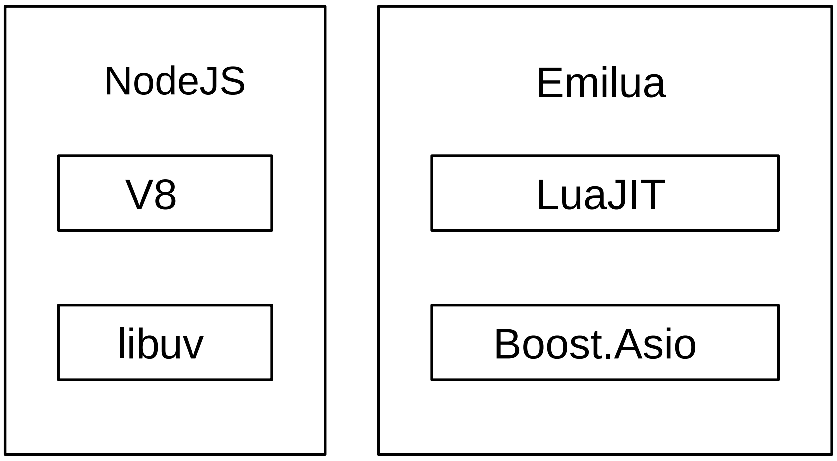
Comparing Emilua to NodeJS



Emilua is an execution engine for Lua. It fills a role similar to NodeJS for Javascript. Apart from being an execution engine, the two have leaps and leaps of differences.

I believe a good first post for this blog would be to clarify the differences between NodeJS and Emilua. NodeJS is very popular and people try to port its API to different languages[1][2][3]. Even non-users of Javascript know NodeJS. By describing Emilua in terms of differences against NodeJS I hope to offer a turbo heads-up. Comparisons also offer me the opportunity to touch on many other topics that I enjoy.

I'll also try to make it brief. If needed be, links to some lengthy posts from other authors that already covered the topic in detail will be included.

One color

The first difference is Emilua's commitment to avoid Bob Nystrom' two colors problem. The problem has been extensively covered elsewhere:

- Christopher Kohlhoff 2015's ignored await-less coroutine proposal to the C++ committee.
- Humble lua users asking for humble APIs.
- Very old manuals on continuation-passing style (for JS).
- Examples why the promises pattern wasn't enough (and the crusade on search for coroutines went on).
- Java's report on project Loom.
- Ron Pressler's talk on high-level APIs for IO concurrency.

NodeJS redundant interfaces

```
tls = require('tls');
fs = require('fs');

options = {
    // sync interfaces
    key: fs.readFileSync('server-key.pem'),
    cert: fs.readFileSync('server-cert.pem')
};

// async interfaces
server = tls.createServer(options, function(socket) {
    // ...
});
```

In a nutshell, every function in Emilua has the ability to suspend the caller function. The runtime will schedule every fiber behind the scenes, and you don't need to wrap the function operator call based on the resources the callee will make use of.

Whereas manuals from some frameworks will instruct you to annotate every call that might block with `await`...

Pseudocode example on `await`

```
n = await read(sock, buf);
await write(sock, buf[:n]);
// ...
```

Emilua won't force you to do the same. IO scheduling is a transparent resource. IO scheduling is not part of the function type signature. You don't need to break the API just because your function implementation just now requires IO activity.

Is it a coroutine… or is it a fiber?

The previous point may have been extensively debated, and it was in this debate that comparisons among the following styles was appropriate:

- Events and polling.
- Callbacks.
- Signals & slots.
- Observers.
- Promises.
- Functional patterns.
- Coroutines.

But Emilua is built around fibers and it is inappropriate to compare fibers and the previous choices. I’ll now compare fibers and coroutines to make this subtle point.

Coroutines may be an old concept that suggest language constructs capable to suspend and resume a function context[4], but fibers come from attempts to provide user-space threads, and, as such, will adhere to threads vocabulary. The presence of threads vocabulary imply a scheduler (i.e. if you call `mutex.lock()`, your code doesn’t need to know which fiber to awake next as this will be taken care of by the scheduler). This difference is noted by authors of fiber libraries, but rarely stated explicitly, so it’s worth to link one of the few texts that make explicit the difference:

> The difference between a fibers facility and just coroutines is that with fibers, you have a scheduler as well.
>
> — Lightweight concurrency in lua, http://wingolog.org/archives/2018/05/16/lightweight-concurrency-in-lua

The difference is important because if you have threads vocabulary, you have not only vocabulary to express concurrent tasks, but also vocabulary to tame this very concurrency with mutexes, condition variables, and the like. None of the previous options of this list — callbacks, promises, coroutines, etc — had such sync primitives.

Ruby 3.0 also has support for fiber-based concurrency. The collective result of what they offer is a form of fiber-based concurrency.

However the single abstraction `Fiber` from their docs is really just a coroutine, not a fiber. The difference between fibers and coroutines is not the lack of a stack. Schedulers are the real defining trait.

The situation got even more confusing as they added “non-blocking fibers” [sic]. If a scheduler for the thread is set, their “non-blocking fibers” [sic] will finally act as fibers and only then the terminology will be correct (except for the “non-blocking” part which is a meaningless adjective).

Promises alone won’t save you from dealing with issues that are inherent to the world of concurrency[5][6][7]. Read Mike Bayer’s case on `await` sharing more properties (and problems) with threads than it’s usually acknowledged for.

Here’s a small example, an implementation for the very simple socket-pair algorithm. First in NodeJS:

```
var net = require('net');

function socket_pair(callback) {
    var result = {}

    var server = net.createServer(function(sock) {
        server.close()
        if (result.err) {
            return
        }
        if (result.sock) {
            callback(null, [ sock, result.sock ])
        } else {
            result.sock = sock
        }
    })

    server.on('error', function(err) {
        if (result.err) {
            return
        }
        result.err = err
        callback(err)
    })

    server.listen(0, '127.0.0.1', function() {
        var sock = new net.Socket()
        sock.connect(server.address().port, '127.0.0.1', function() {
            if (result.err) {
                return
            }
            if (result.sock) {
                callback(null, [ sock, result.sock ])
            } else {
                result.sock = sock
            }
        })
        sock.on('error', function(err) {
            if (result.err) {
                return
            }
            server.close()
            result.err = err
            callback(err)
        })
    });
}
```

Do notice how NodeJS's lack of sync primitives forces you to write your own synchronization (the `result` rendezvous point in the example). Now take a look at how Emilua will make the task much simpler by enabling you to use the `fiber.join()` sync vocabulary:

```
local ip = require 'ip'

function socket_pair()
    local acceptor = ip.tcp.acceptor.new()
    local addr = ip.address.loopback_v4()
    acceptor:open(addr)
    acceptor:bind(addr, 0)
    acceptor:listen()
    local f = spawn(function()
        local sock = ip.tcp.socket.new()
        sock:connect(addr, acceptor.local_port)
        return sock
    end)

    local sock = acceptor:accept()
    acceptor:close()
    return sock, f:join()
end
```

And there is a little something else. Preemptiveness isn't a property exclusive to OS-provided threads. Runtimes from some languages will manage to deliver just this property to fibers as well. Emilua will stay out of preemptiveness (i.e. you're guaranteed to have a safer environment) just like many others. But if you're restricted to cooperative multitasking, you managed to migrate some scheduling decisions from runtime to compile-time. Most frameworks will stop here, but Emilua will go just one mile further.

If you moved some scheduling decisions to compile-time, it makes sense to also move sync primitives to compile-time… or, rather… *scheduling constraints*. Ideally, your code wouldn't compile when these constraints aren't respected. I'm not there yet and there are static analysers waiting to be written, but the vocabulary to encode the user expectation in Lua is here. The vocabulary works like C's `assert()`. One alternative would be to just rely on `mutex`es as usual, but there are these little abusers — like me — of deterministic suspension points that you'll never tame, so I'm adding this little tool anyway to prevent further damage.

Opinionated concurrency style

Bob Nystrom's warning about two colors wasn't enough.

> Choose your async model; we don't mind; we encourage experimentation.
>
> If you don't like callbacks and event emitters, use coroutines and write blocking style code without actually blocking your event loop!
>
> — Luvit homepage, https://luvit.io/

From experience with Boost.Asio, I noticed that you can't just defer the choice to the user and get rid of making one. What happened to Boost.Asio is that you cannot appropriately support any one model.

- Limitations from one model infect other models (e.g. orientation towards IO objects and not threads). This point by itself could give a lengthy article, but nowadays I'm less concerned with convincing people and more concerned with respecting my own precious time, so you'll only have my word here.
- You cannot rely on the strengths that are exclusive to one model (e.g. disable interruption at

critical blocks). It may seem redundant with the previous point because it's just another face of the same coin.

- You just created a new model. *Your "unopinionated" model is a meta-model* that forces every library provider to write convoluted code. Again, another lengthy article that will not receive a share of my time. If you're curious, try looking for libraries built around Boost.Asio that work with the completion token protocol.

Emilua cares about serving one concurrency style and serving it well: fibers.

> >Boost.Fiber
> >(https://www.boost.org/doc/libs/1_67_0/libs/fiber/doc/html/index.html)
> >is another way. This also supports futures, although not currently
> >then-able ones.
>
> Boost.Fiber doesn't need fibers::future::then - just suspend the fiber. If you need more concurrency than that, launch another fiber. then() is redundant with coroutine and fiber concurrency.
>
> — Nat Goodspeed, Boost mailing list, 2018

Active style

A friend of mine teached me this principle early on that affected all my future projects: *design your abstractions where the user is an active party on scheduling decisions*. I've been calling it the active style and people usually don't get it what it is about, but consequences of this design are better understood (e.g. dealing with back-pressure).

Ryan Dahl's successor for NodeJS also got this point covered:

> Node's counterpart to promises was the EventEmitter, which important APIs are based around, namely sockets and HTTP. Setting aside the ergonomic benefits of async/await, the EventEmitter pattern has an issue with back-pressure. Take a TCP socket, for example. The socket would emit "data" events when it received incoming packets. These "data" callbacks would be emitted in an unconstrained manner, flooding the process with events. Because Node continues to receive new data events, the underlying TCP socket does not have proper back-pressure, the remote sender has no idea the server is overloaded and continues to send data. To mitigate this problem, a pause() method was added. This could solve the problem, but it required extra code; and since the flooding issue only presents itself when the process is very busy, many Node programs can be flooded with data. The result is a system with bad tail latency.

> — Deno 1.0, https://deno.land/v1

Emilua is more explicit

Emilua is also just more explicit. Many years ago, NodeJS actually attracted me. Its API (at the time) was better than the HTTP server libs that I'd design. It helped to push me forward. Unfortunately it feels like it stopped in time and didn't preserve this "pusher" feeling.

What attracted me at the time was its lower-level approach to web protocols. I always had trouble understanding layers and layers of web frameworks from Python to Java worlds.

Emilua is therefore also low-level in a few regards — even more than NodeJS. Emilua doesn't have implicit — and conceptually unbounded — write buffers. If you have multiple fibers writing to the same network socket, you better sync them somehow or the receiver will see corrupted streams[8] (just use a mutex to protect the write side and you're done).

Another example would be creating an acceptor socket where you're explicitly required to open + reuse-address + bind + listen to achieve effects that you have by default with NodeJS.

> Although this is the general principle, external native plugins may choose to implement different policies.

You may also see Emilua as a safer training camp before you delve into even lower-level C APIs.

Structured concurrency

There is a growing interest in structured concurrency. Much of the arguments either just lack the required rhetoric or are plain cargo cult programming.

Martin Sústrik has an actual good article on the topic. You just have to be careful to extract the guiding principle behind the specifics. Were it only for the specifics, I could easily dismiss these concerns as not being valid for my use case because the GC will take care of ensuring that values from parent scopes won't be destroyed while they're required.

However, one fact remains: the user can have legitimate reasons to have truly detached "unstructured" tasks. Emilua won't force these users to change their code. Also, structure is not only achieved through `fiber.join()`. The user might very well use a condition variable to add structure. It's dumb to force every structure pattern undergo the same vocabulary.

By default, NodeJS' continuation-passing style discards any structure that the runtime could detect. NodeJS however can detect unhandled promises. If a promise is rejected and no error handler is attached to the promise within a turn of the event loop, the `unhandledRejection process`' event is emitted:

```
process.on('unhandledRejection', function(e, p) {
    console.error(e.stack || e);
});
```

Emilua behaviour is similar. You lose structure when you detach a fiber or a joinable fiber handle is GC'ed. For the case of lost structure, no action is taken unless the fiber errors. If an exception escapes the detached fiber, the uncaught-hook is called. The default hook will just print the stack trace to `stderr`.

```lua
spawn(function()
    -- will print stacktrace to `stderr`
    error('foobar')
end):detach()

-- code here keeps running
```

And the rest of the application can keep running thanks to invariants being preserved by the cleanup handlers (they are similar to Python's `with` blocks), but if a cleanup handler on a detached fiber fails, then all bets are off and the VM is shut.

Threads

NodeJS's solution to threading is cheap. Create another process and make the two communicate through messaging.

It looks cheap. It is cheap. But it makes sense. These scripting languages (JS, lua, Python, …) just don't play well with threading.

However this solution poses other questions that many frameworks stepped on and go unanswered on NodeJS's case. If you create too many threads to speed-up task A, you might end up starving task B. You also face this dilemma in C every time you write a library and it is tempted — an uncommon occurrence FWIW — to spawn a few threads for a small sub-task (e.g. if you want to do a parallel sort inside your function). The threading layout is a property that belongs to the application, not the library. This is one of the issues solved by the executor design in Boost.Asio. Similar efforts exist in different frameworks.

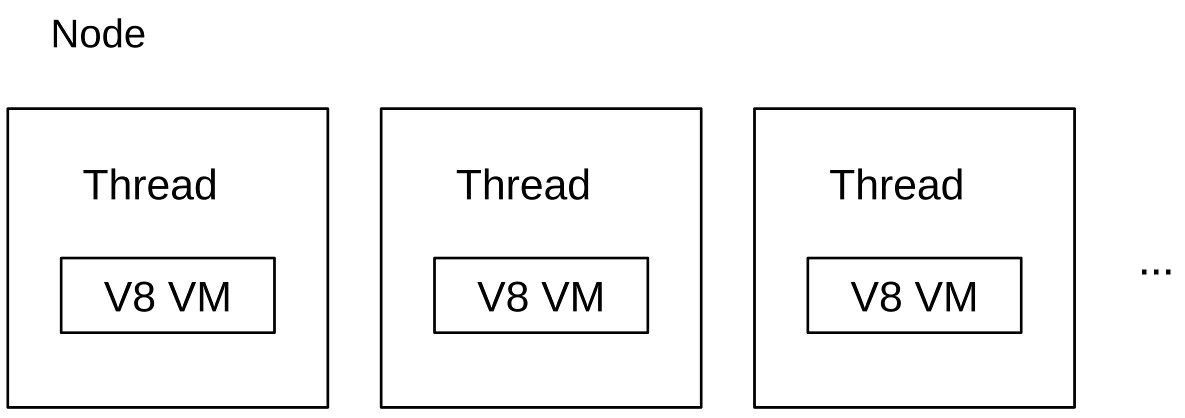


Figure 1. Threading model for NodeJS

As for Emilua, you might spawn a VM and it won't create an extra thread. The child VM will share its parent's thread. And you might freely choose which group of VMs use which thread pools. The threading layout is under your control. I want to keep it brief, so I won't expand more on this point.

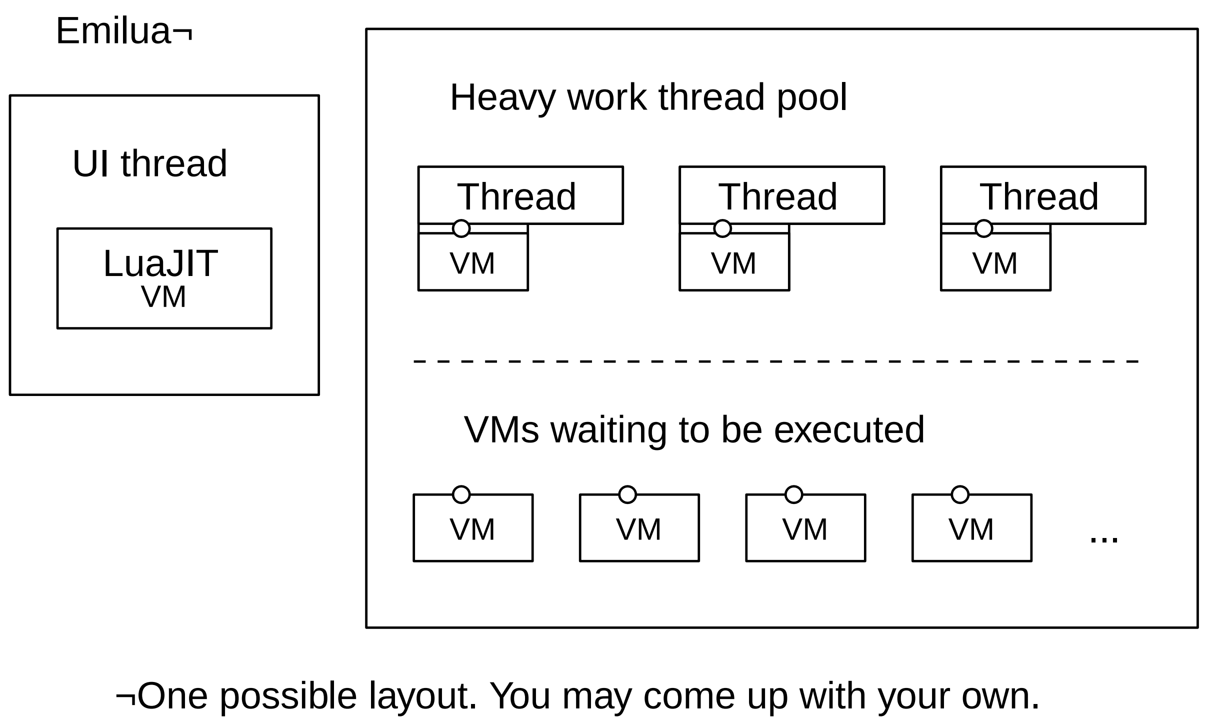


Figure 2. Threading model for Emilua

It may be more interesting for you to know that the same vocabulary to accomplish just the above is also the vocabulary that enables you to use a full actor system. Choose how many extra threads your application should use (if any) and let each actor/VM be scheduled transparently behind the scenes. You should check the tutorial on the documentation to have a brighter picture.

The same actor system might enable us in the future to run a few isolated actors inside Linux namespaces or even qemu guests, but that's a far away milestone.

For now, you can already make use of a system similar to Akka and Quasar. Just one caveat: Akka focuses on distributed systems while Emilua only sees actors as a pattern that retains good scalability to dispatch many work units to a shared thread pool. I'm sure the API will break and evolve as I mature my vision on the actor model.

HTTP

Emilua's HTTP abstractions were designed as a gateway interface so it's easier to develop different backends (but a new backend is always difficult by itself and a time-consuming task nevertheless, so don't get your hopes high).

⚠ As of the 0.1 release, the HTTP module is an experimental feature and must be explicitly enabled at build time. Please report any bugs you find and they will be fixed. One bug that I'm aware of is the lack of limits to protect against DoS attacks.

> That means a remote endpoint might force your application to indefinitely allocate memory to store HTTP headers. This issue will be solved in the next releases.

NodeJS has a hidden state machine that gets into action when you call `writeHead()` and family. Emilua's state machine is explicit. You can query its `read_state` and `write_state`. State transitions are fully documented and it is easy to understand which pieces of the payload will be touched by each `write_*()` method. This is also part of the gateway orientation effort.

```
function handler(req, res) {
    var data = [];
    req.on('data', function(chunk) {
        data.push(chunk);
    });
    req.on('end', function() {
        // handle the request
    });
}
```

Messages are entities separate from sockets. Given the gateway-oriented design, a socket's concrete implementation might not be an embedded HTTP/1.1 server. Only the socket type needs to be polymorphic and it doesn't make sense to turn the message polymorphic as well. More reasons exist: if we desire to offer a socket with some alternative HTTP pipelining support, a socket-message separation makes it clear what's going on. In other words, this design might benefit alternative implementations for our gateway-oriented design.

```
function handler(sock, req, res)
    while sock.read_state ~= 'finished' do
        sock:read_some(req);
    end
    // handle the request
end
```

Backend-specific details are erased from the messages. You'll never query the HTTP version out of a message because it doesn't make sense to other backends. You query capabilities instead and the whole model is oriented around capabilities from HTTP/1.0 and HTTP/1.1. If you're worried about too many layers of engineering, don't be alarmed! The actual API is very small.

```
local ip = require 'ip'
local http = require 'http'
local sleep_for = require 'sleep_for'

local acceptor = ip.tcp.acceptor.new()
acceptor:open('v4')
acceptor:set_option('reuse_address', true)
if not pcall(function() acceptor:bind(ip.address.loopback_v4(), 8080) end) then
    acceptor:bind(ip.address.loopback_v4(), 0)
end
```

```lua
print('Listening on ' .. tostring(acceptor.local_address) .. ':' ..
      acceptor.local_port)
acceptor:listen()

while true do
    local sock = http.socket.new(acceptor:accept())
    spawn(function()
        local req = http.request.new()
        local res = http.response.new()

        res.status = 200
        res.reason = 'OK'

        while true do
            sock:read_request(req)
            sock:write_response_continue()

            print(req.method .. ' ' .. req.target)

            while sock.read_state ~= 'finished' do
                req.body = nil --< discard unused data
                sock:read_some(req)
            end

            if sock.is_write_response_native_stream then
                sock:write_response_metadata(res)

                sleep_for(1000)
                res.body = '3...\n'
                sock:write(res)

                sleep_for(1000)
                res.body = '2...\n'
                sock:write(res)

                sleep_for(1000)
                res.body = '1...\n'
                sock:write(res)

                sleep_for(1000)
                res.body = 'Hello World\n'
                sock:write(res)

                sock:write_end_of_message()
            else
                res.body = 'Hello World\n'
                sock:write_response(res)
            end
        end
    end):detach()
```

```
end
```

Emilua’s HTTP socket is symmetrical. A socket only becomes a server/client socket after the first action is taken. Should you desire to run HTTP sockets on top of obscure rendezvous P2P connections, you’re the one possessing control about what should happen.

There are plenty of small details that went into the design that I cannot possibly cover here. As an example, try to find out how to ignore an HTTP upgrade request w/o closing the connection on NodeJS (i.e. how to treat it as a common request that should receive a common response with no upgrade-protocol action).

Final word

I hope this comparison served the purpose of quickly explaining what Emilua is. I’d also like to make it clear that this project evolved from my needs to have a playground where I can experiment with ideas of my interest and I’d have created it even if a really similar project already existed. In fact, I’m more excited to experiment with unpopular D-Bus scripting using Emilua than running web servers.

[1] https://luvit.io/

[2] https://github.com/ignacio/LuaNode

[3] https://github.com/lipp/nodish

[4] Conway, Melvin E.. "Design of a Separable Transition-Diagram Compiler". Commun. ACM, Volume 6 Issue 7, July 1963, Article No. 7.

[5] https://github.com/taskcluster/docker-worker/pull/332

[6] https://github.com/esamattis/node-promisepipe/pull/9

[7] https://github.com/esamattis/node-promisepipe/pull/8

[8] Look for ASIO composed operations if you’re curious about the internals of this event.