Scenario Developer Manual

This page aims to be a manual to help developing in OpenBACH API.

For a detailed description of the helpers and the reference scenarios, we encourage you to read the introduction to OpenBACH-extra and OpenBACH API.

Philosophy and Conventions

In order to keep a clean API, and ease first-time users understanding, here is some conventions proposed for developing reference scenarios.

If you want to have your scenarios included in OpenBACH, please follow them.

Key principles

Following some of the principles in the Zen of Python, here are a few simple rules to keep in mind:

- Explicit is better than implicit: expose all scenarios parameters in functions signatures;
- Simple is better than complex: a single exposed scenario per file;
- Flat is better than nested: the lesser sub-scenarios, the better.

Reference Scenario File layout

```
"""Module Docstring.

Describe the steps an phylosophy of your scenario.
"""

imports


# Constants
SCENARIO_DESCRIPTION = """Description of what the scenario accomplishes
 - step by step
 - walkthrough
 - if need be
"""
SCENARIO_NAME = 'default_name_for_scenario'
# Other constants
# Custom types needed as parameters


# utility functions
def _utility_function(…):
    …

# "main" scenario functions
def scenario_component(…, scenario_name=SCENARIO_NAME):
    …
    return scenario
```

```
def alternate_scenario_component(…, scenario_name=SCENARIO_NAME):
    …
    return scenario

# exposed "whole" scenario
def build(…, post_processing_entity=None, scenario_name=SCENARIO_NAME):
    scenario = …
    if post_processing_entity is not None:
        …
    return scenario
```

The layout used should be the usual Python layout made of, in this order, module docstring, imports, constant declarations, classes (mostly namedtuples and custom argument types), and functions.

- Constants should contain `SCENARIO_NAME` and `SCENARIO_DESCRIPTION` at the top;
- Functions constructing and returning scenarios should define a `scenario_name=SCENARIO_NAME` parameter at the end;
- Utility functions and other non-exposed variables should use a name with a leading underscore.

Guidelines

- Each time you need to define a new scenario (using `scenario_builder.Scenario`), it must be wrapped in its own function: this ease reusability and parameters discovery through the function signature.
- Scenarios should be built by combining one or several helpers; you may want to use OpenBach parameters as arguments to those helper to ease modification through the HMI, this is fine as long as you use `Scenario.add_constant` instead of `Scenario.add_argument`. The latter making it harder to know, for an unsuspecting user, what values are expected when using the scenario as a sub-scenario.
- The `build` function should be considered the “official” finished scenario of the file: it uses other functions to create a complete scenario and optionally add post-processing capabilities. Usage may look like `import scenario_module_file; scenario_module_file.build(…)`.
- The `build` function is encouraged to define a `scenario_name=SCENARIO_NAME` parameter that will be forwarded to other scenario functions called. Beware, though, that naming two scenario with the same name will have the latter one override the former one in the OpenBach database.
- The `build` function is encouraged to use scenarios constructed by the other functions and add behaviour (such as post-processing) to those scenarios directly; restrict the use of sub-scenario to the minimum (to ease scheduling, for instance).

Naming Conventions

Scenarios

File names and default scenario names (the `SCENARIO_NAME` constant) share the following four options:

- `Protocol_Layer_type_of action`: *e.g.* `network_configuration_link`
- `Protocol_Layer_traffic type`: *e.g.* `service_video_dash`

- `Protocol_Layer_main statistic evaluated`: *e.g.* `network_delay` (if you want to highlight a measurement or performance evaluations (e.g. delay, jitter, rate, etc.)
- `Protocol_Layer_traffic type_main statistic evaluated`

Protocol_Layer is one of: service, transport, network, access or physical.

Arguments

Naming:

- entities names should describe which entity it applies to, when applicable. *e.g.*: `client_entity`, `server_entity`;
- IP and port arguments should describe which entity it applies to, when applicable. *e.g.* `client_ip`, `server_port`;

Order of arguments:

1. entity names (except for post-processing in the `build` function);
2. IPs;
3. Ports;
4. Duration (if exists);
5. Other arguments;
6. Post-processing;
7. Scenario name.

Using the executors, the references scenarios and the helpers

There are three ways to help the user to create and launch scenarios:

- The **helpers** aim to simplify the scripting of your scenarios, by giving wrappers for common functions or tasks (iperf measurements, voip/dash transmission, etc.);
- The **reference scenarios** propose a set of relevant scenarios (for example for evaluating your network in terms of delay, jitter and rate, or to configure OpenSAND, to launch hybridation scenarios, etc.).

Some useful information on helpers and scenarios:

- **Helpers** can be imported in any scenario.
- **Helpers** can provide a simple way to wrap job instances (in one simple code line): e.g. to easily set up a route or configure an interface.
- In terms of scripting, **helpers** can help to easily deploy an iperf3 server along with an iperf3 client.
- Helpers are included in (reference) scenarios.
- Scenarios, whether they are **reference scenarios** or your own, along with **helpers**, whether they are available in OpenBACH or specifically developed, can be included in more complexed scenarios.

How to combine them?

- A **helper** would allow you to launch (in one Python line) an iperf3 server and an iperf3 client to measure the rate of a link.
- A **reference scenario** would allow to prepare your entities (link or tcp stack configuration), to launch sequentially or simultaneously the above iperf3 helper, and another similar helper using the job nuttcp, and finally to launch some postprocessing to compare the results.
- A **executor** would let you launch the above scenario from CLI or generate a JSON to be imported on the web interface as described in Simple Command Line Interface User Manual.

This manual contains guides on how to develop and use the helpers, the reference scenarios, and the executors as you work toward expanding OpenBACH capabilities.

Examples of scripts using the scenario builder are available in the section Reference Scenarios and its subsections.