Scenario Observer and Executors Manual

Scenario Observer

The `scenario_observer.py` script is particular as it is not meant to be used from the command line directly. Instead it serves as a building block to create, save/modify, and run scenarios built using the scenario_builder API. It can also easily generate JSON files instead.

Follow the steps listed here to generate/launch the scenario below:

1. In order to use the API, import the `Scenario` class from scenario_builder and the `ScenarioObserver` from the auditorium-scripts.
2. Create a function that builds your scenario (`build_delay_scenario()`): a fping and a hping launched at the same time, that will be stopped after 20 seconds. The IP destination address should be given as an argument.
3. In the main function, you should create your `ScenarioObserver`, add your arguments (the names of your entities and the argument of your scenario).
4. Pass your arguments to the scenario and use the `launch_and_wait()` function.

delay_scenario.py

```
#Import the Scenario class from scenario_builder and the ScenarioObserver from the
auditorium-scripts.
from scenario_builder import Scenario
from auditorium_scripts.scenario_observer import ScenarioObserver


#Your scenario
def build_delay_scenario(client, server, scenario_name):
    scenario = Scenario(scenario_name, 'Comparison of 2 types of RTT measurements')
    scenario.add_argument('ip_dst', 'Target of the pings and server ip adress')

    hping = scenario.add_function('start_job_instance')
    hping.configure('hping', client, offset=0, destination_ip='$ip_dst')
    fping = scenario.add_function('start_job_instance')
    fping.configure('fping', client, offset=0, destination_ip='$ip_dst')
    stop = scenario.add_function('stop_job_instance', wait_launched=[hping, fping],
wait_delay=20)
    stop.configure(hping, fping)
    return scenario


def main(scenario_name='Delay metrology scenario'):
    observer = ScenarioObserver()
    observer.add_scenario_argument(
            '--client', '--client-entity', default='Client',
            help='name of the entity for the client of the RTT tests')
    observer.add_scenario_argument(
            '--server', '--server-entity', default='Server',
            help='name of the entity for the server of the owamp RTT test')
    observer.add_run_argument(
            'ip_dst', help='server ip address and target of the pings')
    args = observer.parse(default_scenario_name=scenario_name)

    built_delay_scenario = build_delay_scenario(args.client, args.server, scenario_name)
```

```
    observer.launch_and_wait(built_delay_scenario)


if __name__ == '__main__':
    main()
```

Developing and using executors

These executor are wrappers to configure and launch reference scenarios. Reference executors are a 1-to-1 mapping to reference scenarios. Example executors may combine several scenarios in an effort to showcase a complete (setup to teardown) platform benchmark.

A few rules of thumb when developing executors:

- Use the `ScenarioObserver` to handle command-line arguments and scenario execution;
- Do not use arguments names with space: use `-` as a separator on the command-line, it will be converted to `_` as separator in the Python variable name;
- Do not use existing argument names, such as `–path`, `–login` or `–interval`, or it you will run into runtime errors; try to use more specific names;
- Name the argument defining a post-processing entity `–post-processing-entity` for consistency.

You can get a list of existing command-line parameters using

```
PYTHONPATH=~/openbach-extra/apis/ python -c "from auditorium_scripts.scenario_observer import
ScenarioObserver; ScenarioObserver().parse(['-h'])"
```

When defining complex behaviour, especially a scenario needing to be included several times with different parameters, it is advised to define custom types to validate the arguments through `argparse`. Using a custom `Action`, it is possible to validate several arguments of different types at once **and** store them in a list for easier iteration. If need be, you can make the arguments as complex as needed: the command-line users will be able to use a file to store and organize their options; the service traffic mix executor and its associated example arguments file are an example of that, run it using:

```
PYTHONPATH=~/openbach-extra/apis/ python3 executor_service_traffic_mix.py
@executor_service_traffic_mix_arg.txt MyProject run
```

Inspiration from Other Executors

The example folder of executors aims at showcasing extra behavior available when combining several scenarios. You can take inspiration from files in this folder when creating more complex scenarios.

Bear in mind that there are two major ways of combining scenarios, each with their pros and cons:

- Use the `launch_and_wait` method of your `ScenarioObserver` several times in a row with different instantiations of the (reference) scenarios. This have the advantage of an easy setup and short development time as you mostly reuse existing assets. You may however not find

every launched scenarios in the Web Frontend as reusing the same scenario several times in a row will override it over and over again, leaving only the last instance in the OpenBACH database.
- Build a composite scenario using several sub-scenarios and use a single `launch_and_wait` on the whole. This have the advantage that each sub-scenario will be available in the Web Frontend and it make it easier to retrieve data from the one owner scenario. But building such scenario may be more time consuming, error-prone (beware of your sub-scenario names) and/or less maintainable.

Post-Processing

The `scenario_observer.py` script provides a `DataProcessor` class alongside the `ScenarioObserver`. Its purpose is to retrieve data from the executed scenario in order to provide custom post-processing capabilities that are not available through the usual post-processing jobs. You can, for instance, easily compare data from various sub-scenarios on a single plot.

To use the `DataProcessor`, provide it with an instance of a `ScenarioObserver` so it can use the collector credentials to download the data. You can also provide a scenario instance to retrieve data from, but it will use the last instance run by the observer if you don't.

You can then use the `add_callback` method to associate a label (a key) to a callback/openbach_function pair. The openbach function will be mapped to a job instance ID by the `DataProcessor` and the associated data will be fed to the callback. The value returned by the callback will be put into a dictionary under the specified label (key). In order to start this mapping and callback process, you have to call the `post_processing` method, that will return said dictionary ; you can thus specify several callbacks at once before fetching the data.

It is up to you to extract the required data using the callbacks before plotting them using the resulting dictionary.

From:
https://wiki.net4sat.org/ - **Net4sat wiki**

Permanent link:
https://wiki.net4sat.org/doku.php?id=openbach:manuals:2.x:developer_manual:scenario:executors_manual:index

Last update: **2020/06/18 15:54**