Lua as a business logic language in high load application

Ilya Martynov
ilya@iponweb.net
CTO at IPONWEB
Company background

- Ad industry
- Custom development
- Technical platform with multiple components
Custom web server

• One of the components of our technology stack
• Written in C++
• Uses Lua as an embedded scripting language
Adserving requirements

- High load
- Complex logic about what ads to show and how to track them
- Hardware is not always cheaper than developer's time
What is high load
What is high load
How do we come to use Lua?

- First version of adserver is pure C++
- Runs fast
- Development is slow
How do we come to use Lua?

- Developers who can write low level code and can write business logic code are rare animals.
- Operational costs: there is a better balance between cost to run and cost to develop.
How do we come to use Lua?

- Separation of church and state
- C++ for low level and performance critical bits
- Scripting language for business logic
So what do we use as a scripting language?
Why Lua?

• So that I can attend Lua workshop as a speaker!
Why Lua?
Game developers like Lua for good reasons

- Fastest scripting language
- Easiest to embed scripting language
- Simple but expressive
- Can be sandboxed
Why NOT Lua

- Poor libraries (compared to competition)
- But this is NOT as big deal for development in special domain (advertising)
Architecture
Multithreaded C++ server

- Worker thread per CPU core
- One Lua interpreter state per worker
Multithreaded C++ server

- Multiple coroutines in each Lua interpreter state
- New HTTP request → new coroutine in idle Lua interpreter state
Sandbox environment

- Only safe subset of Lua standard library available
- Special high level IO APIs to access external world
- Only allow what is really required
Why coroutines

- Networking IO APIs mean Lua code may wait for responses
- Coroutines can be paused until response so that we can process other requests meanwhile in worker thread
API design

• Hide as much complexity from Lua developers as possible
API design example

- Networking APIs: Allow parallel requests without async or multithreading programming model
- Separate operations to create requests and to wait for results
HTTP client API example

```plaintext
local requests = {}
for _, url in ipairs(urls) do
    local request = http_request({url = url, ...})
    table.insert(requests, request)
end

...

local responses = {}
for _, request in ipairs(requests) do
    local response = request.get()
    table.insert(responses, response)
end
```
Business logic

- Select ad creative (banner) to show from all ad campaigns
- Track important events for ad creative like clicks
Selecting ad creative

- Complex targeting rules
- Ad campaign delivery optimization
- Money calculations
Selecting ad creative

```python
local creatives = {}
for _, campaign in ipairs(data.campaigns) do
    if campaign_passes_targeting(campaign, request) then
        for _, creative in ipairs(campaign.creatives) do
            if creative_passes_targeting(creative, request) then
                table.insert(creatives, creative)
            end
        end
    end
end

local winner_creative = run_auction(creatives)
return winner_creative
```
Business data as native Lua data

● Most of data our business logic works with is read-only

● Amount of data required in real-time is relatively low

● Solution: use Lua data structures as in-memory storage
Business data as native Lua data

• Very natural Lua code – it is all just iterations over Lua data structures

• Very fast – you cannot beat in-memory data
Problem with data

Out of memory.
Out of memory

• You cannot share Lua data between Lua interpreter states

• More CPU cores → higher memory usage

• Projects became bigger too → more data
Out of memory: LuaJIT

<table>
<thead>
<tr>
<th>32 bit Linux</th>
<th>32 bit application</th>
<th>3 GB of RAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>64 bit Linux</td>
<td>32 bit application</td>
<td>4 GB of RAM</td>
</tr>
<tr>
<td>64 bit Linux</td>
<td>64 bit application</td>
<td>all RAM available but LuaJIT can use only 1GB</td>
</tr>
</tbody>
</table>
Memory problem solution

• Switch to stock Lua?
• Not as fast as LuaJIT
• Fixes immediate problem but with higher memory usage breaks due to GC
Memory problem solution

• Move business data out of Lua
• But we need backward compatibility with existing Lua codebases
Attempt #1: userdata

-userdata + metatables to expose C++ managed data storage as “fake” Lua tables
-Each field access via userdata is C function call → slow compared to native Lua data
Attempt #2: FFI cdata

- FFI – alternative interface to C code from Lua available in LuaJIT
- FFI is designed to be LuaJIT friendly
- cdata is sort of like userdata for FFI – also can use metatables to “fake” Lua tables
How does LuaJIT work?

• Runs parts of your code as interpreted and parts of it as JIT compiled
• As long as hot spots are covered you are good
• If code not written with LuaJIT in mind then most of it will not be compiled
FFI: leap of faith
FFI: leap of faith

<table>
<thead>
<tr>
<th></th>
<th>FFI</th>
<th>Lua 1</th>
<th>Lua 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>jit</td>
<td>1.57</td>
<td>1.87</td>
<td>2.00</td>
</tr>
<tr>
<td>nojit</td>
<td>55.1</td>
<td>5.05</td>
<td>5.95</td>
</tr>
</tbody>
</table>

FFI: leap of faith

• If you introduce FFI in your application it will run slower
• Until you manage to get LuaJIT to JIT compile enough parts of it
How to make LuaJIT happy

• Use compilation traces to find why code doesn't compile

• Unfortunately for uninitiated they look like gibberish
Compilation trace

---- TRACE 19 start history.lua:307

0001 MOV 4 0
0002 TGETS 3 0 0 ; "parse"
0003 ISTC 5 1
0004 JMP 5 => 0006
0005 KSTR 5 1 ; ""
0006 CALL 3 2 3

0000 . FUNC0 23 ; history.lua:62
0001 . KSHORT 2 1
0002 . KPRI 3 0
0003 . TGETS 4 0 0 ; "fields"
0004 . TNEW 5 0
0005 . KNIL 6 8
0006 . ISF 2
0007 . JMP 9 => 0089
0008 . LOOP 9 => 0089

---- TRACE 19 abort history.lua:72 -- inner loop in root trace

IPONWEB
LuaJIT challenge

• Requires special low level knowledge to make code run fast
• Sometimes leads to non-intuitive Lua code
LuaJIT quiz

return tonumber(var)

vs

return (tonumber(var))
LuaJIT challenge

- Breaks our abstractions – Lua developers forced to work on lower level than normally needed
Wrapping up

• Lua: unique challenges
• Lua: despite everything very powerful and successful technology