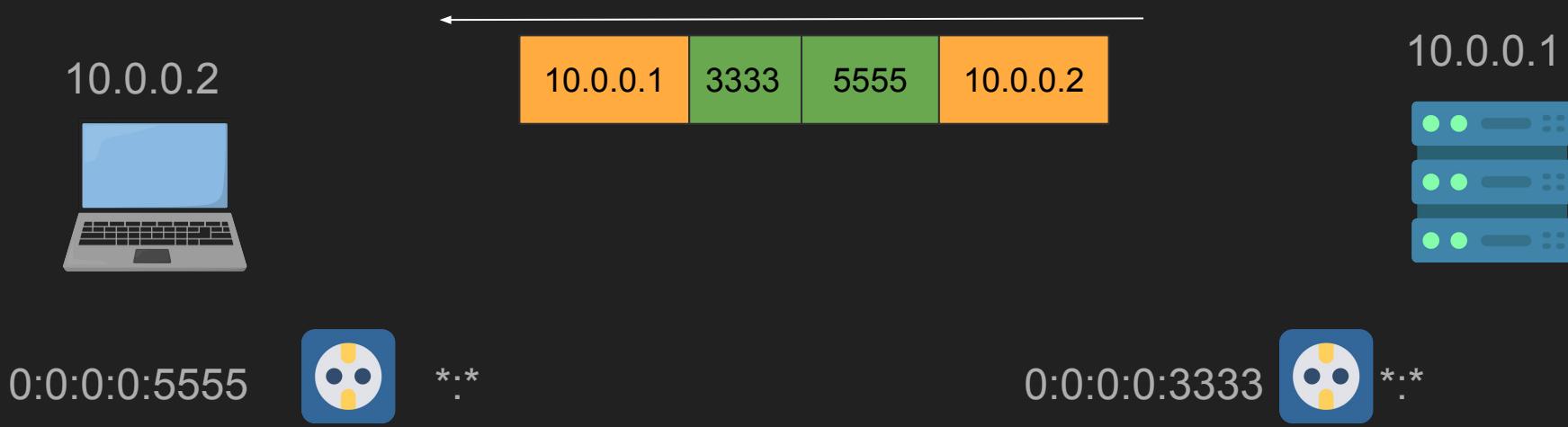
0:0:0:0:3333



.

Server can respond

- Server can send a message 10.0.0.2:5555



Using “connect” with udp

- Simulates a connection
- Use connect to fix the destination
- This way only that destination can respond back to you
- Client connect to 10.0.0.1 3333
- No one else can respond to this client except 10.0.0.1 :3333

10.0.0.2



0:0:0:0:5555



10.0.0.1:3333

10.0.0.1



0:0:0:0:3333



:

UDP Server Example

```
19-udp > js 190-UDP-Server.js > ...
1  const dgram = require('dgram')
2  const server = dgram.createSocket('udp4');           Create a UDP socket
3
4  server.on('message', (msg, rinfo) => {             Got a message
5    console.log(`server got: ${msg} from ${rinfo.address}:${rinfo.port}`);
6    //we can send something back to the destination
7    //we write a udp datagram and set the destination ip/ port
8    //notice it is called send and not write, because it is a whole message
9    server.send("hello client!", rinfo.port, rinfo.address);   Send a message back
10
11});                                                 to the client
12
13server.on('listening', () => {                   The bind was
14  const address = server.address();               successful!
15  console.log(`server listening ${address.address}:${address.port}`);
16});
17
18● server.bind(3333);                           Bind socket to a
19                                         port/address (all)
20  // Prints: server listening 0.0.0.0:3333
```

UDP Client Example

```
19-udp > JS 191-UDP-Client.js > ...
1  const dgram = require('dgram')
2  const client = dgram.createSocket('udp4');           ← Create a UDP socket
3
4  client.on('message', (msg, rinfo) => {             ← Got a message
5    console.log(`client got: ${msg} from ${rinfo.address}:${rinfo.port}`);
6  });
7  //this automatically binds to a random port, (can't send anything otherwise)
8  client.send("hello server", 3333, "127.0.0.1")      ← Send a message to this
9
10 //optionally bind to a source port of your choosing
11 //if you don't do that, kernel will pick one for you
12 //client.bind(55555)
13
14 client.on("error", (err) => console.log(err) )       ← Bind was successful! So we can receive
15 //a bind was successful!
16 client.on('listening', () => {                      stuff back
17   const address = client.address();
18   //this will keep reading
19   console.log(`client listening ${address.address}:${address.port}`);
20 });
21
```

Summary & Demo

- UDP
- Node UDP
- Code exercise →
 - UDP client
 - UDP server

Node Streams

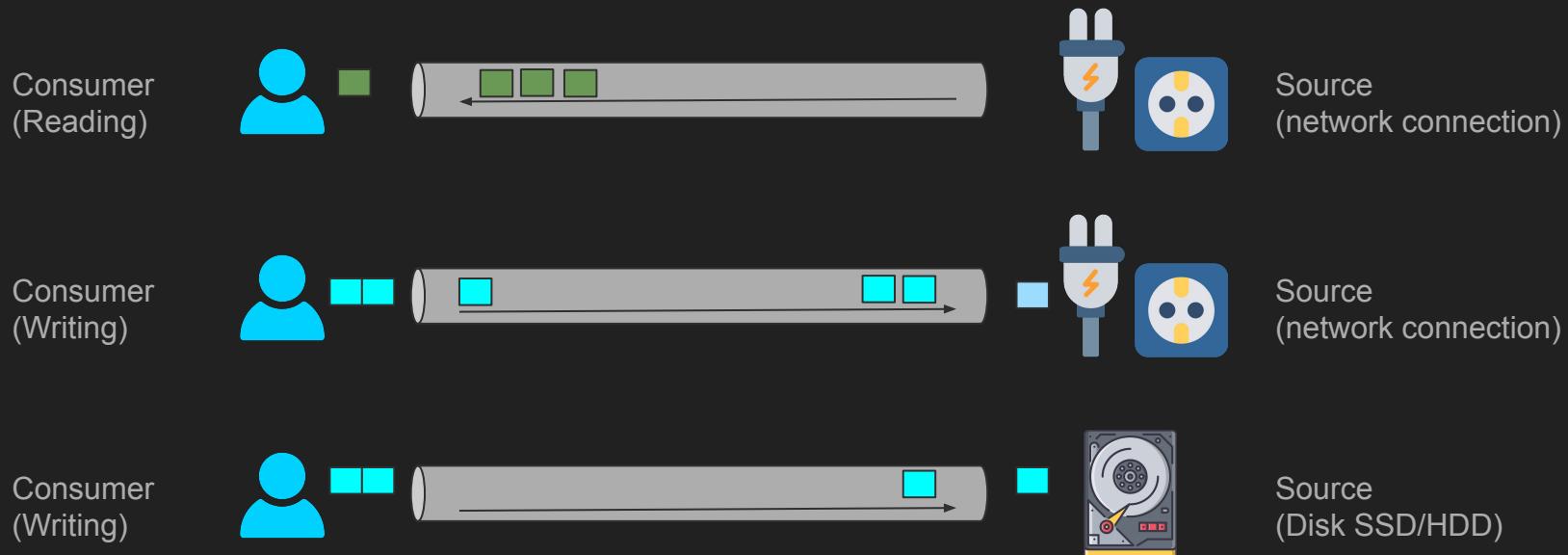
The stream abstraction

What is a stream?

- Source of data to be consumed
- Read or write to a source (often large data)
- Constantly flowing
 - Until source is depleted or closed
- Reading/writing from/to a socket connection
- Reading/writing from/to a file

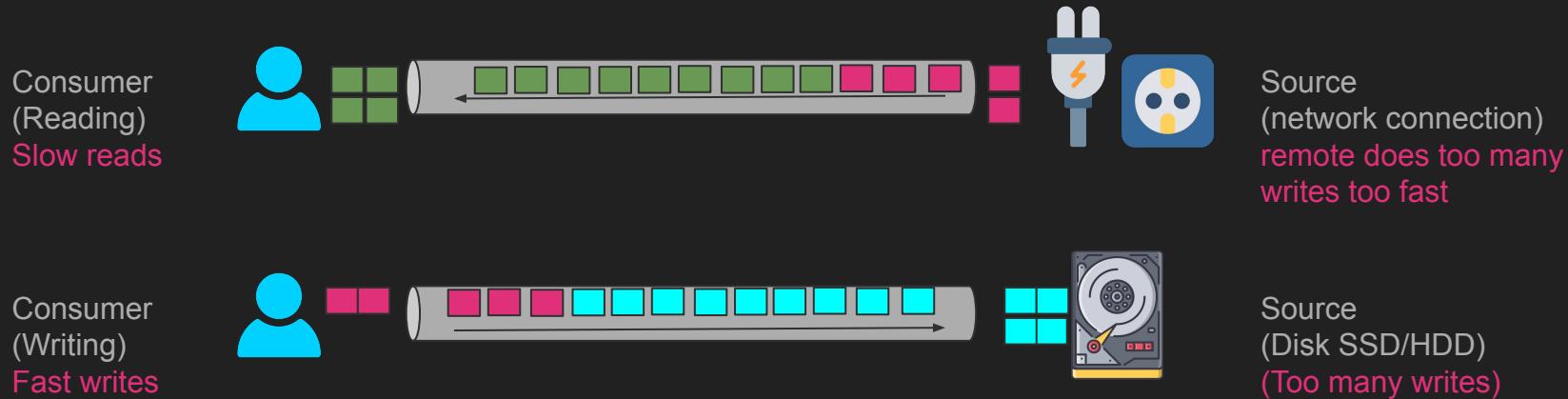
Stream semantics

- Consumer vs Source
- Consumer read from source
- Consumer writes to a source



Why Streams

- Consumer can't process reads fast enough
- Source can't handle writes fast enough
- Need a backoff or control mechanism
- TCP has flow control, OS Kernel has page cache
- Node Invented an application level abstraction called stream
- Kafka solved it with long polling

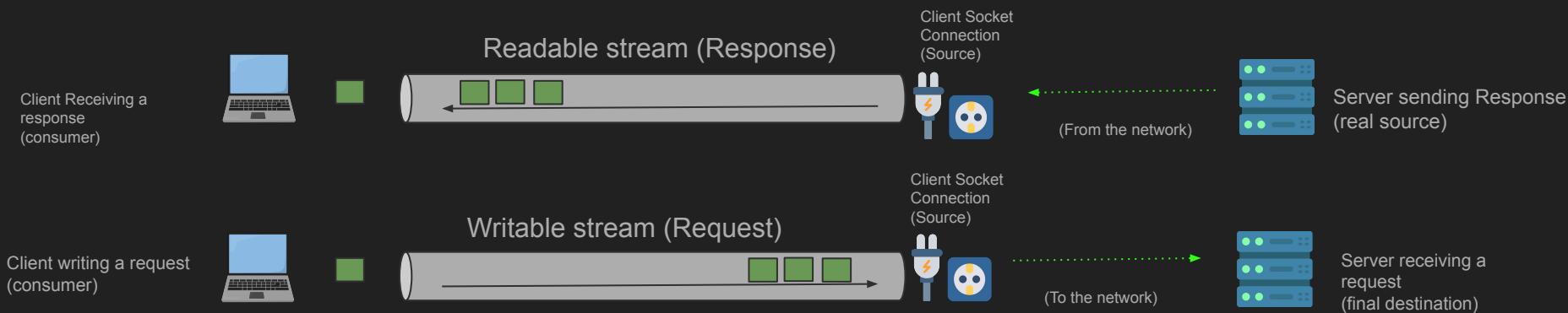


Node Streams

- Readable Stream
 - Reads from a underlying source
 - Connection / Disk / request
- Writable Stream
 - Writes to an underlying source
 - Disk / connection / response

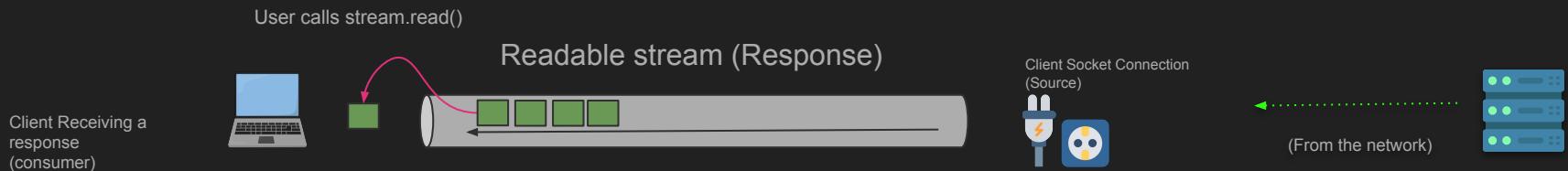
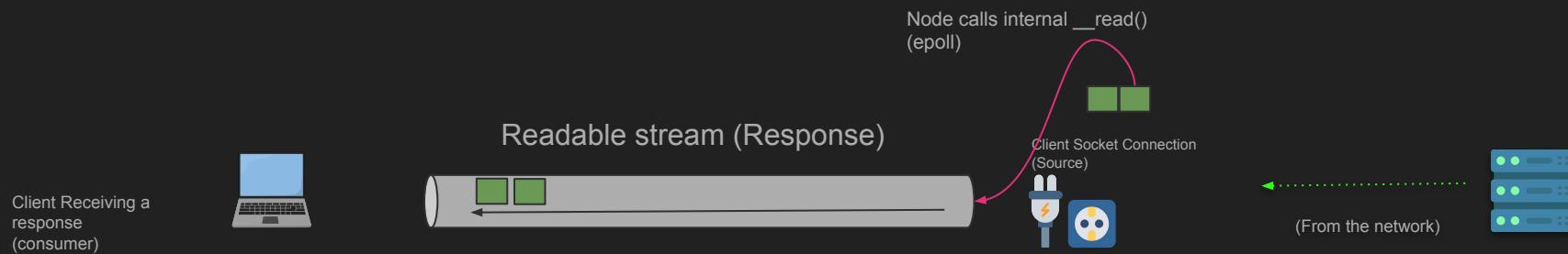
Node Streams

- Client receiving response
 - Readable stream at the client (res object)
- At the server side the Server sending the response
 - Writable stream at the server (res object)
- Same for the request
- Depends where you look



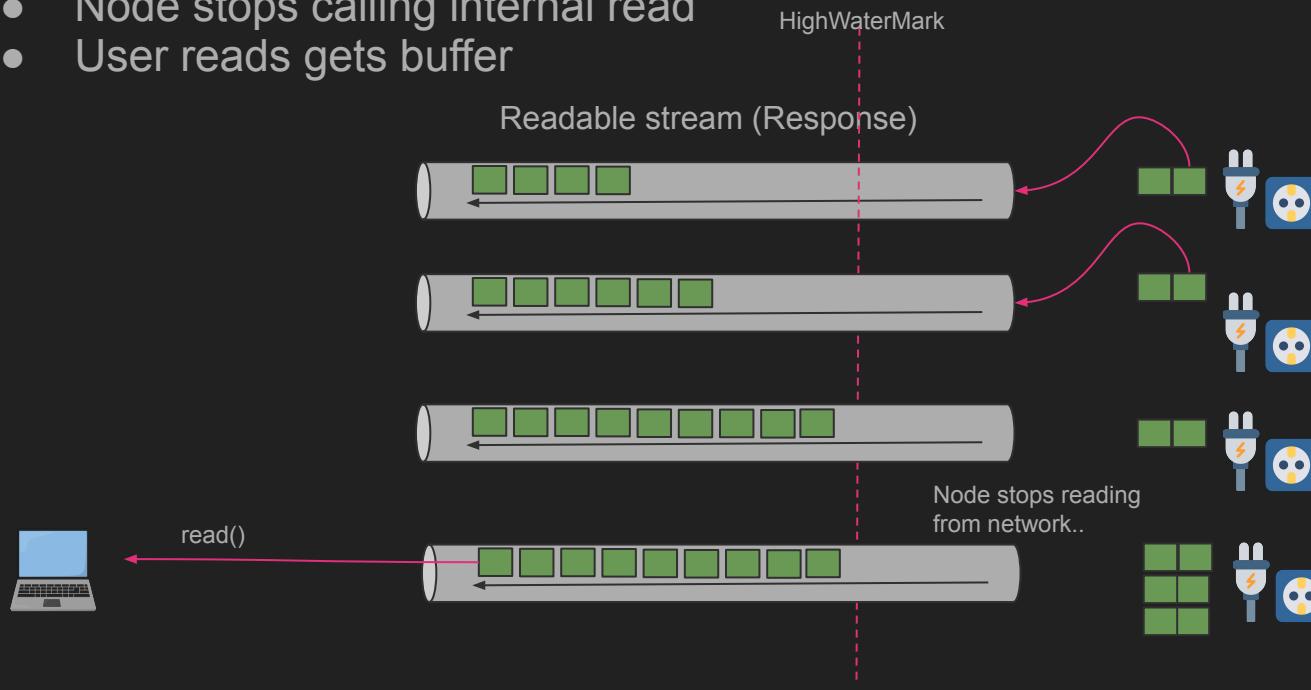
Node internal buffer

- Each stream has an in memory buffer
- Node reads from source and puts it in the buffer
- Consumer reads the stream from the buffer
- Internal read `read()` vs source read `__read()`
- Same with writes



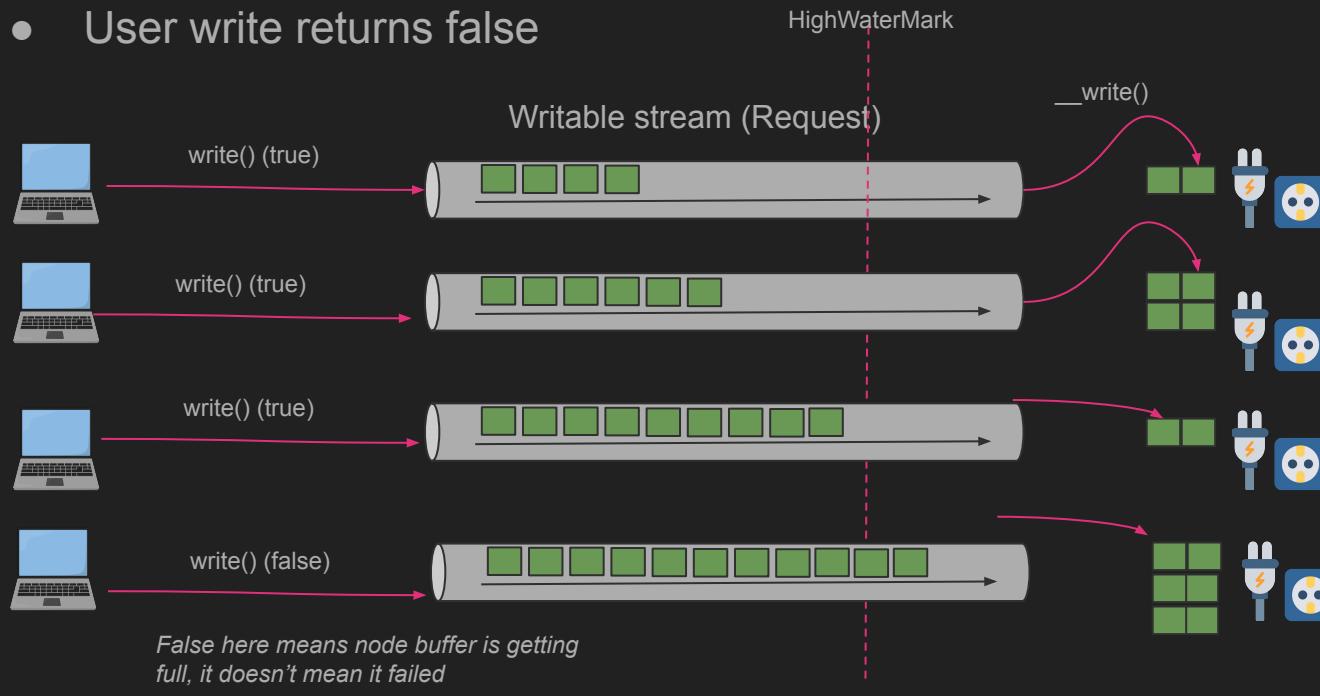
HighWatermark

- Consumer isn't calling read fast enough
- Buffer reaches a specified limit in bytes
- Node stops calling internal read
- User reads gets buffer



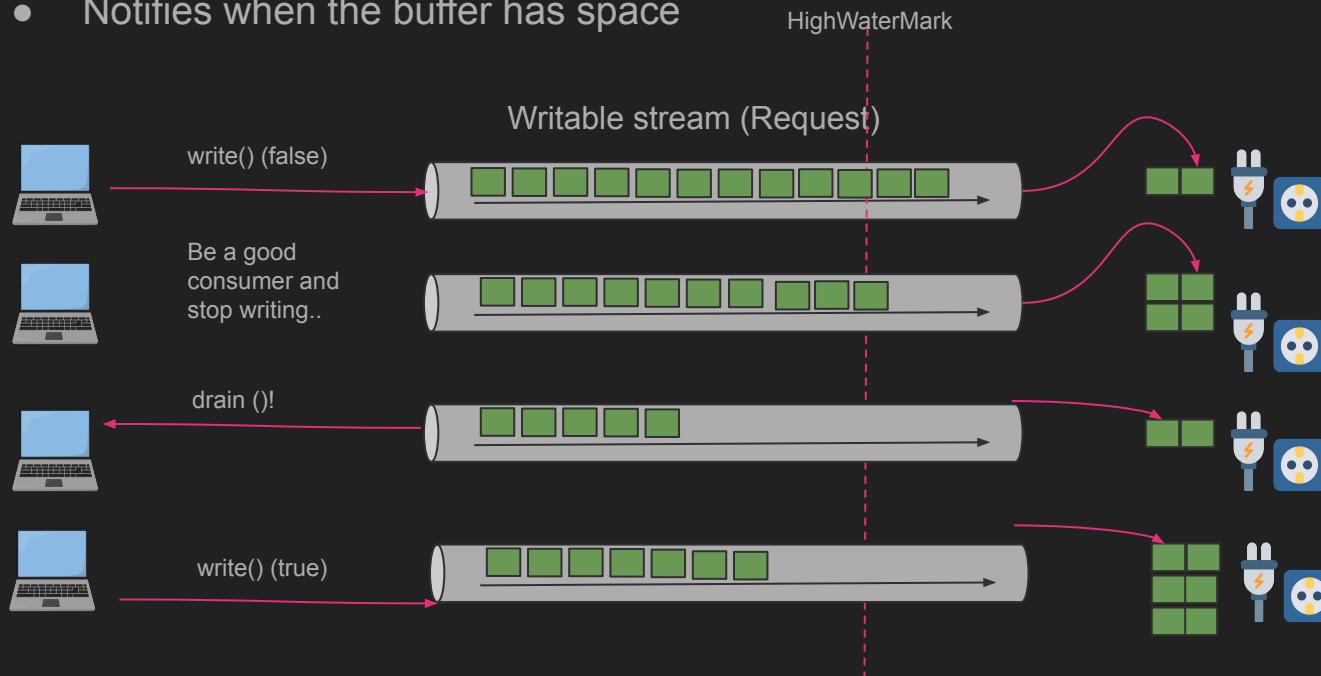
HighWatermark Writable

- Consume is calling write fast
- Buffer reaches a specified limit in bytes
- User write returns false



Draining Writable streams

- The drain event
- Consumer listens to it
- Notifies when the buffer has space



Ending a writable stream

- Calling `end()` on a writable stream
- Indicates that there are no more data
- Critical for request/response streams
- Tells Node to “Wrap up the request”
 - Add necessary endings (`\n\n` http)
- Can’t call `write` after `end`

Chaining streams

- You can chain streams together
- Readable into a writable
- Using pipe or pipeline
- Readablestream.Pipe(Writablestream)
- Reads from a file, pipe it to writable stream
- Pipeline handles errors

Duplex streams - Transformer

- They are both writable and readable stream
- Can act like both
- Example ZIP and Encryption

Example Writing request

17-streams > JS 170-raw-request-stream copy.js > ..

```
• 1 const http = require("node:http");
  2 //request object is a writable stream
  3 const req = http.request("http://example.com", { "method": "GET"});
  4
  5 req.on("response", res => {
  6   //the data events implicitly calls read for us
  7   //response object is a readable stream
  8   res.on("data", data => console.log("some data" + data))
  9 }
10
11 req.end(); //no more data, we are done. wrap up
12 //this doesn't mean "send" , some data might have already been sent
```

Readable

Write data event (this
calls read())

Writable

Example Server

```
17-streams > JS 172-raw-httpserv.js > ...
• 1 const http = require("node:http");
2 //override the highwater mark
3 //default is 64K
4 const server = http.createServer({ "highWaterMark": 300_000 });
5 server.on("request", (req, res) => {
6     //default is 65536
7     console.log("highWaterMark" + req.readableHighWaterMark)
8     //req is a readable stream on the server
9     //body is not read by default (can be large)
10    req.on("data", data => {
11        //only executed when there is a body in the request
12        //like POST, headers are implicitly read and prepared for us
13        //by the http lib
14        console.log("reading request" + data.toString())
15    })
16    //res is a writable stream on the server
17    res.statusCode= 200;
18    res.setHeader("content-type", "text/plain")
19    res.write("hello world" + req.url)
20    res.end();
21 })
22 //send this request curl http://localhost:8080 -X POST -d "key1=value1"
23 server.listen(8080, ()=> console.log("Actualy listening.."));
```

Override highWaterMark

Request (Readable)

Response (Writable)

Summary & Demo

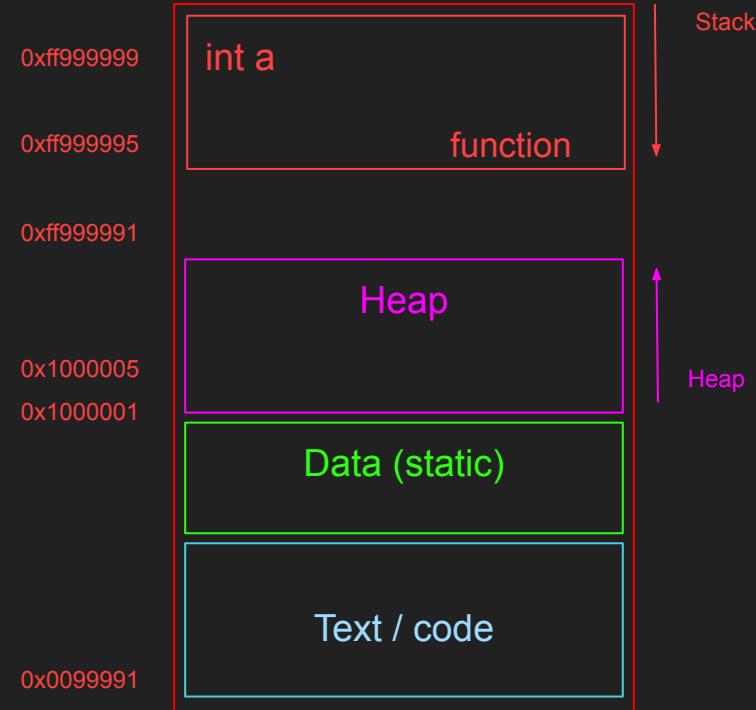
- Node Streams
- Code exercise →
 - Request stream
 - Response stream
 - Pipe
 - Chaining Compress->encrypt

Process vs Thread

Understanding the difference

Process

- An instance of a program
- Has dedicated code, stack, heap, data section
- Has context in the CPU (pc, lr, etc..)
- Process Control Block

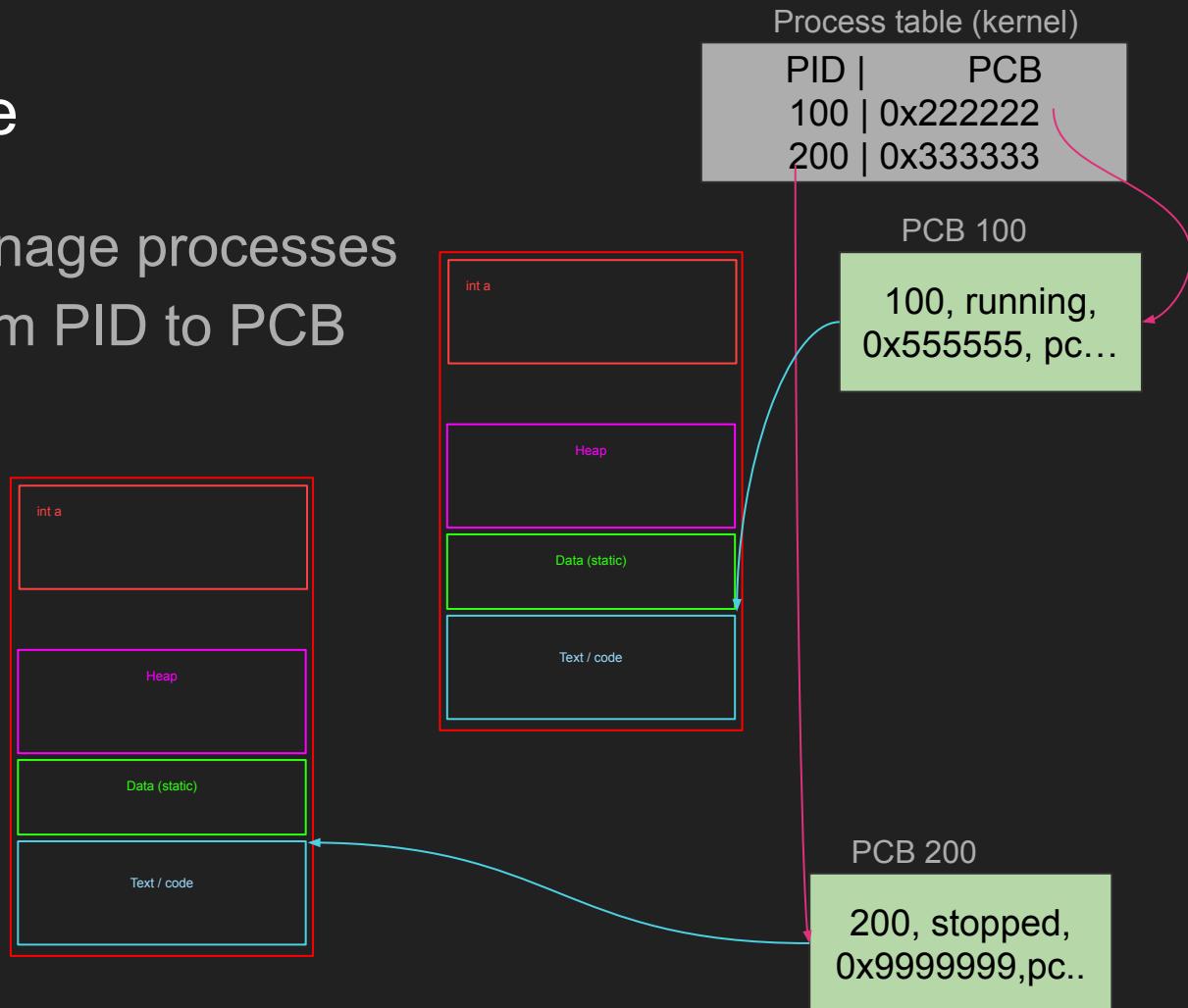


Process Control Block (PCB)

- Kernel needs metadata about the process
- PCB Contains
 - PID, Process State, Program counter, registers
 - Process Control info (running/stopped, priority)
 - Page Table (Virtual memory to physical mapping)
 - Accounting (CPU/memory usage)
 - Memory management info (Pointer to code/stack etc.)
 - IO info (File descriptors)
 - IPC info, semaphores, mutexes, shared memory, messages.

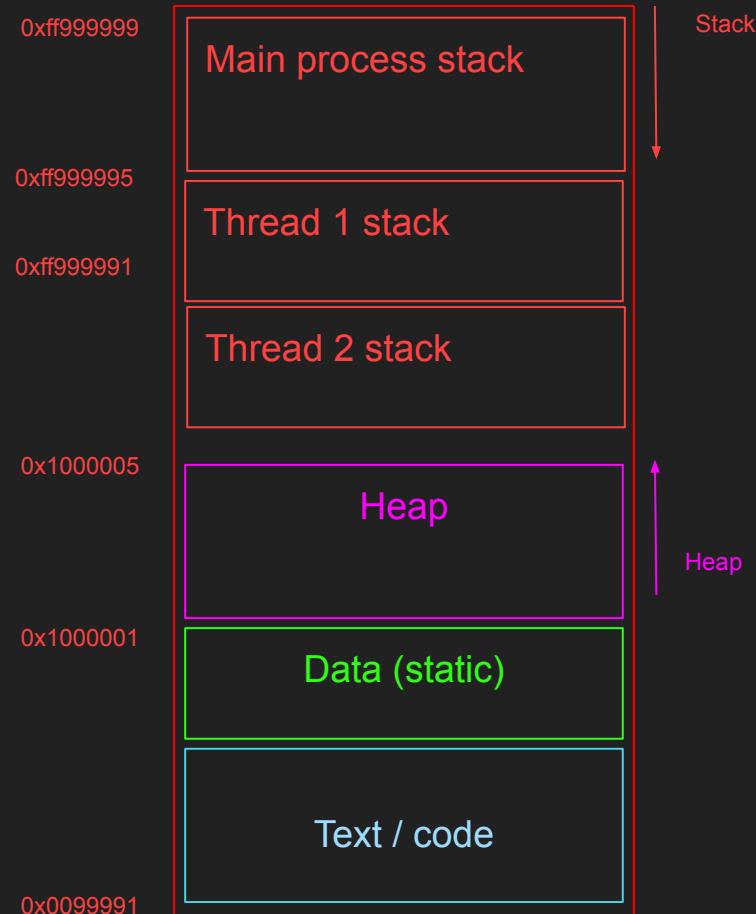
Kernel Process Table

- Kernel needs to manage processes
- A mapping table from PID to PCB
- Process Table
- Quick lookup
- In kernel space



Thread

- A thread is a light weight process
- Shared code/heap,data and PCB
- Stack is different and pc
- Thread Stack lives in same VM

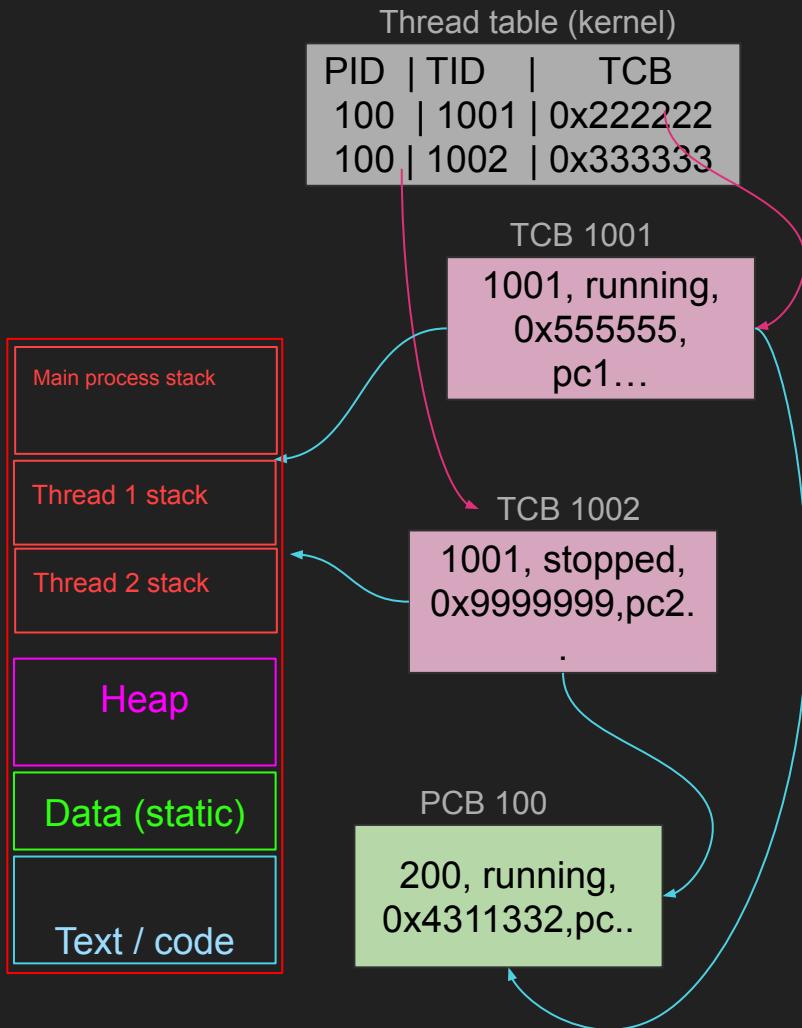


Thread Control Block (TCB)

- Kernel needs metadata about the thread
- TCB Contains
 - TID, Thread State, Program counter, registers
 - Process Control info (running/stopped, priority)
 - Accounting (CPU/memory usage)
 - Memory management info (Pointer to stack etc.)
 - Pointer to parent PCB

Kernel Thread Table

- Kernel needs to manage threads
- A mapping table from TID to TCB
- Thread Table
- Quick lookup
- In kernel space

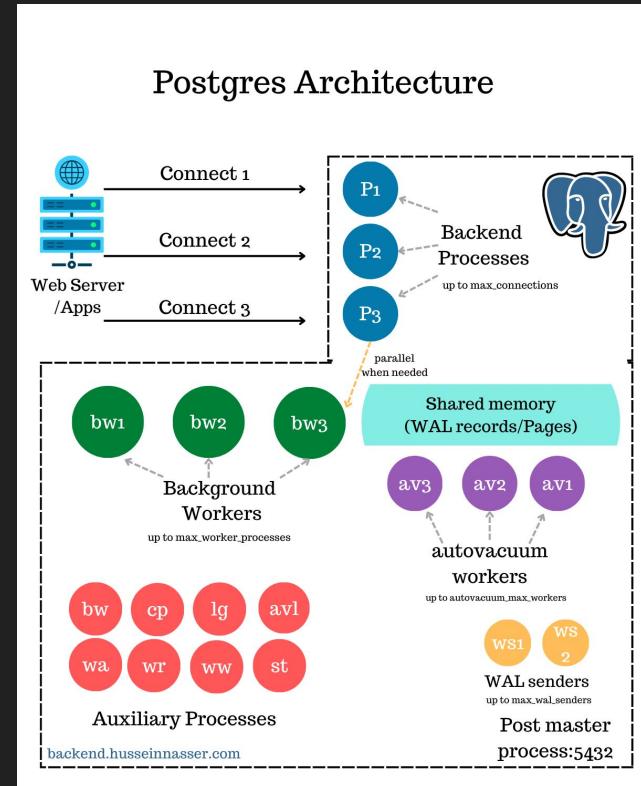


Shared Memory

- Multiple processes/threads can share memory
- mmap
- Virtual memory different, physical memory same
- Shared Buffers in databases

Postgres Processes

- Postgres uses processes
- Should Postgres move to threads?
- Long running discussions

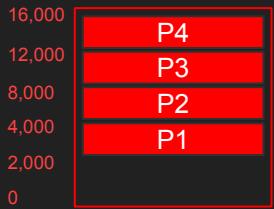


Fork

- Fork creates a new process
- Child must have new virtual memory
- But OS uses CoW so pages can be shared unless a write happens
- Redis Asynchronous durability

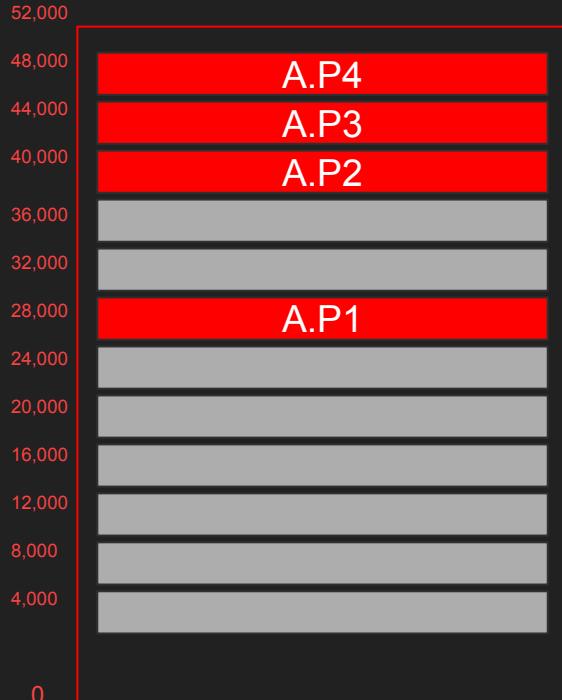
Copy on Write

A



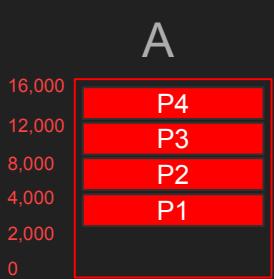
A page table

VMA	PA
4000	28,000
8000	40,000
12,000	44,000
16,000	48,000



A is the only running process., B is about to be forked from A

Copy on Write

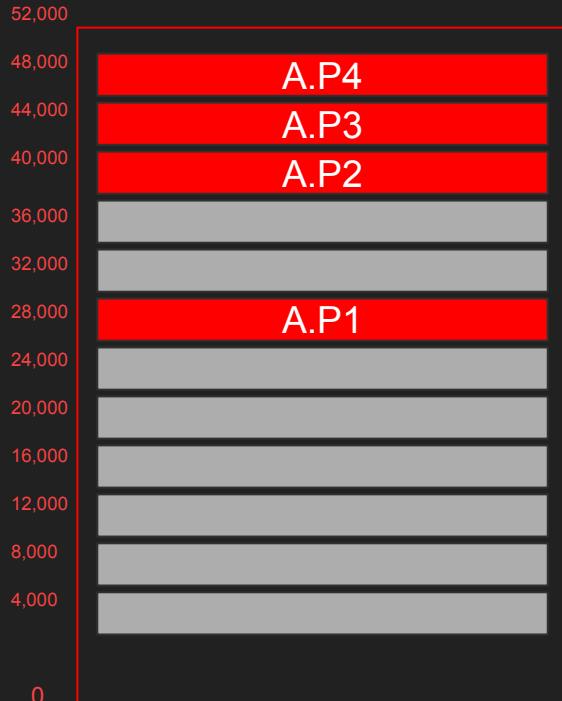


A page table

VMA	PA
4000	28,000
8000	40,000
12,000	44,000
16,000	48,000

B page table

VMA	PA
4000	28,000
8000	40,000
12,000	44,000
16,000	48,000



B is a fork of A, initially they are identical, their page tables point to the same physical memory

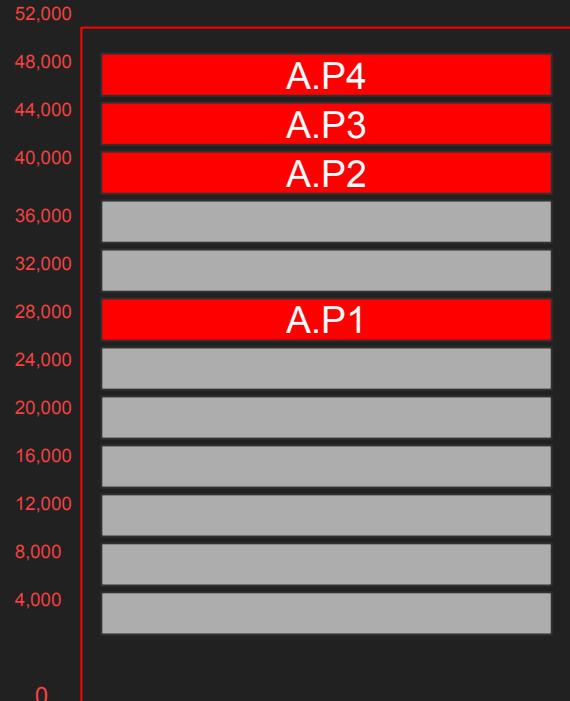
Copy on Write



change

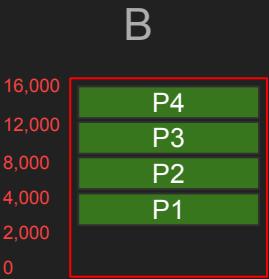
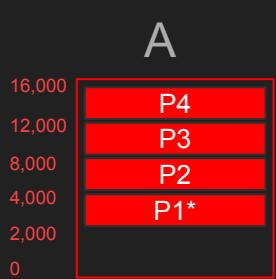
A page table

B page table



A makes a change to an address in Page P1, P1 needs to be copied for B

Copy on Write

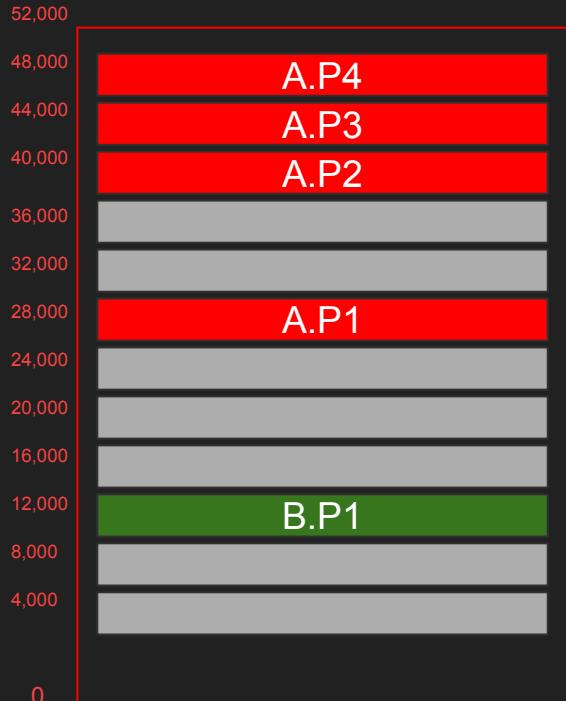


A page table

VMA	PA
4000	28,000
8000	40,000
12,000	44,000
16,000	48,000

B page table

VMA	PA
4000	12,000
8000	40,000
12,000	44,000
16,000	48,000



New page is allocated for B, A.P1 is copied in it, page table is updated to point to it.

Python CoW bug

- Python bug None, True, False
- Refcounting was constantly updated
- Forks were triggering CoW

Summary

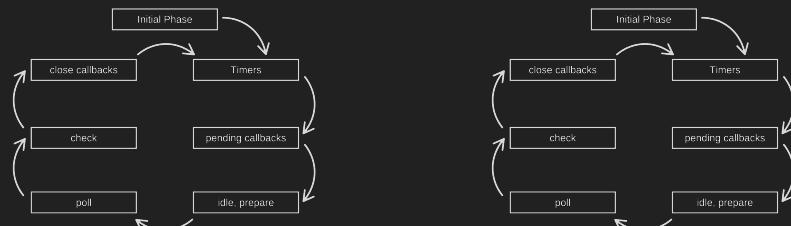
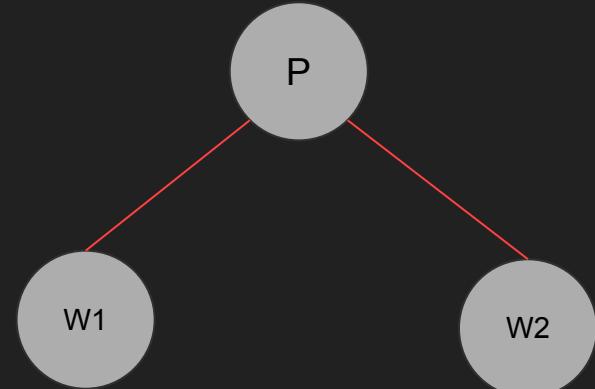
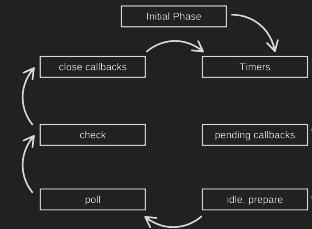
- Processes have dedicated VM
- Threads belong to the same process
- Threads share parent process memory

Node Workers

Threads

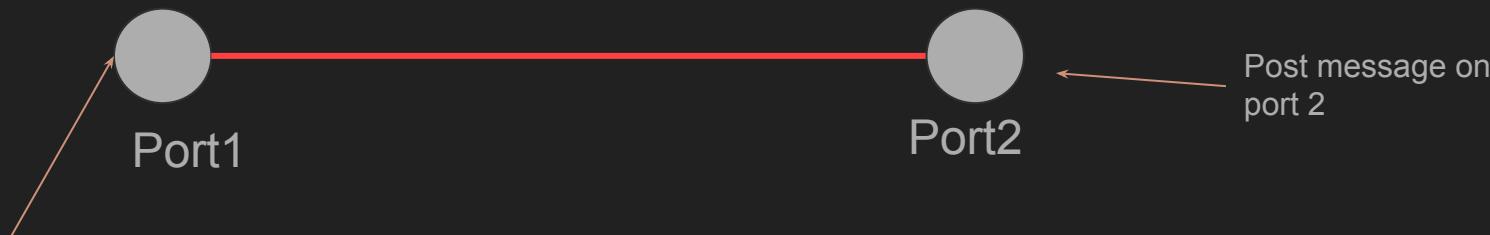
Node threads

- Same process, multiple threads
- Share parent process memory/file descriptors
- Implemented with `worker_threads`
- Each thread gets its own main loop
- Useful for large CPU intensive, spin a thread



Message channel

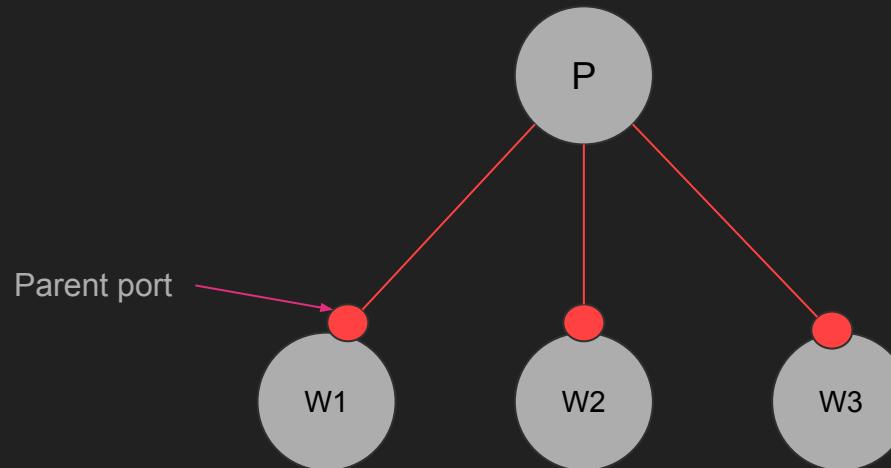
- How do Node threads communicate?
 - Parent and worker thread
 - Thread to another thread
- Meet Message channel
- Message channel has two ports
- By default Parent gets a MessageChannel with every worker



Listen on “message”
event to receive it on
port 1

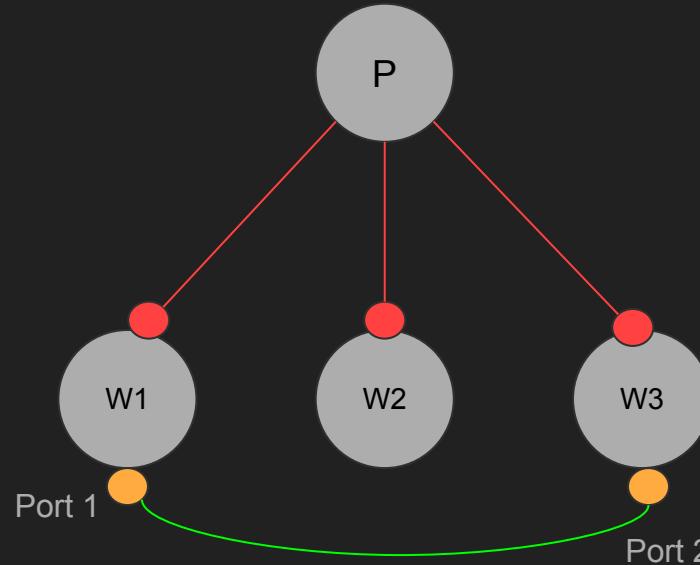
Threads and Parent

- Each thread gets a parent port
- Access to the parent
- Worker can send message to parent through that
- Parent can send message to worker
- Listen on message



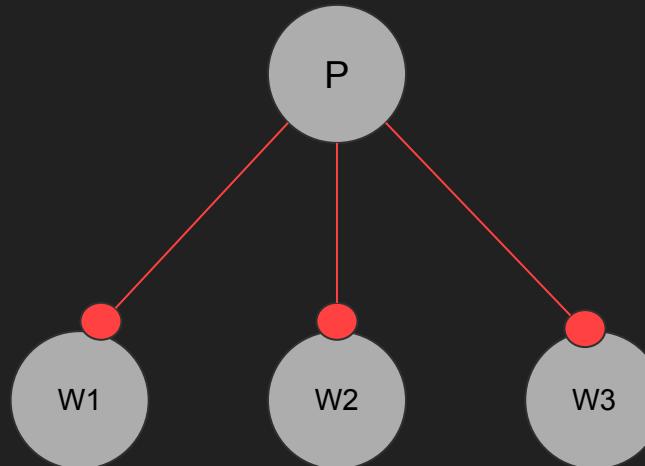
Worker threads messaging

- We can create a message channel
- And send it to the two threads
- Must be on a transferList so that node clones it
- Not everything is transferable!



workerData

- Pass some data from parent to worker
- Available immediately at start
- E.g. want to process 1 million messages
- Break it between 3 threads
 - Each gets $\frac{1}{3}$



How many threads

- Too many vs too few
- Logical cores vs Physical cores
- cgroups, namespaces, containers
- os.CPUS vs os.availableParallelism

Example

```
1 const WorkerThreads = require('node:worker_threads');  
2  
3 //this tells us whether we are on the main thread or worker  
4 if (WorkerThreads.isMainThread) {  
5     //spin a new thread running the same file  
6     console.log("I'm parent thread: " + WorkerThreads.threadId)  
7     const worker = new WorkerThreads.Worker(__filename);  
8     //the thread will be created in the poll phase  
9     setTimeout(()=> console.log("Timer"), 0)  
10    console.log("Start initial phase")  
11    for (let i =0; i< 10000000000;i++);  
12    console.log("End initial phase")  
13 } else {  
14  
15     console.log("I'm a worker thread: " + WorkerThreads.threadId)  
16  
17 }
```

Since we are running same script, we need to know if we are main or worker

Nothing will happen until poll phase

We are in the thread

Example

```
1 const WorkerThreads = require('node:worker_threads');  
2  
3 //this tells us whether we are on the main thread or worker  
4 if (WorkerThreads.isMainThread) {  
5     //spin a new thread running the same file  
6     console.log("I'm parent thread: " + WorkerThreads.threadId)  
7     const worker = new WorkerThreads.Worker(__filename);  
8     //the thread will be created in the poll phase  
9     setTimeout(()=> console.log("Timer"), 0)  
10    console.log("Start initial phase")  
11    for (let i =0; i< 10000000000;i++);  
12    console.log("End initial phase")  
13 } else {  
14  
15     console.log("I'm a worker thread:  
16  
17 }
```

● MacBook-Pro:node-course-content HusseinNasser\$ node
I'm parent thread: 0
Start initial phase
End initial phase
Timer
I'm a worker thread: 1

Summary & Demo

- Node threads
- Code exercise →
 - Node thread simple example
 - Parent to thread messaging
 - Thread to thread messaging
 - Distribute connections between threads
 - Distribute requests between threads

Example 2

```
1 const WorkerThreads = require('node:worker_threads');;
2 //this tells us whether we are on the main thread or worker
3 if (WorkerThreads.isMainThread) {
4     //spin a new thread running the same file
5     console.log("I'm parent thread: " + WorkerThreads.threadId)
6     const worker = new WorkerThreads.Worker(__filename);
7     //the thread will be created in the poll phase
8     setTimeout(()=> console.log("Parent Timer"), 0)
9     console.log("Start parent initial phase")
10    for (let i =0; i< 10000000000;i++);
11    console.log("End parent initial phase")
12 } else {
13     //child thread gets its own loop
14     setTimeout(()=> console.log("Child Timer"), 0)
15     console.log("Start Thread's initial phase")
16     for (let i =0; i< 10000000000;i++);
17     console.log("End Thread's initial phase")
18     console.log("I'm a worker thread: " + WorkerThreads.threadId)
19 }
```

Example 2

```
1 const WorkerThreads = require('node:worker_threads');;
2 //this tells us whether we are on the main thread or worker
3 if (WorkerThreads.isMainThread) {
4     //spin a new thread running the same file
5     console.log("I'm parent thread: " + WorkerThreads.threadId)
6     const worker = new WorkerThreads.Worker(__filename);
7     //the thread will be created in the poll phase
8     setTimeout(()=> console.log("Parent Timer"), 0)
9     console.log("Start parent initial phase")
10    for (let i =0; i< 10000000000;i++);
11    console.log("End parent initial phase")
12 } else {
13     //child thread gets its own loop
14     setTimeout(()=> console.log("Child Timer"),
15     console.log("Start Thread's initial phase")
16     for (let i =0; i< 10000000000;i++);
17     console.log("End Thread's initial phase")
18     console.log("I'm a worker thread: " + Worker
19 }
```

● MacBook-Pro:node-course-content HusseinNasser\$
I'm parent thread: 0
Start parent initial phase
End parent initial phase
Parent Timer
Start Thread's initial phase
End Thread's initial phase
I'm a worker thread: 1
Child Timer

Node Child Processes

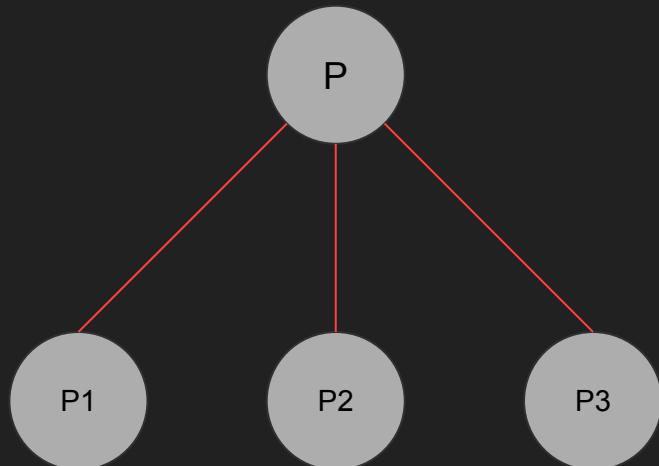
Processes

Process

- Too many threads in one process
 - Large memory (shared)
 - No isolation
- Process gets dedicated memory
- Isolation
- Works with other programs (not just Node)

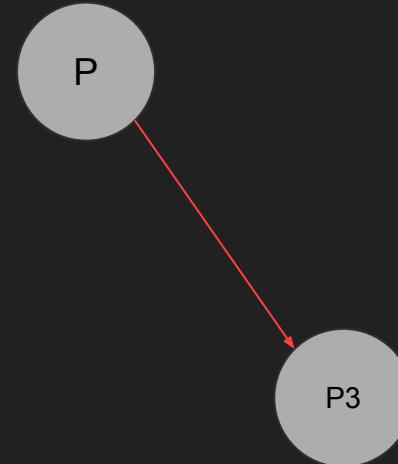
Child process

- When forking a process we get a free IPC channel
- Processes can communicate with each other
- IPC is shared buffer memory in kernel
- Inter-process communication



Node fork

- Forks a Node process
- Run in the initial phase (not poll phase)
- Still asynchronous
- Doesn't clone the process like unix fork
- Users should manually setup child processes
 - E.g. Who is parent who is child
 - Pass env variable to tell us who is child
 - Cluster helps here

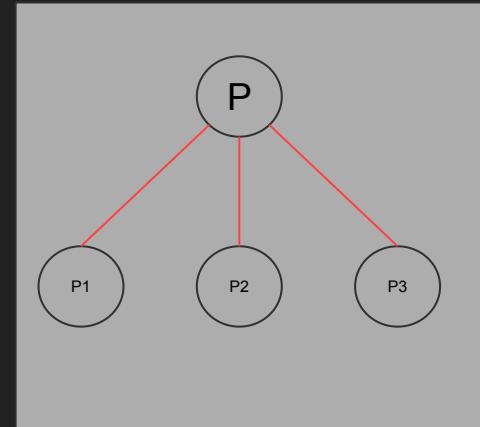


exec vs spawn

- Both spin processes to external commands (not node)
- Exec can be used to run external commands
 - E.g Ifconfig, ls etc
- Gets you back a child process and full stdout
- Not great for streaming large stdout
- Spawn lets you listen for stdout as they come in
 - E.g. top

Cluster mode

- Cluster is a module that helps manage child processes
- Gives you an indication who is parent
- Communications between child and parent
- Great for socket handling
- Scheduling algorithms



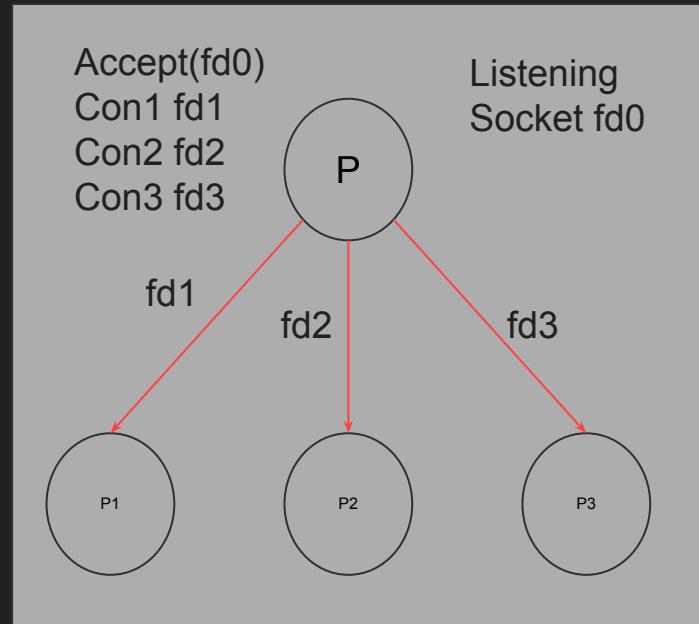
Cluster mode

```
const cluster = require('cluster');
```

- `cluster.isPrimary` //who is the master process
- `cluster.fork()`; //forks a new process and add it to the cluster
- Scheduling policy

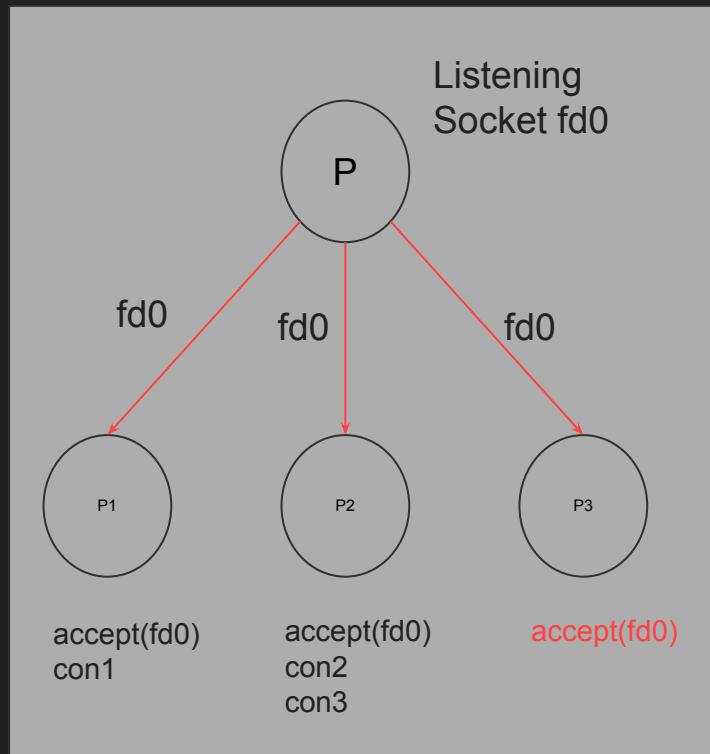
Sockets Scheduling policy - Round Robin

- Scheduling policy for cluster for sockets
- cluster.SCHED_RR (Round robin)
- Parent creates the listening socket
- Parent accept the connection
- Parent sends connection to each worker
- Round robin fashion



Sockets Scheduling policy - None

- cluster.SCHED_NONE
- Parent creates the listening socket
- Parent sends the listening socket to workers
- Workers accept the connection themselves
- Depends on workers performance
- Some workers might not get any



Example

```
22-child-process > js 220-child-process.js > ...
1 //with processes we will need our own indicator
2 const child_process = require('child_process');
3 //this tells us whether we are on the main thread or worker
4 if (!process.env.isChildProcess) {
5   //spin a new process running the same file
6   console.log("Start parent initial phase")
7   console.log("I'm parent thread: " + process.pid)
8   //we have to set a env variable so we know which process is a child
9   console.log("Before forking..")
10  const worker = child_process.fork(__filename, [], { env: { isChildProcess: 'true' } });
11  console.log("After forking..")
12
13  //the thread will be created in the poll phase
14  setTimeout(()=> console.log("Parent Timer"), 0)
15  console.log("About to start a loop ..")
16
17  for (let i =0; i< 10000000000;i++);
18  console.log("End parent initial phase")
19  process.on("exit", ()=>console.log("Parent exit"))
20 } else {
21   //child thread gets its own loop
22   setTimeout(()=> console.log("Child Timer"), 0)
23   console.log("Start Thread's initial phase")
24   console.log("I'm a worker process: " + process.pid)
25
26   for (let i =0; i< 10000000000;i++);
27   console.log("End Thread's initial phase")
28   process.on("exit", ()=>console.log("Child exit"))
29 }
30
```

Manually set env variable

This immediately spins the process (in the initial phase)

Each process gets its own loop

Example

```
22-child-process > js 220-child-process.js > ...
1 //with processes we will need our own indicator
2 const child_process = require('child_process');
3 //this tells us whether we are on the main thread or worker
4 if (!process.env.isChildProcess) {
5   //spin a new process running the same file
6   console.log("Start parent initial phase")
7   console.log("I'm parent thread: " + process.pid)
8   //we have to set a env variable so we know which process is a child
9   console.log("Before forking..")
10  const worker = child_process.fork(__filename, [], { env: { isChildProcess: 'true' } })
11  console.log("After forking..")
12
13  //the thread will be created in the poll phase
14  setTimeout(()=> console.log("Parent Timer"), 0)
15  console.log("About to start a loop ..")
16
17  for (let i =0; i< 10000000000;i++);
18  console.log("End parent initial phase")
19  process.on("exit", ()=>console.log("Parent exit"))
20 } else {
21   //child thread gets its own loop
22   setTimeout(()=> console.log("Child Timer"), 0)
23   console.log("Start Thread's initial phase")
24   console.log("I'm a worker process: " + process.pid)
25
26   for (let i =0; i< 10000000000;i++);
27   console.log("End Thread's initial phase")
28   process.on("exit", ()=>console.log("Child exit"))
29 }
30
```

- HusseinMac:node-course-content |
Start parent initial phase
I'm parent thread: 4713
Before forking..
After forking..
Start Thread's initial phase
I'm a worker process: 4714
End parent initial phase
Parent Timer
End Thread's initial phase
Child Timer
Child exit
Parent exit

Summary & Demo

- Node processes
- Code exercise →
 - Node process simple example
 - Parent to Worker IPC
 - Spawn and Exec
 - Cluster

Node Performance

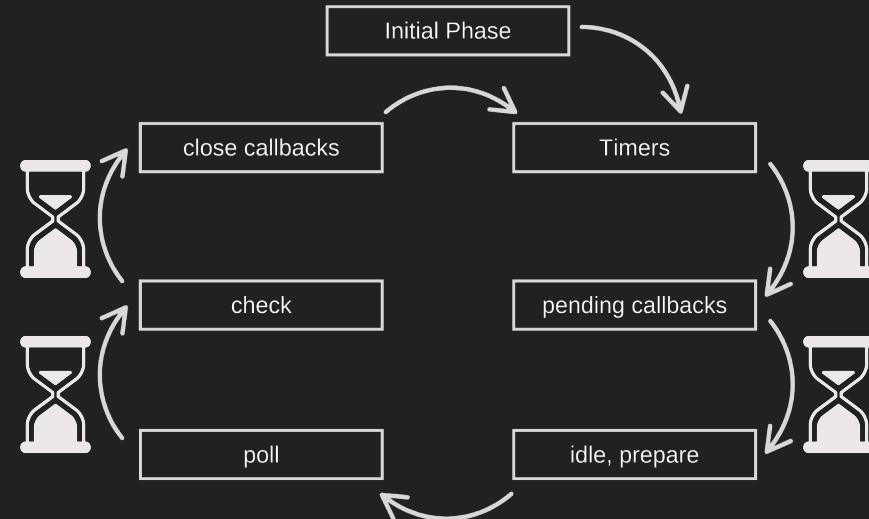
How to keep Node performing

Node Performance

Keep Node Performant

Node Performance

- Node is still single threaded*
- So any CPU intensive starves the loop
- Starving the loop, affects IO
- Sync IO can also starve the loop
- Give breathing room to other phases
 - poll, check, and timers etc..
- So what could we do?



Keep Callbacks Small and Constant!

- Callbacks could be scheduled in any phase
- They are almost always executed on main thread
- Keep callbacks predictable in cost
- Will discuss what to do if you can't

Input should be bounded!

- Building APIs/SDKs/Backends
- You need to expose “input”
 - request, function parameters, objects etc.
- The input should be validated and bounded
- Can’t have users submit:
 - large strings, large numbers, large objects
- Reject or ask the client to send a smaller input
 - HTTP -> request size
 - Try not to read large input only to block it

Try using asynchronous io

- Offload IO when possible to Node's workers
- Instead of `readFileSync` use `readFile`
- These block the main thread
- Same goes for everything “sync”

Watch out for modules

- Watch out for modules you import
- They may run synchronous code that blocks
- Starving your io
- Load them async if possible (import)
- Console.time is your friend

```
5 | console.time('require Time');
6 | require('fetch');
7 | console.timeEnd('require Time');
8 |
9 | console.time('Import Time');
10| import('fetch');
11| console.timeEnd('Import Time');
```

● MacBook-Pro:23-performance
require Time: 7.193ms
Import Time: 1.023ms

Partitioning

- If a callback is slow try breaking it into smaller units
- Schedule the unit of work in the next check phases
- With setImmediate
- Breath!

Break it and schedule it..

Large block..

```
for ( i =0; i< 1_000_000_000; i++); →
```

```
const maxCount = 1_000_000_000;
//let us break the execution
function countingIt (start, end ){
  //if we get to 10m resolve
  if (i >= maxCount) {
    console.log("Promise resolved.")
    resolve("Counted" + i)
    return;
  }
  //console.log (`counting .. ${start} to ${end}`)
  for ( i =start; i< end; i++);
    setImmediate (countingIt, start+ 1_000_000, end + 1_000_000)
```

Offloading

- When partitioning isn't enough
- Offload the workload to another thread or process
- Very useful for scalability
- But communications becomes tricky

Summary & Demo

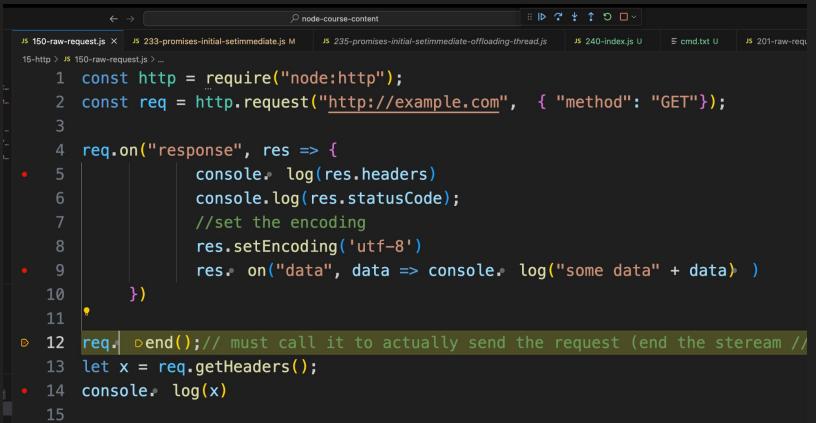
- Node Performance
- Code exercise →
 - Expensive promise
 - Schedule it to the next tick
 - Schedule it to the next check phase
 - Partition it
 - Offload

Debugging NodeJS

Various methods of debug

Debug Node

- Visual studio code add in
- VSCode communicates with Node
- WebSockets



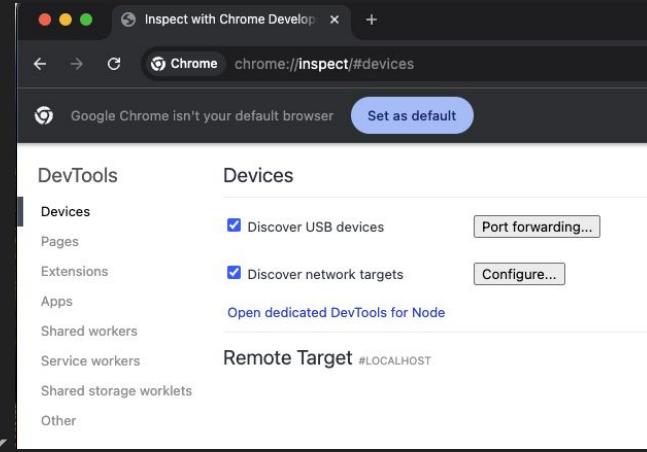
The screenshot shows a browser-based Node.js debugger interface titled "node-course-content". The main area displays a file named "150-raw-request.js" with the following code:

```
1 const http = require("node:http");
2 const req = http.request("http://example.com", { "method": "GET"});
3
4 req.on("response", res => {
5     console.log(res.headers)
6     console.log(res.statusCode);
7     //set the encoding
8     res.setEncoding('utf-8')
9     res.on("data", data => console.log("some data" + data) )
10 }
11
12 req.end(); // must call it to actually send the request (end the stream)
13 let x = req.getHeaders();
14 console.log(x)
```

The code includes several breakpoints indicated by red dots. A tooltip for one breakpoint at line 12 shows the message "Breakpoint hit. You can step into, out of, or over code. Set break points with 'break' statements or click the gutter." The line "req.end();" is highlighted in green, suggesting it is currently being executed or has just been executed.

Remote Debug

- Remote debug
 - With websockets
 - Uses inspect protocol
- \$ node --inspect-brk index.js //run with inspect-brik
- Debugger listening on
ws://127.0.0.1:9229/485b8dd0-e8ae-404b-8049-2da3a136e94b
- Chrome devtools supports inspect protocol



Remote Debug

The screenshot shows the "Inspect with Chrome DevTools" interface in a browser window. The title bar says "Inspect with Chrome DevTools". The address bar shows "chrome://inspect/#devices". A message at the top says "Google Chrome isn't your default browser" with a "Set as default" link. On the left, a sidebar titled "DevTools" lists "Devices", "Pages", "Extensions", "Apps", "Shared workers", "Service workers", and "Shared storage worklets". The "Devices" section is expanded, showing "Discover USB devices" (checked) and "Port forwarding..." button. It also shows "Discover network targets" (checked) and "Configure..." button. Below this is a link "Open dedicated DevTools for Node". The main area is titled "Remote Target #LOCALHOST". It shows "Target (v23.5.0) trace" and the file path "230-nextticks.js file:///Users/HusseinNasser/projects/node-course-content/23-performance/230-nextticks.js inspect".

Summary & Demo

- Node Debug
- Code exercise →
 - Node debug with visual studio code
 - Node with chrome://inspect/

Capturing Node traffic

tcpdump/Wireshark/Proxy

Capturing Node raw packets

- Node makes network calls
- packet be captured with wireshark or tcpdump
- HTTP/TCP/UDP/TLS/IP
- Can't easily decrypt traffic

```
04:20:27.339628 IP 17.253.83.217.80 > 192.168.7.125.49283: Flags [R], seq 2155437383, win 0, length 0
04:20:27.339638 IP 17.253.83.217.80 > 192.168.7.125.49283: Flags [.R], seq 2155437383, win 0, length 0
04:20:27.348362 IP 23.215.0.136.80 > 192.168.7.125.53533: Flags [.], ack 62, win 509, options [nop,nop,TS val 2164206285 ecr 3820397119], length 0
04:20:28.348609 IP 192.168.7.125.53533 > 23.215.0.136.80: Flags [.], ack 1519, win 2048, length 0
04:20:28.418235 IP 23.215.0.136.80 > 192.168.7.125.53533: Flags [.], ack 62, win 509, options [nop,nop,TS val 2164207355 ecr 3820397119], length 0
04:20:29.470812 IP 192.168.7.125.53533 > 23.215.0.136.80: Flags [.], ack 1519, win 2048, length 0
04:20:29.545442 IP 23.215.0.136.80 > 192.168.7.125.53533: Flags [.], ack 62, win 509, options [nop,nop,TS val 2164208483 ecr 3820397119], length 0
04:20:29.635596 IP 192.168.7.125.53533 > 23.215.0.136.80: Flags [.], ack 1519, win 2048, length 0
04:20:30.710866 IP 23.215.0.136.80 > 192.168.7.125.53533: Flags [.], ack 62, win 509, options [nop,nop,TS val 2164209646 ecr 3820397119], length 0
04:20:31.775973 IP 192.168.7.125.53533 > 23.215.0.136.80: Flags [.], ack 1519, win 2048, length 0
04:20:31.852428 IP 23.215.0.136.80 > 192.168.7.125.53533: Flags [.], ack 62, win 509, options [nop,nop,TS val 2164210787 ecr 3820397119], length 0
04:20:33.885478 IP 192.168.7.125.53533 > 23.215.0.136.80: Flags [.], ack 1519, win 2048, length 0
04:20:33.156452 IP 23.215.0.136.80 > 192.168.7.125.53533: Flags [.], ack 62, win 509, options [nop,nop,TS val 2164212893 ecr 3820397119], length 0
04:20:34.224328 IP 192.168.7.125.53533 > 23.215.0.136.80: Flags [.], ack 1519, win 2048, length 0
04:20:34.301381 IP 23.215.0.136.80 > 192.168.7.125.53533: Flags [.], ack 62, win 509, options [nop,nop,TS val 2164213236 ecr 3820397119], length 0
04:20:35.382445 IP 192.168.7.125.53533 > 23.215.0.136.80: Flags [.], ack 1519, win 2048, length 0
04:20:35.377693 IP 23.215.0.136.80 > 192.168.7.125.53533: Flags [.], ack 62, win 509, options [nop,nop,TS val 2164214313 ecr 3820397119], length 0
04:20:35.629982 IP 192.168.7.125.53533 > 23.215.0.136.80: Flags [F.], seq 62, ack 1519, win 2048, options [nop,nop,TS val 3828441165 ecr 2164214313], length 0
04:20:35.706146 IP 23.215.0.136.80 > 192.168.7.125.53533: Flags [F.], seq 1519, ack 63, win 509, options [nop,nop,TS val 2164214642 ecr 3828441165], length 0
04:20:35.706225 IP 192.168.7.125.53533 > 23.215.0.136.80: Flags [.], ack 1520, win 2048, options [nop,nop,TS val 3820441241 ecr 2164214642], length 0
```



HTTP/HTTPS proxy

- We can augment scripts to capture https traffic via proxy
- Popular package https-proxy-agent
- We can spin a proxy and then direct Node towards it
- Using mitmproxy to intercept!
- Node needs to be taught to accept mitmproxy cert



Summary & Demo

- Node Intercept
- Code exercise →
 - Wireshark
 - http/https proxy

Request library showdown

Request vs fetch vs undici vs axios

Demo

- Sending 100 requests
- Measuring memory usage and time
- General idea

C++ Add-ons

When JavaScript is not enough

C++ Addons

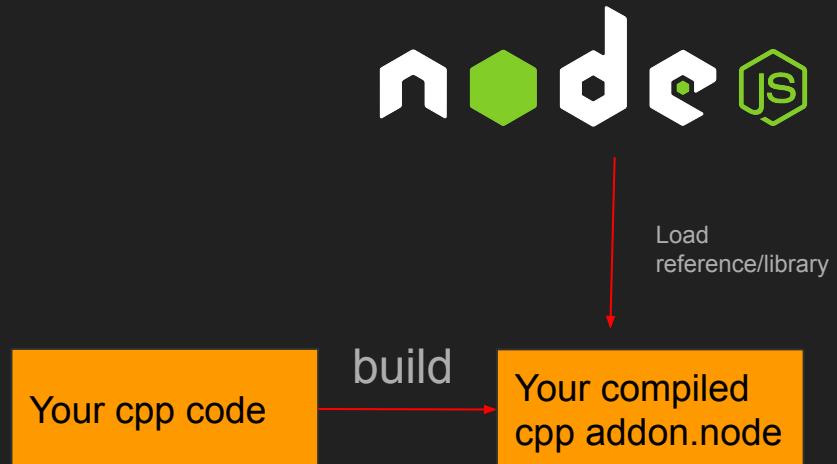
- JavaScript is great for 99% of cases
- However sometimes it cannot hold performance
- Being an interpreted language
- Node allows for native C++ libraries to be included
- Run within Node process

Authoring C++ Addons

- Hook to the native V8
 - Better performance
 - But can break V8 API changes
- N-API, Node API
 - Abstraction C++ on top of V8
 - More compatible, additional layer
- Will use N-API

What are we building?

- We really are building a DLL/object library
- That node then references and use
- It becomes part of Node process



Hooking back to Node

- Interfaces and classes allow “hooks” back to Node
- napi C++ data types
- C++ is great but it's better to respect Node rules
 - Since we are running in the context of Node
- Example you can read file in C++
 - But Node won't know !
 - Better to use uv_lib read file

N-API

- Node C++ API can be installed with node-addon-api
- This installed `<napi.h>` and other dependencies
- We reference `<napi.h>` in our cpp file
- Npm install node-addon-api

How to build?

- We use Google's gyp (Generate your Projects)
- Node-gyp is a special tool for node
- Reads from a binding.gyp file
- Need to tell gyp where to find n-api
- Npm install node-gyp

binding.gyp

28-isprime.cpp > binding.gyp

```
1  {
2    "targets": [
3      {
4        "target_name": "sum",
5        "sources": [ "sum.cpp" ],
6        "include_dirs": [ "node_modules/node-addon-api" ],
7        "dependencies": [ "node_modules/node-addon-api/node_api.gyp:nothing" ],
8        "defines": [ "NAPI_DISABLE_CPP_EXCEPTIONS" ]
9      }
10    ]
11 }
```

The image shows annotations for the binding.gyp file:

- An annotation points to the `target_name: "sum"` line with the text "Output executable name (library really)".
- An annotation points to the `sources: ["sum.cpp"]` line with the text "C++ Sources".
- An annotation points to the `include_dirs: ["node_modules/node-addon-api"]` line with the text "Path for n-api".
- An annotation points to the `dependencies: ["node_modules/node-addon-api/node_api.gyp:nothing"]` line with the text "Path for node-gyp".
- An annotation points to the `defines: ["NAPI_DISABLE_CPP_EXCEPTIONS"]` line with the text "Disable NAPI (compiler specific) exceptions".

Note: there is another way to dynamically locate the napi/node-gyp

output.node

- With all that ready we do npm install
- Node-gyp gets to work and build our cpp file
- Generates the library to be consumed by node
- Release vs debug

Important NApi objects

- `Napi::CallbackInfo`
- `Napi::Env`
- `Napi::Object`
- `Napi::Function`
- `Napi::String`
- `Napi::Number`

Module initialization

```
38 // Register the function as a Node.js module
39 Napi::Object Init(Napi::Env env, Napi::Object exports) {
40     exports.Set(Napi::String::New(env, "addNumbers"),
41                 Napi::Function::New(env, AddNumbers));
42     return exports;
43 }
```

What function to call

What Node sees and calls

Debugging C++ Addon

```
28-nodecpp-addon > sum.cpp > AddNumbers(const Napi::CallbackInfo &)
4 void AddNumbers(const Napi::CallbackInfo& info) {
5     double num1 = info[0].As<Napi::Number>().DoubleValue();
6     double num2 = info[1].As<Napi::Number>().DoubleValue();
7     // Compute sum
8     double sum = num1 + num2;
9
10    // Get the callback function
11    Napi::Function callback = info[2].As<Napi::Function>();
12
13    // Call the callback with the result
14    callback.Call({ env.Null(), Napi::Number::New(env, sum) });
15
16 }
17
18 // Register the function as a Node.js module
```

Summary & Demo

- Node Addons
- Code exercise →
 - Sum.cpp two number
 - Build on release
 - Build on debug