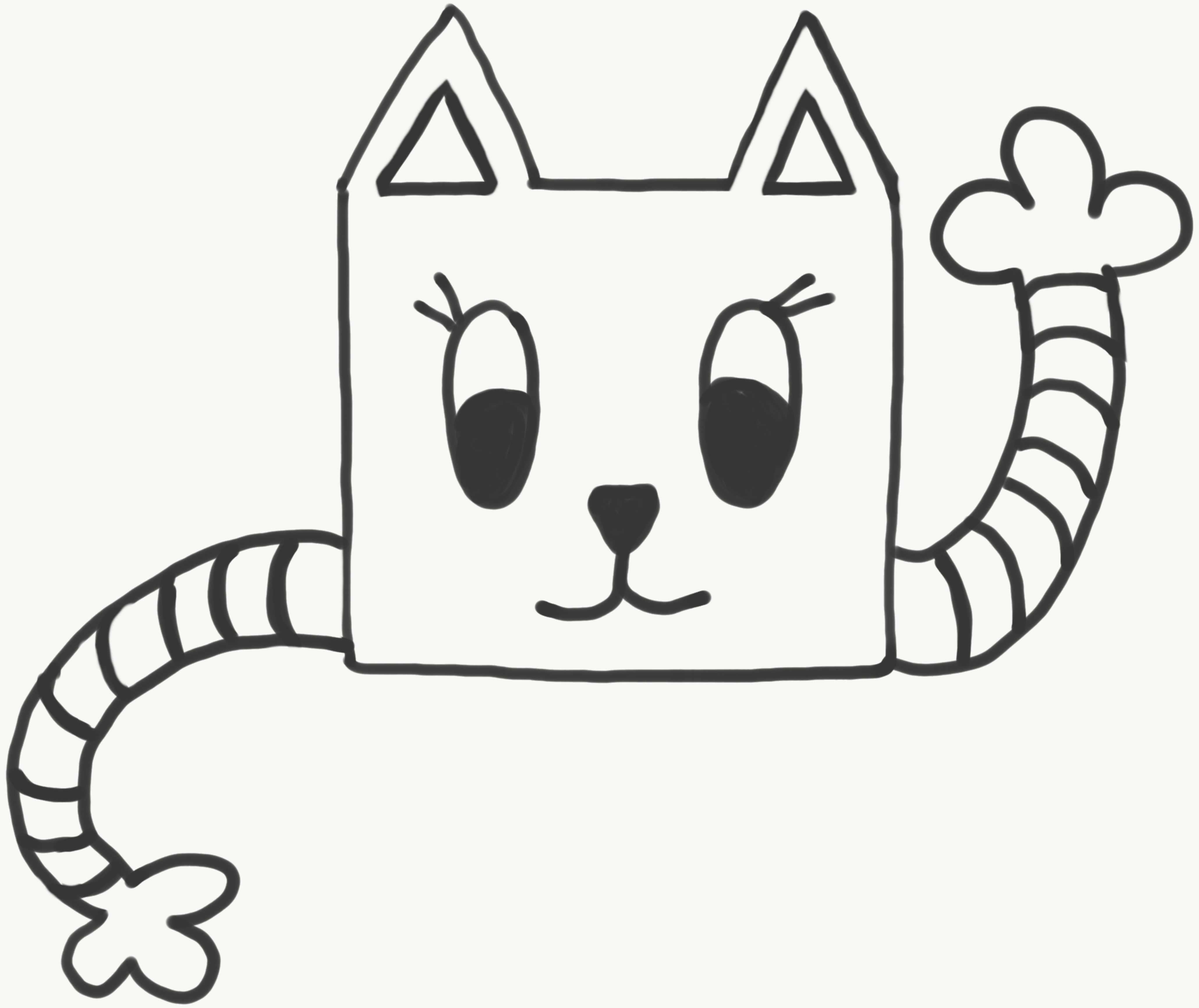
The battle of the event loops

Ujjwal Sharma (@ryzokuken)
featuring
Olga Kobets (@homyasusina)

Ujjwal Sharma (he/him)

- Compilers Hacker at Igalia
- Node.js Core Collaborator
- TC39 Delegate
- Work on V8 and Cranelift (Spidermonkey/wasmtime)
- Student
- Speaker

Барсик B1000



The event loop *has* to be one of the most talked about subjects in JavaScript

Let’s dig a little deeper.

Section I

Concurrency

Concurrency vs Parallelism

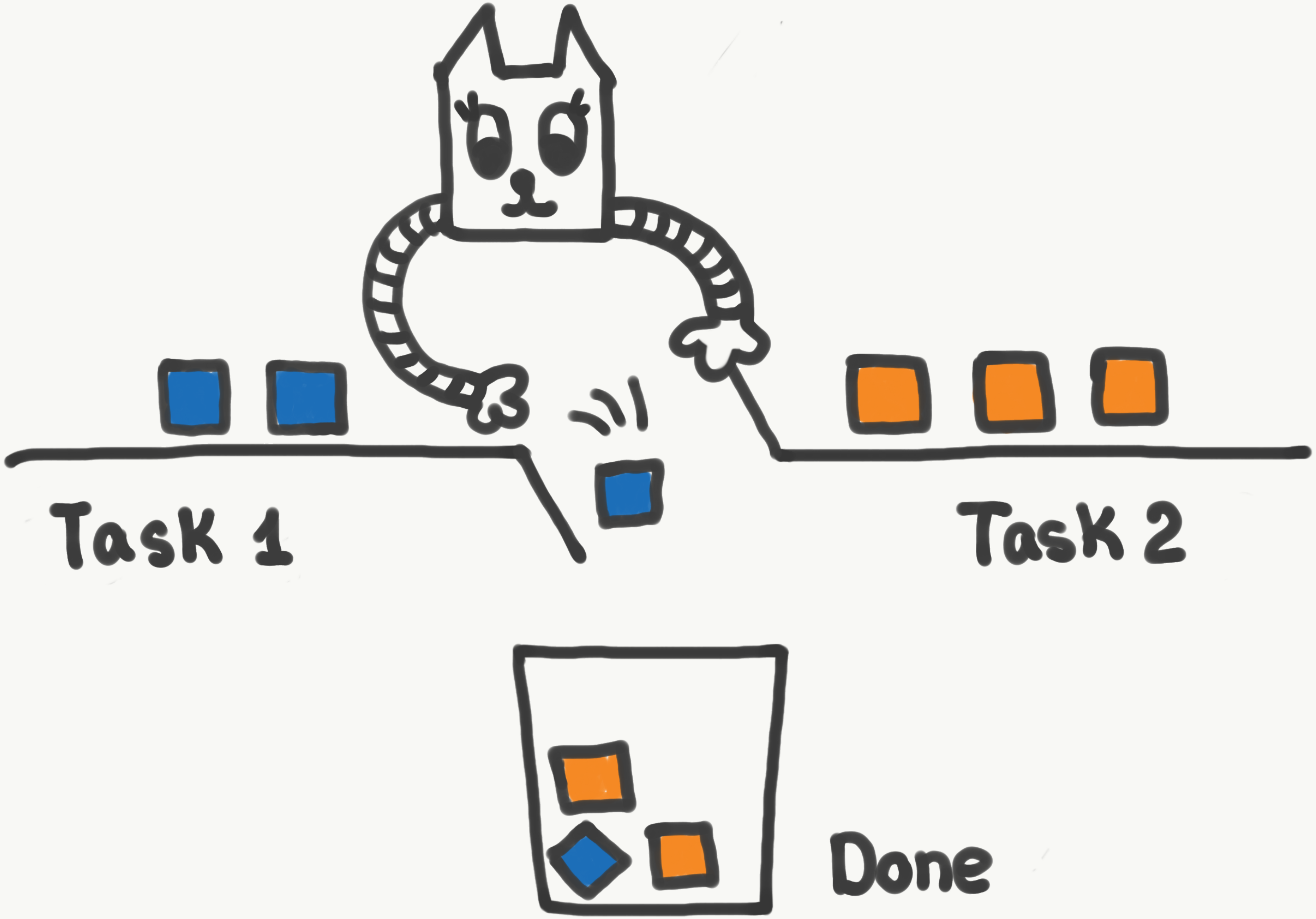
Concurrency

/kənˈkʌr(ə)nsi/

noun

When two or more tasks can start, run, and complete in overlapping time periods.

Example: multitasking on a single-core machine


Task 1
Task 2
Done

Parallelism

/ˈparəlɛlɪzəm/

noun

The state of being parallel or of corresponding in some way.

Example: the Greek thinkers used to believe in the parallelism of microcosm and macrocosm

😅

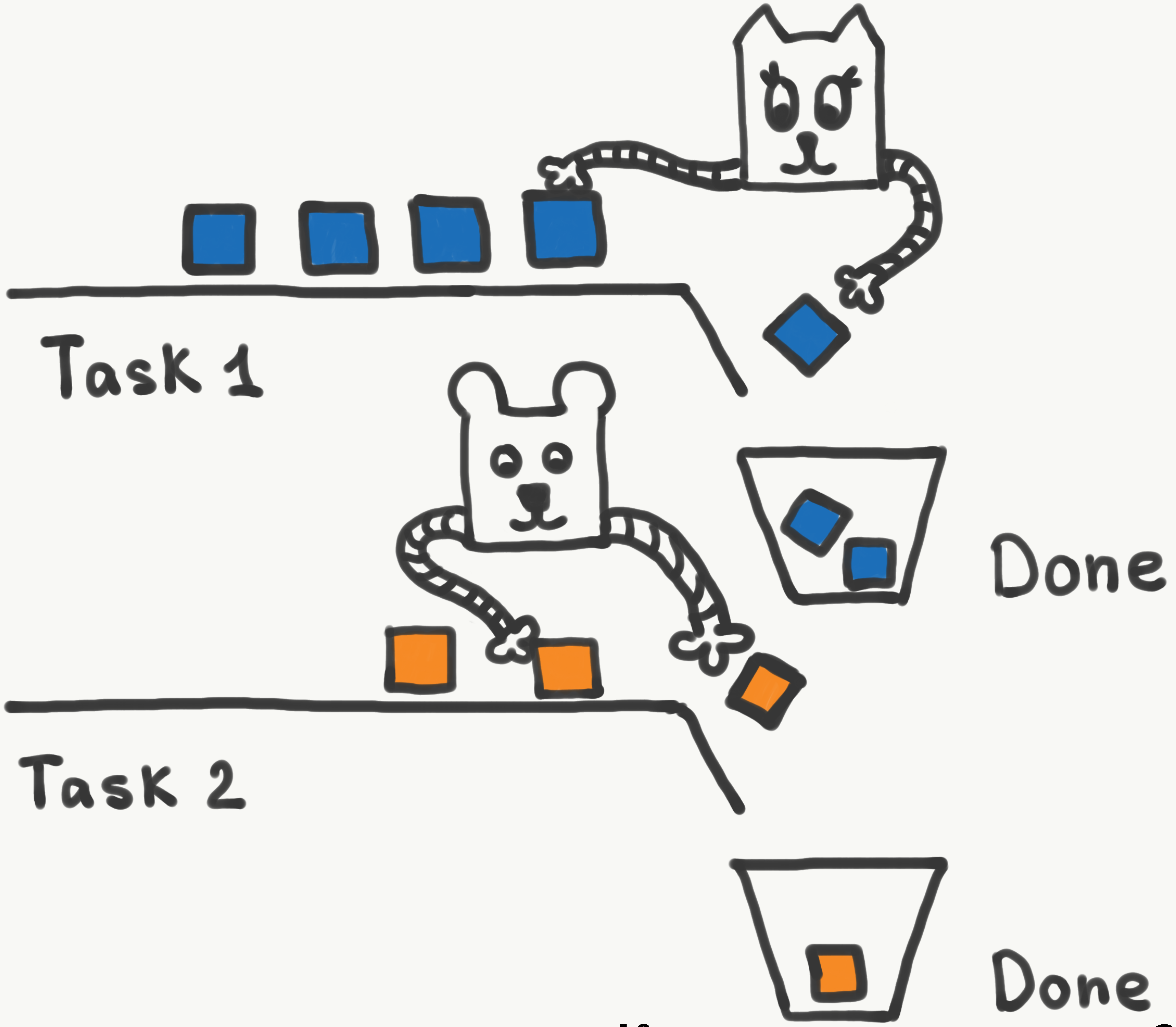
Parallelism

/ˈparəlɛlɪzəm/

noun

When tasks literally run at the same time.

Example: a multicore processor


Task 1
Task 2
Done
Done

If computation is said to be *concurrent*, then it doesn't necessarily dictate how the concurrency is achieved under the hood.

JavaScript is single-threaded

V8 is single-threaded

> “There are many who pretend to despise and belittle that which is beyond their reach.”
>
> *– Aesop (Aesop’s Fables)*

JavaScript does not need multithreading

Two main reasons for operations to be time-consuming:

1. Operations that perform heavy computation.

2. Operations that depend on something.

Two main reasons for operations to be time-consuming:

1. Operations that require CPU time.

~~1. Operations that perform heavy computation.~~

2. Operations that *wait* for something.

~~2. Operations that depend on something.~~

99% of all applications do nothing 99% of the time

Multithreading is useful when

1. Significant CPU time is required.
2. Need to call an awkward synchronous (blocking) API.

Node.js

```
const cluster = require("cluster")

const workers = require("worker_threads")
```

Deno

```
const worker = new Worker(...)
```

So how do you use single-threaded concurrency in the real world?

Section II

Asynchronous Programming

Asynchrony

/eɪˈsɪŋ krəˌni/

noun

The occurrence of events independent of the main program flow and ways to deal with such events.

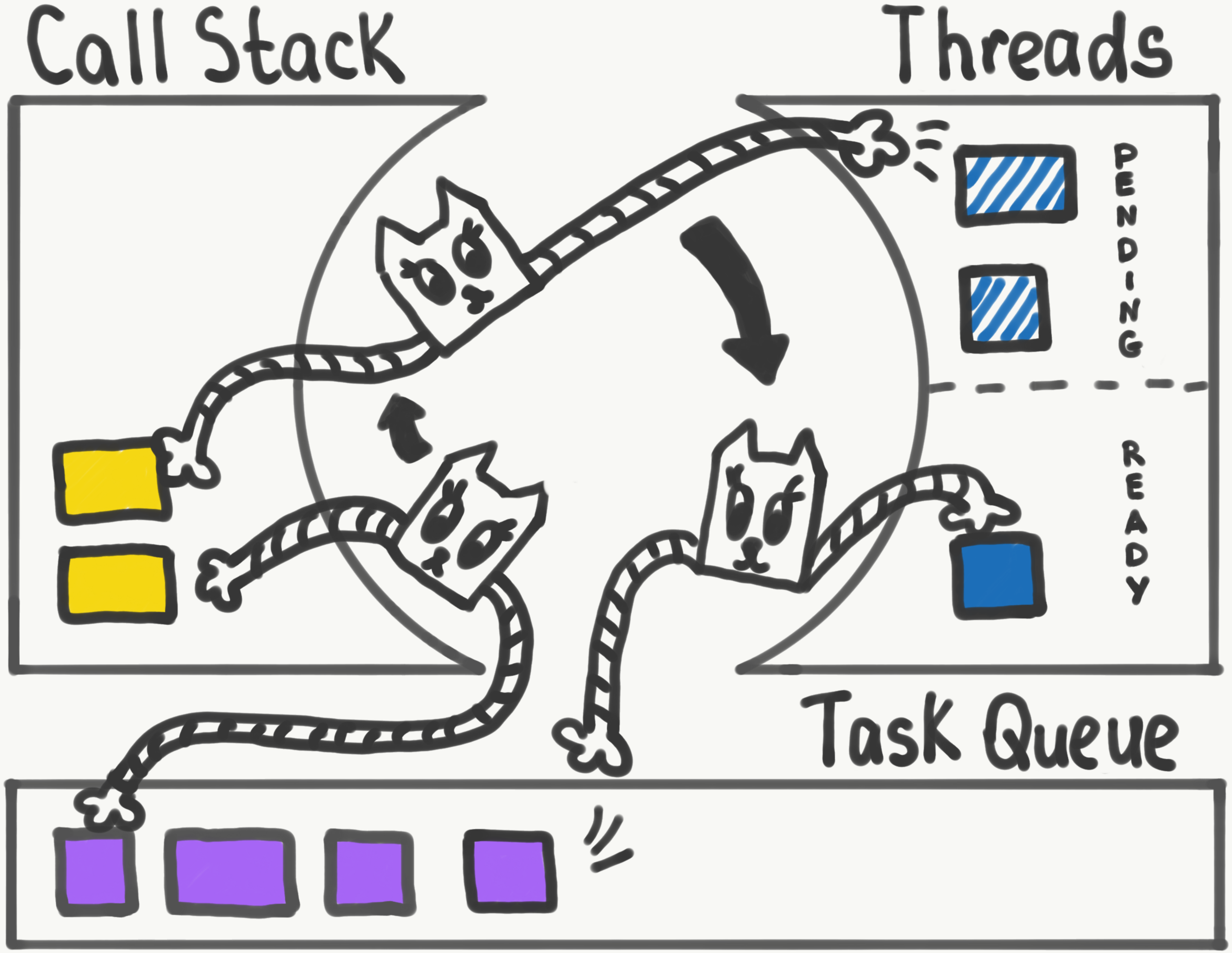
Event-driven programming is by far the most popular paradigm to achieve asynchrony

Green Threads is a popular alternative

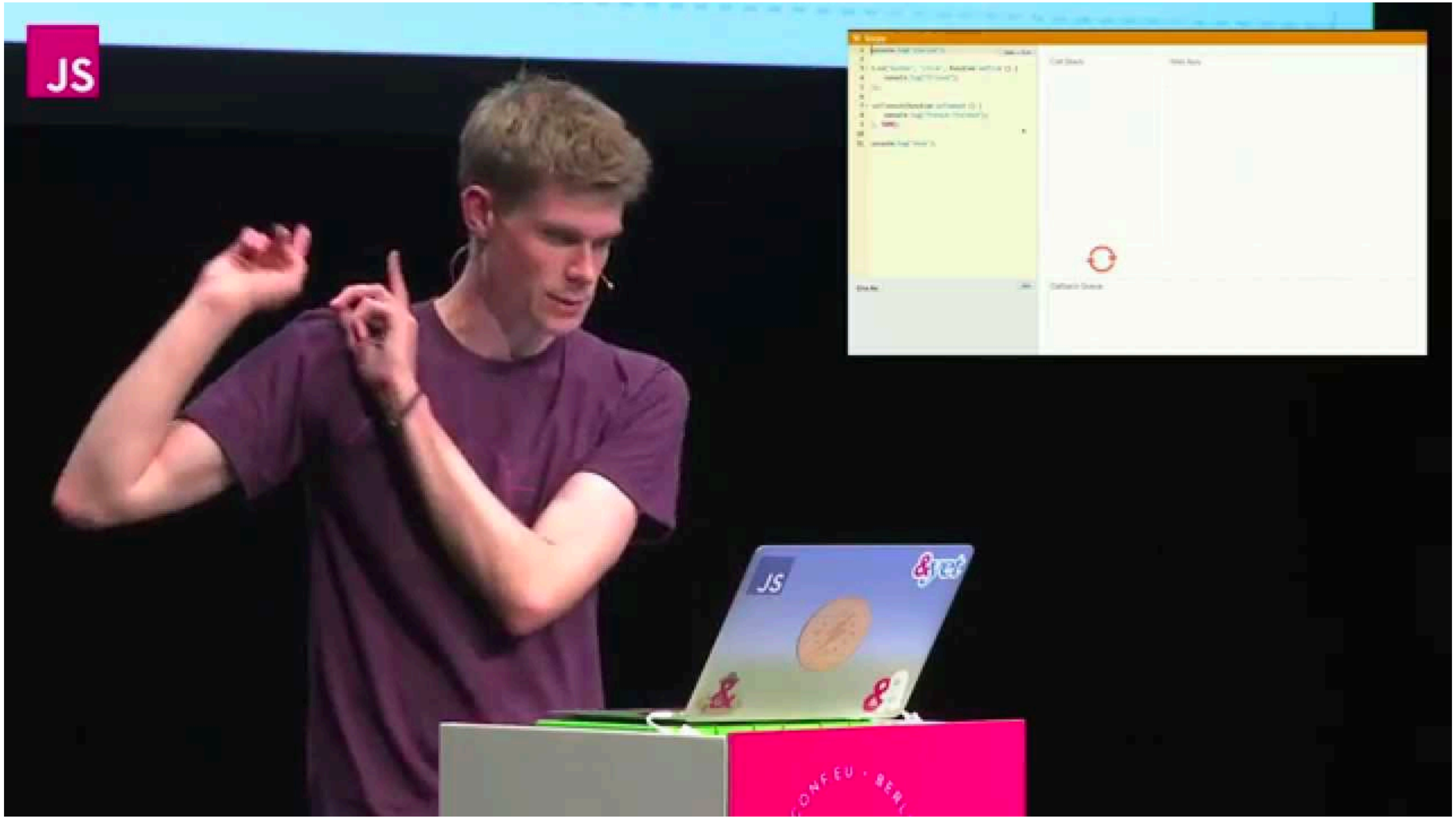
We're not the first ones to use event-driven systems to build web servers

- .NET (C#)
- Spark (Java)
- Twisted (Python)
- Express (JavaScript)
- Vapor (Swift)
- Rocket (Rust)

JavaScript has a concurrency model based on an event loop


Call Stack
Threads
PENDING
READY
Task Queue

What the heck is the event loop anyway? | Philip Roberts | JSConf EU



Иван Тулуп: асинхронщина в JS под капотом / Михаил Башуров (Luxoft)

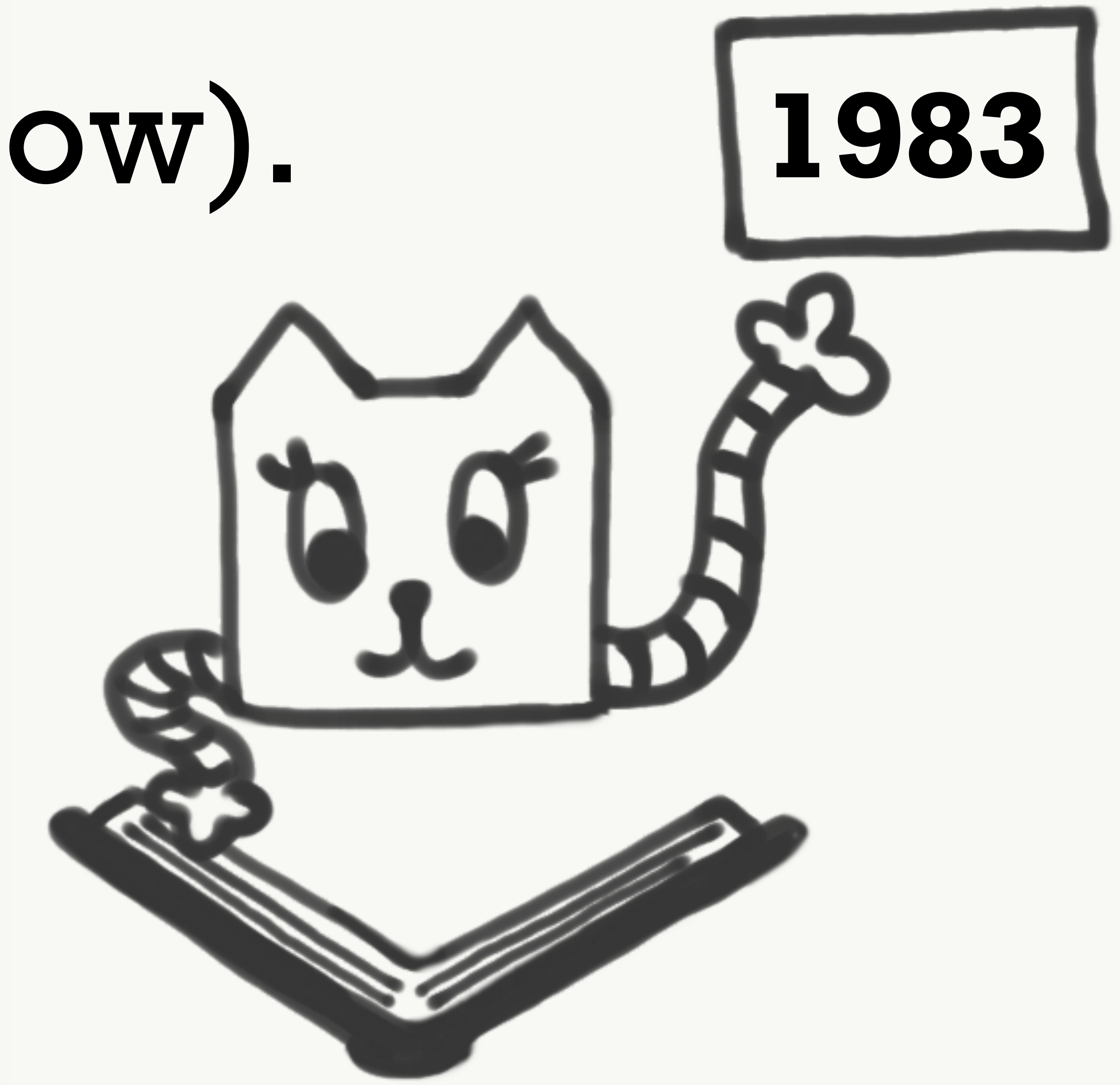
But the “event loop” is a theoretical model

Section III

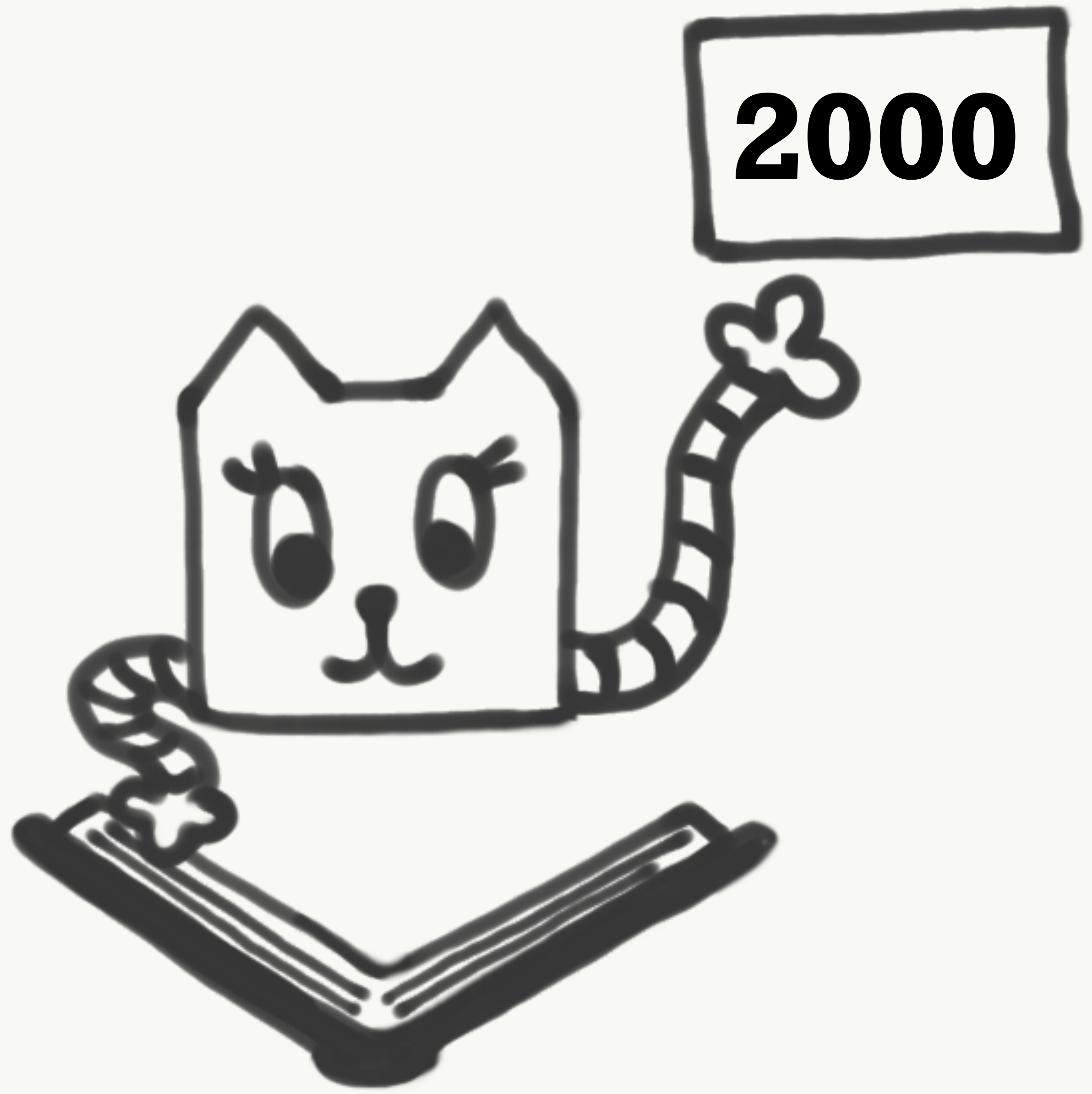
Event Loops

poll and select

- **History**: Introduced in the ~80s-90s (old).
- **Functionality**: More or less the same (boring).
- **Speed**: Perform similarly on benchmarks (slow).
- **Portability**: Everywhere (nice).
- **Complexity**: As simple as it gets (neat).

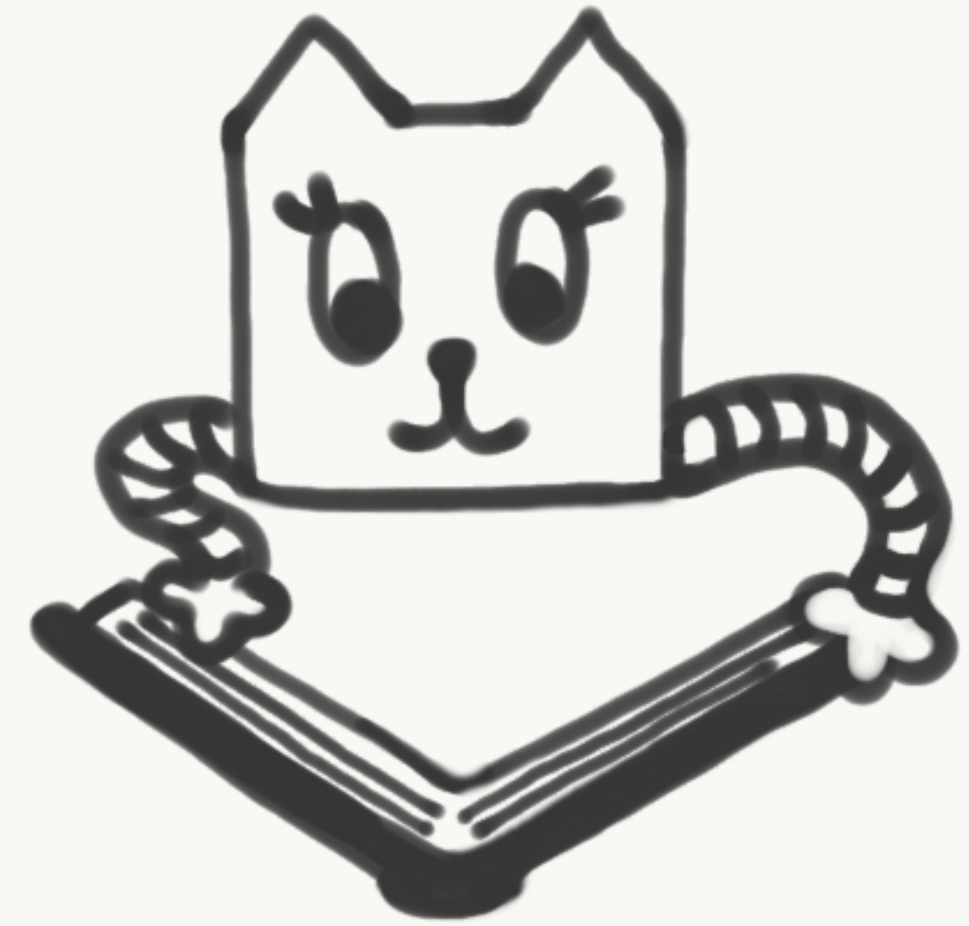


Result: libevent

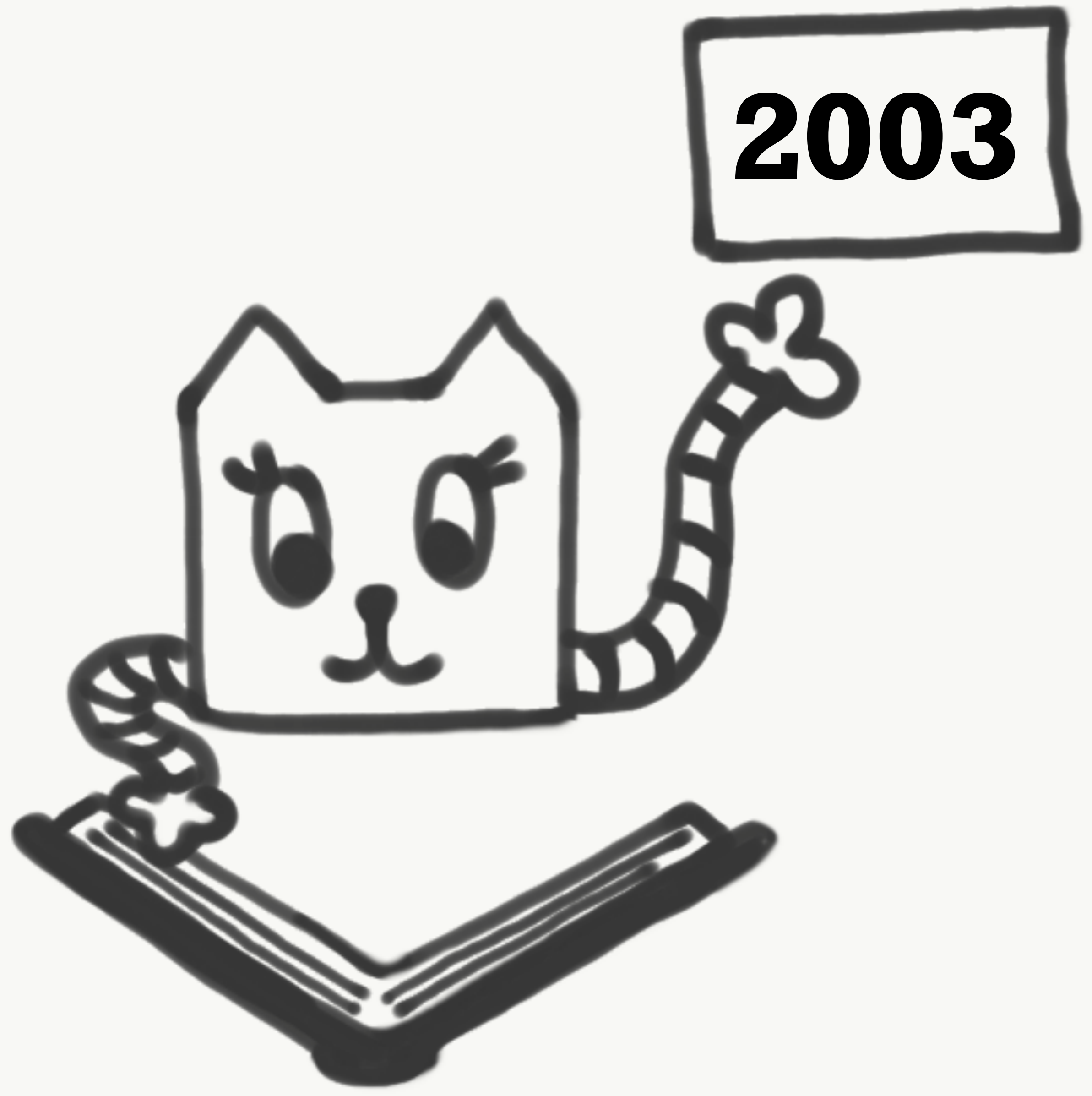


People **loved** poll

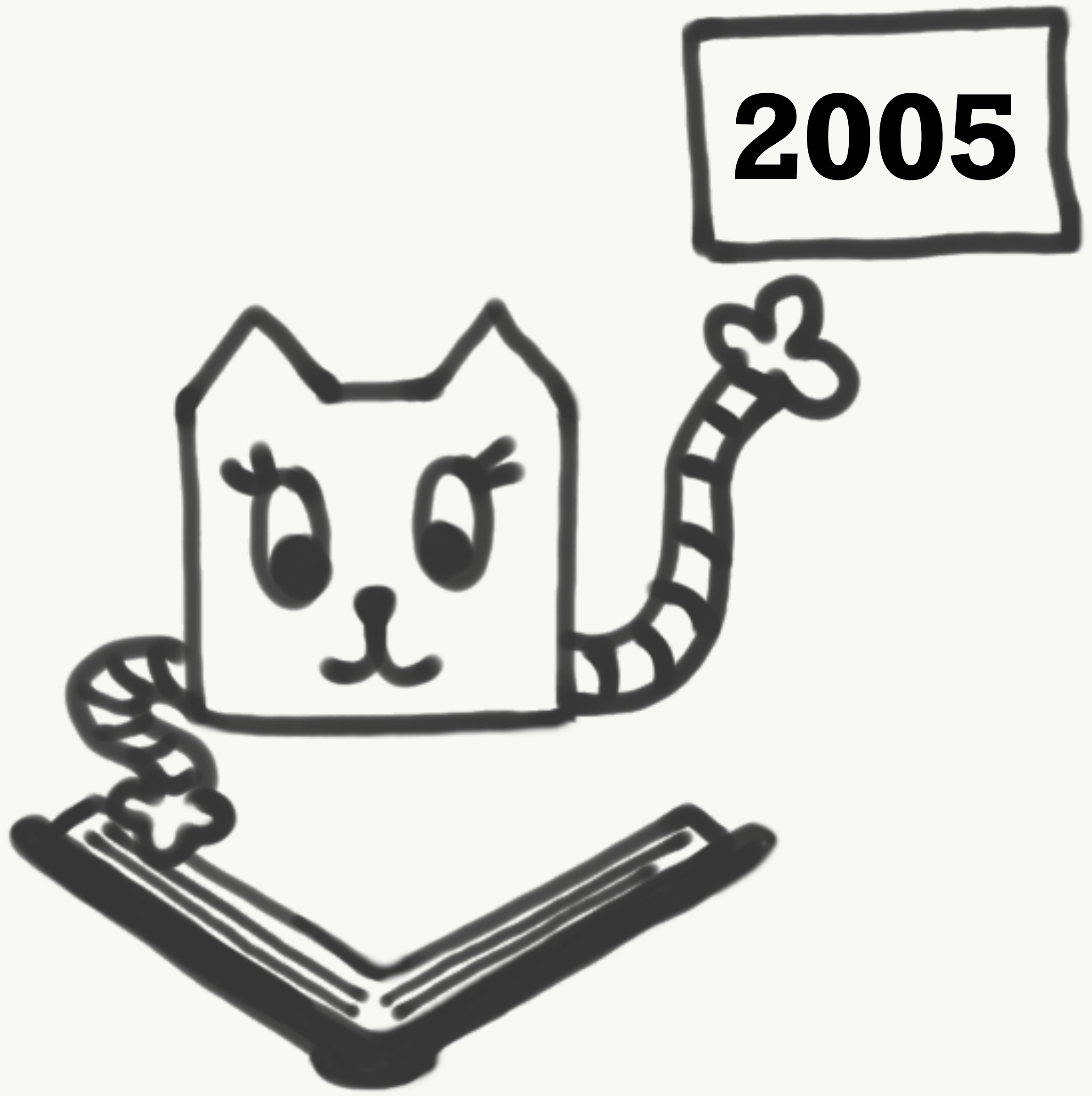
- epoll
- /dev/poll
- kqueue
- pollset
- inotify



Result: libevent

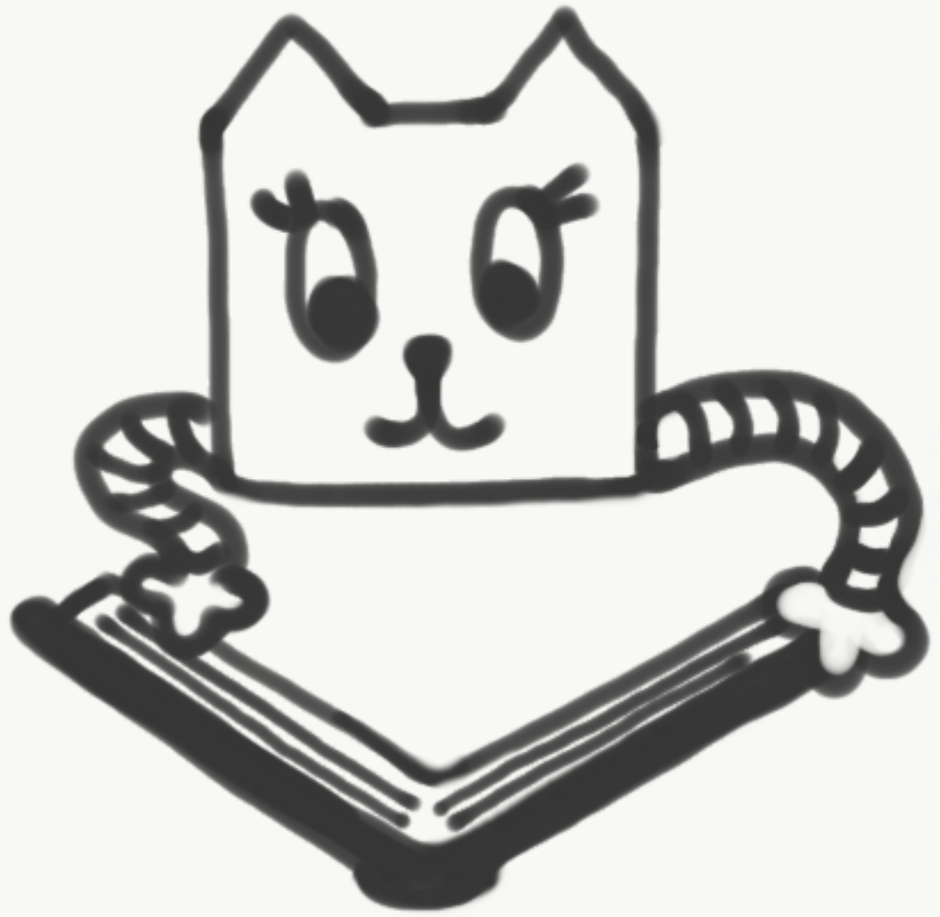


Result: libevent*

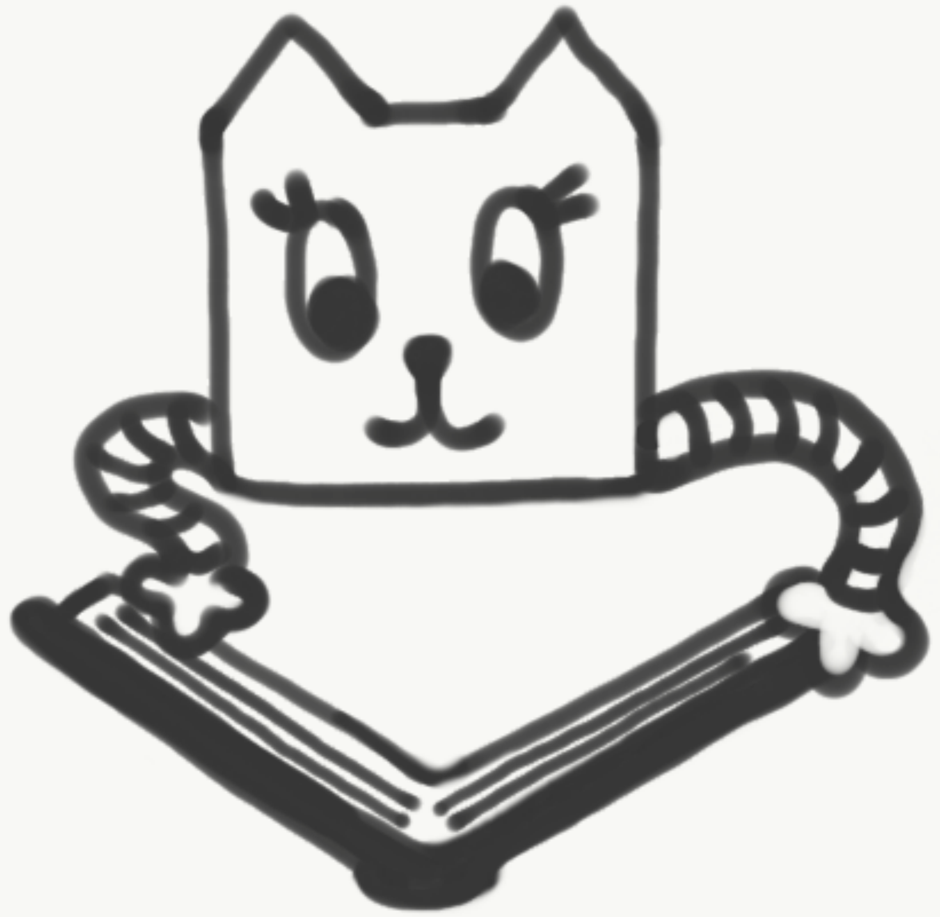


*** Slightly Faster**

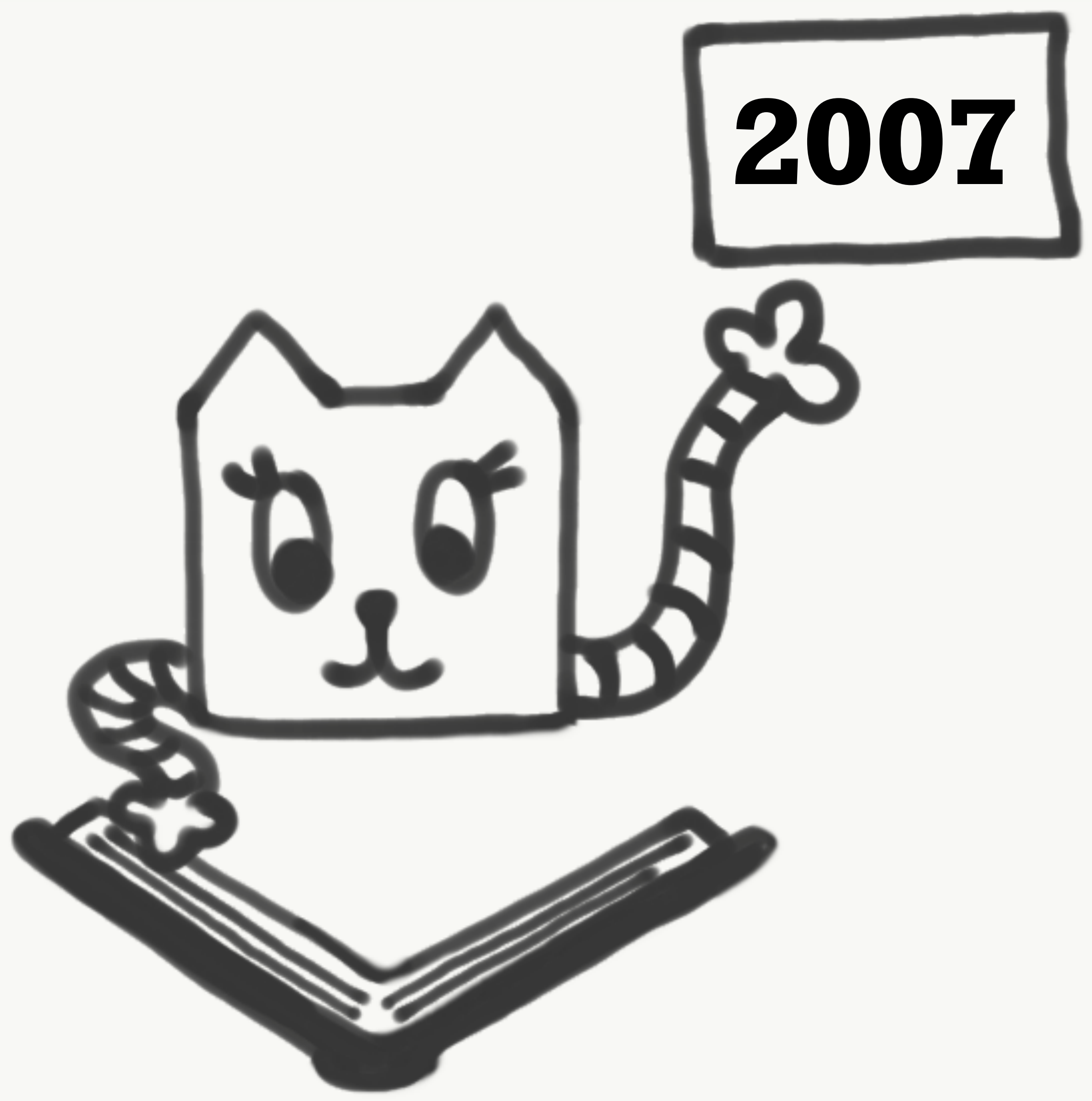
Problem: libevent is too…



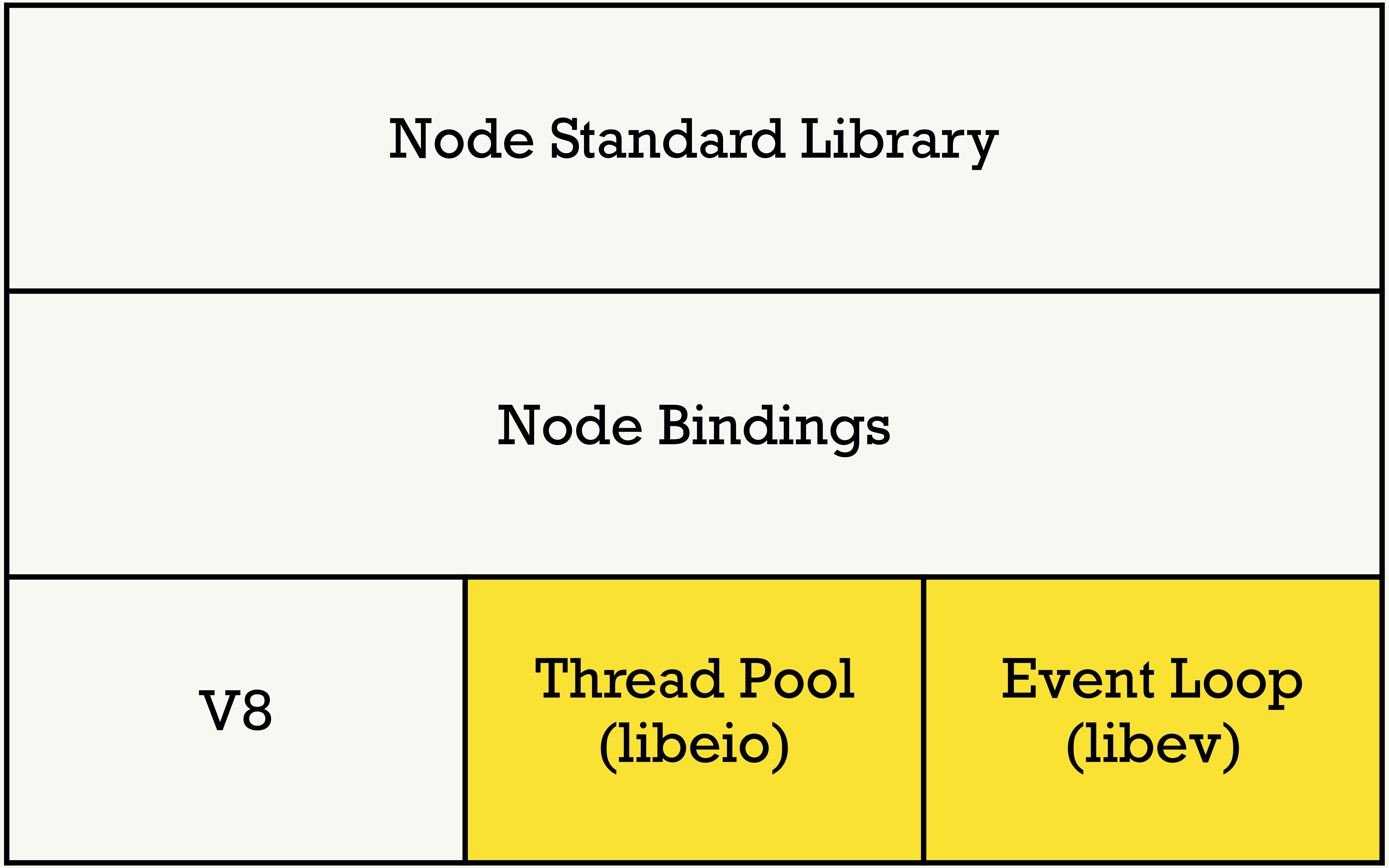
Problem: libevent is too… bloated



Result: libev



Node.js
Lv4
HP
node
JS
Wild Javascript Runtime appeared!



Node Standard Library

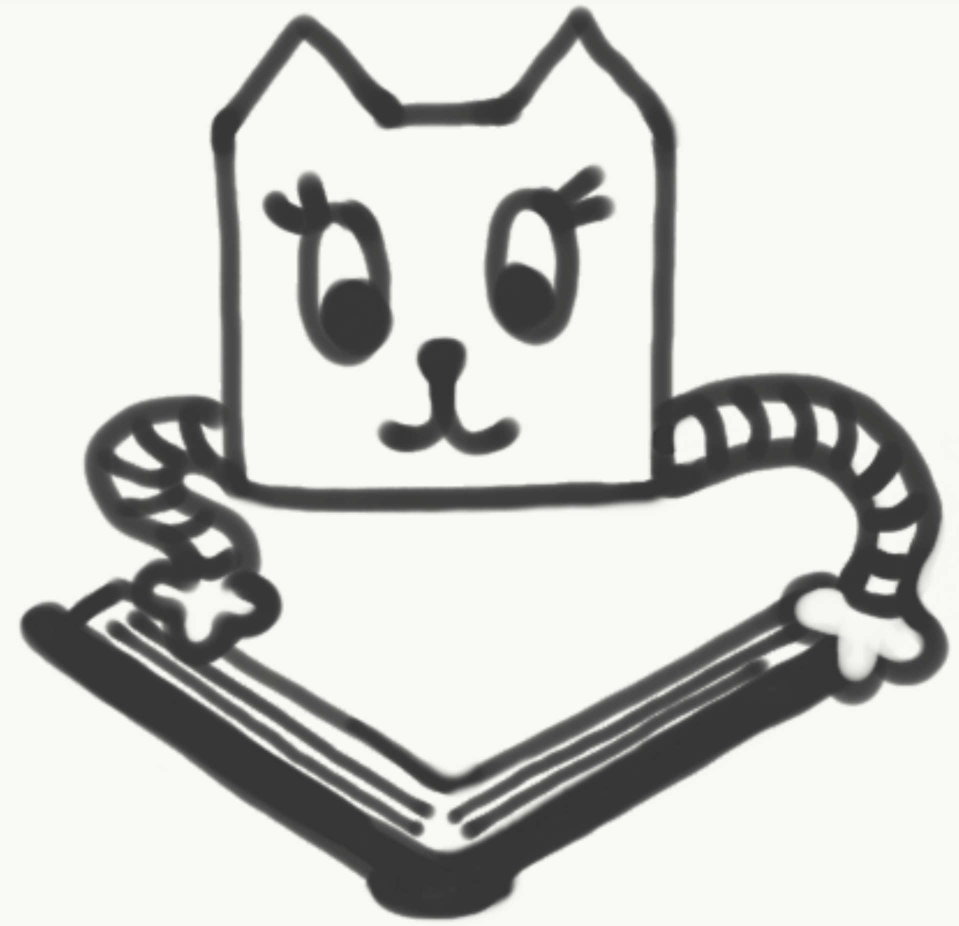
Node Bindings

V8

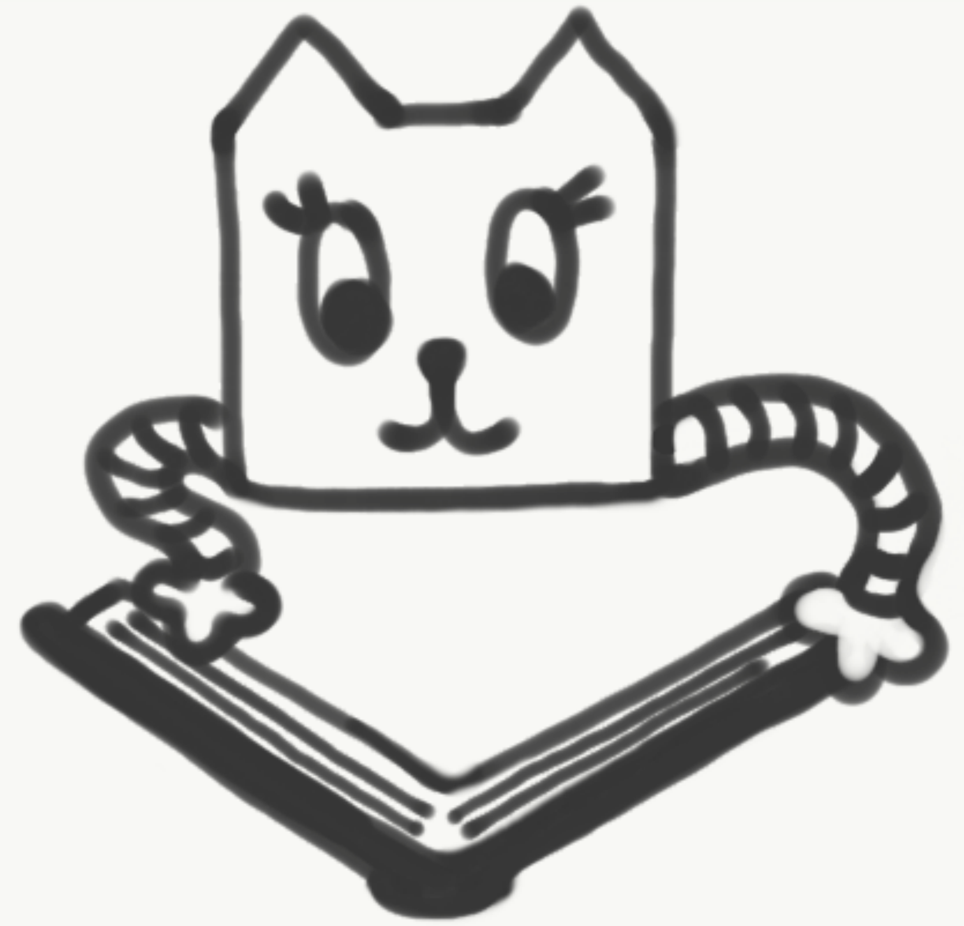
Thread Pool (libeio)

Event Loop (libev)

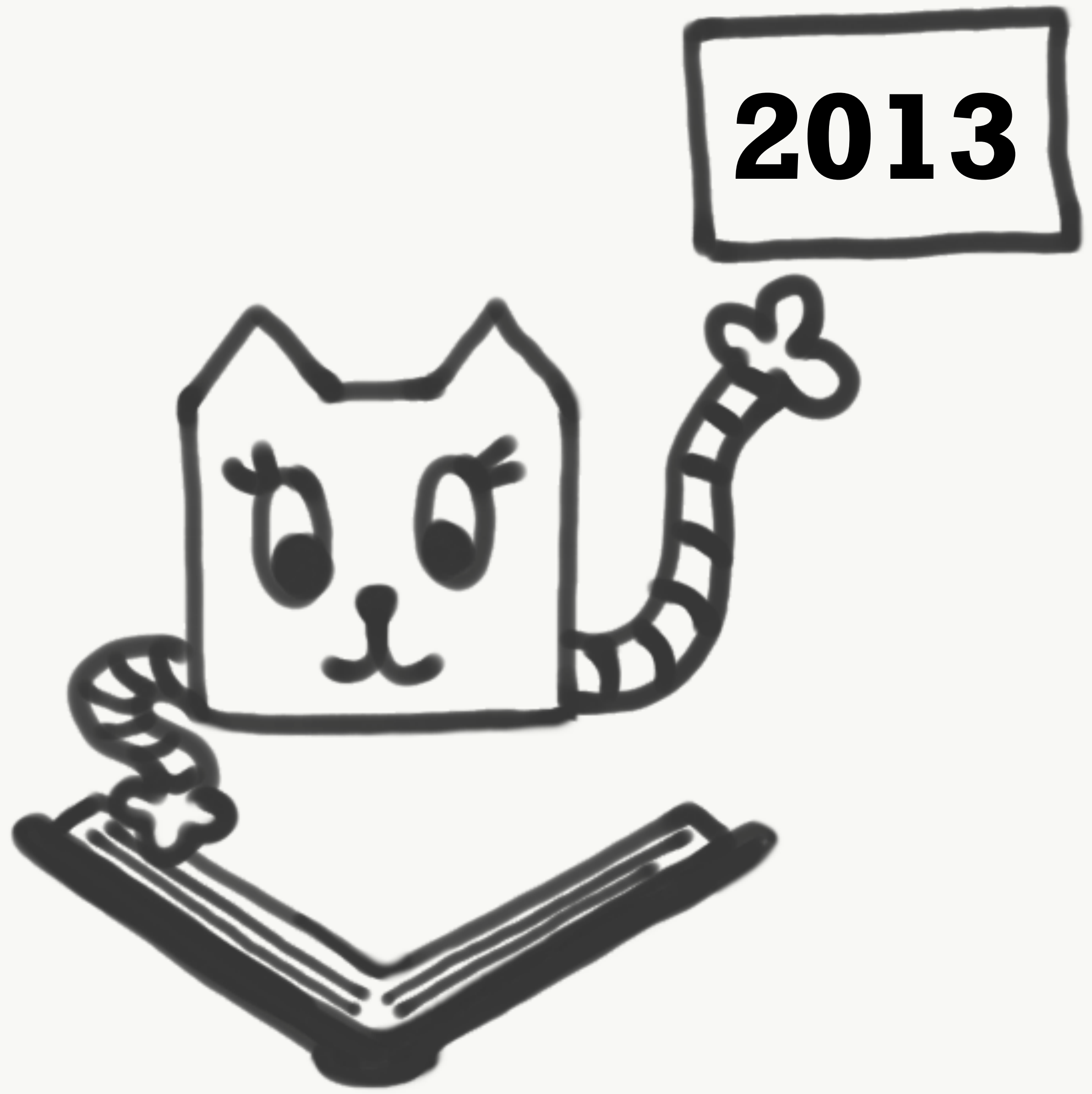
Narrator: There was a problem.



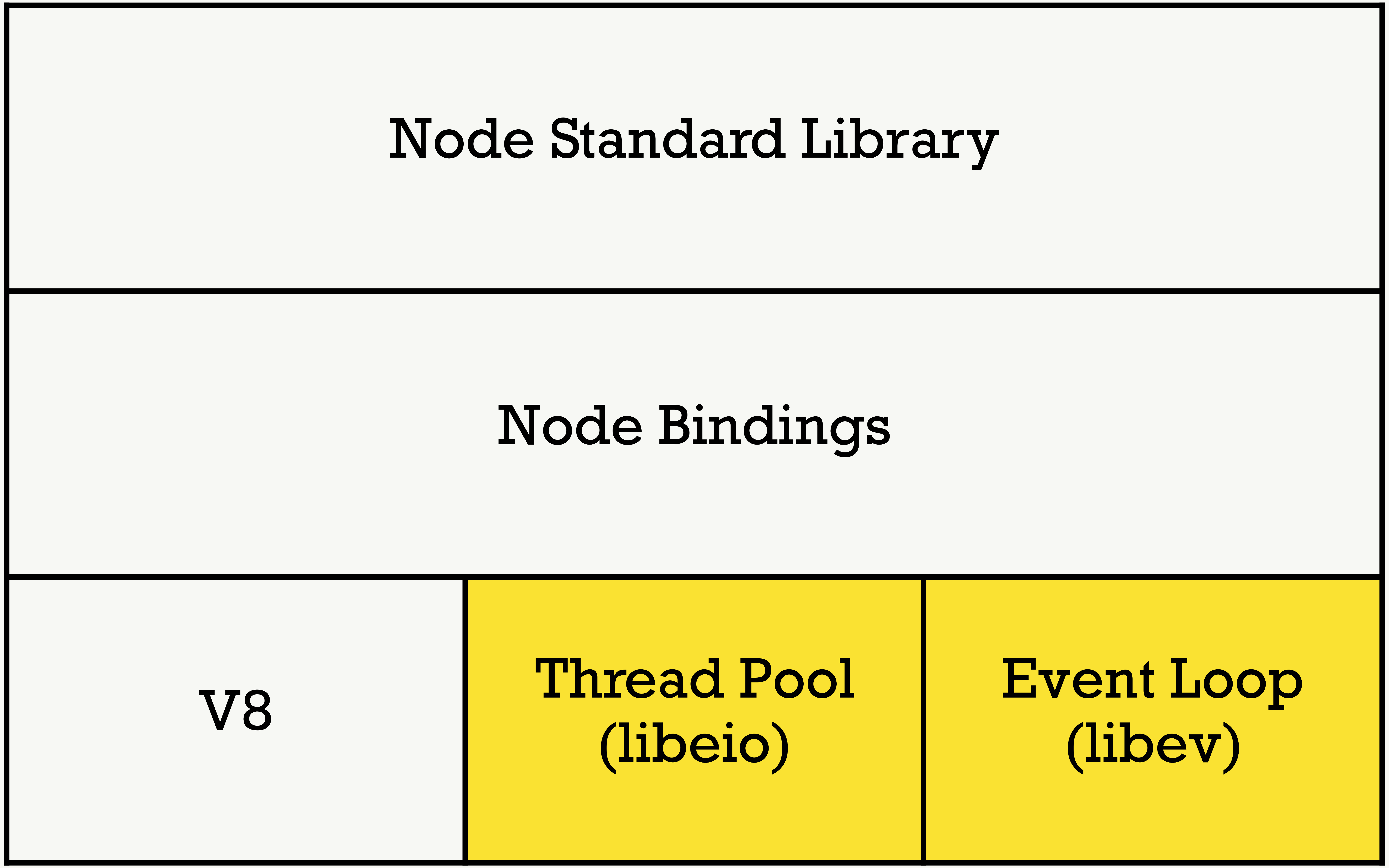
Enter the Dragon



Enter the Unicorn Velociraptor



LXJS 2012 - Bert Belder - libuv



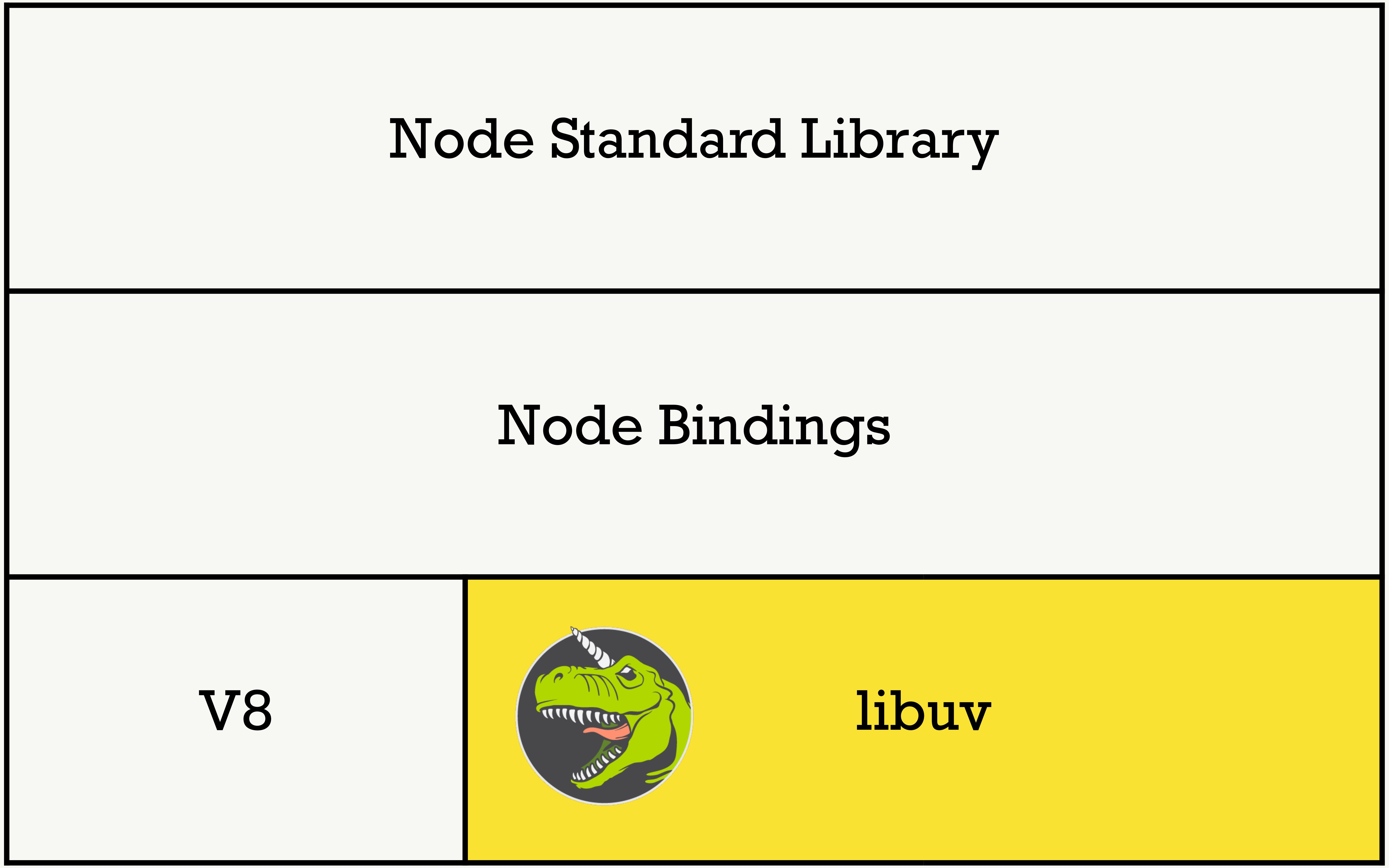
Node Standard Library

Node Bindings

V8

Thread Pool (libeio)

Event Loop (libev)

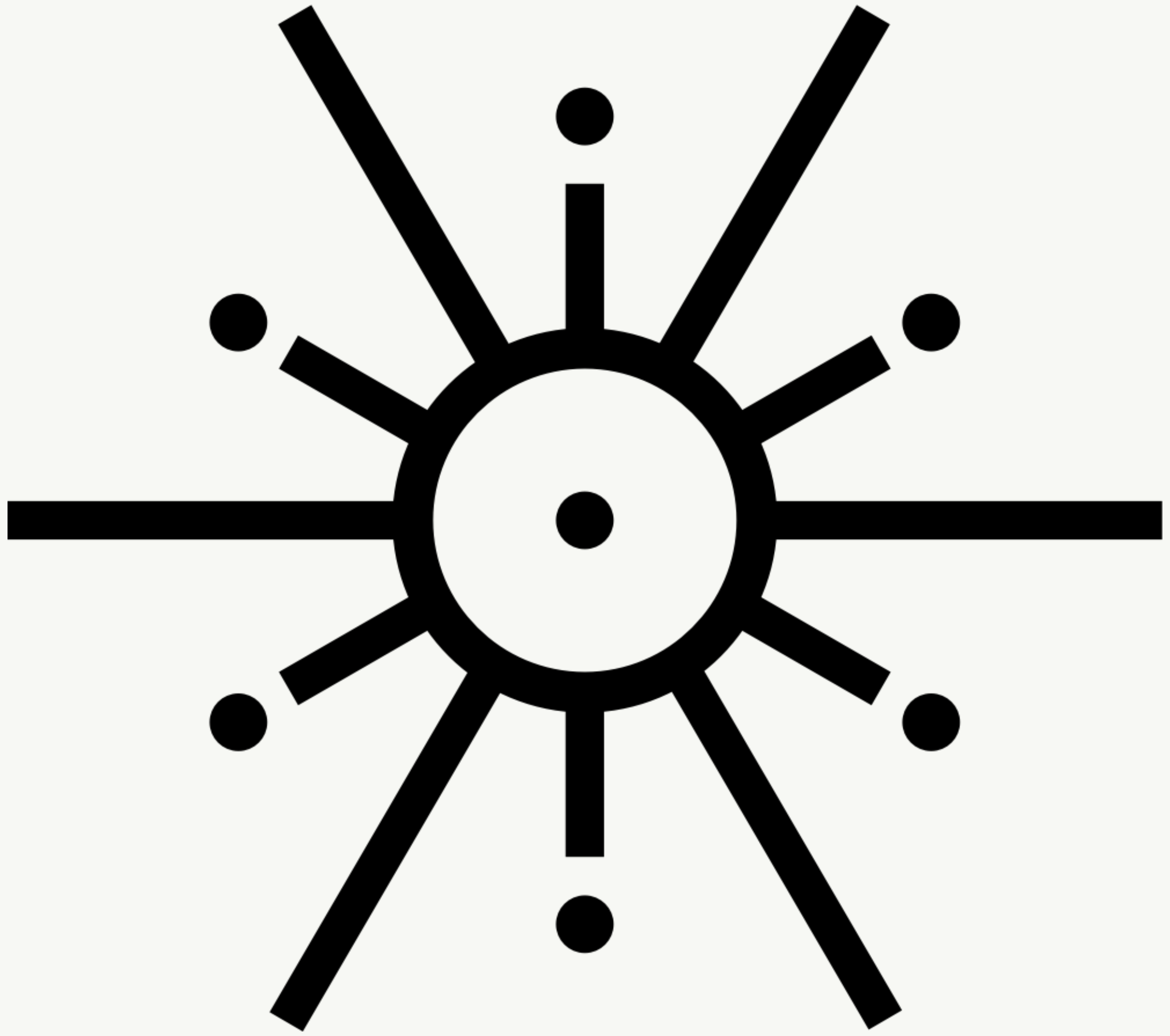


Node Standard Library

Node Bindings

V8

libuv



Section IV

Into the boxing ring

The lifesaver: autocannon

- wrk and wrk2
- “It’s just JavaScript”
- “It just works”
- mcollina is a legend
- TCP?
- Fake TCP?

Introducing
gandiva

Let the benchmarking begin!

Section V

Conclusion

Conclusion 1: tokio is slow

Home
Documentation
Community
Blog
English

Blog Posts

Making the Tokio scheduler 10x faster

Diagnostics with Tracing

Tokio alpha release with async & await

A great 2018, an even better 2019

Announcing the Tokio Doc Push (we need you!)

Experimental async / await support for Tokio

Tokio 0.1.8 with many incremental improvements

New Tokio release, now with filesystem support

New Timer implementation

Announcing the Tokio runtime

Tokio Reform is Shipped and the Road to 0.2

An RFC for a Tokio revamp

Announcing the tokio-io Crate

Announcing Tokio 0.1

Making the Tokio scheduler 10x faster

October 13, 2019

We've been hard at work on the next major revision of Tokio, Rust's asynchronous runtime. Today, a complete rewrite of the scheduler has been submitted as a pull request. The result is huge performance and latency improvements. Some benchmarks saw a 10x speed up! It is always unclear how much these kinds of improvements impact "full stack" use cases, so we've also tested how these scheduler improvements impacted use cases like Hyper and Tonic (spoiler: it's really good).

In preparation for working on the new scheduler, I spent time searching for resources on scheduler implementations. Besides existing implementations, I did not find much. I also found the source of existing implementations difficult to navigate. To remedy this, I tried to keep Tokio's new scheduler implementation as clean as possible. I also am writing this detailed article on implementing the scheduler in hope that others in similar positions find it useful.

The article starts with a high level overview of scheduler design, including work-stealing schedulers. It then gets into the details of specific optimizations made in the new Tokio scheduler.

The optimizations covered are:

- The new `std::future` task system
- Picking a better queue algorithm
- Optimizing for message passing patterns
- Throttle stealing
- Reducing cross thread synchronization
- Reducing allocations
- Reducing atomic reference counting

The major theme is "reduce." After all, there is no code faster than no code!

The article also covers testing the new scheduler. Writing correct, concurrent, lock-free code is really

Conclusion

Conclusion 2: deno is slow

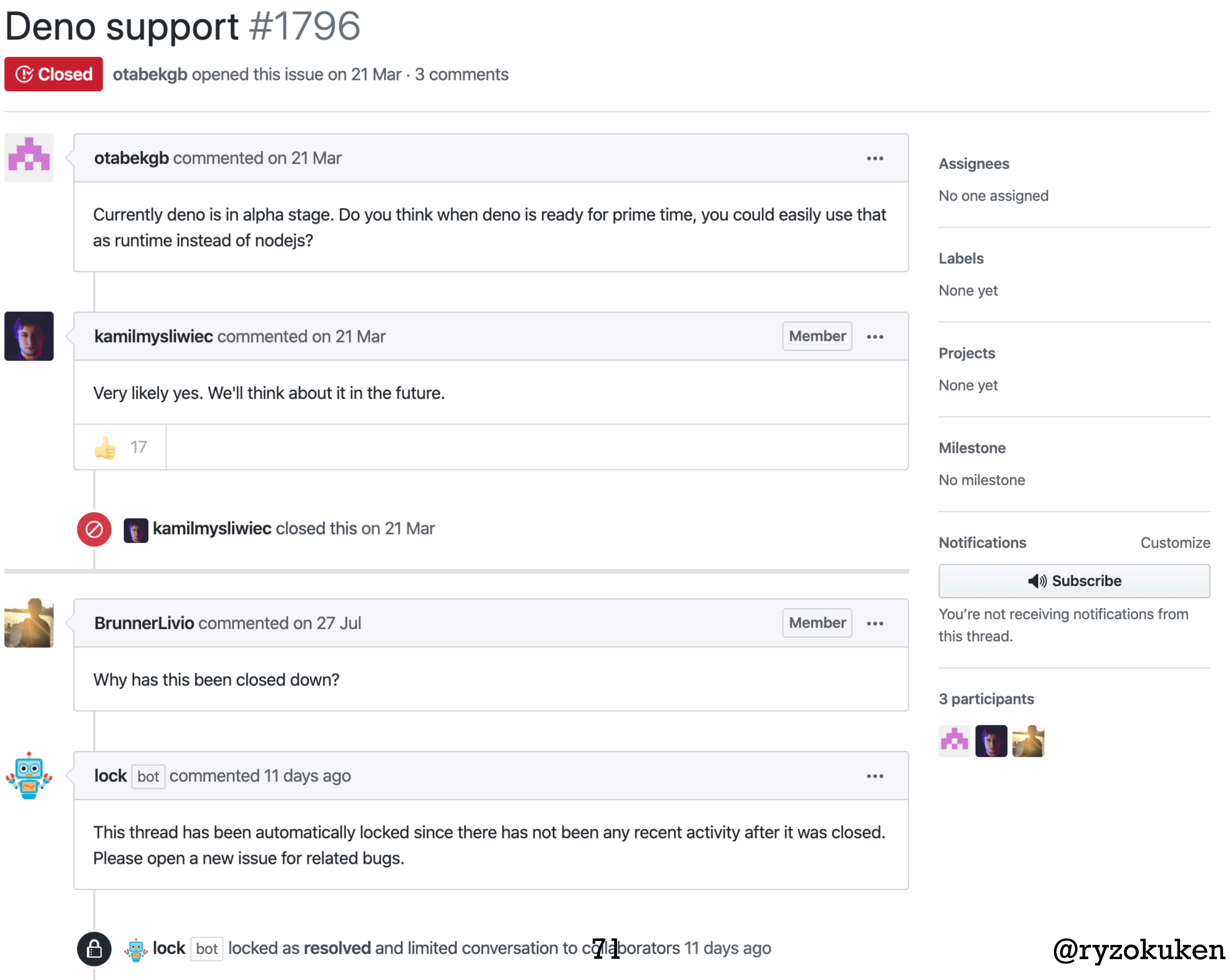
Deno's design is different than Node's in that all native calls are done through zero-copy message passing. This allows for a more uniform bindings, where we have centralised understanding of all calls being made out of the VM.

– *Ryan Dahl*

Ultimately we expect this design to result in better performance, but we're not there yet. Deno's networking is about 50% the speed of Node v13. Follow our progress at https://deno.land/benchmarks

– Ryan Dahl

Conclusion 3: people are still reluctant



Deno support #1796

Closed otabekgb opened this issue on 21 Mar · 3 comments

otabekgb commented on 21 Mar

Currently deno is in alpha stage. Do you think when deno is ready for prime time, you could easily use that as runtime instead of nodejs?

kamilmysliwiec commented on 21 Mar Member

Very likely yes. We'll think about it in the future.

👍 17

kamilmysliwiec closed this on 21 Mar

BrunnerLivio commented on 27 Jul Member

Why has this been closed down?

lock bot commented 11 days ago

This thread has been automatically locked since there has not been any recent activity after it was closed. Please open a new issue for related bugs.

lock bot locked as **resolved** and limited conversation to collaborators 11 days ago

Assignees

No one assigned

Labels

None yet

Projects

None yet

Milestone

No milestone

Notifications Customize

Subscribe

You're not receiving notifications from this thread.

3 participants

But it’s getting there!

Introducing
Deno 1.0

Special Thanks

- Artem Kobzar
- Ryan Dahl
- Olga Kobets
- Kamil Mysliwiec

спасибо!