Who needs fibers?

Emilua has this opinionated style of going full fibered to approach single-VM concurrency. That's an unusual approach in Lua as far as I know and it begs the question: who needs fibers?

So, to illustrate this need, I'll talk about the experience I had in my previous job. Usually I never talk about work outside of my coworkers circle and the reason is I'm just too worried to not reveal any technological or strategic business secrets. Even thou I moved on and now I'm enjoying a well deserved little vacation, I keep my habits intact. Therefore I'll only describe what can be considered standard industry practice or very obvious consequences that anyone implementing the same protocols would infer right away (yeah, I'll hide many optimizations and secret sauces that I developed for my boss). That will be one example where fibers matter, but other examples exist (e.g. some patterns of rate-limiting requests to process-global connections).

In my previous job, I was hired (among other things) to develop a WebSocket gateway that would accept connections and dispatch requests to the correct service in our internal network (e.g. authentication provider, wallet management, market data subscription, order processor, etc).

The usual approach to web development is to serve HTTP requests and forward all synchronization and state management complexities to a database. Therefore you just follow the HTTP *stateless* road and see no state. If that is the case, you'll never understand why fibers are important for you're essentially only implementing a dumb ping-pong protocol. To scale the web application, you just spawn more processes and hope the database will hold out well.

These traditional web applications follow the KISS principle so closely that if you add more concurrency to the same process/thread itself, they already start to fall apart. The applications themselves may not need to manage per-connection state, but the underlying database drivers will do need to handle concurrent accesses by several connection handlers. This is a low-level detail rarely exposed to higher layers so you probably aren't the one who needs to worry about it. Mike Bayer wrote about his experiences with Python database drivers and async madness. If anything, Mike implicitly argues *against* the use of fibers (at least to the developers writing the high-level business logic).

Well… if you don't need fibers vocabulary, what will fibers win over you? **If you don't need to use fibers vocabulary, the fibers will work just as plain coroutines** (that Lua already provides) and you'll have linear clear code. You lose nothing.

However there are people who do need fibers vocabulary. They may just be the ones implementing the database drivers that your async web application will make use of and they'll appreciate the presence of such vocabulary to tame the concurrency beast.

Back to my work, I too needed primitives to tame concurrency issues. Let's explain the gateway task again (this time for real). The gateway wouldn't manage HTTP requests. The gateway would manage WebSocket connections and that implies stateful connections. The user logs in and now any request going through that connection will have the user credentials already applied. So far, there isn't any need for sync primitives and even coroutines will do (that's how I started the project btw, using plain coroutines). However we already have *state*.

Betting on market goods

If you’re already familiar with stock exchanges, skip this section. If you don’t know anything on this business, then follow on and you should have a small introduction that is good enough for the rest of this text.

Putting in simple terms, a stock exchange is a place where you can place buy or sell orders on some goods. These houses grew a lot in complexity by allowing very elaborate contracts (e.g. future options) to be made and still be negotiated in volumes, but this complexity doesn’t really concern us — humble non-traders programmers.

A market data is a constantly-changing price-ordered table that you can query and subscribe to. Essentially it has three pieces of information at least — order id (this one is complex as sometimes you only has access to the book by price or by price band, but it is included here anyway as a simplification of the underlying topic), quantity, and price. The following two tables illustrate an order book (the units don’t matter):

Table 1. Buy orders

ID	Quantity	Price
1243	611	185
334	55	184
1500	1084	183
1501	1951	182

Table 2. Sell orders

ID	Quantity	Price
1	58	186
5	69	187
4	21	188
2	1	189

When the prices advertised on the top rows of the two tables match, an order execution (maybe partially) happens and at least one row disappears. You’re allowed to place orders on this book after you deposit money/goods on the exchange wallets. You can also change or cancel orders that you placed previously according to the house rules. These services are more accessible nowadays due to crypto-markets. You can negotiate on any instrument pair offered by the exchange (e.g. bitcoin against dollars).

When you subscribe to some market data, you process an influx of events (e.g. change row X, insert row Y, delete row Z) to derive its current view.

One of the gateway’s responsibilities was to manage subscriptions to market data. In the traditional scenario (cf. the FIX protocol), you have an internal network where one node per instrument will broadcast two signals — a large snapshot every few seconds, and a small incremental update on every market event. If you're participating in this internal network, you combine these two signals and create a current view of the market data. The incremental applies against the previous view of the market data and produces a new view. If you lose one incremental event (the events would be delivered through UDP-like protocols), then you await until the next snapshot (a full serialized view) and start again.

I'm not a trader. I was only hired for my C++ skills. Any question I had on the business model or traditional industry practice would be promptly answered by other talented guys in the company.

This architecture doesn't really work on web-facing services where broadcast messages aren't really readily available, but that's how the internal pipeline was designed. The WebSocket gateway would subscribe to these two network signals and synthesize new internal in-process signals. There is a single snapshot-incremental channel pair per-instrument consumer to the whole process that will broadcast the derived signal to N separate in-process subscribers. In-process subscribers (i.e. the WebSocket connection handlers) wouldn't be able to tell whether an update was derived from the diff between two snapshots or directly mirrored from a real network status update.

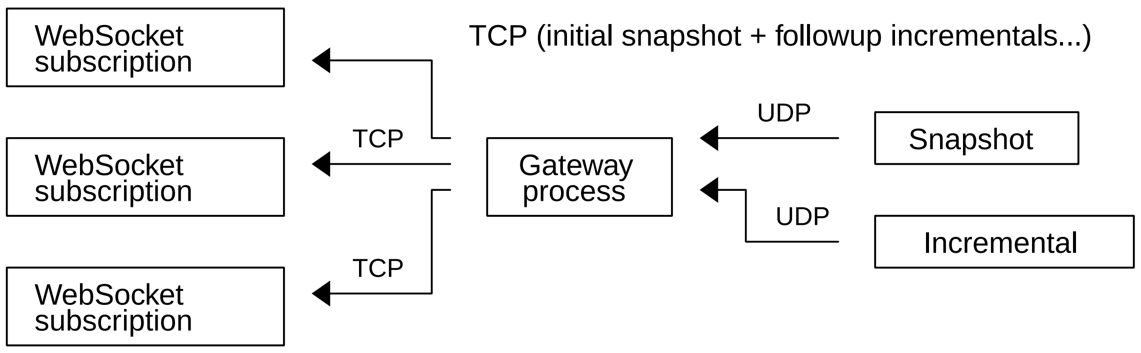

Figure 1. The gateway subscription model

When the WebSocket client requests a market data subscription, he'll receive an initial snapshot containing the current view of the market data and then subsequent incremental updates that must be applied against the last view to produce the most up-to-date market data view. The client has the easy task, no race to deal with, a reliable connection, and a very simple very serial algorithm to apply. However the server… not so lucky. The gateway hides all the complexity from the client. The client will never see a "lost incremental" event.

This architecture allows for a lot of consumers in the internal network (e.g. several gateways as well as co-location paying customers that want minimum latency) while it puts little pressure on the market data providers.

The snapshot-incremental processing code (i.e. the code that only deals with the internal network side to produce in-process events) was a lot of headache already. What if you receive the incremental sooner than the snapshot it applies to (they come from two distinct channels after all, and there is no implicit synchronization between different network channels)? That's just one among several racy events that could happen. When the application starts, it already begins to handle WebSocket connections and these connections will make requests before you synchronized against low-volume slow-pace markets (i.e. you may receive a subscription request from the WebSocket channel for a market that you haven't received the first snapshot yet). That's not an application model that fits well in the database paradigm. That's a truly valid use case for sync primitives (and fiber vocabulary).

Rough skel on how to deal with the first-sync problem

```
function handle_sub_req(self, inst)
    if not snapshots[inst] then
        error('Invalid instrument')
    end

    local msg
    scope(function()
        snapshots[inst].mutex:lock() ①
        scope_cleanup_push(function() snapshots[inst].mutex:unlock() end)

        -- Wait on the cond until the first snapshot for that instrument
        -- is received.
        while not snapshots[inst]:is_valid() do
            snapshots[inst].first_sync:wait(snapshots[inst].mutex) ②
        end

        msg = snapshots[inst]:serialize()
        self:subscribe(inst)
    end)
    self:send_msg(msg)
end
```

① A mutex (fiber vocabulary).

② A condition variable (fiber vocabulary).

However the headache doesn't stop there. One of the order types is the cancel-on-disconnect order. For that order, if the client loses its connection, you should send a cancel-order request to the matching engine. That means you now have a per-process per-connection (non-persistent) state that also doesn't fit in the database model. Say, what happens if the process receives a SIGINT? Now it must cascade-cancel every connection handler (the tasks) and you need some cancellation vocabulary to do the right thing (cancel all the cancel-on-disconnect orders). This is vocabulary also offered by fibers not offered by plain coroutines.

There are several approaches to handle cancel-on-disconnect orders. The amount of responsibility handed to the gateway varies with approach. Nonetheless, the example is still a valid approach and keeps being useful to illustrate why cancellation vocabulary matters.

WebSocket (stateful) channels present plenty of other opportunities to use sync primitives. Remember we synthesized in-process signals out of network events? Well, that means shared event queues. If a client subscribe to a new instrument, there is a new event queue. If the client unsubscribe from an instrument, we must remove one shared event queue (and we better do it right). Guess what happens to the event consumer if the underlying queue being served is destroyed when the fiber reading messages receives an unsubscription request? To be fair, this problem is more severe in C++ where we don't have the luxury of a GC, but Lua would face problems as well if I was to implement all the scheduling optimization tricks that I've done for the C++ project. There are so many possibilities of races here (any point where async IO happens is a context-switch to other tasks that might be touching on the same resources, or could just as well be a point where the task receives a cancellation request to free all the resources)… not only races, but also error-prone scheduling policies (e.g. you must watch for at least starvation scenarios), or even DoS protections to apply (e.g. watch for the queue size in slow consumers). Worse, the high-contention scenario might decrease performance so much that the application is simply unable to really serve any of its functions.

A gateway that allows not only multiple subscriptions multiplexed through the same connection but also unsubscriptions is hard enough to encourage many stock exchanges to just not implement this feature at all. In their houses, if you want to subscribe to a new instrument, you do so in a new connection. And if you want to unsubscribe, you just drop the connection. Do notice that you *still* need some vocabulary here (how would you implement cancel-on-disconnect orders for instance?).

The nightmare is real, but so are the opportunities to optimize lots of these interactions and still perform a correct job. Like I warned previously, I won't share any of the secret sauces that I developed for my boss. You only need to give the problem to someone willing and results will show. I was lucky enough to have worked with particular talents. The market data guy was reliable enough that I could just worry about the concurrency issues and forward all market data algorithms on his back.

Back to this post's subject, shared resources among concurrent tasks means a need for sync primitives (e.g. mutexes, and condition variables) that are non-existent if you rely on sole coroutines. That's how you keep state consistent in the face of multiple tasks trying to access the same resources. This is the answer for "who needs fibers". Some serious applications can't really afford the "stop the world" model from solely database-oriented sync techniques while a slow request is being fulfilled. Some applications must deal with events whose only component possessing the required knowledge to handle the event *is* the web application (e.g. disconnect events). *Some*, but not all, applications need sync primitives and that's where fibers vocabulary kicks in (or other equally valid sync-aware choices).

Do notice that while the gateway deals with a lot of concurrency issues, many of the services in the internal network could work without any consideration to multitasking at all. Therefore, the nightmare is not a combinatorial explosion.

Do notice as well that this application *also* makes use of databases to store a lot of stuff. Sync primitives are not only for those that can do without traditional databases. My team certainly wasn't among the ones that who could do without databases.

Another approach to do away with the nightmare would be to use the actor model instead fibers. The actor model also works, but it's not just simpler. The actor model is also less performant (there are memory copies everywhere under-the-hood). Although a poorly optimized fiber-based application can perform worse than an actor-based application, this is no reason to believe that actors are a better approach in every front. If you explicitly control the message queues (as in the fiber model), you gain the opportunity to optimize these queues by applying rules that lie outside of the concurrency model (i.e. rules from the business logic). I'll limit myself to share that I did a mix of fibers and actor-based concurrency within the same application. That's also something that you're allowed to do (and an approach also supported by Emilua).

Other considerations

- You'll not have a course on the usage of traditional sync primitives going through Emilua docs. Read the POSIX threads manpages instead. Emilua implements traditional sync primitives that were extensively documented for decades already. There is no need to write yet-another-tutorial on how to use a mutex, and so on. I might even reject PRs that try to add these tutorials to the docs as this would be even more maintenance headache to deal with.
- Async IO is the primary driving force to introduce fibers instead plain coroutines in Emilua. Async IO equals to events occurring concurrently to the application process (e.g. mutating buffers partially written while the application could try to access them) and any time you want to apply concurrent IO actions to the same IO object, you'll need sync primitives (hence fibers). The single threaded nature from the Lua VM is not enough to save you from IO concurrency issues.
- Ironically it's easier to stumble upon these problems in client applications (e.g. torrent clients), so I doubt most server guys will ever need to use fibers directly (they have the database to handle all the complexity after all). **Again**, Emilua will not make your application more complex. If you don't need the fibers vocabulary, just don't use it.
- It's easy to go from "fibers to coroutines" (i.e. just ignore fibers and act as if they are plain coroutines), but the opposite is not true. If you're half-way through your project and suddenly stumble on the need to use fibers, you're screwed. You'll have to develop half-baked probably racy sync primitives to work around. That's why I believe an execution engine for Lua should have fibers vocabulary from the get-go (as in Emilua). These applications aren't easy and any help they can get is welcomed. As a comparison, let me just inform you that it was far easier to write this Lua execution engine from the ground-up during my spare time than the WebSocket gateway described in this blog post.
- The Lua VM is still thread-unsafe, so you need some extra vocabulary to exploit possible parallelism (fibers assume shared memory and cannot be used to represent work split among several Lua VMs as the VMs themselves won't be able to share any memory against one another). That's where Emilua allows you to spawn more VMs using a heavily actor-inspired API.
- All concurrency fiber vocabulary available in Emilua follows the cooperative multitasking model. If you know what you're doing, that means you can apply clever scheduling tricks. And if you don't know what you're doing, there's a small chance that your application will starve other tasks due to some infinite loop lying around that doesn't have any IO/timer/sync/yield call (Go also employed cooperative multitasking in the first versions, it's not really a big deal). Emilua will **not** — as Go did — migrate to preemptive models. Its community should grow more intelligent over time, not more stupid (as they saying goes… make it idiot-proof, and someone

will breed a better idiot). For the valid (and rare) case where your application need to strongly guarantee maximum time-slices for each fiber (i.e. the land where only preemptive multitasking has an answer), Emilua is not the runtime you're looking for (although competing API-compatible runtimes could be developed).

- In case you're wondering why our cousin NodeJS doesn't have concurrency issues… well, it has. Check a few of my friend Wander's PRs[1][2][3]. Unfortunately to him, NodeJS has no solid sync vocabulary, so you replace documented problem-solution pairs discovered through the decades by the same problems but requiring the solutions to be written in a new-and-not-so-solid style. Emilua is the one who possesses the tried-and-true tools here.

Wrapping up, I presented an example where sync vocabulary matters (and fibers are one of the choices here). The example happens to be the last paid job I was involved with. Other examples exist in the wild.

A note on correctness

Performance must not be placed before correctness. If your code is incorrect, it's possible to implement a program that delivers the result in exactly 0 milliseconds, and that program is just equally useless. Emilua is young, and so far I only focused on correctness. The fiber API won't break even if you feed it the following code:

```lua
local sleep_for = require 'sleep_for'

local coro = coroutine.create(function()
    sleep_for(1000)
    print('hello')
    coroutine.yield()
    coroutine.yield()
end)

spawn(function()
    coroutine.resume(coro)
end):detach()
coroutine.resume(coro)
```

The execution engine will not crash. The only thing that will happen is a normal Lua error being raised to the appropriate fiber call-stack.

Try to do the same with other fiber implementations to see if they also hold out. Performance can always be improved later, but an incorrect application is always an incorrect application and its advertised performance is useless.

I invested some serious effort into the design and implementation of cleanup handlers to preserve program invariants. Emilua is designed for robustness first, correctness above performance (honestly if I was worried about performance, I'd be coding in C, not Lua).

If you find any crash on the execution engine, please report it and I'll take the issue seriously (feature requests on the other hand may not receive my attention in a

> timely manner).

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

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

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