Writing a Janus plugin in Lua
C can be a scary world, let us come to the rescue!

Lorenzo Miniero
@elminiero

FOSDEM 2018 Real Time devroom
4th February 2018, Brussels 🥧
Remember Janus?

• A door between the communications past and future
  • Legacy technologies (the “past”)
  • WebRTC (the “future”)

Janus
General purpose, open source WebRTC gateway
• https://github.com/meetecho/janus-gateway
• Demos and documentation: https://janus.conf.meetecho.com
• Community: https://groups.google.com/forum/#!forum/meetecho-janus
A quick recap: modular architecture

- The core only implements the WebRTC stack
  - JSEP/SDP, ICE, DTLS-SRTP, Data Channels, ...
- Plugins expose Janus API over different “transports”
  - Currently HTTP / WebSockets / RabbitMQ / Unix Sockets / MQTT
- “Application” logic implemented in plugins too
  - Users attach to plugins via the Janus core
  - The core handles the WebRTC stuff
  - Plugins route/manipulate the media/data
- Plugins can be combined on client side as “bricks”
  - Video SFU, Audio MCU, SIP gatewaying, broadcasting, etc.
Plugins a very powerful way to extend Janus, but...
  ... everything in Janus is written in C! (well, except the web demos of course...)
May be troublesome for some users to write their own (when really needed)
Let’s have a look at the Plugin API (1)

• Plugin initialization and information
  • `init()`: called when plugin is loaded
  • `destroy()`: called when Janus is shutting down
  • `get_api_compatibility()`: must return `JANUS_PLUGIN_API_VERSION`
  • `get_version()`: numeric version identifier (e.g., 3)
  • `get_version_string()`: verbose version identifier (e.g., “v1.0.1”)
  • `get_description()`: verbose description of the plugin (e.g., “This is my awesome plugin that does this and that”)
  • `get_name()`: short display name for your plugin (e.g., “My Awesome Plugin”)
  • `get_author()`: author of the plugin (e.g., “Meetecho s.r.l.”)
  • `get_package()`: unique package identifier for your plugin (e.g., “janus.plugin.myplugin”)

Let’s have a look at the Plugin API (2)

- Sessions management (callbacks invoked by the core)
  - `create_session()`: a user (session+handle) just attached to the plugin
  - `handle_message()`: incoming message/request (with or without a JSEP/SDP)
  - `setup_media()`: PeerConnection is now ready to be used
  - `incoming_rtp()`: incoming RTP packet
  - `incoming_rtcp()`: incoming RTCP message
  - `incoming_data()`: incoming DataChannel message
  - `slow_link()`: notification of problems on media path
  - `hangup_media()`: PeerConnection has been closed (e.g., DTLS alert)
  - `query_session()`: called to get plugin-specific info on a user session
  - `destroy_session()`: existing user gone (handle detached)
Let’s have a look at the Plugin API (3)

- Interaction with the core (methods invoked by the plugin)
  - `push_event()`: send the user a JSON message/event (with or without a JSEP/SDP)
  - `relay_rtp()`: send/relay the user an RTP packet
  - `relay_rtcp()`: send/relay the user an RTCP message
  - `relay_data()`: send/relay the user a DataChannel message
  - `close_pc()`: close the user’s PeerConnection
  - `end_session()`: close a user session (force-detach core handle)
  - `events_is_enabled()`: check whether the event handlers mechanism is enabled
  - `notify_event()`: notify an event to the registered and subscribed event handlers
Sequence diagrams (sessions mgmt)

User → Janus: 
- attach handle XYZ

Janus → Plugin: 
- create_session()

Plugin: 
create session object

Plugin → Janus: 
- handle_message()

Janus → User: 
- send message to handle XYZ

User → Janus: 
- stuff happens (messages, SDP, RTP)
Sequence diagrams (sessions mgmt)

User -> Janus
handle_message()

Janus -> Plugin
prepare synchronous response

Plugin -> Janus
return success+payload

Janus -> User
send response from XYZ

User -> Janus

User -> Plugin

Plugin -> User
Sequence diagrams (sessions mgmt)

User — Janus — Plugin

send message to handle XYZ

handle_message()

prepare asynchronous response

return ack

push_event()

send ack from XYZ

send event from XYZ
Sequence diagrams (sessions mgmt)

User

Janus

Plugin

send message+SDP to handle XYZ

strip WebRTC stuff from SDP

handle_message()

return ack

prepare response and SDP answer

send ack from XYZ

push_event()

add WebRTC stuff to SDP

PeerConnection succeeds

setup_media()

send event+SDP from XYZ
Sequence diagrams (sessions mgmt)

User → Janus
send SRTP to XYZ

Janus
decrypt SRTP and get plain RTP
incoming_rtp()

Plugin
do something with RTP packet
relay_rtp()

Plugin
encrypt plain RTP in SRTP

User → Janus
send SRTP from XYZ
Writing a plugin in a different language

- All the above methods and callbacks need to be implemented in C
  - The core loads a shared module, and the core is written in C

- That said, does the logic really need to be written in C too?
  - As long as stubs are C, the core is happy
  - What these stubs do and return can be done in a different way

- All we need is provide hooks and bindings in C, and delegate the logic

Exactly what we did with the Lua plugin!

- https://github.com/meetecho/janus-gateway/pull/1033
• All the above methods and callbacks need to be implemented in C
  • The core loads a shared module, and the core is written in C
• That said, does the logic really need to be written in C too?
  • As long as stubs are C, the core is happy
  • What these stubs do and return can be done in a different way
• All we need is provide hooks and bindings in C, and delegate the logic

Exactly what we did with the Lua plugin!
• https://github.com/meeteecho/janus-gateway/pull/1033
• http://www.meeteecho.com/blog/tutorial-writing-a-janus-video-call-plugin-in-lua/
Writing a plugin in a different language

- All the above methods and callbacks need to be implemented in C
  - The core loads a shared module, and the core is written in C

- That said, does the logic really need to be written in C too?
  - As long as stubs are C, the core is happy
  - What these stubs do and return can be done in a different way

- All we need is provide hooks and bindings in C, and delegate the logic

Exactly what we did with the Lua plugin!

- https://github.com/meeteecho/janus-gateway/pull/1033
Writing a plugin in a different language

• All the above methods and callbacks need to be implemented in C
  • The core loads a shared module, and the core is written in C

• That said, does the logic really need to be written in C too?
  • As long as stubs are C, the core is happy
  • What these stubs do and return can be done in a different way

• All we need is provide hooks and bindings in C, and delegate the logic

Exactly what we did with the Lua plugin!

• https://github.com/meeteecho/janus-gateway/pull/1033
• http://www.meeteecho.com/blog/tutorial-writing-a-janus-video-call-plugin-in-lua/
Janus Lua plugin: the basics

- Conceptually simple: C plugin, but with an embedded Lua state machine
  - Load a user-provided Lua script when initializing the plugin
  - Implement plugin callbacks in C, and have them call a Lua function
  - Implement core methods as Lua functions in C, that the Lua script can invoke
  - Track users/sessions via a unique ID that the C and Lua code share

- In theory, everything works (simple C ↔ Lua proxy)
  - The core sees a C plugin, but logic is handled in Lua

- In practice, that’s not enough...
  1. Lua is single threaded (how to do things really asynchronously?)
  2. Handling RTP in Lua space would kill performance
Janus Lua plugin: the basics

- Conceptually simple: C plugin, but with an embedded Lua state machine
  - Load a user-provided Lua script when initializing the plugin
  - Implement plugin callbacks in C, and have them call a Lua function
  - Implement core methods as Lua functions in C, that the Lua script can invoke
  - Track users/sessions via a unique ID that the C and Lua code share

- In theory, everything works (simple C↔Lua proxy)
  - The core sees a C plugin, but logic is handled in Lua

- In practice, that’s not enough...
  1. Lua is single threaded (how to do things really asynchronously?)
  2. Handling RTP in Lua space would kill performance
Janus Lua plugin: the basics

- Conceptually simple: C plugin, but with an embedded Lua state machine
  - Load a user-provided Lua script when initializing the plugin
  - Implement plugin callbacks in C, and have them call a Lua function
  - Implement core methods as Lua functions in C, that the Lua script can invoke
  - Track users/sessions via a unique ID that the C and Lua code share

- In theory, everything works (simple C↔Lua proxy)
  - The core sees a C plugin, but logic is handled in Lua

- In practice, that’s not enough...
  1. Lua is single threaded (how to do things really asynchronously?)
  2. Handling RTP in Lua space would kill performance
Hooks and bindings (1)

- Plugin initialization and information

<table>
<thead>
<tr>
<th>C</th>
<th>Lua</th>
</tr>
</thead>
<tbody>
<tr>
<td>init()</td>
<td>init()</td>
</tr>
<tr>
<td>destroy()</td>
<td>destroy()</td>
</tr>
<tr>
<td>get_api_compatibility()</td>
<td>not needed</td>
</tr>
<tr>
<td>get_version()</td>
<td>getVersion()</td>
</tr>
<tr>
<td>get_version_string()</td>
<td>getVersionString()</td>
</tr>
<tr>
<td>get_description()</td>
<td>getDescription()</td>
</tr>
<tr>
<td>get_name()</td>
<td>getName()</td>
</tr>
<tr>
<td>get_author()</td>
<td>getAuthor()</td>
</tr>
<tr>
<td>get_package()</td>
<td>getPackage()</td>
</tr>
</tbody>
</table>

\(^1\) Not really needed, so optional
• Sessions management (callbacks invoked by the core)

<table>
<thead>
<tr>
<th>C</th>
<th>Lua</th>
</tr>
</thead>
<tbody>
<tr>
<td>create_session()</td>
<td>createSession()</td>
</tr>
<tr>
<td>handle_message()</td>
<td>handleMessage()</td>
</tr>
<tr>
<td>setup_media()</td>
<td>setupMedia()</td>
</tr>
<tr>
<td>incoming_rtp()</td>
<td>incomingRtp(²)</td>
</tr>
<tr>
<td>incoming_rtcp()</td>
<td>incomingRtcp(²)</td>
</tr>
<tr>
<td>incoming_data()</td>
<td>incomingData(²)</td>
</tr>
<tr>
<td>slow_link()</td>
<td>slowLink()</td>
</tr>
<tr>
<td>hangup_media()</td>
<td>hangupMedia()</td>
</tr>
<tr>
<td>query_session()</td>
<td>querySession()</td>
</tr>
<tr>
<td>destroy_session()</td>
<td>destroySession()</td>
</tr>
</tbody>
</table>

²Not the right way... more on this later!
Hooks and bindings (3)

- Interaction with the core (methods invoked by the plugin)

<table>
<thead>
<tr>
<th>C</th>
<th>Lua</th>
</tr>
</thead>
<tbody>
<tr>
<td>push_event()</td>
<td>pushEvent()</td>
</tr>
<tr>
<td>relay_rtp()</td>
<td>relayRtp()</td>
</tr>
<tr>
<td>relay_rtcp()</td>
<td>relayRtcp()</td>
</tr>
<tr>
<td>relay_data()</td>
<td>relayData()</td>
</tr>
<tr>
<td>close_pc()</td>
<td>closePc()</td>
</tr>
<tr>
<td>end_session()</td>
<td>endSession()</td>
</tr>
<tr>
<td>events_is_enabled()</td>
<td>eventsIsEnabled()</td>
</tr>
<tr>
<td>notify_event()</td>
<td>notifyEvent()</td>
</tr>
</tbody>
</table>

3 Not the right way... more on this later!
4 Not really needed, so optional
Example of hooks and bindings
Asynchronous logic in the Lua plugin

• We’ve seen how asynchronous events are heavily used by plugins
  • Asynchronous message response, negotiations, etc.
  • Most out-of-the-box Janus plugins are thread based

• Lua is single threaded, though...
  • Coroutines can be seen as threads, but they aren’t
  • Access to the Lua state isn’t thread safe either

Solution: a C “scheduler”

A dedicated thread in the C code of the plugin acts as scheduler

• The Lua script queues tasks, and “pokes” the scheduler via pokeScheduler()
• pokeScheduler() is implemented in C, and wakes the scheduler (queue)
• The C scheduler calls resumeScheduler() in Lua as a coroutine
Asynchronous logic in the Lua plugin

- We’ve seen how asynchronous events are heavily used by plugins
  - Asynchronous message response, negotiations, etc.
  - Most out-of-the-box Janus plugins are thread based

- Lua is single threaded, though...
  - Coroutines can be seen as threads, but they aren’t
  - Access to the Lua state isn’t thread safe either

Solution: a C “scheduler”

A dedicated thread in the C code of the plugin acts as scheduler

- The Lua script queues tasks, and “pokes” the scheduler via `pokeScheduler()`
- `pokeScheduler()` is implemented in C, and wakes the scheduler (queue)
- The C scheduler calls `resumeScheduler()` in Lua as a coroutine
Asynchronous logic in the Lua plugin

- We’ve seen how asynchronous events are heavily used by plugins
  - Asynchronous message response, negotiations, etc.
  - Most out-of-the-box Janus plugins are thread based

- Lua is single threaded, though...
  - Coroutines can be seen as threads, but they aren’t
  - Access to the Lua state isn’t thread safe either

Solution: a C “scheduler”

A dedicated thread in the C code of the plugin acts as scheduler

- The Lua script queues tasks, and “pokes” the scheduler via `pokeScheduler()`
- `pokeScheduler()` is implemented in C, and wakes the scheduler (queue)
- The C scheduler calls `resumeScheduler()` in Lua as a coroutine
Scheduler example: asynchronous reply

Janus → Plugin
handle_message() [C]

Plugin → Lua
incomingMessage() [Lua]

Lua
handle message asynchronously, keep note of task

PokeScheduler() [Lua]

add wake up call to scheduler queue

Lua → Plugin
return ack [Lua]

Plugin → Janus
return ack [C]

Janus
Scheduler example: asynchronous reply

Janus

Plugin

Lua

scheduler thread wakes up

resumeScheduler() [Lua]

resume task from before

Janus

Plugin

Lua

push_event() [C]

pushEvent() [Lua]
Timed callbacks in the Lua plugin

- `pokeScheduler()` and `resumeScheduler()` are great but have limits
  - No arguments can be passed to the scheduler
  - You need to keep track of tasks yourself
  - The `resumeScheduler()` function is called as soon as possible
- You may want to trigger a callback (with a parameter?) after some time instead
  - e.g., “call secondsPassed(5) in 5 seconds”

Solution: a new `timeCallback` function as a C hook

A timed source in the C code of the plugin acts as triggerer
- The Lua script times a callback via `timeCallback()`
- `timeCallback()` is implemented in C, and creates a timed source
- The source fires and calls the specified callback in Lua as a coroutine
Timed callbacks in the Lua plugin

- `pokeScheduler()` and `resumeScheduler()` are great but have limits
  - No arguments can be passed to the scheduler
  - You need to keep track of tasks yourself
  - The `resumeScheduler()` function is called as soon as possible

- You may want to trigger a callback (with a parameter?) after some time instead
  - e.g., “call secondsPassed(5) in 5 seconds”

**Solution: a new `timeCallback` function as a C hook**

A timed source in the C code of the plugin acts as triggerer
- The Lua script times a callback via `timeCallback()`
- `timeCallback()` is implemented in C, and creates a timed source
- The source fires and calls the specified callback in Lua as a coroutine
Timed callbacks in the Lua plugin

- `pokeScheduler()` and `resumeScheduler()` are great but have limits
  - No arguments can be passed to the scheduler
  - You need to keep track of tasks yourself
  - The `resumeScheduler()` function is called as soon as possible
- You may want to trigger a callback (with a parameter?) after some time instead
  - e.g., “call secondsPassed(5) in 5 seconds”

Solution: a new `timeCallback()` function as a C hook

A timed source in the C code of the plugin acts as triggerer
- The Lua script times a callback via `timeCallback()`
- `timeCallback()` is implemented in C, and creates a timed source
- The source fires and calls the specified callback in Lua as a coroutine
What about RTP/RTCP/data?

- As we pointed out, handling data in Lua drags performance down
  - While hooks are there, there’s a cost in going from C to Lua and viceversa
  - Lua state is single threaded, meaning relaying would have a bottleneck
- Arguably this is more of an issue for RTP, less so for RTCP and data
  - ... unless RTCP and data messages are very frequent too

Solution: only configuring routing in Lua (actual relaying still in C)
The C code routes the media according to dynamic changes coming from Lua

- addRecipient() and removeRecipient() dictate who receives user’s media
- configureMedium() opens/closes valves for outgoing/incoming media
- Helper methods (setBitrate(), sendPli(), etc.) do the rest
What about RTP/RTCP/data?

- As we pointed out, handling data in Lua drags performance down
  - While hooks are there, there’s a cost in going from C to Lua and vice versa
  - Lua state is single threaded, meaning relaying would have a bottleneck
- Arguably this is more of an issue for RTP, less so for RTCP and data
  - ... unless RTCP and data messages are very frequent too

Solution: only configuring routing in Lua (actual relaying still in C)

The C code routes the media according to dynamic changes coming from Lua

- addRecipient() and removeRecipient() dictate who receives user’s media
- configureMedium() opens/closes valves for outgoing/incoming media
- Helper methods (setBitrate(), sendPli(), etc.) do the rest
What about RTP/RTCP/data?

- As we pointed out, handling data in Lua drags performance down
  - While hooks are there, there's a cost in going from C to Lua and vice versa
  - Lua state is single threaded, meaning relaying would have a bottleneck
- Arguably this is more of an issue for RTP, less so for RTCP and data
  - ... unless RTCP and data messages are very frequent too

Solution: only configuring routing in Lua (actual relaying still in C)

The C code routes the media according to dynamic changes coming from Lua

- `addRecipient()` and `removeRecipient()` dictate who receives user's media
- `configureMedium()` opens/closes valves for outgoing/incoming media
- Helper methods (`setBitrate()`, `sendPli()`, etc.) do the rest
Routing media (SFU example)
A few examples: EchoTest clone

```
15  -- Example details
16  name = "echotest.lua"
17  logger.prefix(colored("[wblue]" .. name .. "%(reset)\n"))
18  logger.print("Loading...")
19
20  -- State and properties
21  sessions = ()
22  tasks = ()
23
24  -- Methods
25  function init(config)
26     -- This is where we initialize the plugin, for static properties
27     logger.print("initializing...")
28     if config == nil then
29        logger.print("configuration file provided \".. config .. ", but we don't need it")
30     end
31     logger.print("initialized")
32  end
33
34  function destroy()
35     -- This is where we deinitialize the plugin, when Janus shuts down
36     logger.print("deinitialize")
37  end
38
39  function createSession(id)
40     -- Keep track of a new session
41     logger.print("created new session: \".. id")
42     sessions[id] = { id = id, lua = name }
43  end
44
45  function destroySession(id)
46     -- A Janus plugin session has gone
47     logger.print("destroyed session: \".. id")
48     hangupMedia(id)
49     sessions[id] = nil
50  end
51
52  function querySession(id)
53      -- Return info on a session
54      logger.print("queried session: \".. id")
```
Something trickier: VideoRoom clone
VideoCall clone: a tutorial

Astricon 2017 Dangerous Demo

ARI wrapper via Janus (datachannels)

[>>>] help
[<<<] Welcome, brave user, to this Dangerous Demo!
This is the list of supported commands:

help -- Print this message
channels -- List the active channels
call USER from EXTENSION -- Originate a SIP call
hangup CHANNEL -- Hangup a channel
raise hell -- Break this demo

https://gist.github.com/lminiero/9aeeda1be501fb636cad0c8057c6e076
One more cool example... Chatroulette!

Chatroulette Demo (Lua)

Chatroulette Demo (Lua)

https://github.com/lminiero/fosdem18-januslua
One more cool example... Chatroulette!

https://github.com/lminiero/fosdem18-januslua
One more cool example... Chatroulette!

Chatroulette Demo (Lua)

https://github.com/lminiero/fosdem18-januslua
One more cool example... Chatroulette!

Chatroulette Demo (Lua)

Local Stream

Remote Stream

Write a DataChannel message to your peer

[now chatting with Pippo]
What to do next?

- Integrate advanced features recently added to master
  - RTP injection/forwarding, simulcasting, VP9 SVC, ...
- General improvements may be needed once it’s used more
  - Based on refcount branch, which is experimental itself
- Do Lua-based Transport plugins and Event Handlers make any sense?
  - They’re plugins (shared objects) too, after all...
- Why not, write new plugins for other programming languages!
  - Most hooks are already there, after all, we only need bindings
  - A potential “candidate”: JavaScript (e.g., with http://duktape.org/)

Help us improve this!

- Play with it, more testing is important
- Write your own applications, or help expand the Lua plugin itself!
What to do next?

- Integrate advanced features recently added to master
  - RTP injection/forwarding, simulcasting, VP9 SVC, ...
- General improvements may be needed once it’s used more
  - Based on refcount branch, which is experimental itself
- Do Lua-based Transport plugins and Event Handlers make any sense?
  - They’re plugins (shared objects) too, after all...
- Why not, write new plugins for other programming languages!
  - Most hooks are already there, after all, we only need bindings
  - A potential “candidate”: JavaScript (e.g., with http://duktape.org/)

Help us improve this!

- Play with it, more testing is important
- Write your own applications, or help expand the Lua plugin itself!
What to do next?

- Integrate advanced features recently added to master
  - RTP injection/forwarding, simulcasting, VP9 SVC, ...
- General improvements may be needed once it’s used more
  - Based on `refcount` branch, which is experimental itself
- Do Lua-based Transport plugins and Event Handlers make any sense?
  - They’re plugins (shared objects) too, after all...
  - Why not, write new plugins for other programming languages!
    - Most hooks are already there, after all, we only need bindings
    - A potential “candidate”: JavaScript (e.g., with [http://duktape.org/](http://duktape.org/))

Help us improve this!

- Play with it, more testing is important
- Write your own applications, or help expand the Lua plugin itself!
What to do next?

• Integrate advanced features recently added to master
  • RTP injection/forwarding, simulcasting, VP9 SVC, ...

• General improvements may be needed once it’s used more
  • Based on refcount branch, which is experimental itself

• Do Lua-based Transport plugins and Event Handlers make any sense?
  • They’re plugins (shared objects) too, after all...

• Why not, write new plugins for other programming languages!
  • Most hooks are already there, after all, we only need bindings
  • A potential “candidate”: JavaScript (e.g., with http://duktape.org/)

Help us improve this!

• Play with it, more testing is important

• Write your own applications, or help expand the Lua plugin itself!
What to do next?

- Integrate advanced features recently added to master
  - RTP injection/forwarding, simulcasting, VP9 SVC, ...
- General improvements may be needed once it’s used more
  - Based on refcount branch, which is experimental itself
- Do Lua-based Transport plugins and Event Handlers make any sense?
  - They’re plugins (shared objects) too, after all...
- Why not, write new plugins for other programming languages!
  - Most hooks are already there, after all, we only need bindings
  - A potential “candidate”: JavaScript (e.g., with http://duktape.org/)

Help us improve this!

- Play with it, more testing is important
- Write your own applications, or help expand the Lua plugin itself!
Thanks! Questions? Comments?

Get in touch!

- 🔗 https://twitter.com/elminiero
- 🔗 https://twitter.com/meetecho
- 🔗 http://www.meetecho.com