Sec Lope De Vega Project

Release 0.5

Alberto Dominguez

Oct 18, 2022

CONTENTS

Warning: Please, consider this project as **somewhere in between an alpha and beta version** since some capabilities have not been fully tested (mainly the integration with other systems for sending alerts, as well as the support of TLS). In terms of technology maturity, this one is about between TRL-3 and TRL-4 according to the technology readiness level because more testing is needed to verify the fit for purpose.

CHAPTER ONE

CONTENTS

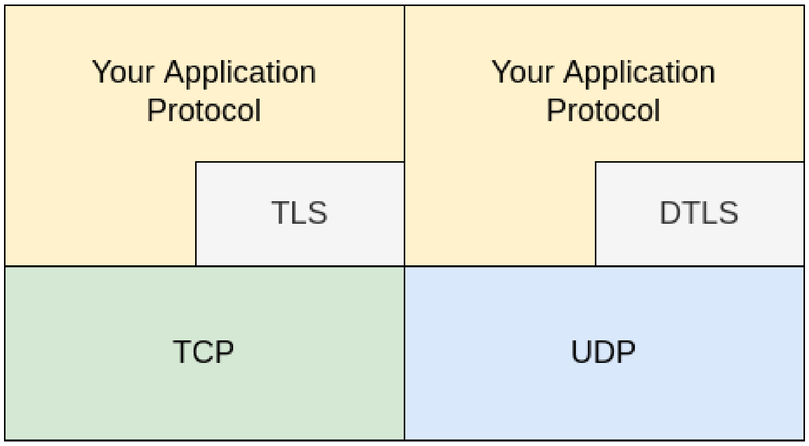
1.1 Intro

Sec Lope de Vega or just **Lope**, is an open source project licensed under the Mozilla Public License v2 . The origin of this was to be as a learning playground for Python, Golang, ZeroMQ and some crypto capabilities. However, this software is transformed into a project since it can be potentially interesting and useful to the community. As far as what this software does, it tries to be a *'Network Application Interactor' (NAI?)*: it interacts with applications, based on a set of rules ('conversation rules') using network protocols. To do so, this software is able to work with **tcp** or **udp** connections in the two typical modes:

- Client mode (*Lope* connects to other systems or applications)
- Server mode (other systems or applications connect to *Lope*)

The *conversation rules* are not fixed, and they are fully configurable by the operator of the software (you). In that sense, this is the main capability of this software: To provide complete flexibility in defining how the interaction should be via some 'configuration files', and optionally, adding some custom code if needed. Since this software works at **tcp** or **udp** level, it tries to be as much as agnostic possible for the application protocol.

This means that this software does not provide directly application level protocols, so it is up to the operator to implement them (or using an existing one). *Lope* is just a kind of 'framework' to allow to define interactions in different application protocols, leaving the implementation as part of the definition of the interaction. Additionally, **TLS** and **DTLS** are supported to some extent, so you may define the interaction on top of those protocols. In the following image you can see the protocol layer capabilities of *Lope*:



This capability of defining your own interaction in the application protocol you wish (if possible) is the main reason for the name of this project. Lope de Vega was a famous Spanish writer in the Spanish Golden Age, who was quite prolific in his work. Therefore, this software enables you to create many 'interactions', potentially making you as 'prolific' as possible thanks to the customization capabilities of this software. The term 'Sec' is a kind of play on words using the idea of 'Sir' (he was indeed a knight of the Order of Malta), and focusing on the potential use of this software for security reasons:

- In client mode, *Lope* can do some security tests when *he* connects to other systems in an interactive approach (not sure if we can call this IAST, and potentially some web crawling/scraping activities. Since you can define how many client connections to establish, it might be useful to try to do some DDoS testing.
- In server mode, *Lope* can work as a kind of honeypot (or a set of them, this is explained in other sections). It also can work to do some dynamic testing on client software via the answers provided from the server.

However, this software may be useful for other kind of tests that are not only security related, such as mocking-up other systems (as a client or as a server), potentially emulating user activity, as well as for doing load and stress testing. **In any case, just remember that this software requires you to define the interaction, so it is not 'plug and play': there is still work to do to cover your needs** (except you have the right interaction files - *conversation rules* files). You can think of this like a gun and its bullets: *Lope* is the 'gun', but the *conversation rules* are the 'bullets'.

Regarding rules, there four three main types of **conversation rules** that *Lope* can execute:

- **Basic Rules:** These rules are executed in specific situations and with a limited set of options (for instance, at the time of the first interaction, or the time-out event).
- **Synchronous Rules (or sync rules):** These rules as executed when a new message is received in the connection with the third party system. Using regular expressions (RegEx) , *Lope* is able to detect what rule or rules are applicable, and then, execute them.
- **Asynchronous Rules (or async rules):** These rules are executed if a previous rule triggers them, so any synchronous or asynchronous rule can trigger one or several asynchronous rules.
- **Hybrid Rules:** These rules are syncrhonous and asynchronous at the same time. They are triggered via a regular expression that is applicable, or asyncrhonously from another rule.

As far as the architecture is respective, this software is made up of two components:

- **The External Connectors:** These elements are those that interact directly with the target system, they get the 'conversation rules' from the **engine**, and they inform it about any interaction they do. There can be several of external connectors working at the same time, working with one **engine**. You can consider this component as the *interaction unit*. This software is coded in Python and the code can be found in this repository.
- **The Engine:** This is the element that work as orchestrator and the 'heart' of the *Lope*, it is the *control unit*. This controls the operation, sends the corresponding 'conversation rules' to the **external connectors**, and receives all activities generated by the **external connectors**. This software is coded in Go and the code can be found here.

Note: This project is the result of the work done in the spare time of the contributors, and the software is **PROVIDED AS IS**, in the complete meaning of those words. There is no official support offered by this project, and proposed changes or bugs might or might not be done. It depends on the willingness of the contributors to use their free time on this project, at their own speed (if they want to do that).

1.2 Architecture

As mentiond previously, *Lope* is composed of two main components: The **external connector** and the **engine**, but there could be more components in the landscape:

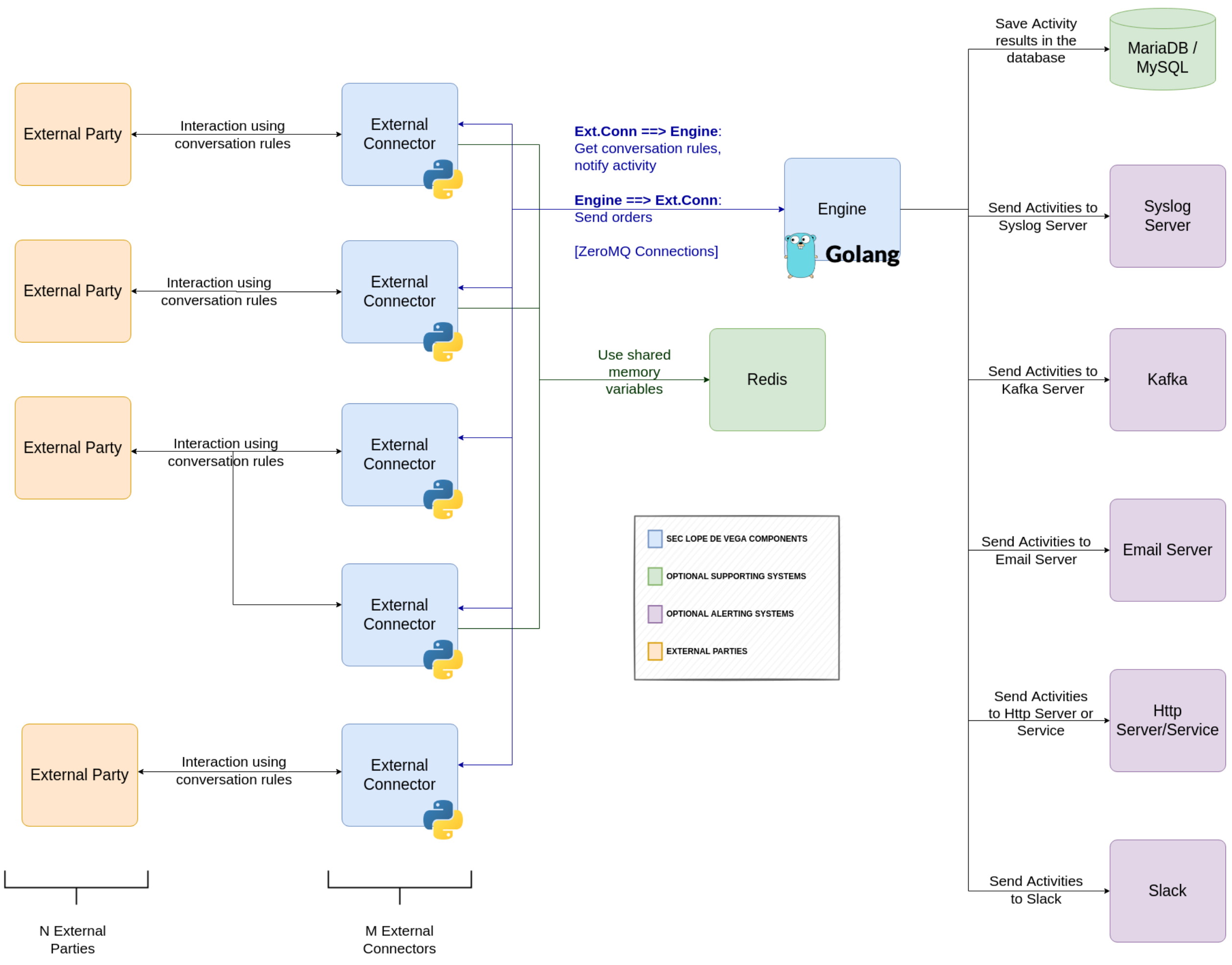
- **The external parties:** They are the elements that connect to *Lope*, or *Lope* connects to them (depending on if the software is working as a client or as a server)
- **Redis server:** In *Session Support & Memory Variables* section, the concept of *memory variables* is explained in detail but here we can say that the interaction can use some variables in the interactions. These variables are

saved in different 'memories', that is why they have that name. One of the memories is shared among different **external connectors**, and this can take place because the redis server. This component is an optional component only needed if you wish to share variables among different **external connectors**, and only for those that are using those variables. This means that not all **external connectors** must be connected to redis, only those that use that capability. Here you can find more information about Redis.

The **engine** is receiving all activities generated by the **external connectors**, and it can do several things with that information. The default option is to save them in a simple file, but other options are also possible. The following components can be used to have specific capabilities about what to do with these activities, so they are optional to run the software and only needed for specific use cases. These components are not provided by this project, and it is up to the operator to install them and configure them, *Lope* only provides the capability to connect with them and use them.

- **MariaDB/MySQL Database:** This component is needed to save the activities in a database. It could be either MariaDB database or MySQL database.
- **Syslog server:** *Lope* needs this component to send the alerts of the activities via syslong.
- **Kafka server:** This component is used when you wish to have the alerts of the activities to be sent via Kafka.
- **Email server:** This component is required if you wish to send alerts via emails.
- **HTTP server/service:** This element is needed if you wish to send alerts to a web server or service using HTTP calls.
- **Slack:** If you wish to send alerts to slack, you should have prepared everything in Slack in advance.

In the following image, you can find a graphical representation of the potential architecture:

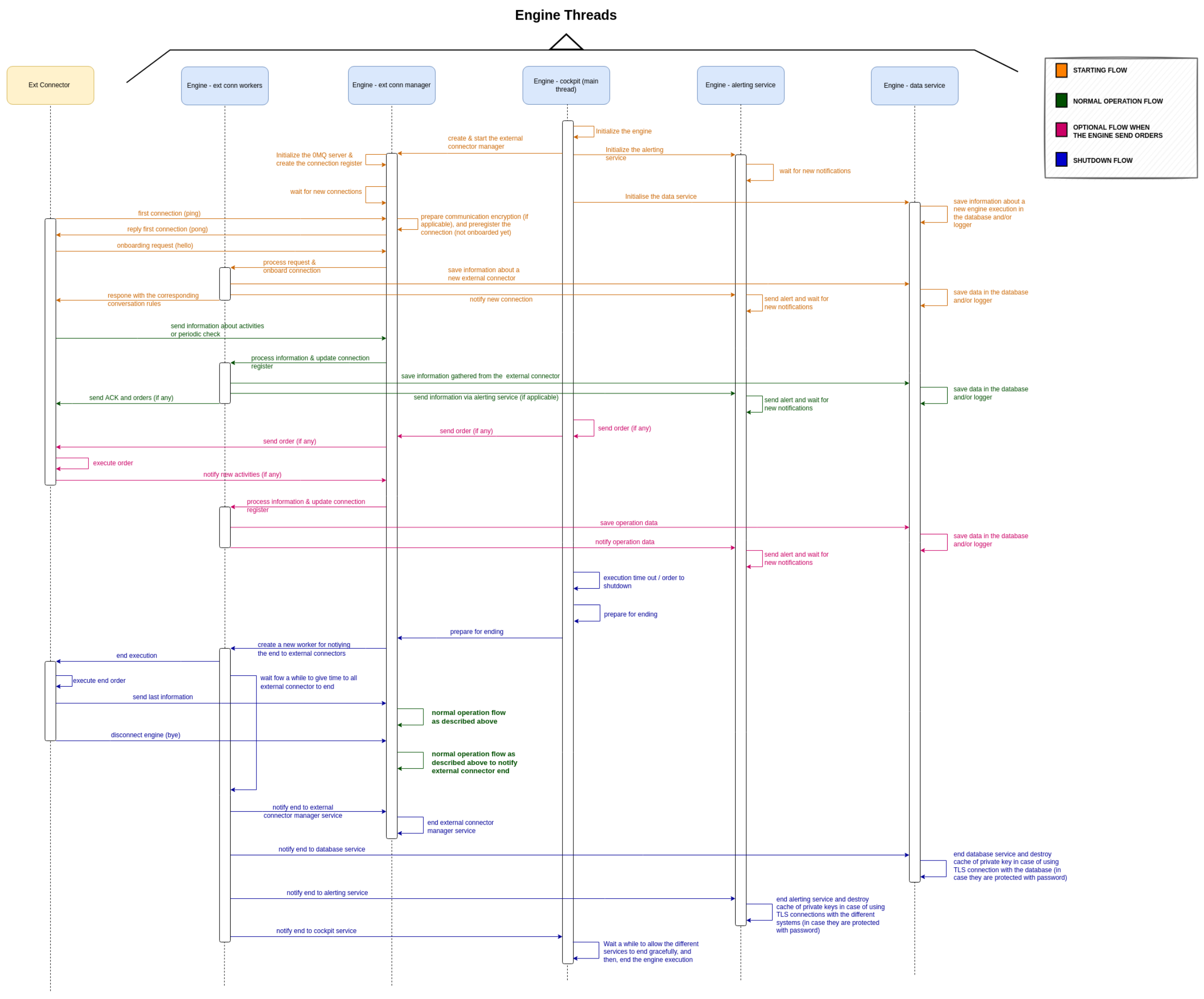

External Party
Interaction using conversation rules
External Connector
Ext.Conn ==> Engine: Get conversation rules, notify activity
Engine ==> Ext.Conn: Send orders
[ZeroMQ Connections]
Engine
Golang
Use shared memory variables
Redis
SEC LOPE DE VEGA COMPONENTS
OPTIONAL SUPPORTING SYSTEMS
OPTIONAL ALERTING SYSTEMS
EXTERNAL PARTIES
N External Parties
M External Connectors
Save Activity results in the database
MariaDB / MySQL
Send Activities to Syslog Server
Syslog Server
Send Activities to Kafka Server
Kafka
Send Activities to Email Server
Email Server
Send Activities to Http Server or Service
Http Server/Service
Send Activities to Slack
Slack

It is important to highlight that every **external connector** interacts with one target system (external party), but one external party can interact with several **external connectors**. It can happen that you have *'N'* external parties interacting with *'M'* **external connectors**. That is why every **external connector** works with a single port, protocol and mode (client or server). Therefore, an external party can interact with several ports at the same time, where each of them is managed by a different **external connector**. This allows the creation of complex scenarios where several **external connectors** can work together with different duties. In any case, all of them must be connected with the **engine** that is controlling that execution of the scenario and centralizing the data gathering.

As described previously, in case a Redis server is present, not all **external connectors** must connect with that component, and the rest of comports are optional. The minimal setup will be one external party interacting with one **external connector**, and one **engine** controlling the execution. One example of the previous scenario can be the following:, if you use *Lope* in a software development pipeline, you can start the **engine** and the **external connector**, do the test you wish against the software under development, and get the results.

1.3 Engine Software Description

This component is the main orquestrator of *Lope*, we can say it is its heart. It controls all **external connectors** that are available in one **engine** execution. Since a picture is worth a thousand words, the best to describe how the **engine** works is an image (you might need to zoom in to see the details):


Engine Threads
Ext Connector
Engine - ext conn workers
Engine - ext conn manager
Engine - cockpit (main thread)
Engine - alerting service
Engine - data service
STARTING FLOW
NORMAL OPERATION FLOW
OPTIONAL FLOW WHEN THE ENGINE SEND ORDERS
SHUTDOWN FLOW
Initialize the engine
create & start the external connector manager
Initialize the alerting service
Initialize the 0MQ server & create the connection register
wait for new notifications
wait for new connections
Initialise the data service
save information about a new engine execution in the database and/or logger
first connection (ping)
prepare communication encryption (if applicable), and preregister the connection (not onboarded yet)
reply first connection (pong)
onboarding request (hello)
process request & onboard connection
save information about a new external connector
respone with the corresponding conversation rules
notify new connection
send alert and wait for new notifications
save data in the database and/or logger
send information about activities or periodic check
process information & update connection register
save information gathered from the external connector
send information via alerting service (if applicable)
send ACK and orders (if any)
send alert and wait for new notifications
save data in the database and/or logger
send order (if any)
send order (if any)
send order (if any)
execute order
notify new activities (if any)
process information & update connection register
save operation data
save data in the database and/or logger
notify operation data
send alert and wait for new notifications
execution time out / order to shutdown
prepare for ending
prepare for ending
create a new worker for notiying the end to external connectors
end execution
execute end order
wait fow a while to give time to all external connector to end
send last information
normal operation flow as described above
disconnect engine (bye)
normal operation flow as described above to notify external connector end
notify end to external connector manager service
end external connector manager service
notify end to database service
end database service and destroy cache of private key in case of using TLS connection with the database (in case they are protected with password)
notify end to alerting service
end alerting service and destroy cache of private keys in case of using TLS connections with the different systems (in case they are protected with password)
notify end to cockpit service
Wait a while to allow the different services to end gracefully, and then, end the engine execution

As you can see, the **engine** is using different threads to leverage the power of parallel computing. The main threads are the following ones:

- **Main thread (a.k.a. engine cockpit service)**: This is the thread that controls the overall execution of this software piece. It initializes the other relevant threads and the logical components, as well as starting the process for shutting down (via interrupt signal or timeout).
- **External Connection Manager Service**: This service is the one that oversees the interaction with **external connectors**, and listens the ZeroMQ socket of the **engine**.
- **External Connection Workers**: These threads are used to process the messages received from the **external connectors** to avoid blocking the *external connection manager service* for processing received messages and activities created by the **external connectors**. Every message received create a new *external connection worker*, except the *ping* message that is replied by the *external connection manager service* directly. Additionally, one worker is created at the time of stopping the execution, as can be seen in the previous image. It orchestrates how the **external connectors** stop and send the last information they have, avoiding losing information.

- **Alerting Service**: This thread is the one that sends the activities created by *Lope* to the different systems available, and it manages the respective connection with them
- **Data Service**: This thread is the main one to storage the activities created by *Lope* and save them in the database, a simple file, or both.

Activities, or external activities, are described in detail in *Activity Alerting & Storage*, but here we can simply say that any action that the *Lope* does, any interaction with the target systems, is considered an activity. **External connectors** start and stop events are considered actions too, as well as the **engine** start and stop. Any activity is sent to the corresponding service to be stored or sent as an alert.

A minor comment about the communcation between **engine** threads for those that might be interested, they are implemented using go channels.

1.4 Engine Configuration

The main configuration file of the **engine** is the *engine_config.yml* file, but the name can be changed if the configuration file is passed as command line option (As explained in *Engine Installation & Execution*). However, the file should be always a yml file. In this file, you configure everything related the **engine** execution, but nothing about the **external connectors** neither *conversation rules* (except in which folder they are located).

1.4.1 External Connectors Connection

ZeroMQ & Encryption

Starting from the basics, the first thing you need to configure is the ZeroMQ communication for the **external connectors**. You should tell if you wish to encrypt the connection with the already encryption functionality. Depending on what you do, there are two results:

- If you enable it, the connection between the **engine** and the **external connectors** will encrypted using RSA as asymmetric algorithm to exchange the session key, and the symmetric algorithm will be ChaCha20-Poly1305. *Lope* has mutual authentication for every exchange of messages, the **engine** authenticates to the **external connectors** in this context via a cryptographic signature using RSA-PSS and SHA-512. So, the `engine_auth_code` is transformed first into a SHA 512 digest, and then, signed using RSA-PSS. This is sent to the external conenctors every time to prove they are talking with the expected engine (and they know this because this `engine_auth_code` is in their configuration file as well). The **external connectors** authenticate against the angine by providing their passwords as hashes using BLAKE2b. Since the messages are encrypted, the hash is always shared encrypted.
- If you do not enable it, then the communication is not encrypted but still there is a mutual authentication in place for the **external connectors** and the **engine**. The `engine_auth_code` then is transformed using BLAKE2b and this is sent to the **external connectors**. In this scenario, they authenticate using BLAKE2b.

Note: Keep in mind that in the second scneario (not enable the `engine_auth_code` field), the content of the messages are visible by third parties, including the authentication hashes.

Below, you can find the part of the configuration file for the networking part:

```
networking:
   # 0MQ Server
   mq:
      port: 5555
```

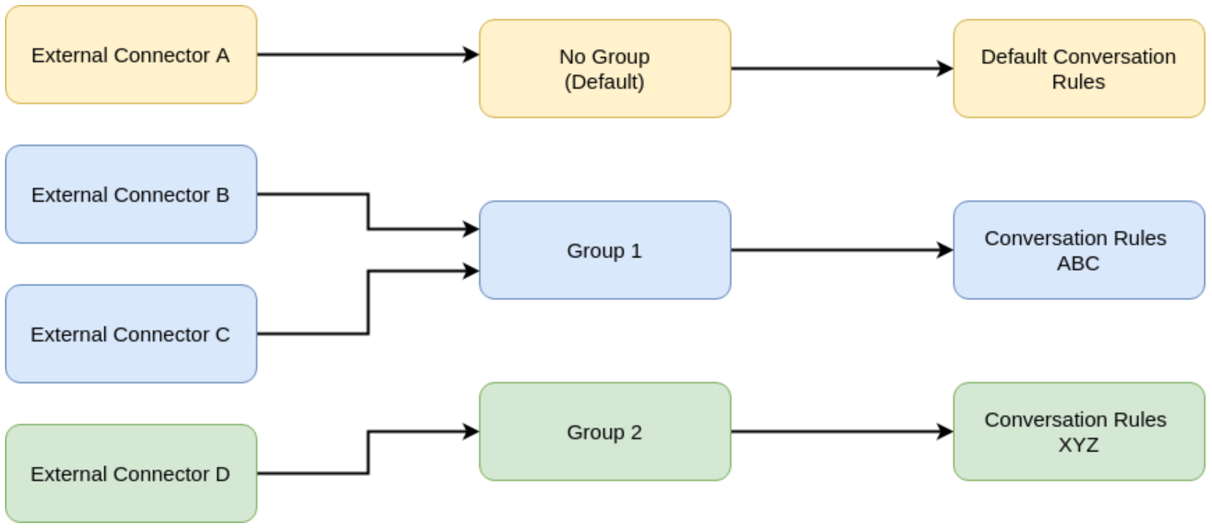
(continues on next page)

(continued from previous page)

```
      ipv6 : no # use ipv6? yes/no (default)

      # For the connections between external connectors
      # and the engine
      engine_auth_code: no # yes/no(default)

      # Minimum possible value is 2048
      rsa_key_lengt: 4096 # default value

      # for the hash code sent to external connectors
      # to ensure they are talking with right engine
      engine_auth_code: "Y0_s0y_0zym4nd14s_r3y_d3_r3y3s!"
```

External Connectors Groups & Conversation Rules

One the ZeroMQ connection is configured, then the next step is to configure the **external connectors** groups and what *conversation rules* to use. In order to deliver *conversation rules* to the **external connectors**, we need to introduce the concept of groups. All **external connectors** belong to one group. If it is not explicitly defined, then the default group is used. In the picture below you can find a grahical description of this idea.



Therefore, you can assign different *conversation rules* to different **external connectors**, according to the groups they belong. This is configured in the configuration file as follows:

```
# configuration about the external connectors,
# depending on the group they belong.
# if they belong to some specific group,
# the corresponding conversation rules are loaded
# The group is described in the conversation rules file
external_connectors:
  # not include the final '/', do not use '.'.
  # For using the current folder, use only this ""
  conv_rules_folder:
    path: "/conv_rules"
    is_relative_path: yes
```

(continues on next page)

(continued from previous page)

```
# reload the conversation files every time a new external connector
# connect with the engine, instead of using the ones loaded
# at the start of the engine (cache copy). This allows to modify
# the configuration files at engine runtime.
reload_conv_rules_in_new_contact: yes # yes/no(default)

# configuration about the number of external connectors
# and external conenctor workers (threads)
max_number_of_workers: 200 # pool size
max_number_of_external_connectors_connected: 15

# This is the password for those external connectors that
# do not belong to any group, they will use the
# 'default conversation rules'
default_secret: "3sT0_3S_s3Cr3t0!"

groups:
  # one example group
  - id: "test_client"
    list:
      - id: "ExtConTest_Client"
        secret: "3sT0_3S_s3Cr3t01!"

  # other example group
  - id: "test_server"
    list:
      - id: "ExtConTest_Server"
        secret: "3sT0_3S_s3Cr3t02!"

  # other example group
  - id: "test_neo2"
    list:
      - id: "ExtConTest4"
        secret: "3sT0_3S_s3Cr3t03!"

  # other example group
  - id: "B"
    list:
      - id: "ExtConX"
        secret: "3sT0_3S_s3Cr3t04!"

      - id: "ExtConY"
        secret: "3sT0_3S_s3Cr3t05!"
```

As you can see, you configure the list of groups, with their respective **external connectors** members with their passwords. You also configure the default password for those external connectors that do not belong to any group and they can autogenerated IDs. Additionally, you can configure where are the *conversation rules* files, the max number of **external connectors** connected at the same time, as well as the max number of external connector worker threads to use.

1.4.2 Engine Logging

Other important part to configure is the logging of the **engine**. It leverage the capabilities of zerolog library, and there are different possibilities for logging:

- **Console logging:** The logs are only showed via the terminal, and they can be *human friendly* if the `console_human_friendly` is enabled (this is applicable in any option that use the console).
- **File logging:** The logs are saved in JSON format
- **Console & file logging:** Logs are saved in a file, as well as shown via console.
- **File logging with log rotation:** In this case, the logs are rotated in several files depending on the configuration parameters.
- **Console & file with log rotation:** Logs are saved in several files as described in the previous point, as well as shown via console.

The created log files have the following name structure: `slv_engine_%Y-%m-%d-%H:%M:%S.log` (example: `2022-August-25_7:3:39.log`). But the prefix and file extension can be changed in the configuration file. If the specified log folder does not exist, it will be created.

```
logging:
  # log level, one of the following: TRACE, DEBUG, INFO (default),
  # WARN, ERROR
  level: "DEBUG"

  # log mode: CONSOLE (default), FILE, BOTH, FILE_ROTATING or
  # BOTH_ROTATING
  mode: "BOTH_ROTATING"

  # https://github.com/rs/zerolog#add-file-and-line-number-to-log
  show_logger_caller: no # yes/no (default)

  # https://github.com/rs/zerolog#pretty-logging (yes/no)
  console_human_friendly: yes

  # to create a different thread for adding information in
  # the logs at the time of logging,
  async_logs: no # (yes/no)

  # not include the final '/', do not use '.'
  # For using the current folder, use only this ""
  log_folder:
    path: "/logs"
    is_relative_path: yes

  # Log file name utils
  log_prefix: "" # if not used, it will be 'slv_engine'
  log_extension: "" # if not used, it will be '.log'

  # log rotation
  log_rotation_max_size: 20 # MB
  log_rotation_number_files: 5
  log_rotation_max_number_days: 180 # days
  log_rotation_compress: yes # yes/no(dafult)
```

1.4.3 Other Engine Configuration Options

You can configure the size of the go channels used to send message among engine threads, as well as some timing for the general time out of the **engine**, or the time to wait before stopping the execution to give some time to get the last information of the **external connectors** and process it. Here you can find how to configure those aspects:

```
# configuration about the Go channels
go_channels:
  # size of the buffer of the GO communication channels
  engine_cockpit_buffer: 30
  ext_conn_manager_buffer: 30
  ext_conn_manager_worker_channel_buffer: 30
  data_service_buffer: 30
  alerting_service_buffer: 30

# configuration about times
timing:
  # seconds (86400s = 24h), '0' means no timeout
  engine_timeout: 0 # seconds (86400s = 24h), '0' means no timeout

  # Time to wait until all ending activities are done before
  # doing the shutdown
  engine_ending_waiting_time: 15 # seconds.
```

In this file, the persistance capabilities of *Lope*, and the integration with different alerting systems is also configured. However, this is explained in *Activity Alerting & Storage*

1.5 Engine Installation & Execution

For intallting the **engine**, there are two options:

- One is using the compiled version provided in the project under the *'bin'* folder of the engine repository. This is ready to run in **linux** operating systems for AMD64 cpu architecture.
- Or compiling the source code. This option is better if you want to run it in a different operating system (OS) or cpu architecture. To do so, you should be aware that one of the libraries of this project ('zeromq/goczmq') is a wrapper of a C library, so you have to have a C compiler for the operating system (OS) and cpu architecture installed too.

1.5.1 Compiling the Project

If you want to compile the project, the first thing you need is to install GO. As mentioned in the previous paragraph, you will need the C compiler in place for the OS and cpu architecture. **Go** allows compiling the project for several different architectures and operating systems, but the mentioned library can do the things a bit harder (especially when you compile the project for a different OS and CPU architecture of your machine). If you have a message error like this at building time:

```
build constraints exclude all Go files in ../go/pkg/mod/gopkg.in/zeromq/goczmq.v4@v4.X.X
```

the cause may be the lack of the right C compiler in place.

For compiling the project, this tutorial can be of interest to you, but for this project you can do just the following (provided you are in the root folder of the project):

```
go build -o {{the place you want to put the executable file}} cmd/main.go
```

or just `make` (or `make build`)

For compiling to a different target system, you should check the following links:

- When using CGO_ENABLED is must and what happens.
- CGO_ENABLED=1 causing the '-marm' unrecognized #16801.
- cmd/go: give a better error message when building Go package with CGO_ENABLED=0.

But in general, you should take into account that you might need to specify the flags `CC`, `CXX`, and `CGO_ENABLED="1"`; apart from the `GOOS` and `GOARCH` as described in the tutorial cited previously.

1.5.2 Location of Configuration Files

The main configuration file for the **engine** is the *engine_config.yml*, and by default the engine tries to get it from `./config` (config folder in the folder where the executable file is located). However, this can be changed using some command line options, as we are going to see in the next section.

The *conversation rules* folder location is described in that configuration file, so it is up to you. This topic is described in detail in *Conversation Rules*

If you import this project in an IDE, the **engine** tries to get the configuration file from the config folder of the project. This is only possible if the `env (environment)` command line option is defined as `DEV`.

1.5.3 Launch Engine and Command Line Options

As mentioned, the **engine** can use command line flags to start. The three command line optines are:

- `-env` (environment): It tells the **engine** if the it is running in an IDE (`DEV`) or not.
- `-nobanner` (no banner): If present, the banner will not be printed at the beginning of the execution.
- `-config` (config): If present, it change the default location of the configuration file *engine_config.yml*. If this is provided, the `env` option will not be used. You can use a different name for that configuration file instead of the default one, since the path is not the location (folder) of the file, is directly the file to be used as configuration file **(but always a .yml file)**.

The default and the alternative values are defined in the following table:

Command Line Option	Default value	Alternative values
`env`	`PRO`	`DEV`
`nobanner`	(not present)	`nobanner` (present)
`config`	(not present)	`/path/to/the/configuration/file/configuration_file_name.yml`

To start the execution of the **engine**, there are two options:

- If the software is not compiled, you can compile and run it by doing `go run cmd/main.go` or just `make run` (provided that you have the environment ready for compilation as described in the previous section)
- If the software is compiled, just run it as any executable. An example using the already compile version in linux:

  ```
  bin/slv_engine_linux_amd64
  ```

or using the command line options:

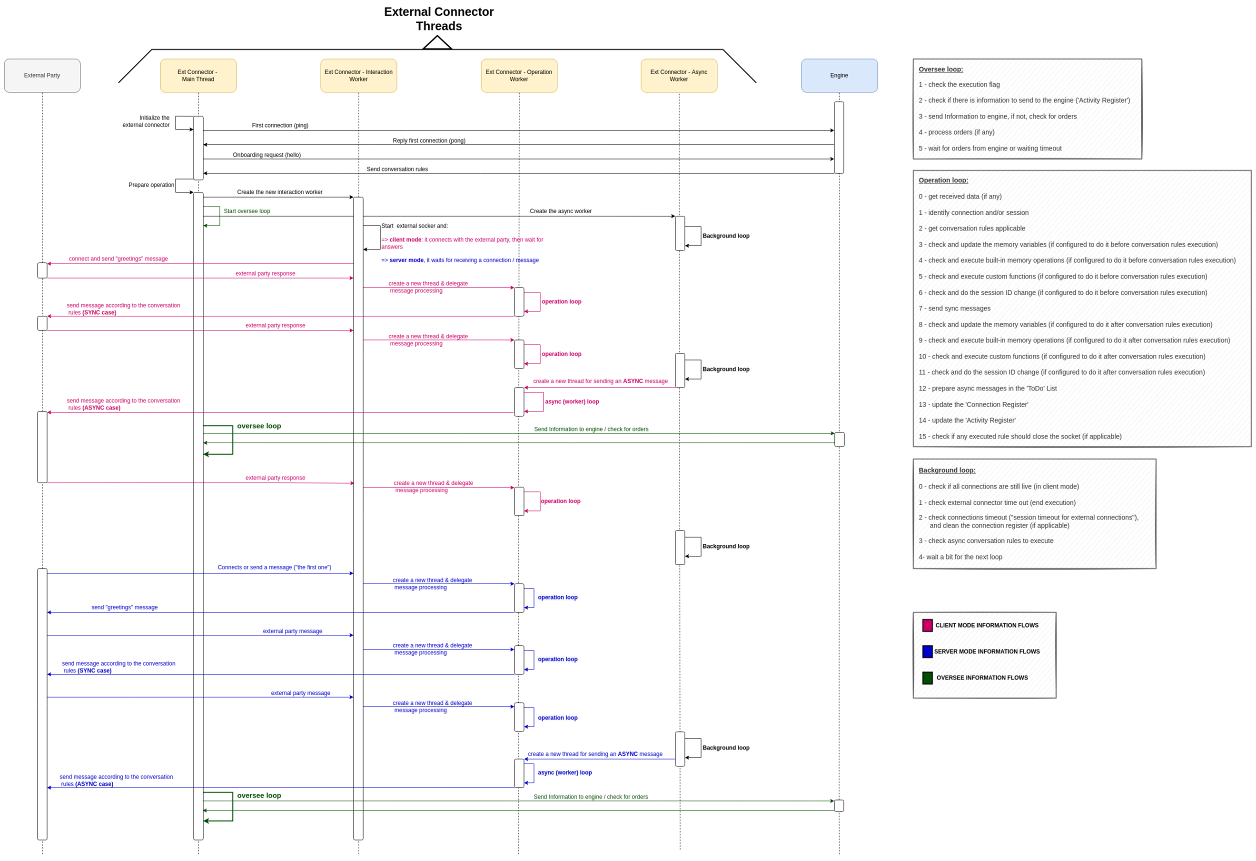
```
bin/slv_engine_linux_amd64 -config=/home/alber/SecLopeDeVega/engine/config/engine_
↪config.yml -nobanner
```

1.5.4 Database Setup

The schema of the database to be used is already provided in the *db* folder in the repository. If you do not know how to import it in your project, check this guide.

1.6 External Connector Software Description

This component is the element of *Lope* that connects with the third parties. As mention in the *Intro*, every external connector works targeting a single port, mode (client or server) and protocol (**tcp** or **udp**); using an specific set of rules for the interaction. Since *Lope* is designed to be able to work with several **external connectors** at the same time, you can run different ones under the control of the same **engine** (as described in the *Architecture*). The idea is that each **external connector** has *one job* in the interaction scenario, and there can be several of them working together. In the following image you can finde how the **external connector** internally works (you might need to zoom in to see the details):


External Connector Threads
External Party
Ext Connector - Main Thread
Ext Connector - Interaction Worker
Ext Connector - Operation Worker
Ext Connector - Async Worker
Engine
Initialize the external connector
First connection (ping)
Reply first connection (pong)
Onboarding request (hello)
Send conversation rules
Prepare operation
Create the new interaction worker
Start oversee loop
Create the async worker
Start external socker and:
=> client mode: it connects with the external party, then wait for answers
=> server mode, it waits for receiving a connection / message
Background loop
connect and send "greetings" message
external party response
create a new thread & delegate message processing
operation loop
send message according to the conversation rules (SYNC case)
create a new thread for sending an ASYNC message
async (worker) loop
send message according to the conversation rules (ASYNC case)
oversee loop
Send Information to engine / check for orders
Connects or send a message ("the first one")
send "greetings" message
external party message
CLIENT MODE INFORMATION FLOWS
SERVER MODE INFORMATION FLOWS
OVERSEE INFORMATION FLOWS

Oversee loop:

1 - check the execution flag

2 - check if there is information to send to the engine ('Activity Register')

3 - send Information to engine, if not, check for orders

4 - process orders (if any)

5 - wait for orders from engine or waiting timeout

Operation loop:

0 - get received data (if any)

1 - identify connection and/or session

2 - get conversation rules applicable

3 - check and update the memory variables (if configured to do it before conversation rules execution)

4 - check and execute built-in memory operations (if configured to do it before conversation rules execution)

5 - check and execute custom functions (if configured to do it before conversation rules execution)

6 - check and do the session ID change (if configured to do it before conversation rules execution)

7 - send sync messages

8 - check and update the memory variables (if configured to do it after conversation rules execution)

9 - check and execute built-in memory operations (if configured to do it after conversation rules execution)

10 - check and execute custom functions (if configured to do it after conversation rules execution)

11 - check and do the session ID change (if configured to do it after conversation rules execution)

12 - prepare async messages in the 'ToDo' List

13 - update the 'Connection Register'

14 - update the 'Activity Register'

15 - check if any executed rule should close the socket (if applicable)

Background loop:

0 - check if all connections are still live (in client mode)

1 - check external connector time out (end execution)

2 - check connections timeout ("session timeout for external connections"), and clean the connection register (if applicable)

3 - check async conversation rules to execute

4- wait a bit for the next loop

As you can see, there are different threads working at the same time:

- **Main Thread**: this thread is the one that starts, initializes and stops the execution. It executes the *oversee loop* where the connection with the **engine** takes place. It also executes any order provided by the **engine**, like the *'end'* order to stop the execution.

- **Interaction Worker**: this thread is the one that listens the open sockets of the **external connector**. If something is recevied, it creates an *operation worker* for managing it.
- **Operation Worker**: this kind of threads are created every time something is received from the third parties to execute the corresponding *conversation rules*. It executes the *Operation loop*, but not all steps are always executed. Depending on the context and the configuration, some of them are skipped.
- **Async Worker**: this thread is the one that controls the timeouts of the **external connection** and its connections, as well as the *async taks (conversation rules)* to exectue. It operates the *backgroud loop* that allows to do those activities.

In the previous image, there are two registers mentioned in the leyend. They are internal registers that are used to control the **external connector** execution:

- *Connection Register*: This allows to control the status of each connection established with third parties (active or not, the *memory variables*, etc.).
- *Activity Register*: This collects any event or *conversation rule* executed for every connection, saving it as an *activity*. The *main thread* checks if there are activities periodically, to send them to the **engine**. These activities, or external activities, are described in detail in *Activity Alerting & Storage*.

Other relevant information about the **external connector** is that it will try 3 times to send a message to the **engine**. In case it does not reply, in any message to send to the **engine**. Additionally, the **external connector** will try to initialize the execution 3 times as well, in case it is not able to do it. This can result that there are 9 attempts to contact the **engine**, if the initialization cannot happen if the engine is not responding.

1.7 External Connector Configuration

The main configuration file of the **external connector** is the *ext_conn_config.ini* file, but the name can be changed if the configuration file is passed as command line option (as explained in *External Connector Installation & Execution*). Just keep in mind that the file should be always a ini file. In this file you can configure those aspects that are relevant for only the **external connector** such as how to connect with the **engine**, the use of certificates for TLS support, the logs, different times, the connection with the Redis server (its role is explained in *Architecture*), and other operation configuration.

1.7.1 Engine Connection

The first thing you have to configure is where to find the engine:

```
# ==========================================================================
[NETWORKING]
# ==========================================================================
# ZEROMQ
# IP or domain
ENGINE_IP = 127.0.0.1
ENGINE_PORT = 5555
```

1.7.2 External Connector Logging

The next interesting section to configure is the logs of the **external connector**. In the same way that the **engine** can generate logs, there are several possibilities:

- **Console logging:** The logs are only showed via the terminal.
- **File logging:** The logs are saved in a single file.
- **Console & file logging:** Logs are saved in a file, as well as shown via console.
- **File logging with log rotation:** In this case, the logs are rotated in several files depending on the configuration parameters.
- **Console & file with log rotation:** Logs are saved in several files as described in the previous point, as well as shown via console.

The created log files have the following name structure: `slv_ext_conn_%Y_%m_%d-%H_%M_%S.log` (example: `slv_ext_conn_2022_09_21-08_29_11.log`). But the prefix and file extension can be changed in the configuration file. If the specified log folder does not exist, it will be created.

```
# ==========================================================================
[LOGGING]
# ==========================================================================
# LEVEL: one of the following: DEBUG, INFO (default), WARNING, ERROR, CRITICAL
LEVEL = DEBUG
# LOG_Folder: not include the final '/', for local folder use '.'
# It creates the the folder if it does not exist
LOG_FOLDER = ./logs
# LOG_MODE: CONSOLE (default), FILE, FILE_ROTATION, BOTH, BOTH_ROTATION
LOG_MODE =  CONSOLE
# Log rotation parameters, by default the max size is 20 MB and 5 files
LOG_ROTATION_MAX_SIZE = 20
LOG_ROTATION_FILE_NUMBERS = 5
# Log file name utils. if not used, LOG_PREFIX = "slv_ext_conn_" and LOG_EXTENSION = ".
↪log"
LOG_PREFIX =
LOG_EXTENSION =
```

1.7.3 External Connector Timing

For the **external connector**, timing is an important aspect. Depending on the context, the different values should be adapted.

```
# ==========================================================================
[TIMES]
# ==========================================================================
# Timeout for the connection with the engine
# 5s by default
ENGINE_TIME_OUT = 5

# Timeout (seconds) for the external connector (lifespan),
# 'NONE' for not using a timeout
# by default = 1 hour = 3600 seconds
EXT_CONNECTOR_TIME_OUT = NONE
```

(continues on next page)

(continued from previous page)

```
# Time (seconds) for main thread operation loop without contacting
# the engine in case if there is not
# information to send to the engine. By default = 5 seconds.
# If this value is less than the
# ENGINE_TIME_OUT, then this last value is used
# for the loop waiting time
MAIN_THREAD_CHECKING_TIME = 5

# Time (float-seconds) between loops when the external connector
# wait for interaction The default value is 0.5 seconds
TIME_BETWEEN_INTERACTION_LOOPS = 0.5

# Time (float seconds) between loops for the async thread.
# The default value us 0.1 seconds
TIME_BETWEEN_ASYNC_LOOPS = 0.10

# Number of async loops to clean the connection register.
# It results in a time value depending on
# the time between async loops
NUMBER_ASYNC_LOOPS_TO_CLEAN_CONN_REGISTER = 10000

# Waiting time (int-seconds) before restarting
# the external connector according to 'restart' order
RESTARTING_WAITING_TIME = 60

# Time (float-seconds) between connections of
# client sockets during starting-up (by deafult = 0.5s)
TIME_BETWEEN_CLIENT_SOCKET_CONNECTIONS = 0.5

# Time (float-seconds) between closing a tcp connection
# and connecting again for tcp client mode (by deafult = 0.5s)
# Or initial delay for having enough time to do the
# TLS handhsake in the reconnection
TIME_BETWEEN_CLIENT_SOCKET_CLOSE_CONNECT = 1

# Time (float-seconds) before allowing the async tasks in
# client mode checks if no active connections are alive (by deafult = 3s)
# Recommended: 5s or more for several TLS or DTLS connections.
TIME_INITIAL_DELAY_ASYNC_THREAD = 5
```

1.7.4 External Connector Operation

The following parameters cover other operational aspects of the **external connector**. The first two are the credentials to authenticate the **external connector** against the **engine**, but if other values will replace them if they are provided via command line options, as described in *External Connector Installation & Execution*. The third one is for being used to authenticate the **engine** against the **external connector**, as mentioned in *Engine Installation & Execution*.

The last two parameters allow to find the custom code that you can add to customize even further the interaction. This is described in detail in *Custom Functions*.

```
# ==========================================================================
[OPERATION]
# ==========================================================================

# external connector ID, by default is autogenerated (AUTO, or empty)
EXTERNAL_CONNECTOR_ID = ExtConTest

# API key for connecting with the engine
EXTERNAL_CONNECTOR_SECRET = 3sT0_3S_s3Cr3t0

# For the hash code sent to external connectors from
# the engine as authentication proof
ENGINE_AUTH_CODE = Y0_s0y_0zym4nd14s_r3y_d3_r3y3s

# Custom functions directory. The software is going to
# look for a python module called with the given name
# ('slv_custom_functions[.py]' in this case). If it is not found,
# no custom functions will be executed. Please do not
# add the extension .py in the name
CUSTOM_FUNCTIONS_DIRECTORY = ../custom_functions/
CUSTOM_FUNCTIONS_MODULE_NAME = slv_custom_functions

# Number of subworkers to divide the task of checking all the regex of custom
# rules for one input. This only is used when the external connector does not
# work in 'only first hit' scenario
NUMBER_RULE_CHECKER_SUBWORKERS = 10
```

1.7.5 TLS/DTLS Support

In case the **external connector** needs to use TLS for **tcp**, or DTLS for **udp**, this will be the place for configuring it. The support of TLS includes the client authentication, as well as the case the private key is encrypted; but always using material created as OpenSSL does.

It is important to mention that for the DTLS case and if the private key is protected, a temporary file will created with the unencrypted key for the exection and deleted once the execution ends.

The following links can help you to configure the TLS or DTLS correctly for the **external connector**:

- TLS support in Python offical documentation.
- OpenSSL tutorial from Jamie Nguyen.
- Certificate chains in Python offical documentation.
- OpenSSL cheat sheet from Cédric Lauradoux.
- Problems using TLS in Python (ssl.SSLCertVerificationError).
- Working with OpenSSL and DNS alternative names.
- Key usage extensions and extended key usage.
- How to create an HTTPS certificate for localhost domains.

```
# ==========================================================================
[D/TLS]
```

(continues on next page)

(continued from previous page)

```
# ==========================================================================
# https://docs.python.org/3/library/ssl.html#module-ssl

# Custom CA:
# - in server mode, it is the CA for validating client certificates
# - in client mode, it is the CA for validating the server certificate if default
#   settings are not used
CLIENT_CABUNDLE_PEM = ../tls/ca/certs/ca.crt.pem

# Is the key of the certificate to use protected with a password? YES (default) / NO
KEY_PROTECTED = YES

# ---------------------
# TLS SERVER MODE
# ---------------------
# https://docs.python.org/3/library/ssl.html#server-side-operation
SERVER_CERTCHAIN_PEM = ../tls/ca/intermediate/certs/ca-chain_server.crt.pem
SERVER_PRIV_KEY_PEM = ../tls/ca/intermediate/private/seclopedevega_server.key.pem
SERVER_KEY_PASSWORD = lope

# ---------------------
# TLS CLIENT MODE
# ---------------------
# https://docs.python.org/3/library/ssl.html#client-side-operation
# Use Machine CAs: YES (default) / NO
CLIENT_DEFAULT_SETTINGS = NO
# Validate Server certificate: YES (default) / NO
CLIENT_VALIDATE_SERVER_CERTIFICATE = YES
CLIENT_CERTCHAIN_PEM = ../tls/ca/intermediate/certs/ca-chain_client.crt.pem
CLIENT_PRIV_KEY_PEM = ../tls/ca/intermediate/private/seclopedevega_client.key.pem
CLIENT_KEY_PASSWORD = lope

# ---------------------
# DTLS MODE
# ---------------------
#https://github.com/mcfreis/pydtls
# One of the following: "DTLS" or "DTLSv1.2" (default)
DTLS_VERSION = DTLSv1.2
```

1.7.6 Redis Integration

As described in *Architecture*, the Redis server is an important element for allowing to share information among different **external connectors**, using memory variables. These variables are explained in detail in *Session Support & Memory Variables*, here we only address what is the configuration capabilities to use Redis.

he following links can help you to install and use Redis, as well as understaning the integration between the external connector and Redis:

- How to Use Redis With Python.
- How To Install and Secure Redis on Ubuntu 20.04.
- Redis TLS support.

- Redis-py TLS Connection Examples.

```
# ==========================================================================
[REDIS] # Multi external connector memory
# ==========================================================================
# IP or domain
REDIS_IP = 127.0.0.1

REDIS_PORT = 6379

REDIS_PASSWORD = 3st33s3lB4uLD3L4sV4r14Bl3sC0mP4Rt1D4s

REDIS_TLS = NO

# use certificate for authenticate against the redis server?
REDIS_USE_CLIENT_CERTIFICATE = YES
REDIS_CLIENT_CERTIFICATE = ../tls/ca/intermediate/certs/ca-chain_client.crt.pem

# Private Key fields for the certificate
REDIS_PRIV_KEY_CLIENT_PROTECTED = YES
REDIS_PRIV_KEY_PASSWORD =  lope
REDIS_PRIV_KEY_CLIENT = ../tls/ca/intermediate/private/seclopedevega_client.key.pem

# Machine CAs in linux: /etc/ssl/certs/ca-certificates.crt
REDIS_CA_CERT = ../tls/ca/certs/ca.crt.pem
```

1.8 External Connector Installation & Execution

Since this component is in Python, you should have Python 3 installed in the environment yo want to run it. Additionally, OpenSSL is also required. `libzmq3-dev` is recommended in some cases, as explained in the pyzmq repository ,if you want to install the dependencies compiling from source.

To have everything ready, you should install the dependencies as usual for any Python project. We recommend to use virtual environments, but it is up to you how to install the dependencies. In case you do not know how to do it, just check the offical Python documentation.

Since a `requirements.txt` is provided, you just can install the dependencies doing this:

```
python3 -m pip install -r requirements.txt
```

In this part of the official documentation you can find more information about how to use the `requirements.txt` file.

1.8.1 Location of Configuration Files

The main configuration file for the **external connector** is the *ext_conn_config.ini*, and by default the engine tries to get it from folder where the *start.py* file is located). However, this can be changed using some command line options, as we are going to see in the next section.

1.8.2 Launch External Connector and Command Line Options

The **external connector** can use command line flags to start too. The four command line options are:

- `--nobanner` or `-NB` (no banner): If present, the banner will not be printed at the beginning of the execution.
- `--config` or `-CFG` (config): If present, it change the default location of the configuration file *ext_conn_config.ini*. You can use a different name for that configuration file instead of the default one, since the path is not the location (folder) of the file, is directly the file to be used as configuration file **(but always a .ini file)**.
- `--ID` or `-ID` (**External Connector** ID): The service account ID to use to authenticate against the **engine**
- `--password` or `-PWD` (**External Connector** password): The password of the service account ID to use to authenticate against the **engine**

The default and the alternative values are defined in the following table:

Command Line Option	Default value	Alternative value
`--nobanner` or `-NB`	(not present)	`nobanner` (present)
`--config` or `-CFG`	(not present)	`/path/to/the/configuration/file/configuration_file_name.ini`
`--ID` or `-ID`	(not present)	`any id`
`--password` or `-PWD`	(not present)	`any password`

To start the execution of the **external connector**, just run it as any python script. An example:

```
python3 start.py
```

or using the command line options:

```
python3 start.py -NB -ID=lope
```

Note: Please, take into account that **external connectors** need to contact the **engine** to run. Otherwise, they will stop the execution since they cannot get the *conversation rules*.

1.9 Conversation Rules

The *conversation rules* are the rules that control how the interaction between the **external connectors** and the external parties. This allows to use the same software to interact differently according to some configuration files. These rules have been slight introduced in the *Intro* and how the **external connectors** are assigned to different *conversation rules* is already explained in *Engine Configuration*. In this section, we are going to explain more the basics of the *conversation rules*.

1.9.1 Conversation Rules File

As described in *Engine Configuration*, the folder that contains the *conversation rules* files have to be defined in the **engine** configuration file. The *conversation rules* files are yml files as it is the **engine** configuration file. The basic structure of these files is as follows:

```
# ====================================================
# EXAMPLE PROTOCOL
# Author: Alberto Dominguez
# ====================================================
# this is just a field just for 'operators', it is not relevant for operation
name: example

# Which external connector group is assigned for this rules.
#  If empty, or 'default' ==> no group (only one file valid without group)
ext_conn_group: B

# This enables the use of this conversation rules.
# This means that you can have some conversation rules files
# that are in the folder, but they are deactivated and then,
# they will no be sent to any external connector.
enable_use: no # yes/no (default).

# -----------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# -----------------------------------------------------------
operation: ... # to be explained later

# -----------------------------------------------------------
# Execution memory
# -----------------------------------------------------------
memory_variables: ... # to be explained in
                      # 'Memory Variables & Operations'

# -----------------------------------------------------------
# Conversation Rules
# -----------------------------------------------------------
conversation: ... # to be explained later
```

Operation Section of Conversation Rules Files

The operation field presented previously has many subfields. The first of them are general one, the following two is for enabling the use of TLS or DTLS (provided that the configuration is already addressed in the configuration of the corresponmding **external connector**); and finally some fields for the capability of closing/reconnecting the socket connection but not the logical connection.

This allows to interact in some cases where the interaction happens not only one single **tcp** connection, and the socket connection is closed and renewed several times. For instance, the interaction with an HTTP API. *Lope* is able to work in two ways:

- Closing the socket connection every time it answers to an input, considering the answering not only the execution of a syncrhonous rule, it is also all the asyncrhonous tasks that are triggered as result.

- Closing the socket connection only in those cases where it is declared in a specific rule.

As previously said, the **external connector** keep the 'logical' connection open, until it reachs the end or it is timed out. You can consider the socket connection as part of the 'logical' connection entity in that regard.

```
# ----------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# ----------------------------------------------------------
operation:

    # GENERAL ASPECTS
    # ===============
    mode: server # server(default)/client

    # In server mode: Origin IP from external connections,
    # empty means any.
    # In client mode: what ip to connect (o domain)
    ip: ""

    # TCP(default)/UDP
    transport_protocol: TCP

    # IPv6 support
    use_ip6: no # yes/no(default)

    # Network port to use
    port: 1883

    # A value of the following list:
    # 'utf-8' (default), 'utf-16', 'utf-32', 'ascii'
    encoding: utf-8

    # Recommendation: Disable this if the tcp
    # connection will be closed after answering
    use_port_for_connection_identification: no # yes/no(default).

    # Max time to consider that a connection with a
    # third party should be timed out.
    # 0 (or negatives values) means no timeout for connections
    interaction_timeout: 300 # seconds,

    # number of unaccepted connections that the
    # system will allow before refusing new connections
    # Recommendation: use same value than number of
    # concurrent connections or larger
    connection_queue: 5 # (default = 5)

    # Max number of bytes accepted as input
    max_input_size: 1024 # default = 1024

    # Max number of concurrent connections allowed,
    # in client mode it will be the number of connections
    # established with the target
```

(continues on next page)

(continued from previous page)

```
    max_concurrent_connection: 5 # (default = 5)

    # Time before starting the execution once the external
    # connector has everything ready to run
    execution_delay: 3 # seconds

    # This means that only the first rule detected if
    # several conversation rules are applicable.
    # The rule with the smallest ID is the one executed.
    # This changes the way of how the external connector
    # checks the regex of custom rules. Instead of being parallel,
    # now it is sequential to ensure that only the first hit
    # is used.
    conversation_use_only_first_hit: no # yes/no(default)


    # TO ENABLE TLS/DTLS USE
    # ======================
    # The external connector TLS configuration should be
    # already in place before enabling the use of encryption
    # for the interaction.

    # This enables TLS or DTLS
    encrypted: no # yes/no (default).

    # This enable TLS authentication
    tls_client_authentication: no # yes/no (default).


    # SOCKET CONNECTION CLOSE
    # =======================
    # It is only aplicable for tcp, tls or dtls.

    # It closes the socket connection every time after answering / replying
    # It disables the functionality 'enable_rule_triggered_closing_socket_connection'.
    close_socket_connection_after_replying: no # yes/no(default)

    # it closes the socket but it not ends the logical
    # conection when the mode of close after answering
    # is not enable and a rule with the 'closing' field is enable
    enable_rule_triggered_closing_socket_connection: yes # yes/no(default)


    # OTHER TOPICS
    # ============
    # Additional aspects should be defined here, but
    # they will be described in the respective sections
    # of this documentation, for the sake of clarity
    ...
```

Conversation Section of Conversation Rules Files

This section contains the rules that steer the interaction. As described in the *Intro*, there are three different of *conversation rules*: the basic one and the custom ones. The last category means that they are defined by the operator (the user) of *Lope*. As described in the *Intro*, they can be syncrhonous, asyncrhonous or hybrid rules.

In this section of the file, the basic rules listed ready to be configured as well the corresponding subsection for defining the custom rules.

```
# -----------------------------------------------------------
# Conversation Rules
# -----------------------------------------------------------
conversation:

    greetings:... # Basic Rule

    default:... # Basic Rule

    empty:... # Basic Rule

    timeout:... # Basic Rule

    ending:... # Basic Rule

    custom_rules:... # Set of Custom Rules
```

1.9.2 Basic Rules

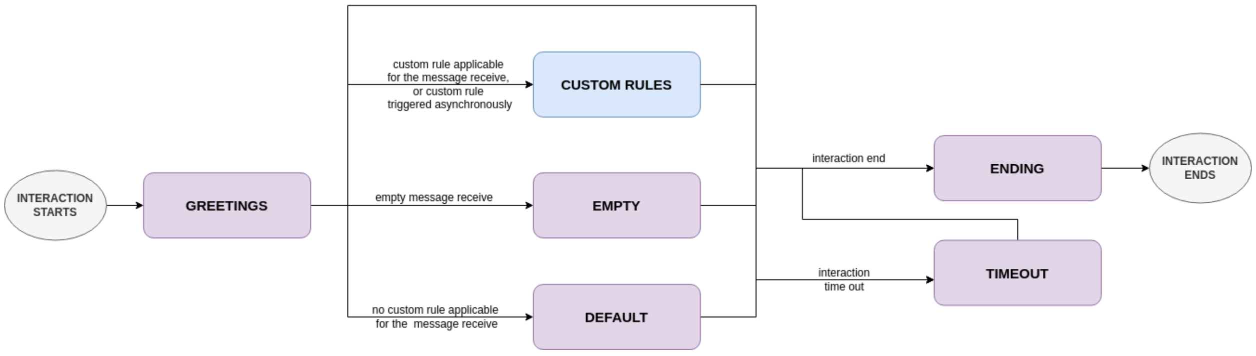
The basic rules are rules that are executed in specific events in the lifecyecle of the interaction. All the basic rules are the same in terms of configuration, they have the same fields:

```
any_basic_rule:
    # What to send to the third parties
    # when the rule is executed
    # If empty or not present,
    # the rule is considered
    # 'executed' without sending
    # anything to the third party
    value: [message to be sent]

    # If the content of the field 'value'
    # is encoded in base64
    b64_flag: no # yes/no(default)

    # If the rule is enabled and
    # can be used for the interaction
    enable: yes # yes/no(default)

    # Other fields related to alerting
    # that will be expalined in
    # 'Activity Alerting & Storage'
    ...
```

The basic rules are executeded in specifc times, understanding 'executed' as sending the content of the field 'value' to the third party.

- `greetings`: The first message sent to the external party in the moment the interaction (session) is established for the first time.
- `default`: This rule is executed if a message is received from an external party, and there is not any custom rule that is applicable.
- `empty`: This rule is executed if a message is received from an external party, but it is an empty message.
- `timeout`: This rule is executed if the interaction (session) is timed out.
- `ending`: This rule is executed if the interaction (session) is ended.

The following diagram represents how the different rules are executed:


INTERACTION STARTS
GREETINGS
custom rule applicable for the message receive, or custom rule triggered asynchronously
CUSTOM RULES
empty message receive
EMPTY
no custom rule applicable for the message receive
DEFAULT
interaction end
ENDING
interaction time out
TIMEOUT
INTERACTION ENDS

1.9.3 Custom Rules

The custom rules are those that are defined by the operator (user) of *Lope*. They are defined in the `custom_rules` field of the `conversation` field. They can be grouped and this is explained in *Advance Conversation Rules*, so here we only will explain the basics about the custom rules.

```
# -----------------------------------------------------------
# Conversation Rules
# -----------------------------------------------------------
conversation:

    greetings:... # Basic Rule

    default:... # Basic Rule

    empty:... # Basic Rule

    timeout:... # Basic Rule

    ending:... # Basic Rule

    custom_rules: # Set of custom rules

        # GROUPED RULES
        # ------------
        groups: ... # To be explained in the
                    # 'Advance Conversation Rules' Section
```

(continues on next page)

(continued from previous page)

```
        # NON-GROUPED RULES
        # -----------------
        rules: ... # list of custom rules that do not
                   # belong to any group
```

The custom rules have the following common structure, regardless the kind of rules they are:

```
any_custom_rule:

    # GENERAL FIELDS
    # ==============
    # An id number. It is not autogenerated,
    # it should be added manually,
    # and it should be unique
    id: 101

    # The kind of rule, one of the following:
    # sync, async or hybrid
    mode: sync

    # What to send to the third parties
    # when the rule is executed.
    # If empty or not present,
    # the rule is considered
    # 'executed' without sending
    # anything to the third party
    response: [message to be sent]

    # If the content of the field 'response'
    # is encoded in base64
    response_b64: no # yes/no(default)

    # If the rule is enabled and
    # can be used for the interaction
    enable: yes # yes/no(default)

    # SOCKET CLOSE SCENARIOS
    # ======================
    # For TCP & DTLS connections (and server mode),
    # it closes the socket but it not ends the logical
    # connection (interaction) when the mode of close
    # after answering is not enable.
    # The field' enable_rule_triggered_closing_socket_connection'
    # should be enable to use this field
    # (in the 'operation' section)
    closing_connection: no # yes/no (default)

    # Only aplicable for tcp transport protocol
    # or DTLS & client mode. It reconnects the
    # socket before execute the rule.The field
    # 'enable_rule_triggered_closing_socket_connection'
```

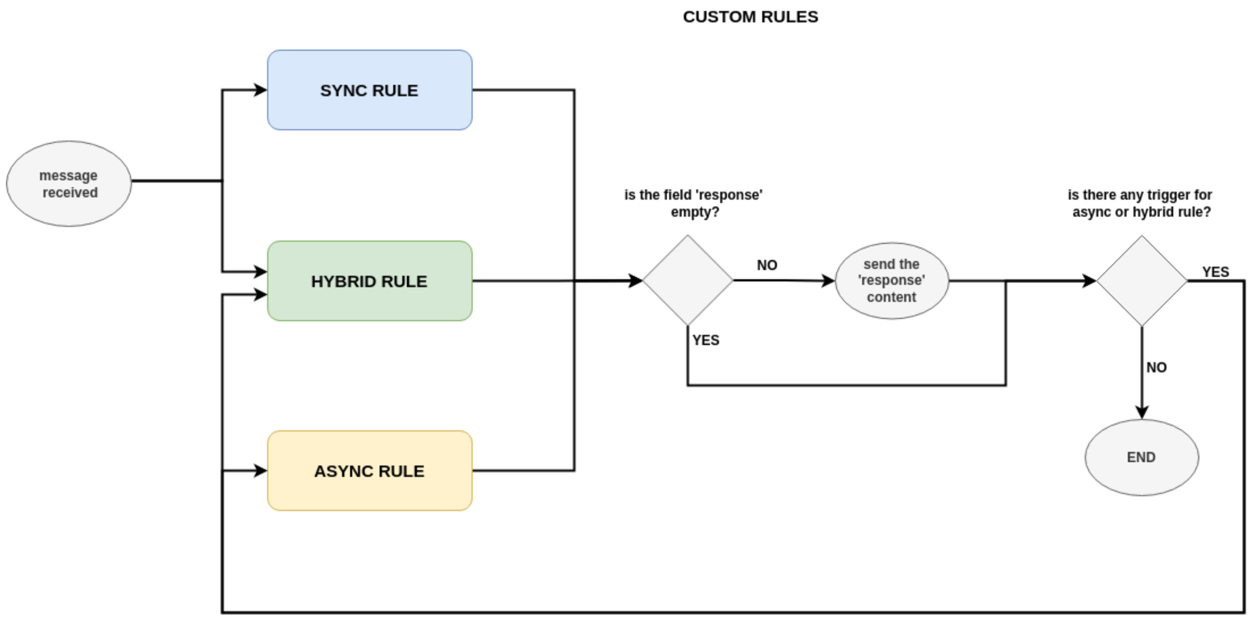
(continues on next page)

(continued from previous page)

```
    # should be enable to use this field
    # (in the 'operation' section)
    reconnect_before_rule_execution: no # yes/no(default)

    # OTHER FIELDS
    # ============

    # this field triggers the interaction ending
    ending_rule: no # yes/no (default)

    # Other fields expalined in this documentation
    ...
```

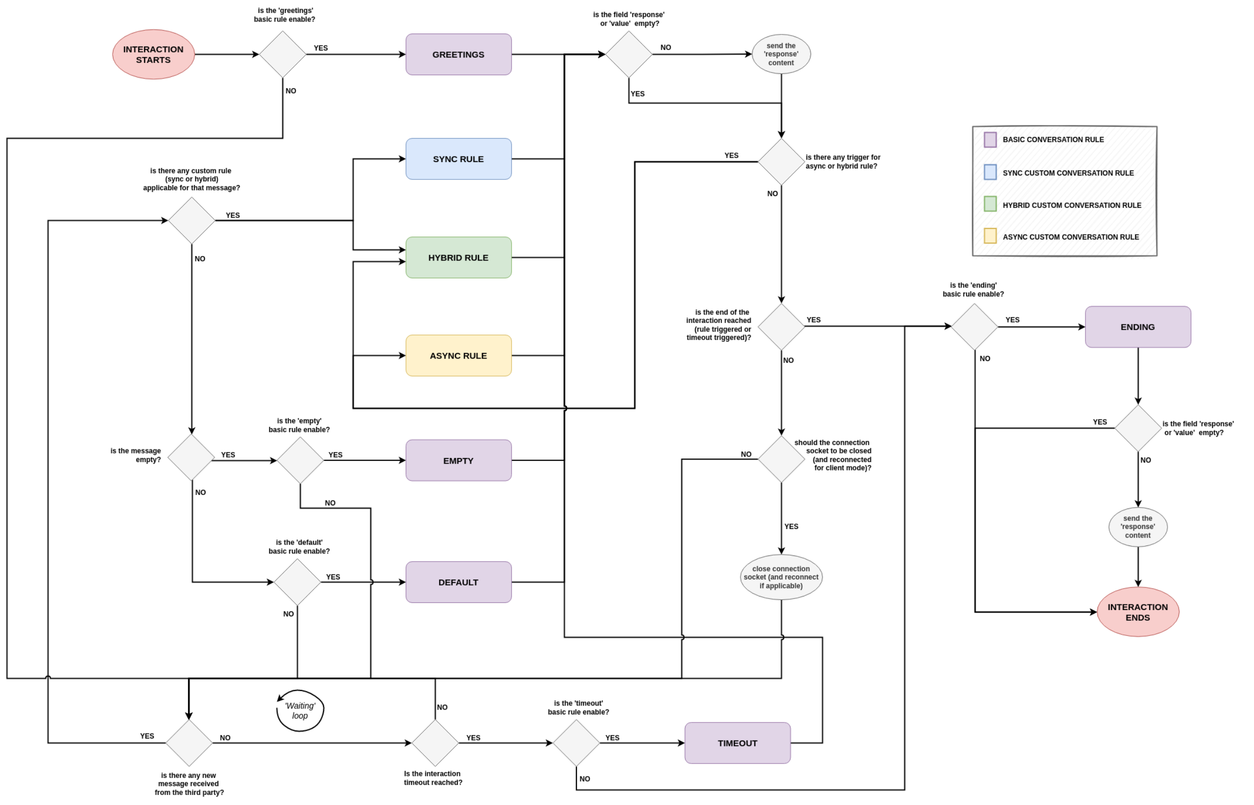
Note that not all fields should be present in all the rules, only when you need you should add it. The mandatory fields are: `id`, and `mode`.

In order to understand how the different custom *conversation rules* can work together, the following diagram represents how the complete process work of replying to any message received from third parties.


CUSTOM RULES
message received
SYNC RULE
HYBRID RULE
ASYNC RULE
is the field 'response' empty?
NO
YES
send the 'response' content
is there any trigger for async or hybrid rule?
YES
NO
END

The Interaction Flow and Conversation Rules

Once that we have seen the basic *conversation rules*, and the custom *conversation rules*; we can combine both diagrams in one to see the interaction as a process flow of *conversation rules*. In this diagram, the concepts of if there is something to reply or not, the socket closing (and reconnection), and the option to enable or disable basic rules are also included as well.


INTERACTION STARTS
is the 'greetings' basic rule enable?
YES
NO
GREETINGS
is the field 'response' or 'value' empty?
NO
YES
send the 'response' content
SYNC RULE
is there any custom rule (sync or hybrid) applicable for that message?
YES
NO
HYBRID RULE
ASYNC RULE
is there any trigger for async or hybrid rule?
YES
NO
BASIC CONVERSATION RULE
SYNC CUSTOM CONVERSATION RULE
HYBRID CUSTOM CONVERSATION RULE
ASYNC CUSTOM CONVERSATION RULE
is the end of the interaction reached (rule triggered or timeout triggered)?
YES
NO
is the 'ending' basic rule enable?
YES
NO
ENDING
is the field 'response' or 'value' empty?
YES
NO
send the 'response' content
INTERACTION ENDS
is the message empty?
YES
NO
is the 'empty' basic rule enable?
YES
NO
EMPTY
is the 'default' basic rule enable?
YES
NO
DEFAULT
should the connection socket to be closed (and reconnected for client mode)?
NO
YES
close connection socket (and reconnect if applicable)
'Waiting' loop
is there any new message received from the third party?
YES
NO
Is the interaction timeout reached?
NO
YES
is the 'timeout' basic rule enable?
YES
NO
TIMEOUT

1.9.4 Synchronous Rules

The custom syncrhonous rules are those that are applicable when their RegEx is applicable for a given input. It could happen that several of them are applicable, and they rest of them or only the first one will executed depeneding on the field `conversation_use_only_first_hit` of the `operation` section.

```
any_custom_sync_rule:

    # GENERAL FIELDS
    # ==============
    ...
    mode: sync

    # SYNC FIELDS
    # ===========
    # RegEx that make the rule applicable
    regex: [any regex]

    # Just in case the 'regex' field is
    # encoded in base 64
```

(continues on next page)

(continued from previous page)

```
    regex_b64: no # yes/no (default)

    # OTHER FIELDS
    # ============
    # Other fields expalined in this documentation
    ...
```

1.9.5 Asynchronous and Hybrid Rules

These rules are those that can be executed without any input, they are triggered by other rules. This happens because the triggering (custom) rule has the field `trigger` where you list all of the async or hybrid rules that should be triggered, as well as if it should wait a certain period of time before being executed.

However, some async rules can be triggered from the beginning, and they have specific fields to be marked as an async rule to be called from the start of the interaction: `beginning_async_rule` and `beginning_async_delay`.

The following example is the fields for a triggering rule, and async rule only needs to have the `mode` field as `async`.

Async Rules

```
any_triggering_rule:

    # GENERAL FIELDS
    # ==============
    ...

    # ASYNC TRIGGERING FIELDS
    # =======================
    trigger:

          # One Async rule to execute
        - rule_id: 3 # Async ID rule to execute
          # Time to wait until executing the rule
          # descibed in the 'rule trigger' field
          delay: 30 # seconds

          # Another async rule to execute,
          # In this case, without delay
        - rule_id: 5

    # OTHER FIELDS
    # ============
    # Other fields expalined in this documentation
    ...
```

And this is an example of an async rule to be executed from the beginning of the interaction:

```
any_async_rule:

    # GENERAL FIELDS
    # ==============
```

(continues on next page)

(continued from previous page)

```
    ...
    mode: async

    # ASYNC FIELDS FOR BEGINNING ASYNC RULES
    # =====================================
    # mark this rules as an async rule to execute
    # from the beginning of the interaction
    beginning_async_rule: yes # yes/no (default)

    # time to wait from the beginning of the interaction
    # until the rule is executed
    beginning_async_delay: 30 # seconds

    # OTHER FIELDS
    # ============
    # Other fields expalined in this documentation
    ...
```

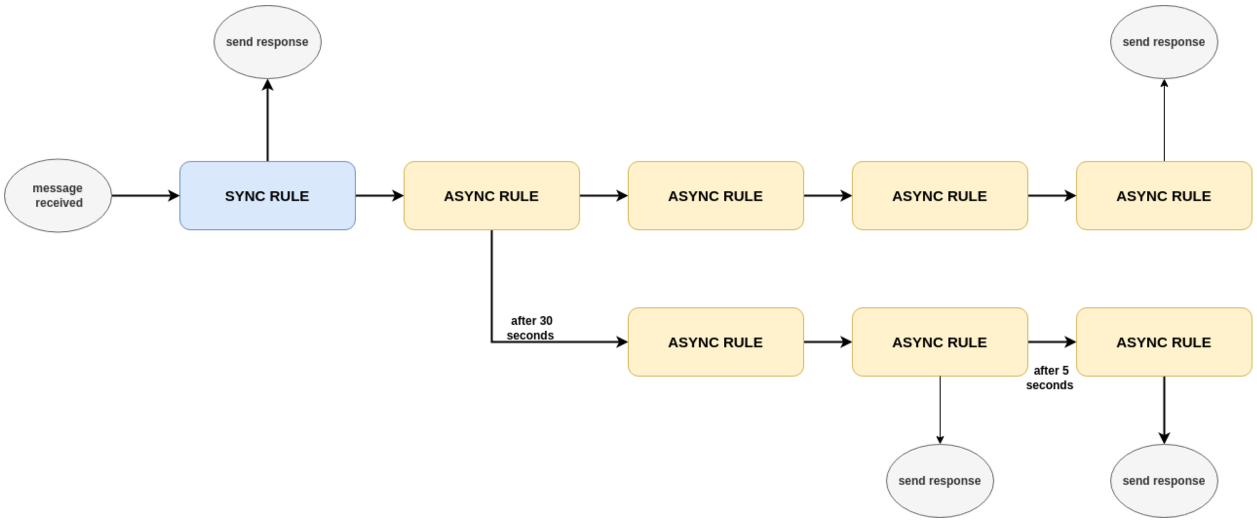
Hybrid Rules

This rules are Synchronous and asyncrhonous at the same time, they can be triggered by a regular expression, or being triggered by another rule.

```
any_custom_hybrid_rule:

    # GENERAL FIELDS
    # ==============
    ...
    mode: hybrid

    # SYNC FIELDS
    # ============
    # RegEx that make the rule applicable
    regex: [any regex]

    # Just in case the 'regex' field is
    # encoded in base 64
    regex_b64: no # yes/no (default)

    # OTHER FIELDS
    # ============
    # Other fields expalined in this documentation
    ...
```

Async or Hybrid Rules as Set of Steps

You can see, sometimes you ‘execute’ a rule, but it does not reply anything to the third party. You may think this is usless, but you will see that this case makes sense once you have other capabilities that will be explained in other chapters of this documentation (e.g.: calling or executing memory operations). Then, they work as a kind of ‘steps’ for these actions, especially when several of them are triggered one after the another in a kind of ‘steps’ list or chain.



1.10 Session Support & Memory Variables

1.10.1 Session Support

We have already seen that the connection socket can be closed during the interaction. Either to be renew for the **external connector**, or the third party, depending on the scenario (client or server mode). This introduces a challenge that is the capability of recognising the interaction session once the socket is renewed, and not considering it as a new interaction.

So as to do that, *Lope* has some mechanisms to have session support. In the `operation` section, there si a subsection not explained so far that is `session`. In server mode, this adds a session ID for each connection once it is created, and used later on to recognise the connection / interaction session. This is done via checking this ID for each request before executing any rule. The session information is expected to have a key-value structure, where the symbol to separate the key and the value can be configured, as well as what is the ending character of the session information (it can be empty).

Therefore, if the structure is something like this KEY+KEY_VALUE_SEPARATOR+{session_value}+END_VALUE

In this example `jsession:srfksdhbq234r;` we can identify the different parts of the structure:

- KEY = `jsession`
- KEY_VALUE_SEPARATOR = `:`
- `session_value` = `srfksdhbq234r`
- END_VALUE = `;`

In server mode, the `session_value` can be In client mode, identifying the session can autogenerated using different options. You can add the session key (KEY) and the session value (`session_value`) by using the reserved words {{SESSION_KEY}} and {{SESSION_VALUE}} in the `response` or `value` field in any rule (curvy brackets included). The following is how the session support configuration can be configured:

```
# -------------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# -------------------------------------------------------------
operation:

    # GENERAL ASPECTS
    # ===============
    ...


    # TO ENABLE TLS/DTLS USE
    # ======================
    ...


    # SOCKET CONNECTION CLOSE
    # =======================
    ...

    # SESSION SUPPORT
    # ===============
    session:
        # To use or not the session support
        enable: no # yes/no(default)

        # session key
        key: id

        key_value_separator: ":"

        # character after the session value (if not present,
        # it will try to limit the session value via a 'space'
        # character, 'brake line' character, or 'end' character)
        end_value: ","

        #  for server mode
        autogenerated:

            enable: yes # yes(default)/no

            number_characters: 20 # default 12

            # One of the following: "numbers", "hex_lower", "hex_upper",
            # "hex_mix", "alphanumeric_upper", "alphanumeric_lower"(default),
            # "alphanumeric_mix", "alphanumeric_and_symbols_upper",
            # "alphanumeric_and_symbols_lower", "alphanumeric_and_symbols_mix"
            characters_type: alphanumeric_lower

        # Session ID can be updated via conversation rules
        update:

            # is it possible to change the session ID during the interaction?
```

(continues on next page)

(continued from previous page)

```
            enable: yes # yes/no(default)

            # One of the following:
            # "rule_detected", "rule_executed(default)"
            when: rule_executed

    # OTHER TOPICS
    # ============
    # Additional aspects should be defined here, but
    # they will be described in the respective sections
    # of this documentation, for the sake of clarity
    ...
```

As you can see, the session values can be changed via custom rules. This can be at the time of ‘rule detection’ (when the rule is analysed as applicable), or after its execution (‘rule execution’). This can be done by adding the following fields:

```
any_custom_rule:

    # GENERAL FIELDS
    # ==============
    ...

    # SESSION UPDATE FIELDS
    # =====================
    # If several rules are applicable, they are overwritten.
    # The last one is the one that remains
    # (usually, the one with the larger ID)
    session_update:
        # allow that this rule can do the session update
        enable: yes # yes/no (default)

        # Several options available. In case several of them,
        # the preference orderis the following:
        #1-memory variable, 2-autogenerated value, 3-fixed value

        # OPTIONS:

        # 1 - Using memory variable, the content of the
        #  memory variable (they are explained below in this chapter)
        memory_variable: var1

        # OR

        # 2 - Create a new session ID using the configuration
        # of the 'operation' section
        # only used when 'memory_variable' is empty or not used
        autogenerated_value: yes # yes/no (default)

        # OR

        # 3 - Fixed value (hardcoded value)
```

(continues on next page)

(continued from previous page)

```
        # only used when 'memory_variable' is empty or not used,
        # and 'autogenerated_value' is 'no' or not used
        fixed_value: [any value you wish]


    # OTHER FIELDS
    # ============
    # Other fields expalined in this documentation
    ...
```

1.10.2 Memory Variables

We have already seen the session support, and this could be considered as an example of a memory variables (`SESSION_KEY` and `SESSION_VALUE`). These variables ar the way to save some data and have 'memory' about the interaction. The variables must be declared in the *conversation rules* file and they are typed, so they shuold be used always in the same way (if it is an `int`, then do not use it as a `string`).

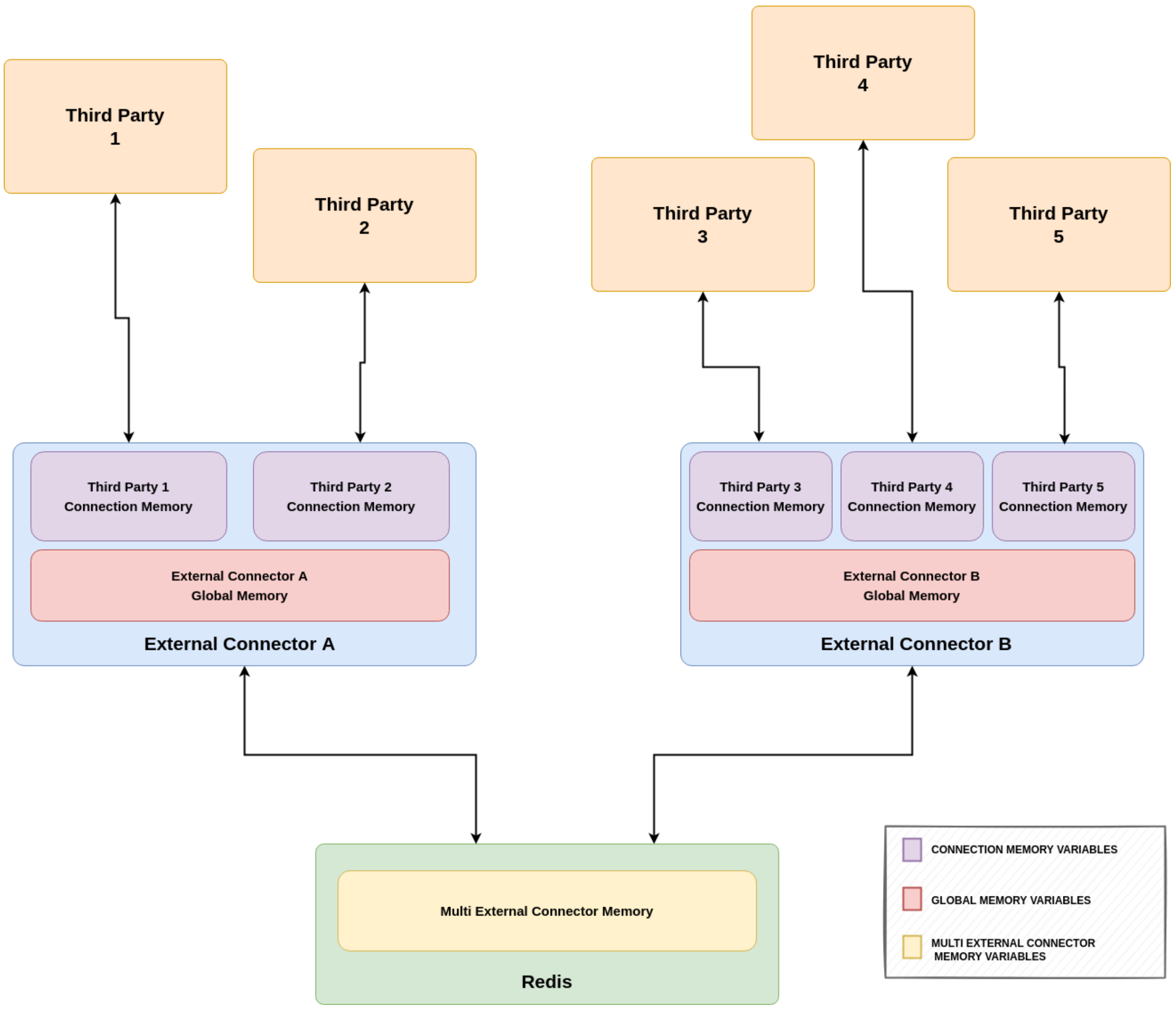
There are different scopes for this variables:

- **Connection (Interaction) Level**: These variables 'live' in the context of a connection or interaction, they are only accesible and usable in the context of a specific connection with a third party. Every time a new connection is established, a new set of the these memory varibales are provided for that connection or interaction.
- **Global Level**: The variables of this memory level are shared among different connections within one **external connector**, and they are created at the beginning of the execution. They are accesible and usable for any connection that happens with the third parties that are interacting with that **external connector**.
- **Multi External Connector Level**: These variables are located in the redis server, and they are shared among all the **external connectors**. This kind of memory is the 'execution' memory level of *Lope* and it is created for the first **external conenctor** that is executed.

The use of multi external connector memory variables is not enable by default, and you can do it using by 'enabling' the field `memory_variables_multi_ext_connector_enable` in the `operation` section. Just remember that you also have to configure the connection between the **external connection** and the Redis server, as explaind in *External Connector Configuration*.

Additionally, since several **external connectors** may try to initialize the same memory variable in the same execution, we need to put some control there. This can be carried out using the field `multi_ext_connector_memory_overwrite_during_init` under the `operation` section. This allows us to mark if a new **external connector** should overwrite the exisisting variable in Redis at the initialization phase, or not. By default, any new execution checks if the memory variable already exists in Redis, and if not, the **external connector** creates it. If it exist, it depends on the value of that field in `operation` to overwrite it or not.

The following diagram represents the different kind of memories that can contain the memory variables, according to their scope:


Third Party 1
Third Party 2
Third Party 3
Third Party 4
Third Party 5
Third Party 1 Connection Memory
Third Party 2 Connection Memory
External Connector A Global Memory
External Connector A
Third Party 3 Connection Memory
Third Party 4 Connection Memory
Third Party 5 Connection Memory
External Connector B Global Memory
External Connector B
Multi External Connector Memory
Redis
CONNECTION MEMORY VARIABLES
GLOBAL MEMORY VARIABLES
MULTI EXTERNAL CONNECTOR MEMORY VARIABLES

They can be used in any response for any rule, by puting the variable name under double curvy brackets: `{{variable_name}}`, and used in many different operations as explained in this page. However, they should be defined previously before being used. In the definition, they can be initialized as well, and the initialization can be either using a fixed value or autogenerated value.

You can define memory variables within the memory variables. At the time of using it to send a response to the third party, the first memory variable will be added, and later on, the second. This feature allows that memory variables can work as 'templates for replying' as well. Since this functionality is replacing memory variables until there is none, you can create as many template levels as you wish.

In case of several memory variable names have the same name, the one with the reduced scope is the one used at the time of being used. Therefore, the preference order is: connection level > global level > milti external connector level. The declaration of the memory variables is done in the `memory` section (same level than `operation`). The supported types are `int`, `float`, `string`, `bool`; where the `string` one is the default one used if the type is not declared. The following example shows how the declaration of memory variables is done using fixed values.

```
# -----------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# -----------------------------------------------------------
```

(continues on next page)

(continued from previous page)

```
operation:

    # GENERAL ASPECTS
    # ===============
    ...

    # TO ENABLE TLS/DTLS USE
    # ======================
    ...

    # SOCKET CONNECTION CLOSE
    # =======================
    ...

    # SESSION SUPPORT
    # ===============
    ...

    # MEMORY VARIABLES
    # ================
    # To enable the use of Redis server to share memory
    # variables between external connectors
    memory_variables_multi_ext_connector_enable: yes # yes/no(default)

    # To overwrite exisisting memory variables in Redis
    # during the initialization phase
    multi_ext_connector_memory_overwrite_during_init: yes # yes/no(default)

    # OTHER TOPICS
    # ============
    # Additional aspects should be defined here, but
    # they will be described in the respective sections
    # of this documentation, for the sake of clarity
    ...

# -----------------------------------------------------------
# Execution memory
# -----------------------------------------------------------
# list of memory variables to be used in the simulation
memory_variables:

  multi_extconn_level:
    - name: var1
      default_value: 0
      type: int

    - name: var2
      default_value: False
      type: bool

  global_level:
    - name: var3
```

(continues on next page)

(continued from previous page)

```
        default_value: 2.71
        type: float

      - name: var4
        default_value: False
        type: bool

    connection_level:
    - name: var5
      default_value: 14qwefa234rt
      type: string

    - name: var6
      default_value: I am a string :)
```

And this another example shows how to define memory variables using autogenerated values at the connection level, but it works in the same way for any other memory level:

```
# -----------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# -----------------------------------------------------------
operation:

    # GENERAL ASPECTS
    # ===============
    ...

    # TO ENABLE TLS/DTLS USE
    # ======================
    ...

    # SOCKET CONNECTION CLOSE
    # =======================
    ...

    # SESSION SUPPORT
    # ===============
    ...

    # MEMORY VARIABLES
    # ================
    # To enable the use of Redis server to share memory
    # variables between external connectors
    memory_variables_multi_ext_connector_enable: yes # yes/no(default)

    # To overwrite exisisting memory variables in Redis
    # during the initialization phase
    multi_ext_connector_memory_overwrite_during_init: yes # yes/no(default)

    # OTHER TOPICS
    # ============
```

(continues on next page)

(continued from previous page)

```
    # Additional aspects should be defined here, but
    # they will be described in the respective sections
    # of this documentation, for the sake of clarity
    ...

# ------------------------------------------------------------
# Execution memory
# ------------------------------------------------------------
# list of memory variables to be used in the simulation
memory_variables:

  connection_level:

    # String autogenerated memory variable
    - name: random_string__token
      type: string

      autogenerated:
        enable: yes # yes/no(default)

        number_characters: 6 # default 12

        # One of the following: "numbers", "hex_lower",
        # "hex_upper", "hex_mix", "alphanumeric_upper",
        # "alphanumeric_lower"(default), "alphanumeric_mix",
        # "alphanumeric_and_symbols_upper",
        # "alphanumeric_and_symbols_lower",
        # "alphanumeric_and_symbols_mix"
        characters_type: alphanumeric_mix

    # Float autogenerated memory variable
    - name: random_float
      type: float

      autogenerated:
        enable: yes # yes/no(default)

        min_limit_interval: 0
        max_limit_interval: 1

    # Int autogenerated memory variable
    - name: random_int
      type: int

      autogenerated:
        enable: yes # yes/no(default)

        min_limit_interval: 0
        max_limit_interval: 100

    # Bool autogenerated memory variable
    - name: random_bool
```

(continues on next page)

(continued from previous page)

```
      type: bool

      autogenerated:
        enable: yes # yes/no(default)
```

If no value is provided in the definition, then the default values are:

- '0' for `float` and `int` types
- 'false' for `boolean` types
- an empty string for `string` types

1.11 Memory Operations

Having memory variables is great, but using them is even better. In this section of the documentation we are going to see how you can work with them.

1.11.1 Memory Modifications Using Rules

The first use of the memory variables is that any custom rule can change the value of those variables. This can be done via directly changing the value, or by using the information received from the third party systems.

Direct Memory Variable Update

The direct change of the memory variables needs first to decide 'when' this is going to happen, either in the 'rule detection' step (when the rule is considered as applicable to be executed), or in the 'rule execution' step (when the rule is executed and the response is sent, if there is any). This is configured in in the `operation` section:

```
# -----------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# -----------------------------------------------------------
operation:

  # GENERAL ASPECTS
  # ===============
  ...

  # TO ENABLE TLS/DTLS USE
  # ======================
  ...

  # SOCKET CONNECTION CLOSE
  # =======================
  ...

  # SESSION SUPPORT
  # ===============
  ...
```

(continues on next page)

(continued from previous page)

```
  # MEMORY UPDATE & CONDITIONS
  # ==========================
  # # One of the following: "rule_detected", "rule_executed"(default).
  # (Before session update and custom functions)
  memory_update_when: rule_executed

  # To enable the use of Redis server to share memory
  # variables between external connectors
  memory_variables_multi_ext_connector_enable: yes # yes/no(default)

  # To overwrite exisisting memory variables in Redis
  # during the initialization phase
  multi_ext_connector_memory_overwrite_during_init: yes # yes/no(default)

  # OTHER TOPICS
  # ============
  # Additional aspects should be defined here, but
  # they will be described in the respective sections
  # of this documentation, for the sake of clarity
  ...
```

And the memory variable update should be defined under the `memory` field, and `update` subfield of the custom rule. Several memory variables can be updated at onces since you define a list of them to be updated. The updating process can use a fixed (hardcoded) value, or another memory variable. In this second case, the content of the reference memory variable is copied into the memory variable to update. In case both options are present (fixed value and reference memory variable), the fixed value case is omitted.

```
any_custom_rule:

  # GENERAL FIELDS
  # ==============
  ...

  # MEMORY UPDATE & CONDITIONS
  # ==========================
  memory:
      # new values for some memory variables.
      # In case or a reference variable is present then
      # the updating process uses the value of the
      # reference memory variable, the value is omitted
      updates:
        # One memory variable is updated
        - var_name: var2
          value: test2

        # Another memory variable is updated
        - var_name: var1
          value: 2 # This value is not used!
          reference_variable: var2 # This is used to update
                                   # the memory variable
```

(continues on next page)

(continued from previous page)

```
  # OTHER FIELDS
  # ============
  # Other fields expalined in this documentation
  ...
```

Memory Modifications Using Captured Data

The memory variables can be updated via capturing specific information from the messages that are received from the third parties. This allows to save specific set of information in a memory variable. This only can be done in the sync or hybrid custom conversation rules, and it uses RegEx as well:

```
any_custom_rule:

  # GENERAL FIELDS
  # ==============
  ...

  # CAPTURING INFORMATION
  # =====================
  capturing_data:
        enable: yes # yes/no(default)

        # list of different information to capture
        captures:

            # RegEx to capture the information
          - regex: [any regex]
            # Memory variable to save the captured information
            mem_var_name: var1
            # Is the regex in base64 format?
            regex_b64: no # yes/no (default)

            # RegEx to capture the information
          - regex: [any regex]
            # Memory variable to save the captured information
            mem_var_name: var2
            # Is the regex in base64 format?
            regex_b64: no # yes/no (default)


  # OTHER FIELDS
  # ============
  # Other fields expalined in this documentation
  ...
```

You can use memory variables of different scopes for doing the update. For instance, you can update a connection level memory variable using a global level memory variable. In case the reference memory variable is in ‘collision’ with others that have the same name, the one with the more limited scope will be used (in the same way that we have seen about how to add a memory variable in a response).

1.11.2 Custom Rule Conditional Execution

One of the potential uses of the memory variables is to modify the behavior of the execution, according to different values they may have. This is done similarly than the update process of memory variables. This means that the custom rule is only executed if the memory conditions are satisfied, after being considered as applicable. Therefore, in this case, there are two steps of allowing the execution of the rule:

- Being applicable synchronous or asynchronously.
- Having the memory conditions satisfied

In the syncrhonous case, the memory conditions are checked at the time the RegEx is evaluated. In the asyncrhonous case, the conditions are checked just before the rule execution. This means the conditions are not checked at the time to trigger the rule, you trigger it but later on, the memory conditions are reviewed.

```
any_custom_rule:

  # GENERAL FIELDS
  # ==============
  ...

  # MEMORY UPDATE & CONDITIONS
  # ==========================
  memory:
      # conditions to satisfy for executing the rule.
      # In case or a reference variable is present,
      # then the comparison is if the value of the memory variable
      #  is the same of the reference memory variable.
      # In case the 'value' field and the 'reference_variable' are present,
      # only the 'reference_variable' field
      # is used
      conditions:
        # var2 == 'test'?
        - var_name: var2
          value: test2

        # var1 == var2?
        - var_name: var1
          value: 2 # This value is not used!
          reference_variable: var2 # This is used for comparison

  # OTHER FIELDS
  # ============
  # Other fields expalined in this documentation
  ...
```

You can add as many conditions as you wish, all of them will be evaluated to allow the rule execution. Similarly to the update process, you can compare memory variables of different scopes. For instance, you can compare a connection level memory variable using a multi external connector level memory variable. In case of the reference memory variable is in ‘collision’ with others that have the same name, the one with the more limited scope will be used (in the same way that we have seen about how to add a memory variable in a response).

1.11.3 Built-in Memory Operations

You can do some operations using the memory variables to allow you to control more how the interaction takes place. Every custom rule can call a built-in memory operation, where you define what are the input values to use and what memory variables will save the output.

The reserved word `EXT_IN` allow you to use the external input (message received from the third party) as input of the operation. You can use memory variables and fixed values as inputs as well. For the output, you need to specify what memory variables will save the results of the operations (if they are not provided, the results are lost). Please, be also aware that the order of the list of inputs or outputs is important for the correct use of the operation, as well as the data 'type' (int, float, bool...) of the result should be the same with the one declared in the section 'memory_variables'.

In case you need to do more than one of memory operations, we recommend using a set of async rules as a set of steps, where each of them is calling a memory operation. The reason to enforce one memory operation per conversation rule is for the sake of performance: Using async rules that are called in a sequential order, avoid blocking the interaction with one rule.

```
any_custom_rule:

  # GENERAL FIELDS
  # ==============
  ...

  # BUILT-IN MEMORY OPERATIONS
  # ==========================
  builtin_memory_operation:

    # built-in memory operation enable for this rule?
    enable: yes # yes/no(default)

    # Operation to execute
    operation: STR_CONCAT

    # List of inputs
    # EXT_IN = 'EXTERNAL INPUT'
    input:
      - EXT_IN
      - var1
      - 3

    # List of memory variables
    # to save the results
    output:
      - var3
      - var4

  # OTHER FIELDS
  # ============
  # Other fields expalined in this documentation
  ...
```

There are other reserved words that you can use as input for the memory operations:

- `IP`: the ip of the connection
- `PORT`: the port of the connection

- SESSION_KEY: the session key of the connection
- SESSION_VALUE: the session value of the connection

String Operations

The list of the string memory operations available, indicating the expected input and type before the arrow, and the output after the arrow. In case of several inputs/outputs are expected, they are grouped inside a parenthesis.

- STR_CONCAT = String concatenation, merge several strings into one:

```
(string, string, string,...) => string
```

- STR_REPLACE = String replace, replace a string or regex by the value provided:

```
(original_string, string_regex_to_replace, string_to_add) => modified_string
```

- STR_SUBTRACT = String substract, remove a string within another:

```
(original_string, string_to_replace) => modified_string
```

- STR_UPPER = String upper, put in uppercase the content of a string:

```
(string) => string
```

- STR_LOWER = String lower, put in lowercase the content of a string:

```
(string) => string
```

- STR_SPLIT = String split, it separte a string in several pieces. If there are more pieces that elements in the output field, they will be lost.

```
(string_to_split, string_separator) => (string, string, string, ...)
```

- STR_TRIM = String trim, it trims a string:

```
(string) => string
```

- STR_MATCH = String match, it checks if a regular expresion is present in a string:

```
(string, string_regex) => bool
```

- STR_CAPTURE = String capture, it captures a string using a reguex:

```
(original_string, string_regex) => captured_string
```

- STR_COUNT = String count, it counts the number of characters in a string:

```
(string) => number
```

- STR_MODE = String mode, it calculates the mode (central tendency) of the given nominal data set:

```
(string, string, ...) => string
```

- STR_ENCODE_B64 = String encode base 64, it encodes any string into base64:

```
(string) => string
```

- STR_DECODE_B64 = String decode base 64, it decodes any string already encoded in base64:

```
(string) => string
```

- STR_ENCODE_HEX = String encode hexadecimal, it encodes any string into hexadecimal:

```
(string) => string
```

- STR_DECODE_HEX = String decode hexadecimal, it decodes any string already encoded in hexadecimal:

```
(string) => string
```

Number Operations

The list of the number memory operations available, indicating the expected input and type before the arrow, and the output after the arrow. In case of several inputs/outputs are expected, they are grouped inside a parenthesis.

- NBR_SUM = Number sum, it sums a set of values added in the input field:

```
(number, number, ...) => number
```

- NBR_SUBTRACT = Number substract, it substracts to the first element of the input field, the rest of values added there:

```
(number, number_to_substract, number_to_substract ...) => number
```

- NBR_MULTIPLY = Number multipy, it multiplies a set of values added in the input field:

```
(number, number, ...) => number
```

- NBR_DIVIDE = Number divide, it divides the first element by the rest of elements in the input field in a sequential approach:

```
(number, number_to_divide, number_to_divide ...) => number
```

- NBR_FLOOR = Number divide, it does a floor division of the first element by the rest of elements in the input field in a sequential approach:

```
(number, number_to_floor_divide, number_to_floor_divide ...) => number
```

- NBR_MODULO = Number modulo, it gets the reminder of a set of modulo operator calls. For more information, please check Python Modulo Operator.

```
(number, number_for_modulo_operator, number_for_modulo_operator, ...) => number.
```

- NBR_POWER = Number power, it applies a set of sequential power operations to the first element in the list:

```
(number, power_number, power_number, ...) => number
```

- NBR_INVERSE_SIGN = Number inverse sing, change the sign of the number

```
number => number
```

- NBR_GREATER = Number greater than, compare tu numbers (a>b):

```
(number,number) => bool
```

- NBR_LOWER = Number lower than, compare tu numbers (a<b):

```
(number,number) => bool
```

- NBR_GREATEREQ = Number greater than or equal, compare tu numbers (a>=b):

```
(number,number) => bool
```

- NBR_LOWEREQ = Number lower than or equal, compare tu numbers (a<=b):

```
(number,number) => bool
```

- NBR_MEAN = Number mean, Arithmetic mean value (average) of data:

```
(number, number, ...) => number
```

- NBR_GEOMETRIC_MEAN = Number geometric mean, geometric mean value of data:

```
(number, number, ...) => number
```

- NBR_HARMONIC_MEAN = Number harmonic mean, harmonic mean value of data :

```
(number, number, ...) => number
```

- NBR_MEDIAN = Number median, median value (middle value) of data:

```
(number, number, ...) => number
```

- NBR_MEDIAN_LOW = Number median low, low median value of data:

```
(number, number, ...) => number
```

- NBR_MEDIAN_HIGH = Number median high, high median value of data:

```
(number, number, ...) => number
```

- NBR_MEDIAN_GROUPED = Number median grouped, it calculates the median of grouped continuous data, calculated as the 50th percentile:

```
(number, number, ...) => number
```

- NBR_MODE = Number mode, it calculates the mode (central tendency) of the given numeric or nominal data set:

```
(number, number, ...) => number
```

- NBR_POP_STD_DEV = Number standard deviation, it calculates the standard deviation from an entire population:

```
(number, number, ...) => number
```

- NBR_STD_DEV = Number standard deviation, it calculates the standard deviation from a sample data set:

```
(number, number, ...) => number
```

- `NBR_POP_VAR` = Number standard deviation, it calculatesthe variance of an entire population:

```
(number, number, ...) => number
```

- `NBR_VAR` = Number standard deviation, it calculates the variance from a sample of data:

```
(number, number, ...) => number
```

Boolean Operations

The list of the boolean memory operations available, indicating the expected input and type before the arrow, and the output after the arrow. In case of several inputs/outputs are expected, they are grouped inside a parenthesis.

- `LGC_NOT` = Logical 'NOT' operation:

```
bool => bool
```

- `LGC_AND` = Logical 'AND' operation:

```
(bool, bool, bool, ...) => bool
```

- `LGC_OR` = Logical 'OR' operation:

```
(bool, bool, bool, ...) => bool
```

- `LGC_XOR` = Logical 'XOR' operation:

```
(bool, bool, bool, ...) => bool
```

- `LGC_NAND` = Logical 'NAND' operation:

```
(bool, bool, bool, ...) => bool
```

- `LGC_NOR` = Logical 'NOR' operation:

```
(bool, bool, bool, ...) => bool
```

1.12 Custom Functions

One of the best features that *Lope* provides is to allow that you can add your own Python code to expand the current capabilities already provided. This has already mentioned in the *External Connector Configuration*, in the `operation` section of the **external connector** configuration file. You need to provide the location and name of the Python module where *Lope* can find your own code, as explained in the preiously cited chapter. If the custom function module can not be imported, the execution of the **external connector** can take place but the interactions will not use those

You can add your own code in three different ways in the interaction:

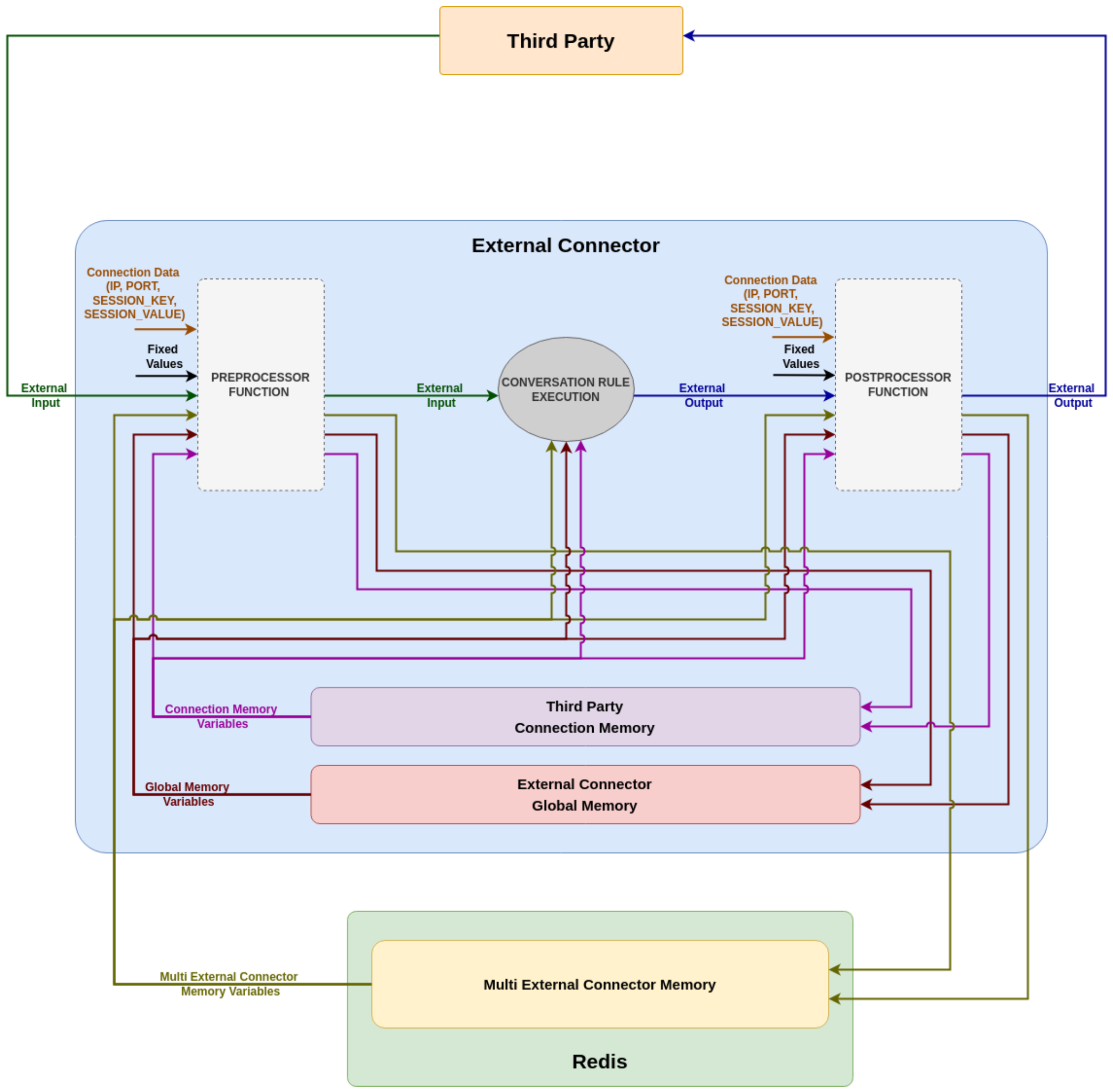
- **Pre-processor function**: If it is used, this function is called every time a new message is received from a third party before any action of *Lope*. This can be interesting in interactions that you need something before executing the *Lope* actions. For instance, if your interaction is using a binary protocol, you should convert the binary into text based information before using *Lope*.
- **Post-processor function**: If it is used, this function is called every time after *Lope* has something to send to third party as part of the process of replying them. This covers the need of doing something at the end of the response process. In the precious example, this can allow you to convert into binary the text *Lope* have already prepared.

- **Function in Custom Rule**: Any custom rule can execute a function in the same way that it can execute a built-in memory operation, as explained in *Memory Operations*.

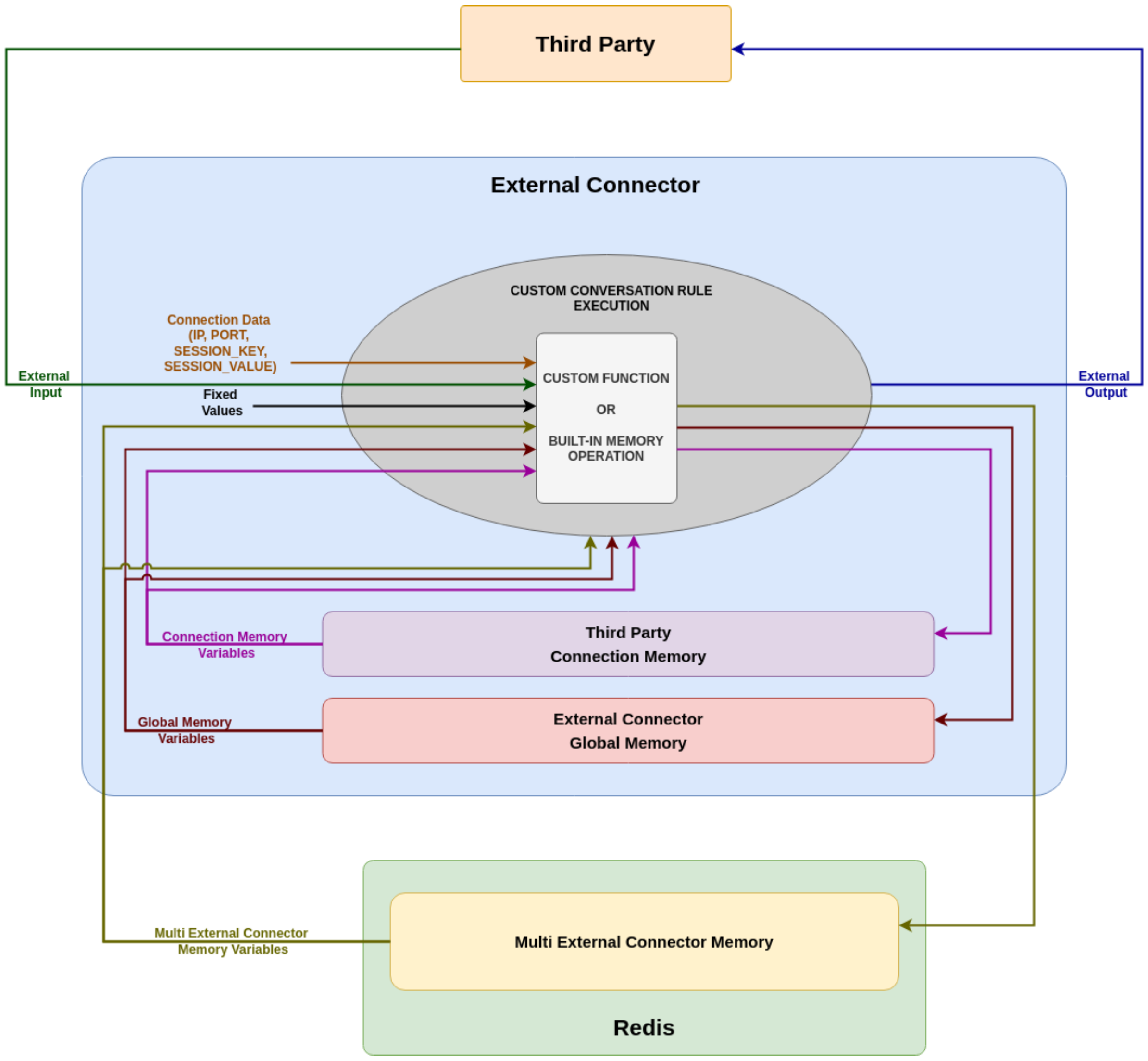
In any case, the functions called has to be called using the exact name, and you should provide the same number and in the same order of parameters as they are defined. In case of not, those not provided parameters will be autoprovided using an empty string, a '0' or 'false', depending on the case. For the number of memory variablesfor saving the output of a function is not the same of the results the function provides, then some values will be lost. In case of providing more memory variables than results, they will remain unchanged.

Please, keep in mind that the *pre-processor* and *post-processor* will always use the 'external input' or 'external output' (depending on the preprocessor function used) as the first input and output of those functions. Therefore, you only can add additional inputs and outputs to the pre/post-processor functions. This is one of the the main differences with the normal custom functions (those called in any custom rule execution), the pre/post-processor functions have the mandatory input and output that is the 'external input' for the *pre-processoror* function, and the 'external output' for the *post-processor* function. The rest of potential inputs and outputs are optional, and can be any memory variable or fixed (hardcoded) values, or even some connection data (**ip**, **port**, the **session key** or **session value**). For the functions called from any custom rule, all the values are optional and even they can have none.

The following image represents how the *pre-processor* and *post-processor* functions work in the response process. Please keep in mind that you can use only one of those functions if needed, you do not need to use both always.


Third Party
External Connector
Connection Data (IP, PORT, SESSION_KEY, SESSION_VALUE)
Fixed Values
External Input
PREPROCESSOR FUNCTION
External Input
CONVERSATION RULE EXECUTION
External Output
Connection Data (IP, PORT, SESSION_KEY, SESSION_VALUE)
Fixed Values
POSTPROCESSOR FUNCTION
External Output
Connection Memory Variables
Third Party Connection Memory
Global Memory Variables
External Connector Global Memory
Multi External Connector Memory Variables
Multi External Connector Memory
Redis

The next one represents the potential inputs of custom functions called in custom rules, or any built-in memory operations, since both work in the same way. They can use the same potential inputs and outputs, you only have to choose what you need.


Third Party
External Connector
CUSTOM CONVERSATION RULE EXECUTION
Connection Data (IP, PORT, SESSION_KEY, SESSION_VALUE)
External Input
Fixed Values
CUSTOM FUNCTION
OR
BUILT-IN MEMORY OPERATION
External Output
Connection Memory Variables
Third Party Connection Memory
Global Memory Variables
External Connector Global Memory
Multi External Connector Memory Variables
Multi External Connector Memory
Redis

As you can see in the image, there is another big difference between pre/post-processor functions and these other operations: the use of the 'external output'. *Post-processor* function can use it, but these functions or memory operations they cannot: they only modify the memory variables. Depending on the configuration of when the custom functions in rules should be executed ('rule detection' vs 'rule execution'), you can use the modified memory variable in the response. In case you execute the custom functions after the rule execution, the potential reply is already sent. Nevertheless, this is not an issue because if you need to do an operation and then replying to the third party, you can call the function or memory operation in the sync custom rule, and trigger an async rule to send the reply right after the sync rule. Following that idea, you can concatenate several memory operations (built-in or custom functions) as a set of async or hybrid rules (as we have already seen at the end of the *Conversation Rules*).

1.12.1 Custom Pre-processor Function

Once we have seen the concepts behind the use of custom functions, it is time to see how to use them. To use the *pre-processor* function, you have to add the following information in the `operation`:

```
# -----------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# -----------------------------------------------------------
operation:

    # GENERAL ASPECTS
    # ===============
    ...


    # TO ENABLE TLS/DTLS USE
    # ======================
    ...

    # SOCKET CONNECTION CLOSE
    # =======================
    ...

    # SESSION SUPPORT
    # ===============
    ...

    # CUSTOM FUNCTIONS
    # ================
    custom_function_preprocessor:

      # name of the function to call
      name: testing_preprocessor

      # the external input will always be the first
      # input, others are defined here:

      # additional
      input:
        - var1
        - var2
        - 2
        - 'IP'
      # In this case the input of the function is:
      # ('external input', var1, var2, 2, ip value) => testing_preprocessor

      # the external input will always be the first
      # output, others are defined here:

      # additional
      output:
        - var1
```

(continues on next page)

(continued from previous page)

```
        - var2
      # In this case the output of the function is:
      # testing_preprocessor => ('external input', var1, var2)

    # OTHER TOPICS
    # ============
    # Additional aspects should be defined here, but
    # they will be described in the respective sections
    # of this documentation, for the sake of clarity
    ...
```

1.12.2 Custom Post-processor Function

The use of the custom *post-processor* is almost identical to the *pre-processor* function:

```
# -----------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# -----------------------------------------------------------
operation:

    # GENERAL ASPECTS
    # ===============
    ...

    # TO ENABLE TLS/DTLS USE
    # ======================
    ...

    # SOCKET CONNECTION CLOSE
    # =======================
    ...

    # SESSION SUPPORT
    # ===============
    ...

    # CUSTOM FUNCTIONS
    # ================
    custom_function_postprocessor:

      # name of the function to call
      name: testing_postprocessor

      # the external output will always be the first
      # input, others are defined here:

      # additional
      input:
        - var1
        - var2
```

(continues on next page)

(continued from previous page)

```
        - 'hello!'
      # In this case the input of the function is:
      # ('external output', var1, var2, 'hello!') => testing_postprocessor

      # the external input will always be the first
      # output, others are defined here:

      # additional
      output:
        - var1
        - var2
      # In this case the output of the function is:
      # testing_postprocessor => ('external output', var1, var2)

    # OTHER TOPICS
    # ============
    # Additional aspects should be defined here, but
    # they will be described in the respective sections
    # of this documentation, for the sake of clarity
    ...
```

1.12.3 Custom Functions in Custom Rules

In contrast of the pre/post-processor functions, the functions in custom rules are defined in the rules that execute them, in the same way of the built-in memory operations. In case of a custom rule contains a built-in memory operation and a custom function, then the built-in memory operation will be executed first and later, the custom function call.

The use of the same reserver words that we have seen in *Memory Operations* is also applicable here:

- `EXT_IN`: the external input from the third party
- `IP`: the ip of the connection
- `PORT`: the port of the connection
- `SESSION_KEY`: the session key of the connection
- `SESSION_VALUE`: the session value of the connection

The execution of custom function can be adjusted to decide 'when' takes place, as we have already seen for *Memory Operations*: in the 'rule detection' step (when the rule is considered as applicable), or in the 'rule execution' step (when the rule is executed and the response is sent). This is configured in in the `operation` section as follows:

```
# ----------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# ----------------------------------------------------------
operation:

  # GENERAL ASPECTS
  # ===============
  ...

  # TO ENABLE TLS/DTLS USE
```

(continues on next page)

(continued from previous page)

```
  # ======================
  ...

  # SOCKET CONNECTION CLOSE
  # =======================
  ...

  # SESSION SUPPORT
  # ===============
  ...

  # CUSTOM FUNCTIONS
  # ================
  # One of the following: "rule_detected", "rule_executed"(default)
  custom_functions_when: rule_detected

  # OTHER TOPICS
  # ============
  # Additional aspects should be defined here, but
  # they will be described in the respective sections
  # of this documentation, for the sake of clarity
  ...
```

and the way to declare the use of the custom function within a rule can be found here:

```
any_custom_rule:

  # GENERAL FIELDS
  # ==============
  ...

  # CUSTOM FUNCTION
  # ===============
  custom_function:

    # custom function enable for this rule?
    enable: yes # yes/no(default)

    name: function_name

    # List of inputs
    input:
      - EXT_IN # 'external input'
      - var1
      - 3.14

    # List of memory variables
    # to save the results
    output:
      - var3
      - var4
```

(continues on next page)

(continued from previous page)

```
  # In this example:
  # ----------------
  # Inputs to the custom function
  # ('external input', var1, 3.14) => function_name

  # Output of the custom function
  # function_name => (var3, var4)

  # OTHER FIELDS
  # ============
  # Other fields expalined in this documentation
  ...
```

1.13 Advance Conversation Rules

In this chapter, we are going to see more advance options that are available for the use of the conversation rules to provide more tools for defining more complete scenarios, or just easy the use of *Lope* to operators.

1.13.1 Conditional External Connector Execution

This feature allows the **external connector** to wait for certain conditions before starting to interact with third parties. This feature needs the use of multi external connector memory varibales, because the **external connector** starts its execution but it does not interact directly with third parties. It will wait until some conditions are satisfied (memory variables in Redis have the right values), before starting any interaction. We can say that the **external connector** is 'sleeping' until the right time to 'wake up'.

In order to use this functionality, you have to configure it in the `operation` section of the conversation rules. It can get the IP and port for the execution from some memory variables as well, so as to start the execution using dynamic data. The memory variables used for the conditions can be different for those declared to be used in the interaction, therefore you do not need to declare them as you do for those you are going to use. In the following example you can see how this can be configured.

```
# ---------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# ---------------------------------------------------------
operation:

  # GENERAL ASPECTS
  # ===============
  ...

  # TO ENABLE TLS/DTLS USE
  # ======================
  ...

  # SOCKET CONNECTION CLOSE
  # =======================
  ...
```

(continues on next page)

(continued from previous page)

```
  # SESSION SUPPORT
  # ===============
  ...

  # CONDITIONAL EXT CONN EXECUTION
  # ==============================
  conditional_execution:
      # use conditional execution in this execution?
      enable: no # yes/no(default)

      # list of conditions: memory values and the value
      conditions:
          # One condition
          - var_name: order_66
            value: True

          # Another condition
          - var_name: order_67
            value: False

      # If present, the ip value will be
      # gathered from this memory variable
      mem_var_ip: var1

      # If present, the port value will be
      # gathered from this memory variable
      mem_var_port: var2

  # OTHER TOPICS
  # ============
  # Additional aspects should be defined here, but
  # they will be described in the respective sections
  # of this documentation, for the sake of clarity
  ...
```

1.13.2 Async Switch

We have already seen that one custom rule can trigger several async or hybrid custom rules as well. However, the 'OR' is missing, this means that you may trigger this set of custom rules asynchronously, or this another set of them; depending on some conditions (memory variables and their values). This provides flexibility to interact differently depending on the execution context, so alternative interactions now can take place.

In order to configure that, you can follow the below example. Any 'case' of the 'switch' has to define the list of `conditions` that make it applicable, and the list of `rules` to trigger asynchronously when that happens. The 'switch' structure is made up of `options` (cases) and the `dafault` (when no option is applicable). However, the `default` one is optional.

```
any_custom_rule:

  # GENERAL FIELDS
  # ==============
```

(continues on next page)

(continued from previous page)

```
...

# ASYNC SWITCH
# ============
# Switch for executing some async (or hybrid) rules depending
# on conditions. E.g., ff memory variable 'AAAA' has the value
# 'X' (value field), or it has the same content of the memory
# variable 'BBB', then execute the async Rule 'N'. In case no
# option fits, use the async rule of the default field (if present,
# and the list length is greater than 0)
async_switch:

    options:
        # first option or 'case' wit a list of conditions to satisfy
        - conditions:
            # list of conditions
            - var_name: var1
                value: 2
                reference_variable: var2

            # if conditions are satisfied, then a set of async
            # (or hybrid) rules to execute
          rules:
            - rule_id: 3
              delay: 3

        # second option or 'case'
        - conditions:
            - var_name: var1
                value: 2
                reference_variable: var2
          rules:
            - rule_id: 3
              delay: 3

    # other cases:
    #   Another case
    #   - conditions:...
    #     rules:...
    #
    #   and another one
    #   - conditions:...
    #     rules:...

    default:
    # set of async ruls to execute if no 'case' statement
    # is applicable
    - rule_id: 3
        delay: 3


# OTHER FIELDS
```

(continues on next page)

(continued from previous page)

```
  # ============
  # Other fields expalined in this documentation
  ...
```

1.13.3 Async Loop

The async loop is another interesting feature to model interactions and provide more flexibility. This feature allows you execute a set of rules asynchronously while some conditions are satisfied. Every time the rules are going to be executed, the loop conditions are reviewed for checking if a new iteration should take places. This means that iterations will may take place in different times, according to the different delays of the rules. Howevever, every rule will have its loop. Not all the rules of the loop are executed in every iteration, only the rule that is going to be executed. Therefore, we can say that we have several parallel virtual loops in place with the memory same conditions.

```
any_custom_rule:

  # GENERAL FIELDS
  # ==============
  ...

  # ASYNC LOOP
  # ==========
  # Switch for executing some async (or hybrid) rules depending
  # on conditions. E.g., ff memory variable 'AAAA' has the value
  # 'X' (value field), or it has the same content of the memory
  # variable 'BBB', then execute the async Rule 'N'. In case no
  # option fits, use the async rule of the default field (if present,
  # and the list length is greater than 0)
  async_loop:

    # list of the conditions of the loop
    # they can use another memory variable
    # or a fixed value. In case of both are
    # present, the memory variable will be used
    conditions:
      - var_name: var1
        value: 2
        # OR
        reference_variable: var2

    # list of rules to execute
    rules:
      # this rule will be executed every 3 seconds, 5 times
      # as maximum
      - rule_id: 3
        delay: 3
        # if present and grater than 0, this means the number
        # of times the iteration can take place per each rule
        max_number_iterations: 5

      # this rule will be executed every 5 seconds
      - rule_id: 17
```

(continues on next page)

(continued from previous page)

```
        delay: 5


  # OTHER FIELDS
  # ============
  # Other fields expalined in this documentation
  ...
```

1.13.4 External Connector Fork

If having an **external connector** (or some of them) is great, why not inviting more to the party?!. This is exactly what this feature does: It allows you to create new **external connectors** during the interaction.

```
any_custom_rule:

  # GENERAL FIELDS
  # ==============
  ...

  # EXT CONN FORK
  # =============
  # You can create new instances of external connectors
  # if a rule is executed and optionally, passing a
  # new configuration file, external connector id and
  # password (secret). If these parameters are not provided,
  # the new instance of external connector will start as
  # as the original did (this is powered by python subprocess module)
  fork:
    # is this feature enable?
    enable: yes

    # Optional parameters.
    # --------------------
    # If they are not provided, the paremeters of the current
    # external connector will be used

    # config file to use for the new external connector
    config_file: "/path/to/a/different/config/file"

    # ID to use for the new external connector
    new_id: "Terminator"

    # password to use for the new external connector
    new_secret: "sshhhhhh...it's_a_secret"

    # number of instances to create
    number: 3 # Number of instances to create


  # OTHER FIELDS
  # ============
```

(continues on next page)

(continued from previous page)

```
  # Other fields expalined in this documentation
  ...
```

1.13.5 Conversation Rules Groups

As you may have already realised, for complex interaction scenarios you will need a lot of rules. This could be a performance issue when you have a new input, despite of having a parallel analysis of the potential RegEx. In order to overcome that issue, the rule groups comes into play: they group rules in blocks, and that block is only applicable under some conditions (It could be a RegEx, or some values for some memory variables). In any case, the rules of the group are not evaluated (or disabled) until the group applicable.

This measure can really help in the reduction of the number of RegEx evaluations, apart of providing a bit of order in the conversation rule file. Just for your information, groups are evaluated **after** the evaluation of the non-grouped *conversation rules*. Below you can find how you can implement that feature in the conversation rule file:

```
# -----------------------------------------------------------
# Conversation Rules
# -----------------------------------------------------------
conversation:

    greetings:... # Basic Rule

    default:... # Basic Rule

    empty:... # Basic Rule

    timeout:... # Basic Rule

    ending:... # Basic Rule

    custom_rules: # Set of custom rules


        # GROUPED RULES
        # ------------
        groups:
        # list of groups

          # One group that is enabled using regex
        - id: group_A
          # The regex that makes this group applicable
          regex: [any regex]

          # list of custom rules
          rules: ...


          # Another group is enabled using memory variables
        - id: group_B

          # List of memory conditions that make the group applicable
```

(continues on next page)

(continued from previous page)

```
          memory_conditions:
              # var1 == fixed_value?
            - var_name: var1
              value: [any fixed_value]

              # var2 == var3?
            - var_name: var2
              reference_variable: var3

          # list of custom rules
          rules: ...


        # NON-GROUPED RULES
        # -----------------
        rules: ... # list of custom rules that do not
                   # belong to any group
```

1.13.6 Multiple Conversation Rules Files

We have seen that there are plenty of possibilities about what you can do with the rules and the memory variables. However, having everything in one file could be a madness to work with that. The proposed solution is to split the conversation rules file into a primary one, and a set of auxiliary ones that will be imported at the time of loading the primary conversation rules file.

This also allows that you can have some things defined in specific files and imported into several different conversation rules files. The things that you can put in this secondary kind of files are custom conversation rules, and memory variables declaration. The auxiliary files can import other auxiliary files as well, providing more flexibility at the time of creating smaller piece of conversarion rules in different files.

In order to this, you have to do the following in the primary conversation rule file:

```
# ====================================================
# EXAMPLE PROTOCOL
# Author: Alberto Dominguez
# ====================================================
# this is just a field just for 'operators', it is not relevant for operation
name: example

# Which external connector group is assigned for this rules.
#  If empty, or 'default' ==> no group (only one file valid without group)
ext_conn_group: B

# This enables the use of this conversation rules.
# This means that you can have some conversation rules files
# that are in the folder, but they are deactivated and then,
# they will no be sent to any external connector.
enable_use: no # yes/no (default).

# -----------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
```

(continues on next page)

(continued from previous page)

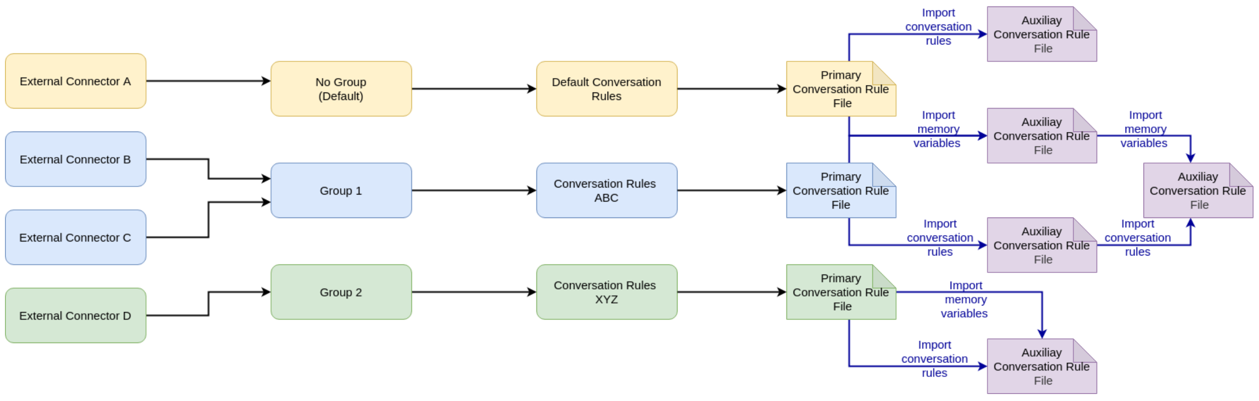
```
# -----------------------------------------------------------
operation: ... # to be explained later

# -----------------------------------------------------------
# Execution memory
# -----------------------------------------------------------
memory_variables:

  # some memory variables defined (if any)
  # ...
  multi_extconn_level:...

  global_level:...

  connection_level:...

  # list of files to import
  import:

    # ALWAYS IMPORT .YML FILES!
  - path: /path/to/auxiliary/file/for/more/memory/variables.yml
    is_relative: yes

# -----------------------------------------------------------
# Conversation Rules
# -----------------------------------------------------------
conversation:

  greetings:... # Basic Rule

  default:... # Basic Rule

  empty:... # Basic Rule

  timeout:... # Basic Rule

  ending:... # Basic Rule

  custom_rules:

    rules:...

    groups:...

    # list of files to import
    import:

        # ALWAYS IMPORT .YML FILES!
      - path: /path/to/auxiliary/file/for/more/custom/rules.yml
        is_relative: yes
```

And, in any auxiliary conversation rule file, you can add the following information:

```
# ====================================================
# AUXILIARY CONVERSATION RULE FILE
# Author: Alberto Dominguez
# ====================================================
# this is just a field just for 'operators',
# it is not relevant for operation
name: additional file x32

# Fields used at the time of importing memory variables
# -----------------------------------------------------
multi_extconn_level:...

global_level:...

connection_level:...


# Fields used at the time of importing conversation rules
# -------------------------------------------------------
rules:...

groups:...


# Fields used to import additional auxiliary files
# ------------------------------------------------
# list of files to import
import:

    # ALWAYS IMPORT .YML FILES!
  - path: /path/to/another/auxilary/file.yml
    is_relative: yes
```

The auxiliary files can have any of those fields, bnut they are not mandatory. It could happen that one file has one one field of them, or you have rules and memory variables in the same file, but this file should be used in the memory and in the rules sections to really import the whole content.

The following image expand the previous used image of how the **external connectors** are linked with the *conversation rules*, adding some different imported files to the existing example.



1.14 Activity Alerting & Storage

In this chapter we are going to talk about the result of the interactions: activities (a.k.a. external activities). They are the result of the rules execution, or any relevant event of the interaction life cycle of the interactions. These activities are created in the **external connector** and sent to the **engine** for being saved, or being sent as alerts to other systems.

The integrity of the activities has been considered in the design of *Lope*, since the **engine** also signs them cryptographically to ensure their integrity. This makes especially sense when you are using encryption between **external connectors** and the **engine**, because the encryption mechanism is ChaCha20-Poly1305. This algorithm is considered an authenticated encryption which simultaneously provides the confidentiality and authenticity of data in motion. The combination of both assure that any generated activity is genuine, and the integrity is preserved from its creation to the reporting.

The activity signature is done using Elliptic Curve Digital Signature Algorithm (ECDSA), especially the curve which implements NIST P-256 (FIPS 186-3, section D.2.3).

1.14.1 Activities Reporting

There are plenty of options in the reporting aspect of the activities. in *Architecture* you can see the different reporting channels available to send alerts, and below in this section of the documentation you will find how to configure that integration. You can customize even how the reporting is done, in general you just tell report all activites for all enabled options doing this:

```
# -----------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# -----------------------------------------------------------
operation:

   # GENERAL ASPECTS
   # ===============
   ...


   # TO ENABLE TLS/DTLS USE
   # ======================
   ...


   # SOCKET CONNECTION CLOSE
   # =======================
   ...

   # ACTIVITY REPORTING
   # ==================

   # Report the acitivities for all reporting channels enabled
   alert_all_flag: yes # yes/no(default)

   # OTHER TOPICS
   # ============
   # Additional aspects should be defined here, but
   # they will be described in the respective sections
```

(continues on next page)

(continued from previous page)

```
    # of this documentation, for the sake of clarity
    ...
```

Or you can decide for which activity what reporting mechanism you want to use. You can do this by adding the `alert` field in the corresponding rule. If you enable the case for using all available channels (globally in the `operation` section, or in each rule), this disable the selection of specific reporting channels.

Basic Rules Reporting

For any basic rule you can do the following:

```
any_basic_rule:
    # What to send to the third parties
    # when the rule is executed
    # If empty or not present,
    # the rule is considered
    # 'executed' without sending
    # anything to the third party
    value: [message to be sent]

    # If the content of the field 'value'
    # is encoded in base64
    b64_flag: no # yes/no(default)

    # If the rule is enabled and
    # can be used for the interaction
    enable: yes # yes/no(default)

    # Alerting fields
    alert:
        # all channels, default option
        all: no # yes/no(default)

        email: yes # yes/no(default)

        http: no # yes/no(default)

        kafka: no # yes/no(default)

        syslog: yes # yes/no(default)

        slack: yes # yes/no(default)
```

For `timeout` and `ending` the basic rules, *Lope* will report their corresponding events, not the rule execution.

Custom Rules Reporting

For custom rules, the configuration is the same:

```
any_custom_rule:

   # GENERAL FIELDS
   # ==============
   ...

   # SOCKET CLOSE SCENARIOS
   # ======================
   ...

   # ALERTING FIELDS
   # ===============
   alert:
      # all channels, default option
      all: no # yes/no(default)

      email: yes # yes/no(default)

      http: no # yes/no(default)

      kafka: no # yes/no(default)

      syslog: yes # yes/no(default)

      slack: yes # yes/no(default)


   # OTHER FIELDS
   # ============
   ...
```

1.14.2 Memory Reporting

You can report snapshots of the memory variables in every activity as well. This will produce a large amount of data, so please take that in mind. This feature could be relevant for debugging purposes, but it is not recommended for real operation scenarios. To enable this, you can do it using the following fields in the engine configuration file:

```
# -----------------------------------------------------------
# Operational parameters of the interation
# and connection with third parties
# -----------------------------------------------------------
operation:

   # GENERAL ASPECTS
   # ===============
   ...
```

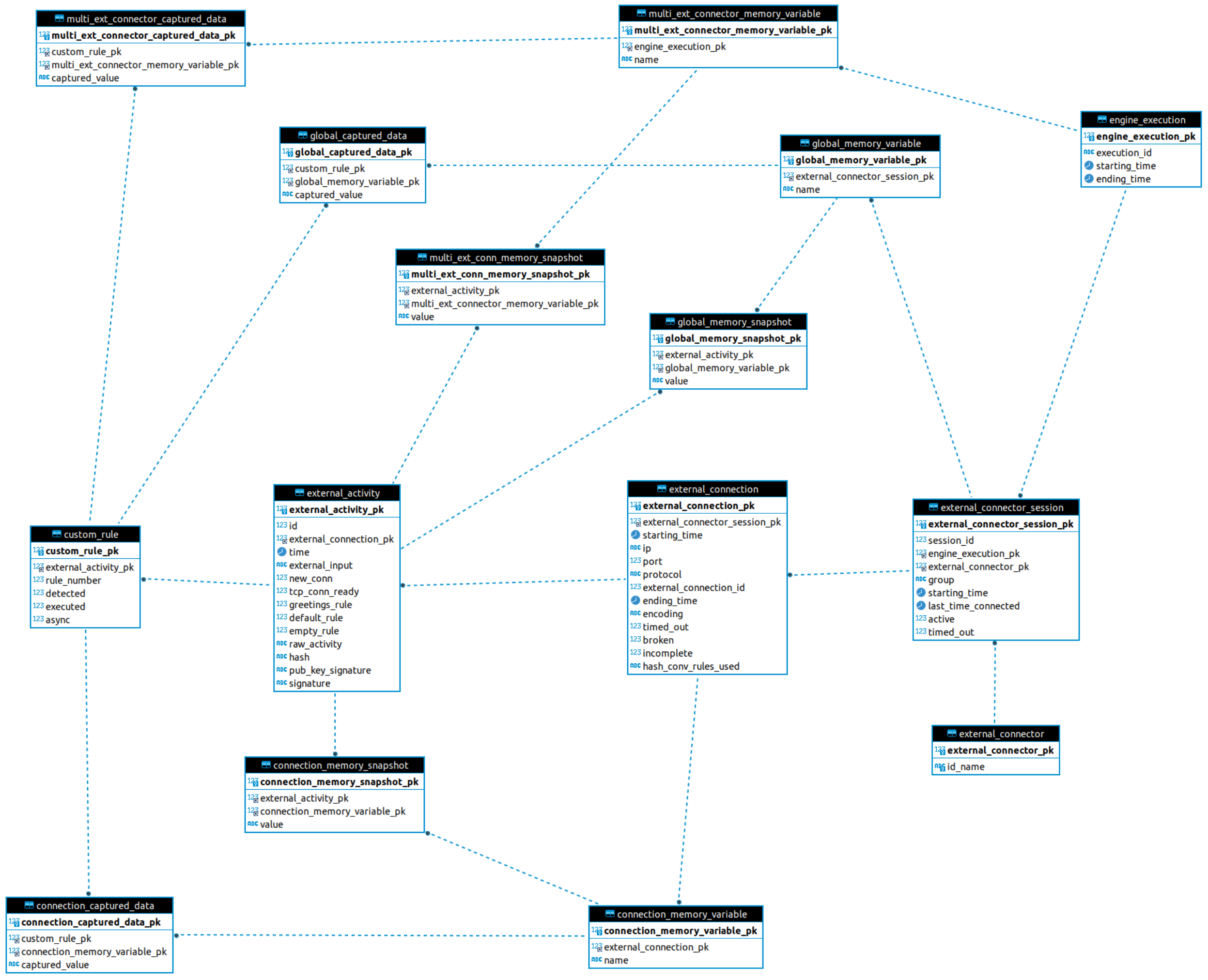
(continues on next page)

(continued from previous page)

```
# TO ENABLE TLS/DTLS USE
# ======================
...


# SOCKET CONNECTION CLOSE
# =======================
...

# ACTIVITY REPORTING
# ==================

# Report the acitivities for all reporting channels enabled
alert_all_flag: yes

# Report the status of the memory for each external activity generated
# (This adds many entries in the database)
report_memory: yes # yes/no(default).

# Encode b64 the memroy variables content at the time of reporting memory
encode_b64_memory_reported: yes # yes/no(default).


# OTHER TOPICS
# ============
# Additional aspects should be defined here, but
# they will be described in the respective sections
# of this documentation, for the sake of clarity
...
```

1.14.3 Activities Storage

Lope comes with a database schema ready to be imported into MariaDB or MySQL, ready to save any activity or any captured data for any interaction. Below you can find the Entity Relationship (ER) Diagram that allow to save all the data. In this diagram, an `external_connector_session` is any session that the **external connector** establishes with the **engine**. The `external_connection` is any session (or interaction) that the **external connector** establishes with a third party. As you will see, in any activity you can save the status of the memory variables in use as we have already explained.



Additionally, *Lope* provides simple file storage capabilities to create a file where all activities will be saved during the execution. The use of the database or the file case are independent, in the sense of using one does not affect the another. In order to configure the different persistence capabilities, you have to add the following in the engine configuration file (explained previously in *Engine Configuration*):

```
data_service:

    # Max number of data workers for the data service
    # to save information in parallel.
    # If '0' or negative values, then the default
    # value is used ('200')
    max_number_data_workers: 200


    # To just save the information in a simple file
    simple_file_storage:

       enable: no # yes/no(default)

       # not include the final '/', do not use '.'.
       # For using the current folder, use only this ""
```

(continues on next page)

(continued from previous page)

```
    folder_path: "/raw_data"
    is_relative_path: yes

    # To encode in b64 the activities to be saved
    encode_b64: no #yes/no(default).

  # Database use
  database:
    # mariadb / mysql: https://github.com/go-sql-driver/mysql
    # is the database enable for this execution?
    enable_sql_db: yes

    # Database credentials
    user: root
    password: toor

    # Database data
    ip: "127.0.0.1" # or url
    port: 3306
    schema: "slv_engine"

    # --------------
    # TLS
    # --------------
    # in case the connection with the database is encrypted using TLS
    tls_config:
      # is the connection with the database encrypted?
      enable: no

      ca_cert: ""
      # If a CA o custom CA is in use,
      # they are not self-signed certs (InsecureSkipVerify option)
      skip_certificate_verification: no
      relative_path_for_certificates: yes

      # For client TLS authentication (Lope => MariaDB/MySQL)
      engine_client_cert: ""
      engine_client_key: ""

      # in case the private key of the certificate is protected
      # using a password
      engine_client_key_protected: yes
      engine_client_key_protected_password: ""
```

1.14.4 Activities Alerting

To configure the integration with the respective alerting channels, you have to configure the `alerting` section of the engine configuration file, and the number of workers to parallelize the delivery of alerts. Then, you have to add the configuration channels you want to use.

```
alert_service:

 # Max number of data workers for the data service
 # to save information in parallel.
 # If '0' or negative values, then the default
 # value is used ('200')
 max_number_alert_workers: 200

 # Alerting channel configuration
 # (for those in use)
 syslog: ...

 kafka: ...

 email: ...

 http: ...

 slack: ...
```

Syslog Alerting

```
alert_service:

 # Max number of data workers for the data service
 # to save information in parallel.
 # If '0' or negative values, then the default
 # value is used ('200')
 max_number_alert_workers: 200

 # Alerting channel configuration
 # (for those in use)
 syslog:
    enable: no # yes/no(dafult)

    remote_flag: no # yes/no(dafult)

    # One of the following: tcp/udp(default)
    remote_syslog_protocol: "tcp"

    remote_syslog_server_ip: "localhost" # or domains

    remote_syslog_server_port: 515

    tag: "lope"

```

(continues on next page)

(continued from previous page)

```
    # send the activities encoded base64?
    encode_activity_base64: yes # yes/no(default)

    # --------------
    # TLS
    # --------------
    # in case the connection with the Syslog is encrypted using TLS
    tls_config:
      # is the connection with the database encrypted?
      enable: no

      ca_cert: ""
      # If a CA o custom CA is in use,
      # they are not self-signed certs (InsecureSkipVerify option)
      skip_certificate_verification: no
      relative_path_for_certificates: yes

      # For client TLS authentication (Lope => Syslog)
      engine_client_cert: ""
      engine_client_key: ""

      # in case the private key of the certificate is protected
      # using a password
      engine_client_key_protected: yes
      engine_client_key_protected_password: ""

  # Other alerting channels
  ...
```

Kafka Alerting

```
alert_service:

  # Max number of data workers for the data service
  # to save information in parallel.
  # If '0' or negative values, then the default
  # value is used ('200')
  max_number_alert_workers: 200

  # Alerting channel configuration
  # (for those in use)
  kafka:
    enable: no # yes/no(dafult)

    server_ip: 127.0.0.1 # Or domain

    server_port: 8888

    topic: "lope"

```

(continues on next page)

(continued from previous page)

```
    create_topic_if_not_exist: yes # yes/no(default)

    # Distribution or balancer, it must be one of the following:
    # 'LEAST_BYTES'(default),'CRC32BALANCER','MURMUR2BALANCER', 'HASH'
    distribution: "LEAST_BYTES"

    # One of the following: 'PLAIN','SCRAM' or 'NONE'(default)
    authentication_mechanism: "NONE"

    # Only in use for scam authentication, one of the following: 'SHA256' or 'SHA512
↪'(default)
    auth_scram_hash: "SHA512"

    # topic partition
    event_key: "lope"

    timeout: 10 # seconds

    user: ""

    password: ""

    # send the activities encoded base64?
    encode_activity_base64: yes # yes/no(default)

    # --------------
    # TLS
    # --------------
    # in case the connection with the Kafka is encrypted using TLS
    tls_config:
      # is the connection with the database encrypted?
      enable: no

      ca_cert: ""
      # If a CA o custom CA is in use,
      # they are not self-signed certs (InsecureSkipVerify option)
      skip_certificate_verification: no
      relative_path_for_certificates: yes

      # For client TLS authentication (Lope => Kafka)
      engine_client_cert: ""
      engine_client_key: ""

      # in case the private key of the certificate is protected
      # using a password
      engine_client_key_protected: yes
      engine_client_key_protected_password: ""

# Other alerting channels
...
```

Email Alerting

```
alert_service:

 # Max number of data workers for the data service
 # to save information in parallel.
 # If '0' or negative values, then the default
 # value is used ('200')
 max_number_alert_workers: 200

 # Alerting channel configuration
 # (for those in use)
 email:
    enable: no # yes/no(dafult)

    smtp_ip: "smtp.gmail.com" # or domains

    smtp_port: 587

    # One of the following: 'PLAIN'(default),'CRAMMD5'
    authentication_mechanism: PLAIN

    # SMTP
    # https://pkg.go.dev/net/smtp
    smtp_plain_auth_identity: ""

    smtp_auth_user: "seclopedevega@gmail.com"

    smtp_auth_password: ""

    smtp_plain_auth_host: "smtp.gmail.com"

    # email fields
    from: "seclopedevega@gmail.com"

    # lists of email addresses for different fields
    reply_to:
       - "seclopedevega@gmail.com"

    to:
       - ""

    cc:
       - ""

    bcc:
       - ""

    subject: "[SecLopeDeVega][Activity Report]"

    # The activity is added after the body intro and before the body end
    body_intro:
       "
```

(continues on next page)

(continued from previous page)

```
      <h2> Hello there, it's Lope</h2>\n
      ---------------------------------\n
      ACTIVITY = "

   body_end:
      "
      \n\n---------------------------------\n
      "

   # send the activities encoded base64?
   encode_activity_base64: yes # yes/no(default)

   # --------------
   # TLS
   # --------------
   # in case the connection with the SMTP is encrypted using TLS
   tls_config:
      # is the connection with the database encrypted?
      enable: no

      ca_cert: ""
      # If a CA o custom CA is in use,
      # they are not self-signed certs (InsecureSkipVerify option)
      skip_certificate_verification: no
      relative_path_for_certificates: yes

      # For client TLS authentication (Lope => SMTP)
      engine_client_cert: ""
      engine_client_key: ""

      # in case the private key of the certificate is protected
      # using a password
      engine_client_key_protected: yes
      engine_client_key_protected_password: ""

 # Other alerting channels
 ...
```

HTTP Alerting or Webhook

```
alert_service:

 # Max number of data workers for the data service
 # to save information in parallel.
 # If '0' or negative values, then the default
 # value is used ('200')
 max_number_alert_workers: 200

 # Alerting channel configuration
 # (for those in use)
```

(continues on next page)

(continued from previous page)

```
http:
   enable: no # yes/no(dafult)

   url: "http://127.0.0.1"

   # One of the following: 'POST'(default),'PUT' or 'GET'
   method: GET

   timeout: 10 # seconds

   # HTTP Headers
   headers:
      # header_key: header_value
      aaa: "aaaa"
      bbb: "bbb"

   # Query string parameters. If the activity must be sent via parameters,
   # use '{{ACTIVITY}}' in the parameter value
   url_parameters:
      # parameter_key: parameter_value
      aaa: "aaaa"
      bbb: "bbb"

   # This parameter allows to send the activity in the HTTP body,
   # it is recommended to disable it for HTTP GET method
   use_body_flag: yes # yes/no(default).

   # The activity is added after the body intro and before the body end
   body_intro:
      "
      Hello there, it's Lope</h2>\n
      ---------------------------------\n
      ACTIVITY = "

   body_end:
      "
      \n\n---------------------------------\n
      "

   # send the activities encoded base64?
   encode_activity_base64: yes # yes/no(default)

   # --------------
   # TLS
   # --------------
   # in case the connection with the HTTP Server/Service is encrypted using TLS
   tls_config:
      # is the connection with the database encrypted?
      enable: no

      ca_cert: ""
      # If a CA o custom CA is in use,
```

(continues on next page)

(continued from previous page)

```
      # they are not self-signed certs (InsecureSkipVerify option)
      skip_certificate_verification: no
      relative_path_for_certificates: yes

      # For client TLS authentication (Lope => HTTP Server/Service)
      engine_client_cert: ""
      engine_client_key: ""

      # in case the private key of the certificate is protected
      # using a password
      engine_client_key_protected: yes
      engine_client_key_protected_password: ""

 # Other alerting channels
 ...
```

Slack Alerting

```
alert_service:

 # Max number of data workers for the data service
 # to save information in parallel.
 # If '0' or negative values, then the default
 # value is used ('200')
 max_number_alert_workers: 200

 # Alerting channel configuration
 # (for those in use)
 slack:
    enable: no # yes/no(dafult)

    # Slacks tokens
    oauth_token: [slack OAuth token]
    app_level_token : [slack app level token]

    # Slack URL or IP
    Url:

    # Slack Channel to use
    channel_id: [the id of your channel]

    # Debug the connection flag
    debug_flag: no # yes/no(default)

    # Pretext to use in the Slack messages.
    # If not present, or empty, it will not be used
    pretext: [any pretext]

    # The color to use for your messages:
    text_color: [hex color]
```

(continues on next page)

(continued from previous page)

```
    # The activity is added after the body intro and before the body end
    body_intro:
       "
       Hello there, it's Lope</h2>\n
       ---------------------------------\n
       ACTIVITY = "

    body_end:
       "
       \n\n---------------------------------\n
       "

    # send the activities encoded base64?
    encode_activity_base64: yes # yes/no(default)

# Other alerting channels
...
```

1.15 Quick Start

Lope comes with example interaction ready to show the capabilities it provides. In order to 'play' with *Lope*, first you have to decide if you want to compile the **engine**, or just using the compiled version for linux x64. In any case, take a look at the instructions described in *Engine Installation & Execution*. For the **external connector**, yo need to install Python 3, as the corresponding libraries as explained in *External Connector Installation & Execution*.

Lope is configured to run this example interaction by default, working as a server in the port 8888. The interaction that will execute is the one described in the `cr_test_neo-yml`. The version 2 is almost the same, it only use different concersation rules files, instead of using only one file. The approach is a kind of self-guided tutorial where you can just add simple inputs (e.g. `x1x`, `x2x`, `y1y`, `n1n`, etc.) to see what happens. To do that, you have first to prepare the environment and, once everything is ready, then you have to start the **engine** first. Take a look at the logs to see if you have the following message:

`Engine up and running, ready to receive messages of external connectors!`

This means the **engine** has started correctly. Then, you have to start the **external connector** and review the logs. If everything is ok, then you should see something like this:

`[InteractionWrkr][22-10-15][09:57:34][INFO]: Listening the socket in the port:  8888`

Then, you can interact with *Lope*, our recommendation is using ncat and do the following:

`ncat 127.0.0.1 8888` (provided *Lope* is running in your local machine).

Just navigate in the *conversation rules* file and check the options you have and then, add the right input to see the effects. Additionally, you have other two example interactions the test client and the server. You can start an **external connector** as client (providing the id: `ExtConTest_Client`) and another as server (providing the id: `ExtConTest_Server`), and then, the client **external connector** will interact with the server one. For this test, you will need to have the Redis server in place, and we recommend to use wireshark to see what is happening in the interaction (port 8080). In any test, please review the configuration files to check if everything is correct anyway.

1.16 Potential Use Cases

In this section, we provide some ideas about in what use cases we think *Lope* might help you. Please, be aware the success of implementing those potential use cases

1.16.1 Server Mode

In this mode, *Lope* is waiting for third party connections to start interacting. The main use here is about being a honeypot, or a set of them by providing a more realistic experience. Since this software may emulate real scenarios, this could be used as a fake training IT environment for some cases. Since all the generated activities are signed, you can prove that the things happened in the interaction, have indeed happened. The following list is some use cases that we have identified that could be relevant:

- **Simple Honeypot**: Just create an interaction to be executed by an **external connector**, and wait to get the activities about what happens.
- **Complex Honeypot Scenario**: Create a set of **external connectors** working together, where what happens none of them is affecting how others react. You can start new ones dynamically by using your own code, or using the *fork* functionality.
- **Perform Security Tests in a Client by Sending Tests from the Server Side**: Sometimes, perform security test in client software is hard because it is not easy to send back some things to test the software (e.g.: input validation). *Lope* can help you in this by pretending to be the corresponding server and send something to validate the client software.
- **'Training Arena'**: As mentioned, since *Lope* may behave as a real system (or set of them), provided that you have worked the right interactions, you might create an environment for 'training', and even to evaluate the performance more officially (activities are signed, do you remember?)

1.16.2 Client Mode

This mode is more about checking if a running server software is working as expected, in the sense of security or quality. This can perform different tests in different application protocols by having the right *conversation rules* for doing so. As you know, the activities results are tamper-proof, so you can attest how a server software status is (in terms of security, quality, or what ever.).

- **Navigation in Web Applications**: You can navigate in web applications to test how the software is working, and even to perform web-scrapping in the target websites with the right interactions. You can emulate user interactions at some extent, that allows you to test navigation and performance.
- **Testing APIs**: Similarly to the previous point, you can also work with APIs and verify their behavior. This cal help you to verify API contracts, and perform some dynamic testing to ensure the API is working as it should be.
- **Perform Security Tests**: If you can test any exposed software in terms of quality, you can do it in terms of security (provided that you have the right *conversation files*). Since you can create new **external connectors** during the test dynamically, you can get information from other place and go on with the interaction (for instance, being able to pass single sign-on). TLS authentication is also supported, so you might test systems that require it as well. In case of application protocols where there are not other security tools able to work with that, this tool might help you to do security test because it tries to be as agnostic as possible from the application protocols, but limited to TCP or UDP.
- **Perform Load and Stress Testing**: Considering that one **external connector** can perform several connections against a target (M), and you can have several external connectors working in parallel (N); this means you can have M x N connections against a target performing interactions (that can be simple, or can be complex emulating real expected interaction). This can help you in checking if the software is able to support, and to what extent, the high load situations.

- **Perform Denial of Service Testing**: In the same way of thinking with the previous point, you can directly try to test if the software can support some denial of service attacks. Not only some that are based on brute force, even other that are more 'intelligent' or based on interactions (IDoS?).
- **Assist Other Testing Tools**: Since you can add your code to this tool, this allows you to mix *Lope* capabilities with other tools. *Lope* can work as a kind of orchestrator and perform some tests, while other tools perform others. Each tool will generate their corresponding results independently, but *Lope* can get help in organizing a unified testing experience. A simple example could be that you need to pass a single sign-one, and provide the session cookie to another security tool to access the system.

1.17 Contact

If you want to contact us, we recommend you to do it via gitter, or in the google groups. For more private talks, just send us an email.

CHAPTER

TWO

INDEX AND SEARCH

- genindex
- search

INDEX