RFC #1999 - 2020-03-06 - API extensions for `lua` transform

This RFC proposes a new API for the `lua` transform.

Motivation

Currently, the `lua` transform has some limitations in its API. In particular, the following features are missing:

- **Nested Fields**

  Currently accessing nested fields is possible using the field path notation:

  ```
  event["nested.field"] = 5
  ```

  However, users expect nested fields to be accessible as native Lua structures, for example like this:

  ```
  event["nested"]["field"] = 5
  ```

  See #706 and #1406.

- **Setup Code**

  Some scripts require expensive setup steps, for example, loading of modules or invoking shell commands. These steps should not be part of the main transform code.

For example, this code adding custom hostname

```lua
if event["host"] == nil then
  local f = io.popen ("/bin/hostname")
  local hostname = f:read("*a") or ""
  f:close()
  hostname = string.gsub(hostname, "\n$", "")
  event["host"] = hostname
end
```

Should be split into two parts, the first part executed just once at the initialization:

```lua
local f = io.popen ("/bin/hostname")
local hostname = f:read("*a") or ""
f:close()
hostname = string.gsub(hostname, "\n$", "")
```

and the second part executed for each incoming event:

```lua
if event["host"] == nil then
  event["host"] = hostname
end
```

See #1864.

- **Control Flow**

  It should be possible to define channels for output events, similarly to how it is done in `swimlanes` transform.

  See #1942.

Motivating Examples

Fields Manipulation

The following example illustrates fields manipulations with the new approach.

```
[tranforms.lua]
  type = "lua"
  inputs = []
  version = "2"
  hooks.process = """
    function (event, emit)
      -- add new field (simple)
      event.new_field = "example"
      -- add new field (nested, overwriting the content of "nested" map)
      event.nested = {
        field = "example value"
      }
```

```
      -- add new field (nested, to already existing map)
      event.nested.another_field = "example value"
      -- add new field (nestd, without assumptions about presence of the parent map)
      if event.possibly_existing == nil then
        event.possibly_existing = {}
      end
      event.possibly_existing.example_field = "example value"

      -- remove field (simple)
      event.removed_field = nil
      -- remove field (nested, keep parent maps)
      event.nested.field = nil
      -- remove field (nested, if the parent map is empty, the parent map is removed too)
      event.another_nested.field = nil
      if next(event.another_nested) == nil then
        event.another_nested = nil
      end

      -- rename field from "original_field" to "another_field"
      event.original_field, event.another_field = nil, event.original_field

      emit(event)
    end
  """
```

Log to Metric

This example is a log to metric transform which produces metric events from incoming log events using the following algorithm:

1. There is an internal counter which is increased on each incoming log event.
2. The log events are discarded.
3. Each 10 seconds the transform produces a metric event with the count of received log events.
4. Edge cases are handled in the following way:
    1. If there are no incoming invents, the metric event with the counter equal to 0 still has to be produced.
    2. On Vector's shutdown the transform has to produce the final metric event with the count of received events since the last flush.

Two versions of a config running the same Lua code are listed below, both of them implement the transform described above.

Inline Functions

This config uses Lua functions defined as inline strings. It is easier to get started with runtime transforms.

```
[transforms.lua]
  type = "lua"
  inputs = []
  version = "2"
  hooks.init = """
    function init (emit)
      event_counter = 0
      emit({
        log = {
          message = "starting up"
        }
      }, "auxiliary")
    end
  """
  hooks.process = """
    function (event, emit)
      event_counter = event_counter + 1
    end
  """
  hooks.shutdown = """
    function shutdown (emit)
      emit {
        metric = {
          name = "counter_10s",
          counter = {
            value = event_counter
          }
        }
      }

      emit({
        log = {
          message = "shutting down"
        }
      }, "auxiliary")
    end
  """
  [[timers]]
  interval_seconds = 10
  handler = """
    function (emit)
      emit {
        metric = {
          name = "counter_10s",
          counter = {
            value = event_counter
```

```
          }
        }
      }
      counter = 0
    end
  """
```

Single Source

This version of the config uses the same Lua code as the config using inline Lua functions above, but all of the functions are defined in a single `source` option:

```
[transforms.lua]
  type = "lua"
  inputs = []
  version = "2"
  source = """
    function init (emit)
      event_counter = 0
      emit({
        log = {
          message = "starting up"
        }
      }, "auxiliary")
    end

    function process (event, emit)
      event_counter = event_counter + 1
    end

    function shutdown (emit)
      emit {
        metric = {
          name = "counter_10s",
          counter = {
            value = event_counter
          }
        }
      }

      emit({
        log = {
          message = "shutting down"
        }
      }, "auxiliary")
    end
```

```
    function timer_handler (emit)
      emit {
        metric = {
          name = "counter_10s",
          counter = {
            value = event_counter
          }
        }
      }
      counter = 0
    end
  """
  hooks.init = "init"
  hooks.process = "process"
  hooks.shutdown = "shutdown"
  timers = [{interval_seconds = 10, handler = "timer_handler"}]
```

Loadable Module: Global Functions

In this example the code from the `source` of the example above is put into a separate file:

`example_transform.lua`

```lua
function init (emit)
  event_counter = 0
  emit({
    log = {
      message = "starting up"
    }
  }, "auxiliary")
end

function process (event, emit)
  event_counter = event_counter + 1
end

function shutdown (emit)
  emit {
    metric = {
      name = "counter_10s",
      counter = {
        value = event_counter
      }
    }
  }
```

```lua
  emit({
    log = {
      message = "shutting down"
    }
  }, "auxiliary")
end

function timer_handler (emit)
  emit {
    metric = {
      name = "counter_10s",
      counter = {
        value = event_counter
      }
    }
  }
  counter = 0
end
```

It reduces the size of the transform configuration:

```toml
[transforms.lua]
  type = "lua"
  inputs = []
  version = "2"
  search_dirs = ["/example/search/dir"]
  source = "require 'example_transform.lua'"
  hooks.init = "init"
  hooks.process = "process"
  hooks.shutdown = "shutdown"
  timers = [{interval_seconds = 10, handler = "timer_handler"}]
```

Loadable Module: Isolated Functions

The way to create modules in previous example above is simple, but might cause name collisions if there are multiple modules to be loaded.

It is recommended to create tables for modules and put functions inside them:

example_transform.lua

```lua
local example_transform = {}
local event_counter = 0
function example_transform.init (emit)
  emit({
    log = {
      message = "starting up"
```

```
    }
  }, "auxiliary")
end

function example_transform.process (event, emit)
  event_counter = event_counter + 1
end

function example_transform.shutdown (emit)
  emit {
    metric = {
      name = "counter_10s",
      counter = {
        value = event_counter
      }
    }
  }

  emit({
    log = {
      message = "shutting down"
    }
  }, "auxiliary")
end

function example_transform.timer_handler (emit)
  emit {
    metric = {
      name = "counter_10s",
      counter = {
        value = event_counter
      }
    }
  }
  counter = 0
end

return example_transform
```

Then the transform configuration is the following:

```
[transforms.lua]
  type = "lua"
  inputs = []
  version = "2"
  search_dirs = ["/example/search/dir"]
  source = "example_transform = require 'example_transform.lua'"
```

```
hooks.init = "example_transform.init"
hooks.process = "example_transform.process"
hooks.shutdown = "example_transform.shutdown"
timers = [{interval_seconds = 10, handler = "example_transform.timer_handler"}]
```

Reference-level Proposal

Versions

Lua transform configuration have to be versioned in order to distinguish between the old and the new APIs.

The old API is identified by version `1` and the new one, which is proposed in the present RFC, is identified by version `2`. The version can be set using a `version` option in the configuration file. During the transitional period, omitting the version should result in using version `1`. After all changes proposed here are implemented and sufficiently tested, version `1` could be deprecated and version `2` used as the default version.

New Concepts

In order to enable writing complex transforms, such as the one from the motivating example, a few new concepts have to be introduced.

Hooks

Hooks are user-defined functions which are called on certain events.

- `init` hook is a function with signature

  ```lua
  function (emit)
    -- ...
  end
  ```

  which is called when the transform is created. It takes a single argument, `emit` function, which can be used to produce new events from the hook.

- `shutdown` hook is a function with signature

  ```lua
  function (emit)
    -- ...
  end
  ```

  which is called when the transform is destroyed, for example on Vector's shutdown. After the shutdown is called, no code from the transform would be called.

- `process` hook is a function with signature

```
function (event, emit)
  -- ...
end
```

which takes two arguments, an incoming event and the `emit` function. It is called immediately when a new event comes to the transform.

Timers

Timers are user-defined functions called on predefined time interval. The specified time interval sets the minimal interval between subsequent invocations of the same timer function.

The timer functions have the following signature:

```
function (emit)
  -- ...
end
```

The `emit` argument is an emitting function which allows the timer to produce new events.

Emitting Functions

Emitting function is a function that can be passed to a hook or timer. It has the following signature:

```
function (event, lane)
  -- ...
end
```

Here `event` is an encoded event to be produced by the transform, and `lane` is an optional parameter specifying the output lane. In order to read events produced by the transform on a certain lane, the downstream components have to use the name of the transform suffixed by `.` character and the name of the lane.

Example

An emitting function is called from a transform component called `example_transform` with `lane` parameter set to `example_lane`. Then the downstream `console` sink have to be defined as the following to be able to read the emitted event:

```
[sinks.example_console]
  type = "console"
  inputs = ["example_transform.example_lane"] # would output the event from `example_la
  encoding = "text"
```

Other components connected to the same transform, but with different lanes names or without lane names at all would not receive any event.

Event Schema

Events passed to the transforms have `userdata` type with custom implementation of the `__index` metamethod. This data type is used instead of `table` because it allows to avoid copying of the data which is not used.

Events produced by the transforms through calling an emitting function can have either the same `userdata` type as the events passed to the transform, or be a newly created Lua tables with the same schema outlines below.

Both log and metrics events are encoded using external tagging.

- Log events could be seen as tables created using

  ```
  {
    log = {
      -- ...
    }
  }
  ```

  The content of the `log` field corresponds to the usual log event structure, with possible nesting of the fields.

  If a log event is created by the user inside the transform is a table, then, if default fields named according to the global schema are not present in such a table, then they are automatically added to the event. This rule does not apply to events having `userdata` type.

  Example 1 > The global schema is configured so that `message_key` is `"message"`, `timestamp_key` is `"timestamp"`, and `host_key` is is `"instance_id"`. > > If a new event is created inside the user-defined Lua code as a table > > `lua > event = { > log = { > message = "example message", > nested = { > field = "example nested field value" > }, > array = {1, 2, 3}, > } > }` > > > and then emitted through an emitting function, Vector would examine its fields and add `timestamp` containing the current timestamp and `instance_id` field with the current hostname.

  Example 2 > The global schema has default settings. > > A log event created by `stdin` source is passed to the `process` hook inside the transform, where it appears to have `userdata` type. The Lua code inside the transform deletes the `timestamp` field by setting it to `nil`: > > `lua > event.log.timestamp = nil` > > > And then emits the event. In that case Vector would not automatically insert the `timestamp` field.

- Metric events could be seen as tables created using

  ```
  {
    metric = {
      -- ...
  ```

```
  }
}
```

The content of the `metric` field matches the metric data model. The values use external tagging with respect to the metric type, see the examples.

In case when the metric events are created as tables in user-defined code, the following default values are assumed if they are not provided:

Field Name	Default Value
`timestamp`	Current time
`kind`	`absolute`
`tags`	empty map

Furthermore, for `aggregated_histogram` the `count` field inside the `value` map can be omitted.

Example: `counter` > The minimal Lua code required to create a counter metric is the following: > > "'lua > { > metric = { > name = "example_counter", > counter = { > value = 10 > } > } > }

Example: `gauge` > The minimal Lua code required to create a gauge metric is the following: > > "'lua > { > metric = { > name = "example_gauge", > gauge = { > value = 10 > } > } > }

Example: `set` > The minimal Lua code required to create a set metric is the following: > > "'lua > { > metric = { > name = "example_set", > set = { > values = {"a", "b", "c"} > } > } > }

Example: `distribution` > The minimal Lua code required to create a distribution metric is the following: > > "'lua > { > metric = { > name = "example_distribution", > distribution = { > values = {"a", "b", "c"} > } > } > }

Example: `aggregated_histogram` > The minimal Lua code required to create an aggregated histogram metric is the following: > > "`lua > { > metric = { > name = "example_histogram", > aggregated_histogram = { > buckets = {1.0, 2.0, 3.0}, > counts = {30, 20, 10}, > sum = 1000 -- total sum of all measured values, cannot be inferred from`counts`and`buckets> } > } > } > Note that the field [count](https://vector.dev/docs/about/data-model/metric/#count) is not required because it can be inferred by Vector automatically by summing up the values from`counts'.

Example: `aggregated_summary` > The minimal Lua code required to create an aggregated summary metric is the following: > > "'lua > { > metric = { > name = "example_summary", > aggregated_summary = {

> quantiles = {0.25, 0.5, 0.75}, > values = {1.0, 2.0, 3.0}, > sum = 200, > count = 100 > } > } > }

Data Types

The mapping between Vector data types and Lua data types is the following:

Vector Type	Lua Type	Comment
`String`	`string`	
`Integer`	`integer`	
`Float`	`number`	
`Boolean`	`boolean`	

Vector Type	Lua Type	Comment
`Timestamp`	`userdata`	There is no dedicated timestamp type in Lua. However, there is a standard library function `os.date` which returns a table with fields `year`, `month`, `day`, `hour`, `min`, `sec`, and some others. Other standard library functions, such as `os.time`, support tables with these fields as arguments. Because of that, Vector timestamps passed to the transform are represented as `userdata` with the same set of accessible fields. In order to have one-to-one correspondence between Vector timestamps and Lua timestamps, `os.date` function from the standard library is patched to return not a table, but `userdata` with the same set of fields as it usually would return instead. This approach makes it possible to have both compatibility with the standard library functions and a dedicated data type for timestamps.

Vector Type	Lua Type	Comment
`Null`	empty string	In Lua setting a table field to `nil` means deletion of this field. Furthermore, settin an array element to `nil` leads to deletion of this element. In order to avoid inconsistencies, already present `Null` values are visible represented as empty strings from Lua code, and it is impossible to create a new `Null` value in the user-defined code.
`Map`	`userdata` or `table`	Maps which are parts of events passed to the transform from Vector have `userdata` type. User-created maps have `table` type. Both types are converted to Vector's `Map` type when they are emitted from the transform.

Vector Type	Lua Type	Comment
`Array`	`sequence`	Sequences in Lua are a special case of tables. Because of that fact, the indexes can in principle start from any number. However, the convention in Lua is to to start indexes from 1 instead of 0, so Vector should adhere it.

Configuration

The new configuration options are the following:

Option Name	Required	Example	Description
`version`	yes	`2`	In order to use the proposed API, the config has to contain `version` option set to `2`. If it is not provided, Vector assumes that API version 1 is used.
`search_dirs`	no	`["/etc/vector/lua"]`	A list of directories where `require` function would look at if called from any part of the Lua code.

Option Name	Required	Example	Description
`source`	no	`example_module = require("example_module")`	Lua source evaluated when the transform is created. It can call `require` function or define variables and handler functions inline. It is **not** called for each event like the `source` parameter in version 1 of the transform
`hooks.init`	no	`example_function` or `function (emit) ... end`	Contains a Lua expression evaluating to `init` hook function.
`hooks.shutdown`	no	`example_function` or `function (emit) ... end`	Contains a Lua expression evaluating to `shutdown` hook function.
`hooks.process`	yes	`example_function` or `function (event, emit) ... end`	Contains a Lua expression evaluating to `shutdown` hook function.

Option Name	Required	Example	Description
`timers`	no	`[{interval_seconds = 10, handler = "example_function"}]` or `[{interval_seconds = 10, handler = "function (emit) ... end"}]`	Contains an array of tables. Each table in the array has two fields, `interval_seconds` which can take an integer number of seconds, and `handler`, which is a Lua expression evaluating to a handler function for the timer.

Prior Art

The implementation of `lua` transform supports only log events. Processing of log events has the following design:

- There is a `source` parameter which takes a string of code.
- When a new event comes in, the global variable `event` is set inside the Lua context and the code from `source` is evaluated.
- After that, Vector reads the global variable `event` as the processed event.
- If the global variable `event` is set to `nil`, then the event is dropped.

Events have type `userdata` with custom metamethods, so they are views to Vector's events. Thus passing an event to Lua has zero cost, so only when fields are actually accessed the data is copied to Lua.

The fields are accessed through string indexes using Vector's field path notation.

Sales Pitch

The proposal

- gives users more power to create custom transforms;
- supports both logs and metrics;
- makes it possible to add complexity to the configuration of the transform gradually when needed.

Plan Of Attack

- [ ] Implement support for `version` config option and split implementations for versions 1 and 2.

- [ ] Add support for `userdata` type for timestamps.
- [ ] Implement access to the nested structure of logs events.
- [ ] Implement metrics support.
- [ ] Support creation of events as table inside the transform.
- [ ] Support emitting functions.
- [ ] Implement hooks invocation.
- [ ] Implement timers invocation.
- [ ] Add behavior tests and examples to the documentation.