OpenBACH design

David PRADAS

Table of Contents

Chapter 1. Context of OpenBACH

The objective of this study is to design and implement a benchmark for performing tests and collect different metrologies to allow the evaluation of next proposed transport protocols and mechanisms on a terrestrial, satellite and hybrid network. It is important to implement an adequate measurement environment to provide an appropriate evaluation framework.

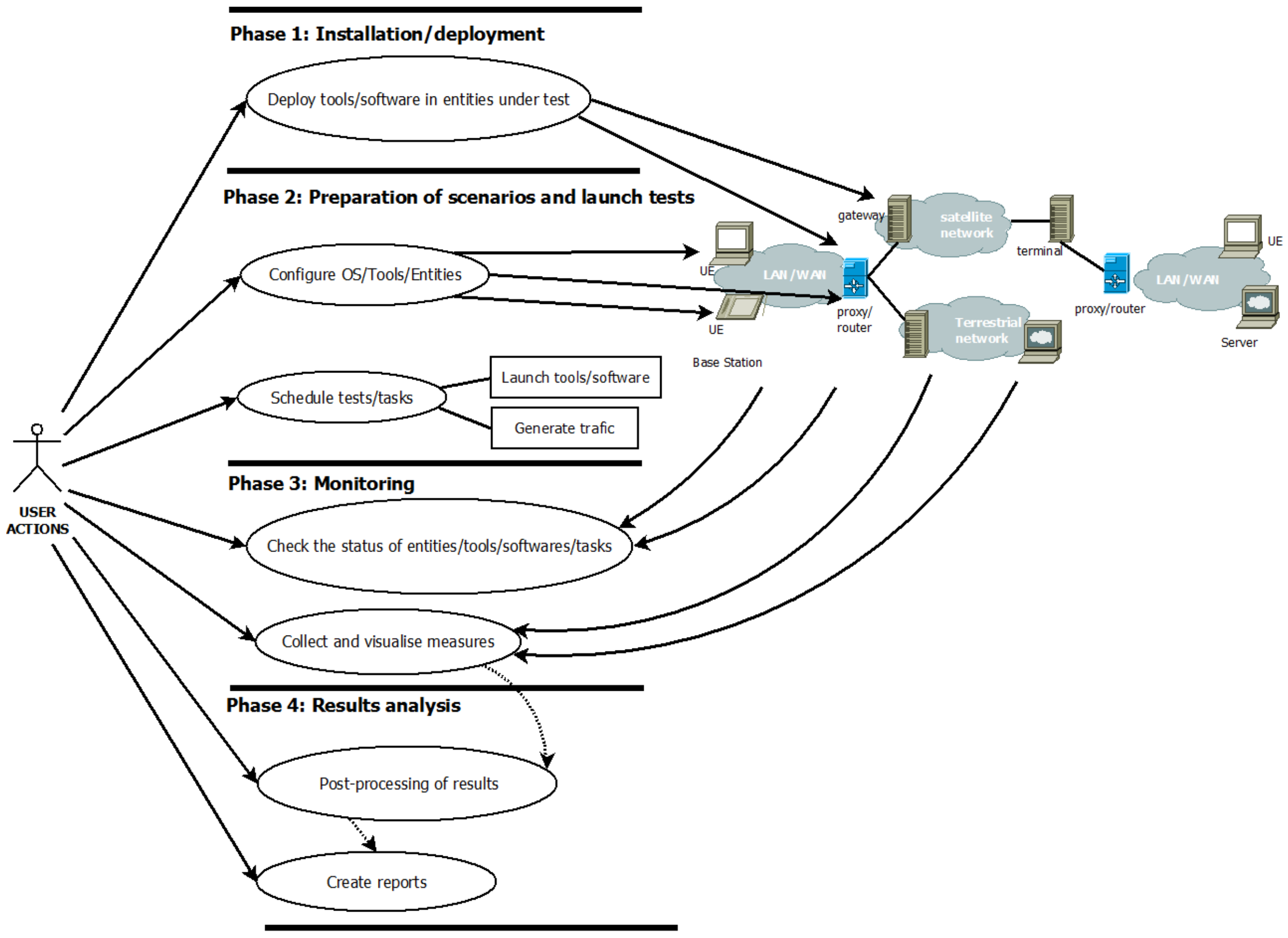
The transport benchmark would be able to quickly benefit different actors for the development of satellite network and telecom technologies, but also the actors of terrestrial networks, which would allow to extend the number of potential users.

This benchmark must propose to integrate existing metrology tools, or exceptionally developed according to specific needs. It must be easily operated and at the same time deliver a maximum of useful information for the understanding, the validation and the design of existing or under development transport mechanisms. The benchmark shall also be capable of analyzing the deployment of protocols and the options supported on clients or remote servers. In addition, different visualizations tools and statistical analysis shall be proposed. Moreover, tools allowing to configure the network must be implemented in order to facilitate the implementation of exhaustive testing, reproducible or even automated. Traffic injection or traffic generation will also be deployed within the benchmark.

The test bed will be developed in an open source logic similar to the one used for OpenSAND. The components will eventually be interfaced and / or integrate OpenSAND, in a logic that will be jointly defined by CNES, Thales Alenia Space, and Viveris Technologies at the end of the study.

Chapter 2. Generic use case of OpenBACH

The objective of this project is thus to create a benchmark allowing a user to perform a set of actions presented in Figure 1. These actions are grouped in 4 different phases:



Phase 1: Installation/deployment
Deploy tools/software in entities under test
Phase 2: Preparation of scenarios and launch tests
Configure OS/Tools/Entities
Schedule tests/tasks
Launch tools/software
Generate trafic
Phase 3: Monitoring
Check the status of entities/tools/softwares/tasks
Collect and visualise measures
Phase 4: Results analysis
Post-processing of results
Create reports
USER ACTIONS
gateway
satellite network
terminal
UE
LAN/WAN
proxy/router
Terrestrial network
Base Station
Server

Figure 1. Generic use case of OpenBACH

- **Phase 1**: Installation/deployment
  - In this phase, a user would wish to perform and easy and straightforward installation/deployment of any tool and/or software that he needs in order to evaluate a network under test. That includes the installation of OpenBACH itself.
- **Phase 2**: Preparation of scenarios and launch tests
  - In particular, the user would wish to configure all software/tool and entities under evaluation (including OS parameters, IP, TCP stack, etc), and
  - Schedule and launch all the required tasks/software/tools for evaluating protocols, mechanisms in the network under test. That might include traffic generators, the launch of measures acquisition.
- **Phase 3**: Monitoring
  - The user would wish to verify that the tests are correctly running by checking the status of all entities under test, tools, running software and/or tasks (via logs and status values), and

    - Collect the measures that have been scheduled in the previous phase and visualize real-time results.

- **Phase 4**: Results analysis
    - Once the tests are finished, the user may wish to perform post-processing actions according to the user needs. In this phase the user shall be able to perfom from simple actions such as shaping the obtained results (obtain averages/max/min) or more complex actions such as computing new results from using different inputs measures.
    - Finally, the user shall be able to visualize this new results and stock them (i.e. images, tables, etc) for creating reports of the study that has been carried out.

OpenBACH is designed in order to be capable of fulfilling the needs of a user that wishes to perform any of the actions detailed above.

Chapter 3. Functional design

This sections aims at describing the overall view of the functional architecture design for OpenBACH (Benchmark Automation tools for Communication and Hypervision).

3.1. Terminology of OpenBACH

First of all, the following table allows to define some terms used in this document and in particular, those that compose the backbone of an OpenBACH scenario:

Table 1. OpenBACH terminology

Terms	Definition
Job[1]	A number of individual tasks (one or more) with a common purpose and to be executed in a sole Agent. A job might be able to launch/configure other software tools (e.g. ping/iperf) and OS tools (e.g. iptables), configure OS parameters, collect/generate information/stats from tools/OS, etc.
Job instance	An execution of a job configured with a set of parameters.
openbach-function	Function defined and launched in the Controller allowing to perform tasks related to: install agents/Jobs, configure and schedule Job/scenario instances, perform information/status requests regarding Agents/Jobs/Scenarios and their instances, etc.
openbach-function instance	An execution of an openbach-function with a set of parameters.
Scenario	A set of openbach-functions that allow to perform different tasks on one or more Agents.
Scenario instance	An execution of a scenario with a set of parameters.

[1] **A classification of Job types depending on their purpose is defined in this document.**

Other terms regarding the design of OpenBACH and used in this document are defined in the following table:

Table 2. Design terminology

Terms	Definition
Network Under Test	Network under test allowing to interconnect different network entities. The real traffic (e.g. HTTP, Video streaming, etc.) is sent through this network, and it will be possibly monitored by OpenBACH
Management network	Logical or physical network independent from the Network under test (or dedicated bandwidth of the physical network) allowing to interconnect each network entity with the collector and the controller of OpenBACH. This network is used to send all the signalization/messages of control, monitoring, etc., related to OpenBACH.
Frontend	It is the presentation layer and what the user is able to see, i.e. the interface between the user and the data access layer (in the backend). In summary, a mix of programming and layout that powers the visuals and interactions of the web.

Terms	Definition
Backend	It is seen as the servers-side code which has access to the data, and implements functions to manipulate this data and to use it for different purposes. In the case of OpenBACH, the backend contains the intelligence of the benchmark, i.e. the functions that allow to perform different tasks.

3.2. Design components

OpenBACH shall implement the components Controller, Collector, Auditorium and different Agents. Their roles are detailed next:

- A Controller shall centralize and deploy the configuration functionalities of OpenBACH as well as the jobs/scenarios to be launched.
- The Collector shall be able to collect all the statistics, data, logs/errors and other messages generated by the Job instances and the overall structure of OpenBACH.
- The Agents shall be deployed in the different end network entities (work stations, terminals, etc.), middle entities (server, proxy, etc.) that are supposed to be controlled by OpenBACH, or even in the same entities where the Controller and Collector are deployed. The Agents shall control (schedule/launch/stop) the jobs within a network entity according to the Controller commands, and collect the local stats/logs sent by these jobs. As we will see, an Agent might be placed next to the Collector and/or the Controller.
- The Auditorium component shall centralize the different frontend interfaces for configuring and monitoring (logs and statistics) the benchmark.

A basic functional scheme of OpenBACH is represented in Figure 2. From the Auditorium, a user shall be able to configure OpenBACH and request information of it (status of entities and components). The configuration is centralized at the Controller, which is in charge of deploying this configuration to the required Agents (the configuration might also include the deployment of new Agents and Jobs) and asking for status information. The Agents execute/schedule/stop the Jobs and relay the informations to be collected (statistics/logs/status) to the Collector, which centralizes all the data from all the available Agents/Jobs. Once the information is stocked in the Collector, the Controller is able to perform requests of data regarding the status of OpenBACH (in order to be sent to the Auditorium), and the Auditorium is able to make requests logs and statistics in order to allow the visualization in the user PC screen.

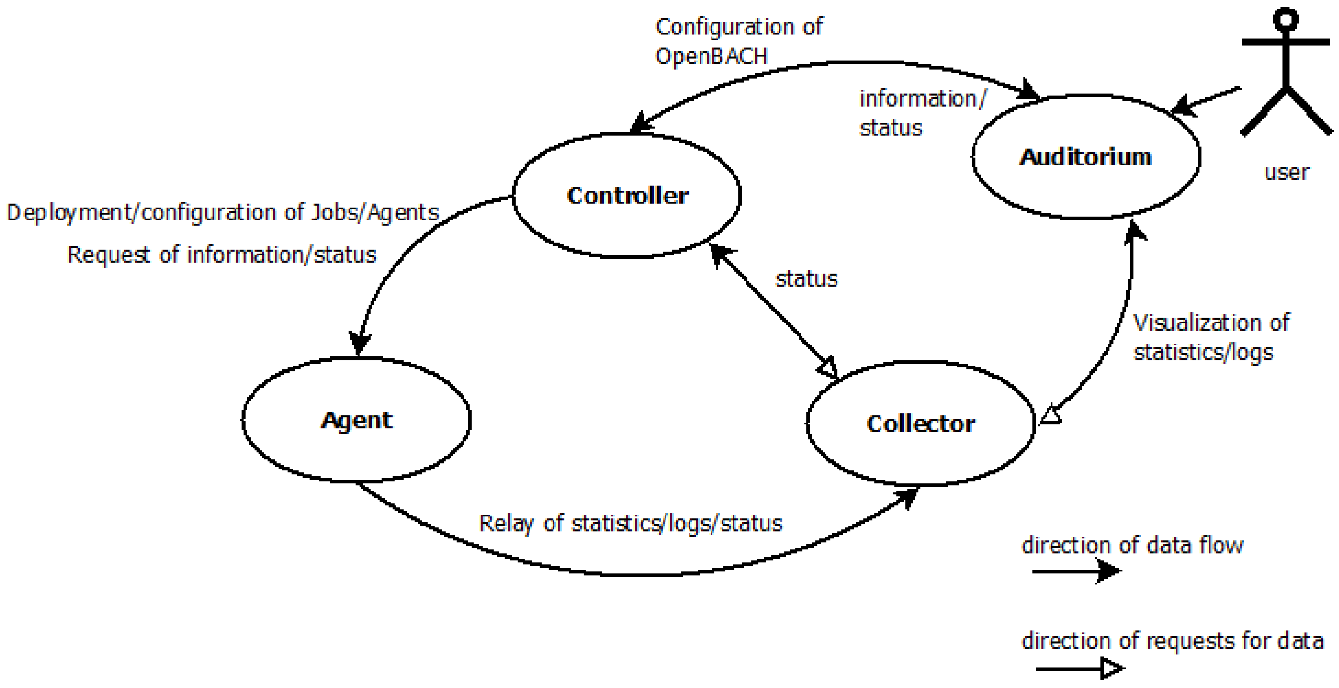

Configuration of OpenBACH
information/ status
Auditorium
user
Controller
Deployment/configuration of Jobs/Agents
Request of information/status
status
Visualization of statistics/logs
Agent
Collector
Relay of statistics/logs/status
direction of data flow
direction of requests for data

Figure 2. Design of OpenBACH interfaces

3.3. Functionality groups

OpenBACH shall propose two main functionalities: the configuration of the benchmark (including the available jobs) and the collection of relevant data.

These two types of functionalities are well identified by color in the architecture shown in Figure 3:

- Configuration (purple boxes/arrows): includes configuration of jobs, scenarios, entities, scheduling of jobs/scenarios.
- Collection and display of statistics and logs/status (blue boxes/arrows) allowing to monitor the Network under Test.

3.3.1. Functional blocks per component

Below, we list the functional blocks per component as well as the types of data flows between them that OpenBACH shall implement. The functional architecture is shown in Figure 3.

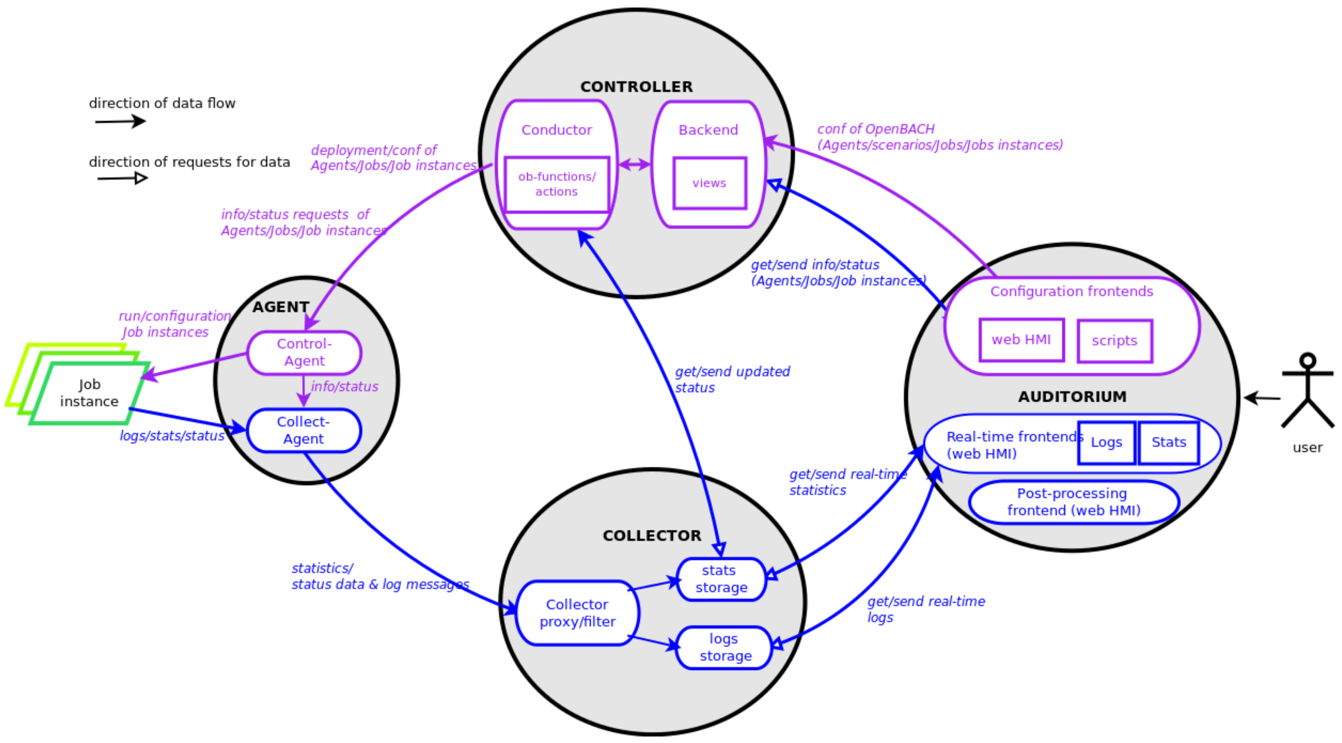

direction of data flow
direction of requests for data
CONTROLLER
Conductor
ob-functions/ actions
Backend
views
deployment/conf of Agents/Jobs/Job instances
info/status requests of Agents/Jobs/Job instances
conf of OpenBACH (Agents/scenarios/Jobs/Jobs instances)
get/send info/status (Agents/Jobs/Job instances)
AGENT
Control-Agent
info/status
Collect-Agent
run/configuration Job instances
Job instance
logs/stats/status
get/send updated status
AUDITORIUM
Configuration frontends
web HMI
scripts
Real-time frontends (web HMI)
Logs
Stats
Post-processing frontend (web HMI)
user
get/send real-time statistics
get/send real-time logs
statistics/ status data & log messages
COLLECTOR
Collector proxy/filter
stats storage
logs storage

Figure 3. Design of OpenBACH components

The Auditorium shall implement several frontends, one per type of display:

- Two frontends for the configuration of OpenBACH:
    - Web interface-based: a user web interface allowing to configure and schedule the available Jobs/scenarios of OpenBACH
    - Python scripts-based: allowing also to configure/schedule the available Jobs on each Agent from a Linux shell terminal. These scripts allow a more fine-grained control to the available actions as well as unique ones that are not available from the Web interface.
- Two frontends (web interface-based) for displaying real-time data:
    - A dashboard frontend for log messages allowing to get, filter and show the collected logs of the benchmark.
    - A dashboard frontend for real-time statistics allowing to display the collected metrics.

The Controller shall implement:

- A backend: a web server allowing to listen for user interface requests (from frontend) regarding the deployment, the configuration and the scheduling of OpenBACH (i.e. Agents, Jobs, Jobs instances, scenarios and scenarios instances), as well as requests regarding OpenBACH information and status from Agents. These requests are then dispatched to a daemon called openbach-conductor for further processing and proper response.
- A daemon (openbach-conductor): it is in charge of taking the demands of the Backend and processing them. It can contact databases or Agents to do so and may optionally spawn threads to ensure fast response for long-running processes.

The Agent shall implement:

- A Control-Agent: It shall be able to configure and execute/schedule/stop different Job instances depending on the Controller commands (openbach-functions). It also shall be able to get status/information of the Agent itself and the available Jobs, as well as the Job instances status.
- A Collect-Agent: it shall allow to collect statistics/data and logs from the different running Job instances of the Agent and relay them to the Collector and locally store them.
- Job instances: One or several executions of a Job configured with a set of parameters. A job instance might be able to perform different tasks and/or to collect statistics to be sent. They might be started/stopped (e.g. start/stop a ping), activated/deactivated (e.g. iptables rules), etc. Different types of Jobs are differentiated within OpenBACH depending on the tasks it performs, such as administration tasks or telecom/network related tasks.

The Collector shall implement:

- A stats collector daemon: it shall centralize the data/statistics collection received from the Agents and store them into data storages.
- A logs collector: it shall centralize the log messages collection received from the Agents and store them into data storages.

3.4. Projects, Scenarios, OpenBACH-functions, OpenBACH actions and Jobs (and instances)

The comprehension of these main terms is one of the keys to well understand the OpenBACH design described herein and in particular, the way to configure the benchmark.

Besides the definition of each term (see Table 1 at the beginning of section Terminology), the purpose of this section is to explain the relationship between this terms.

As it has been previously explained, the Jobs are the groups of tasks (under the form of scripts) that are deployed in the Agents. An execution of this script configured with a set of parameters is known as a Job instance (an execution of the Job launched by the Agent in the same machine). The job instances might be scheduled by openbach-functions when they are implemented within a Scenario context, or regular actions when they are independent of any scenario. Later, we will focus on the different between these two types of functions.

The OpenBACH actions aim at performing many other tasks (other than scheduling job instance); such as the installation of Agents, Jobs, status requests, creation of projects, etc. Some of these actions are available as openbach-function to be used within a Scenario.

3.4.1. Within a scenario context

From the controller point of view (Figure 4), the Controller provide openbach-functions to run (they provide the structure to check for well formedness of their parameters); the Controller also owns different projects (identified by a name and operated by one or several users), different scenarios (identified by a name and associated to a project) and provide the overall structure to schedule openbach-functions within a Scenario. A project owns up to several scenarios. A scenario owns a group of unordered openbach-functions (identified by a function id). These openbach-function might be ordered through "wait_for" elements, which are able to add execution dependencies to the openbach-function (e.g. the openbach-function will be launched only when a specific openbach-function instance has been launched or when a specific job instance has finished).

The scenario instance is defined as a scenario with a date, an id, and a user who started it, and it is composed of a group of scheduled openbach-function instances. These instances can:

- be waiting on their "wait_for" condition; that either an or several other openbach-function complete; or that a Job instance / Scenario instance finishes its execution.
- be waiting for a fixed amount of time; possibly after the previous state.
- be completed as they have executed the action they were planned for; possibly after either one of the previous states.

The scheduler of the Controller is in charge of launching the scenario instance with all the openbach-function instances.

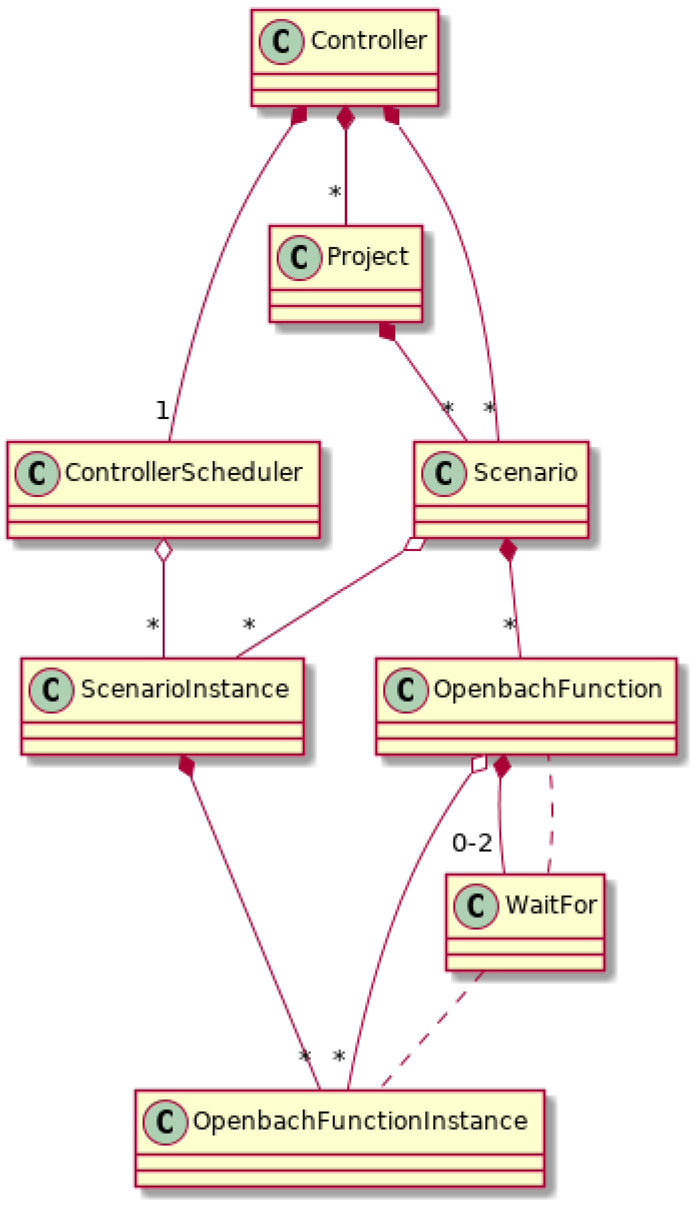


Figure 4. Relationship between scenario instances and openbach-functions in the Controller (and project)

From the Agent point of view (Figure 5), the controller also owns a scheduler, the Jobs that are installed in the Agent, defined by a job name and a description of the job. The scheduler is in charge of launching the Job instances, which are defined as the Jobs with a date of execution, an id and the arguments. Finally, each Job instance shall be associated to a scenario instance context represented by an id.

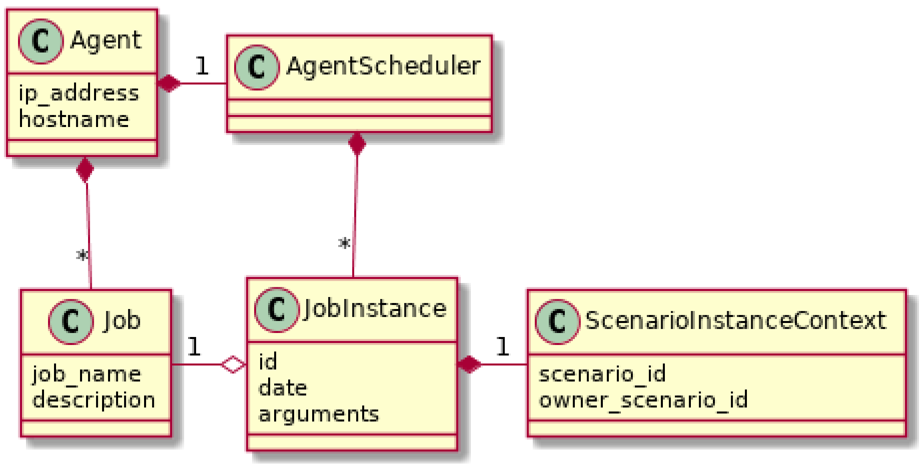


Figure 5. Organisation of Jobs and Job instances in the Agent

The steps that shall be performed to schedule and launch the job instance by means of the scenario and the openbach-functions are described below and represented in Figure 6.

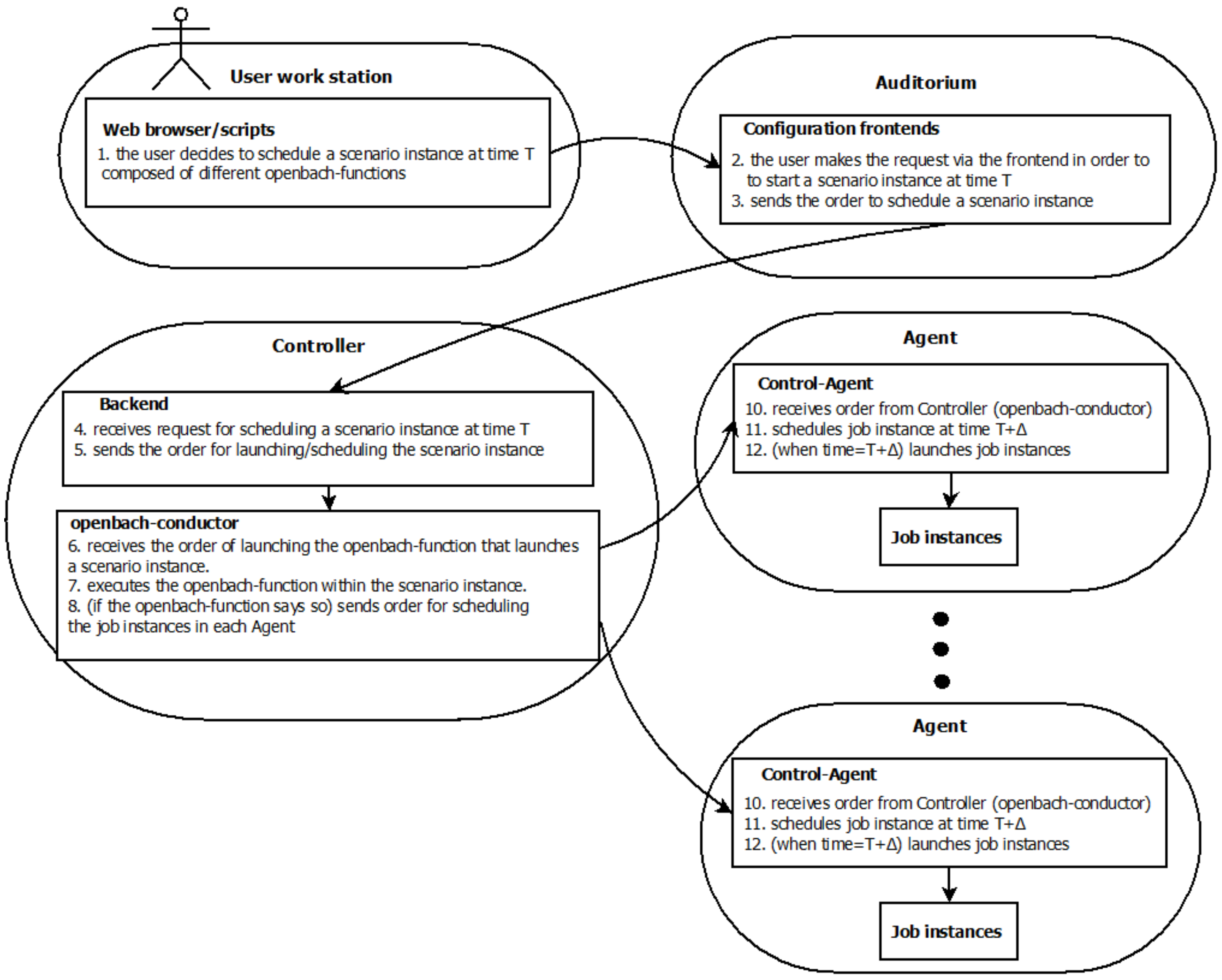

User work station
Web browser/scripts
1. the user decides to schedule a scenario instance at time T composed of different openbach-functions
Auditorium
Configuration frontends
2. the user makes the request via the frontend in order to to start a scenario instance at time T
3. sends the order to schedule a scenario instance
Controller
Backend
4. receives request for scheduling a scenario instance at time T
5. sends the order for launching/scheduling the scenario instance
openbach-conductor
6. receives the order of launching the openbach-function that launches a scenario instance.
7. executes the openbach-function within the scenario instance.
8. (if the openbach-function says so) sends order for scheduling the job instances in each Agent
Agent
Control-Agent
10. receives order from Controller (openbach-conductor)
11. schedules job instance at time T+Δ
12. (when time=T+Δ) launches job instances
Job instances
Agent
Control-Agent
10. receives order from Controller (openbach-conductor)
11. schedules job instance at time T+Δ
12. (when time=T+Δ) launches job instances
Job instances

Figure 6. Steps to launch a Job instance via the scenario and the openbach-functions concepts

1. (step 1) A user choses to launch a scenario instance from the web browser (web HMI) or the shell terminal (scripts). The request shall thus be sent via the configuration frontends (step 2). For that, the scenario shall be already created by the user and available in the backend data base.
2. (step 3-4) The configuration frontend shall send a request to the backend for launching the scenario instance (via HTTP).
3. (step 5-6) The backend shall transfer to the openbach-conductor the order of launching the openbach-function to start the scenario instance.
4. (step 7) When the scenario instance is launched, the openbach-function instances that are defined within the scenario shall be launched/scheduled by the openbach-conductor. Some of them might imply performing tasks in the Agent, and others in the backend (e.g. install an Agent). In fact, the openbach-functions shall specify when to launch the job instance in the Agent with respect to a "reference starting time" of the scenario instance plus an increment delta/offset (Δ).
5. (step 8-9) (if at least one of these functions specifies to perform a task in the Agent) The openbach-conductor shall send the order to the Agent via TCP sockets.
6. (step 10-11) The Agent shall schedule the job instance when it receives the order of scheduling

the job instance from the Controller (openbach-conductor).

7. (step 12) The launch of the Job instance is performed by the scheduler of the Control-Agent (when time = "reference starting time" + Δ, i.e. a "reference starting time" of the scenario instance plus an increment (Δ) parameter).

The "reference starting time" of a scenario instance is the time at which the scenario instance shall be launched in the backend.

3.4.2. openbach-functions and regular actions

In the Controller, the openbach-conductor implements actions that can be requested from the frontends. Each function has a defined set of input parameters, possible restrictions about the users able to call it and respond with JSON result that can be turned into an HTTP response. Most of the actions are rather specific and limited in scope and require the users to call them explicitly through the frontends. But some of them are generic enough to be exposed as openbach-functions available as Scenario actions. Thus two methodologies exist to execute an action:

- The action is available as the result of an HTTP REST call and return a proper HTTP response with a JSON body to the caller;
- The openbach-function is available as an action within a Scenario instance. Only the action "start_scenario" can trigger the execution of the openbach-functions contained within the launched Scenario. These openbach-functions are wrappers around regular actions and usually end up executing the same code.

Herein and example with the core function "start_job_instance".

1. If the user wants to start a job instance independently of any scenario. The action "start_job_instance" will perform the core function and return a proper response to the Backend (and from there to the frontend) with the "OK" status and the ID of the job instance, or a bad request "404".
2. On the other hand, within a scenario context, the openbach-function "start_job_instance" might need to associate the openbach-function and the scenario instance to the job instance, or it start a watch to check the status of the job instance and associate this watch to the current scenario, etc.

As openbach-functions are defined from regular actions, any action that makes sense as a task in a Scenario may be available as openbach-function. But there are two exceptions: the "while" and the "if" openbach-function are only available as openbach-functions and not as regular OpenBACH actions.

3.4.3. Functional definitions of a Scenario/Scenario instance

The scenario instance is managed by the openbach-conductor (in the Conctroller) and centralizes the status of all the job instances received from Agents and the status of the openbach-functions. The states of a scenario instance are described below (see Figure 7):

- **scheduling**: when a user decides to launch a scenario, the Controller starts validating locally the arguments provided to the openbach-functions and create resources to schedule them.

- **running**: a scenario instance is considered in this state when at least one of the openbach-functions, job instances, or scenario instances is still running. It keeps running until an error occurs, a user asked to stop the scenario or every scheduled instance comes to an end.
- **finished KO**: when one Agent or an openbach-function send an error status. If the error is considered not critical, the scenario might keep running. If it is considered critical, the scenario instance should stop all started job instances, scenario instances, and openbach-functions.
- **stopped**: in this state, the backend tries to stop the scenario instance (and thus all the job instances, scenario instances, and openbach-functions running/scheduled) because a user asked to stop the scenario instance.
- **stopped out of control**: when stopping the scenario on behalf of a user action, if a resource (openbach-function) is unreachable, the state of the scenario is set to "stopped out of control".
- **finished OK**: when the end time of the scenario instance is reached meaning the scenario instance is correctly finished.

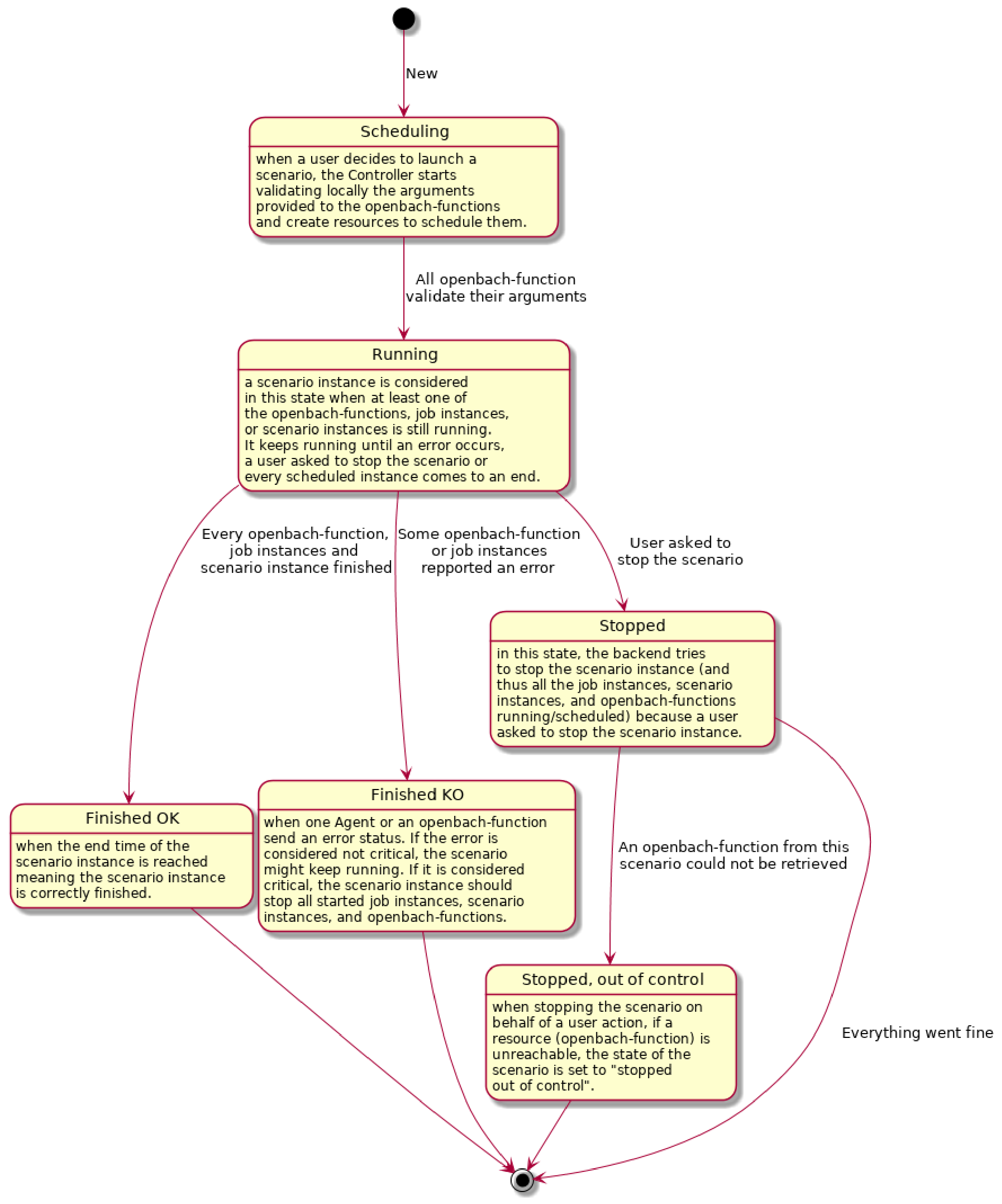

New
Scheduling
when a user decides to launch a scenario, the Controller starts validating locally the arguments provided to the openbach-functions and create resources to schedule them.
All openbach-function validate their arguments
Running
a scenario instance is considered in this state when at least one of the openbach-functions, job instances, or scenario instances is still running. It keeps running until an error occurs, a user asked to stop the scenario or every scheduled instance comes to an end.
Every openbach-function, job instances and scenario instance finished
Some openbach-function or job instances repported an error
User asked to stop the scenario
Stopped
in this state, the backend tries to stop the scenario instance (and thus all the job instances, scenario instances, and openbach-functions running/scheduled) because a user asked to stop the scenario instance.
Finished OK
when the end time of the scenario instance is reached meaning the scenario instance is correctly finished.
Finished KO
when one Agent or an openbach-function send an error status. If the error is considered not critical, the scenario might keep running. If it is considered critical, the scenario instance should stop all started job instances, scenario instances, and openbach-functions.
An openbach-function from this scenario could not be retrieved
Stopped, out of control
when stopping the scenario on behalf of a user action, if a resource (openbach-function) is unreachable, the state of the scenario is set to "stopped out of control".
Everything went fine

Figure 7. States diagram of scenario instance

3.4.4. Functional definitions of a Job/Job instance

The job instance is managed by the Agent. The states of a Job instances are described below (Figure 8):

- **not scheduled**: default state of a job on an Agent. When the Agent starts or the Job is just installed on the Agent.

- **scheduled**: when a start job instance has been correctly received by the Agent.
- **running**: when a job has been scheduled and its starting date has been reached. If the job is persistent it is "running" until a user stops it. Otherwise it can be running if the job is executing or its scheduling is recurent.
- **not running**: when a job has been scheduled and stopped executing by itself. A return code of 0 has been collected for this job.
- **error**: when a job has been scheduled and stopped executing by itself. A non-zero status code has been collected for this job.
- **stopped**: when the execution of the job has been canceled by user request.

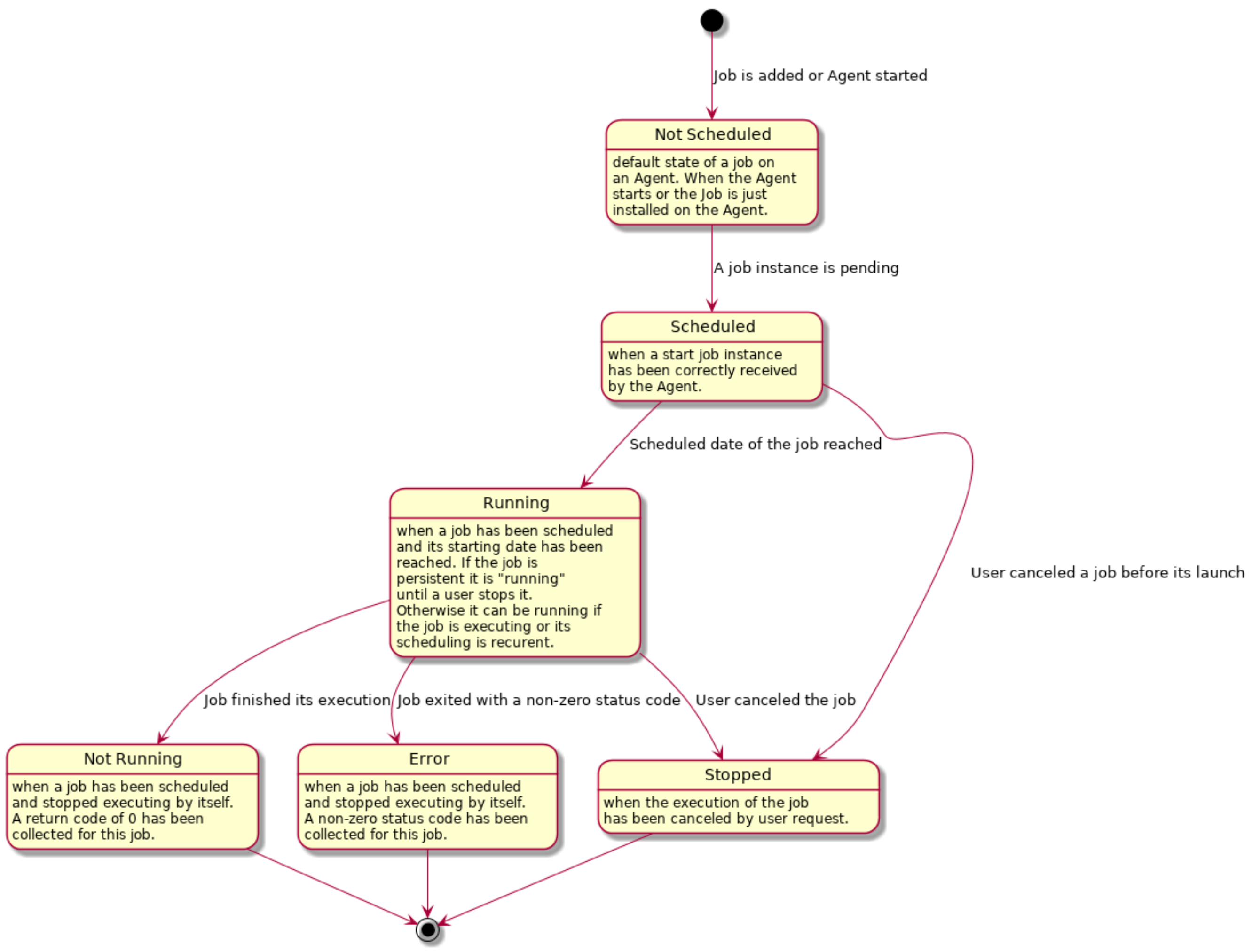

Job is added or Agent started
Not Scheduled
default state of a job on an Agent. When the Agent starts or the Job is just installed on the Agent.
A job instance is pending
Scheduled
when a start job instance has been correctly received by the Agent.
Scheduled date of the job reached
User canceled a job before its launch
Running
when a job has been scheduled and its starting date has been reached. If the job is persistent it is "running" until a user stops it. Otherwise it can be running if the job is executing or its scheduling is recurent.
Job finished its execution
Job exited with a non-zero status code
User canceled the job
Not Running
when a job has been scheduled and stopped executing by itself. A return code of 0 has been collected for this job.
Error
when a job has been scheduled and stopped executing by itself. A non-zero status code has been collected for this job.
Stopped
when the execution of the job has been canceled by user request.

Figure 8. States diagram of a job instance

3.4.4.1. Job types

The Jobs can be classified in different types depending on their purpose, domain or even the purpose they aim at performing.

First of all, there is a clear separation of jobs that are included in OpenBACH and jobs that are not. Included jobs are called "core_jobs" and are provided for convenience as they monitor tools that are first choice actors in metrology. OpenBACH guaranty their integration within the platform.

Other jobs can be found in an "openbach-extra" repository. This repository contains "stable" and "experimental" jobs. Stable jobs are maintained by the OpenBACH developpers and are guarantied to integrate well but are not included as "core" jobs as their usage may be more specific or less automatic. "experimental" jobs are jobs that were developed at some point in the OpenBACH ecosystem but no maintenance effort is provided; they can work as is, or not at all…

In each "core", "stable", and "experimental" kind of jobs, a classification shall be made to separate the job depending on the network layer it operates: admin, network, post-processing, service, transport, metrology.

3.5. Material aspects: Entities

The following section describes the deployment of OpenBACH in different entities. In particular, Figure 9 shows the architecture and the components of the proposed design. An example of network topology where OpenBACH could be deployed is available at the top-left corner of the figure. In such topology, the network entities are interconnected by means of heterogeneous physical links (satellite, terrestrial, LTE, WiFi, etc.).

The scheme also shows the components of OpenBACH, the functions (and the associated functional blocks), the entities (servers, work stations, etc.) where the components are deployed, and a management network (recommended but optional) allowing the interaction between these components.

3.5.1. Types of entities

Five types of entities (identified as grey boxes in the figure) are defined in the Figure 9 OpenBACH design: network entities, user entity, controller entity, collector entity and auditorium entity.

- A "network entity" is defined as any machine, server, or workstation, able of hosting a Linux OS (and possibly Windows OS in further evolutions of OpenBACH) and an OpenBACH Agent component. Some examples of roles performed by these "network entities" are: a user terminal, a server, a proxy, a gateway, a satellite terminal, a terrestrial base station.
- A "controller entity" is defined as any machine, server, or workstation, able of hosting a Linux OS where the Controller is deployed.
- A "collector entity" is defined as any machine, server, or workstation, able of hosting a Linux OS where the Collector is deployed.
- An "auditorium entity" is defined as any machine, server, or workstation, able of hosting a Linux OS where the different frontends of the Auditorium are deployed.
- Finally, the "user entity" is defined as any personal computer (or workstation) from which a user would be capable of supervising and interacting with OpenBACH. This entity requires at least a shell terminal access and a web browser (Firefox or Chrome) for accessing the OpenBACH interfaces.

For the sake of simplicity, the Collector, the Controller and the Auditorium might be deployed in the same entity.

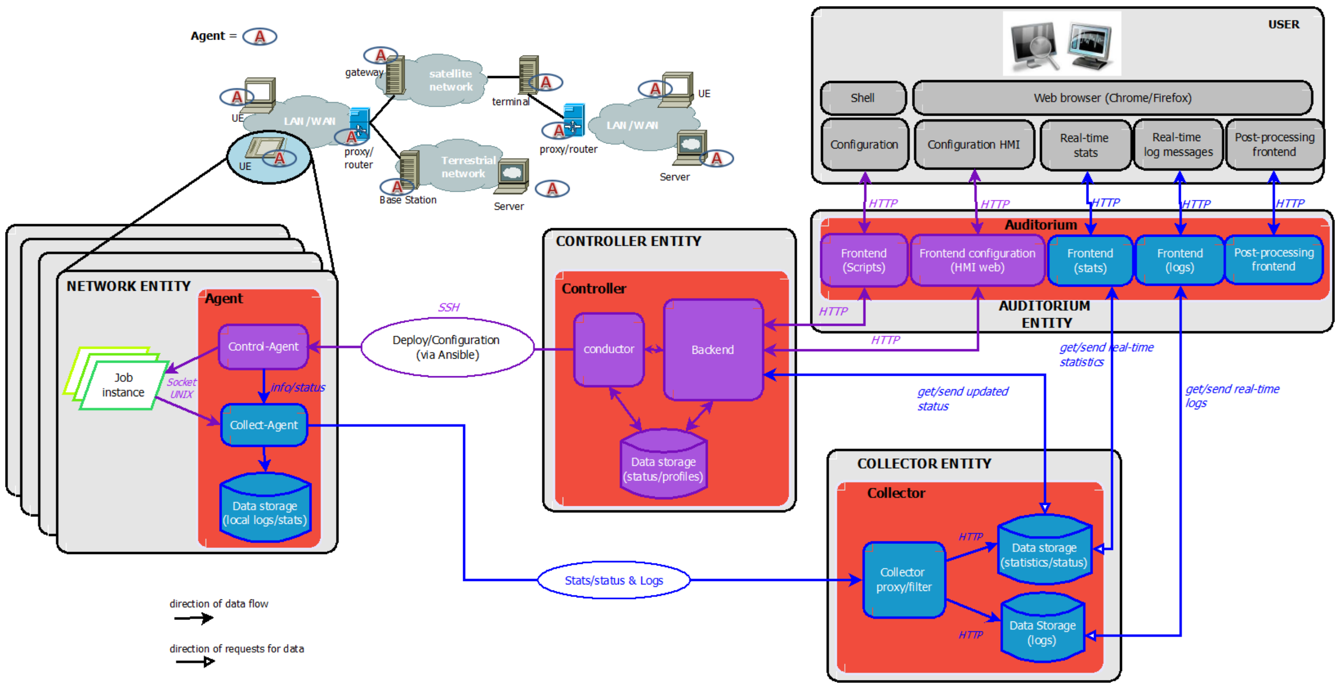


Figure 9. Architecture, components and interfaces of OpenBACH

3.5.2. Functional blocks per entity

Below, we list the functional blocks, types of storage and components for each considered entity that OpenBACH shall implement:

- A "Network entity" shall have:
  - An Agent :
    - A Control-Agent
  - A Collecting agent
    - Jobs (deployed) and Instances of Jobs (running/scheduled)
    - A path towards an available data storage: it shall allow to locally store data/logs. It is useful for offline scenarios where the network entity is not accessible during the tests (e.g.: when a management network is not available).
- The "Collector entity" shall have:
  - A Collector daemon for statistics and log messages.
  - A data base for storing logs.
  - A data base for storing statistics/data.
- A "Controller entity" shall have:
  - A backend (web server)
  - A daemon (openbach-conductor).
  - A data Storage managed by the backend for storing information related to the benchmark (available agents and entities information, information of jobs available, status of Jobs instances, scenarios, etc).

- An "Auditorium entity" shall have several frontends: one per type of display (configuration of benchmark, statistics display and logs display). In particular:
    - A frontend of configuration (web interface)
    - A dashboard frontend for real-time statistics dashboard (web interface)
    - A dashboard frontend for real-time log messages (web interface)
- A "User entity" shall dispose of:
    - A web browser (Chrome/Firefox) client to access the different available frontends, i.e.:
        - Configuration web interface
        - Real-time statistics
        - Logs/errors/status
        - Post-processing or offline statistics
    - Linux/Unix shell terminals for jobs/scenarios configuration (related to the Python script frontend).

Chapter 4. Detailed conception

4.1. Detailed conception of the Auditorium

4.1.1. Configuration frontends

Herein, we describe the design of the configuration frontends, and in particular the available supervision functions allowing to configure OpenBACH and the different jobs/scenarios. On the other hand, the design and requirements of the other OpenBACH frontend, i.e. those aiming at displaying the statistics/data and the log messages, are detailed in section [section_display] (after the description design of the Collector and the Agents). This order is preferable since it makes the comprehension of the chosen solution easier as well as the provided requirements of the frontends.

By means of the configuration frontends, the user shall be able to ask for different types of information regarding Agents and Jobs, in particular, the user shall be able to ask for:

- the list of Agents installed and their status (running/not running)
- the list of Jobs that might be installed in an Agent (i.e. available for installation in OpenBACH). This might help a user decide the jobs that can be installed.
- the list of jobs available in each Agent (not necessarily running, only available)
- the list of job instances per Job that are scheduled/started for each Agent.
- The scenarios available.
- The list of scenario instances scheduled/started and their status.

This information is used by the user to have an update knowledge of the benchmark, so that he would be able to correctly perform different tasks. The tasks that a user shall be able to carry out are:

- Install/uninstall Agents in the network entities. The procedure for installing new Agents is explained in section [install-agent] (TBD) and in the wiki OpenBACH (http://opensand.org/support/wiki/doku.php?id=openbach:manuals:index).
- Install/remove a job to/from an Agent
- Schedule/start/stop a job instance in an Agent with different configuration parameters.
- Create/delete/modify scenarios.
- Start/stop a scenario instance over different Agents.
- After the implementation of a new Job performed by a user, the user shall be able to make the Job available for installation.

The configuration frontend will thus serve as user interface, allowing the user to perform different tasks (as detailed above). These tasks will be performed by calling the "openbach-actions" from the frontend in order to send the request to the core of the Controller, also known as Backend, which will perform different actions according to the requested tasks. The benchmark shall implement two different configuration frontends, one for basic users, which will perform different tasks through the web interface, and a second frontend, based on python scripts, allowing for more flexibility and implemented for advanced users.

In order to maximize the evolutivity and the clarity of the backend implementation, both frontends shall be able to call/use the same functions implemented in the backend. For this reasons, we propose a backend based on web services.

The communication between the Backend and the configuration frontends shall be carried out via an HTTP Restful API.

All the responses of the backend shall be implemented in JSON format.

4.1.1.1. Web interface (Basic user)

In this section, we list some of the requirements that the frontend shall implement.

The web interface dedicated to configuration of the benchmark shall:

- Be able to authenticate users,
- Display the list of Projects owned by the current user.
- Display the list of Scenarios in a given project.
- Allow to configure/launch a Scenario.
- Display past and current Scenario instances.
- Allow to access statistics of Jobs of a given Scenario instance.
- Allow to access logs of Jobs of a given Scenario instance.
- Allow to list/edit the network Entities associated to a given project.
- Allow to list/install/uninstall Jobs in the Agent of a given Entity.
- Display the status of installed Agents for the admin users.
- Allow to list/add Jobs in the Controller for the admin users.
- Allow to manage user rights for the admin users.

4.1.1.2. Python scripts (Advanced users)

In this section, we list the requirements that the frontend shall implement.

The Python scripts dedicated to the configuration of the benchmark shall:

- Be able to authenticate users;
- Allow to manage (install/list/uninstall) Agents;
- Allow to manage (install/list/modify/uninstall) Collectors;
- Allow to manage (create/list/modify/delete) Projects;
- Allow to manage (add/list/delete) Jobs on the Controller;
- Allow to manage (install/list/uninstall) Jobs on the Agents;
- Allow to manage (create/list/modify/delete) Scenarios;

- Allow to manage (start/status/restart/stop) Job instances on the Agents;
- Allow to manage (start/status/stop) Scenario instances on the Controller;
- Allow to send files on Agents;
- Allow to terminate any Job instance or Scenario instance for admin users.

4.2. Detailed conception of Controller

The Controller is in charge of centralizing and deploying the configuration of OpenBACH, the Agents the Jobs and scenarios and commands the Agents to schedule the Jobs instances to be launched within a scenario instance.

As it can be observed in Figure 10 (and previously detailed, see section Functional blocks per component), the controller shall implement different functional blocks. It mainly consists of a backend for receiving the requests from the various frontends, data storage for saving informations related to OpenBACH (status, users, scenarios, projects, etc.) and a daemon (openbach-conductor) to interact with the Agents and the data.

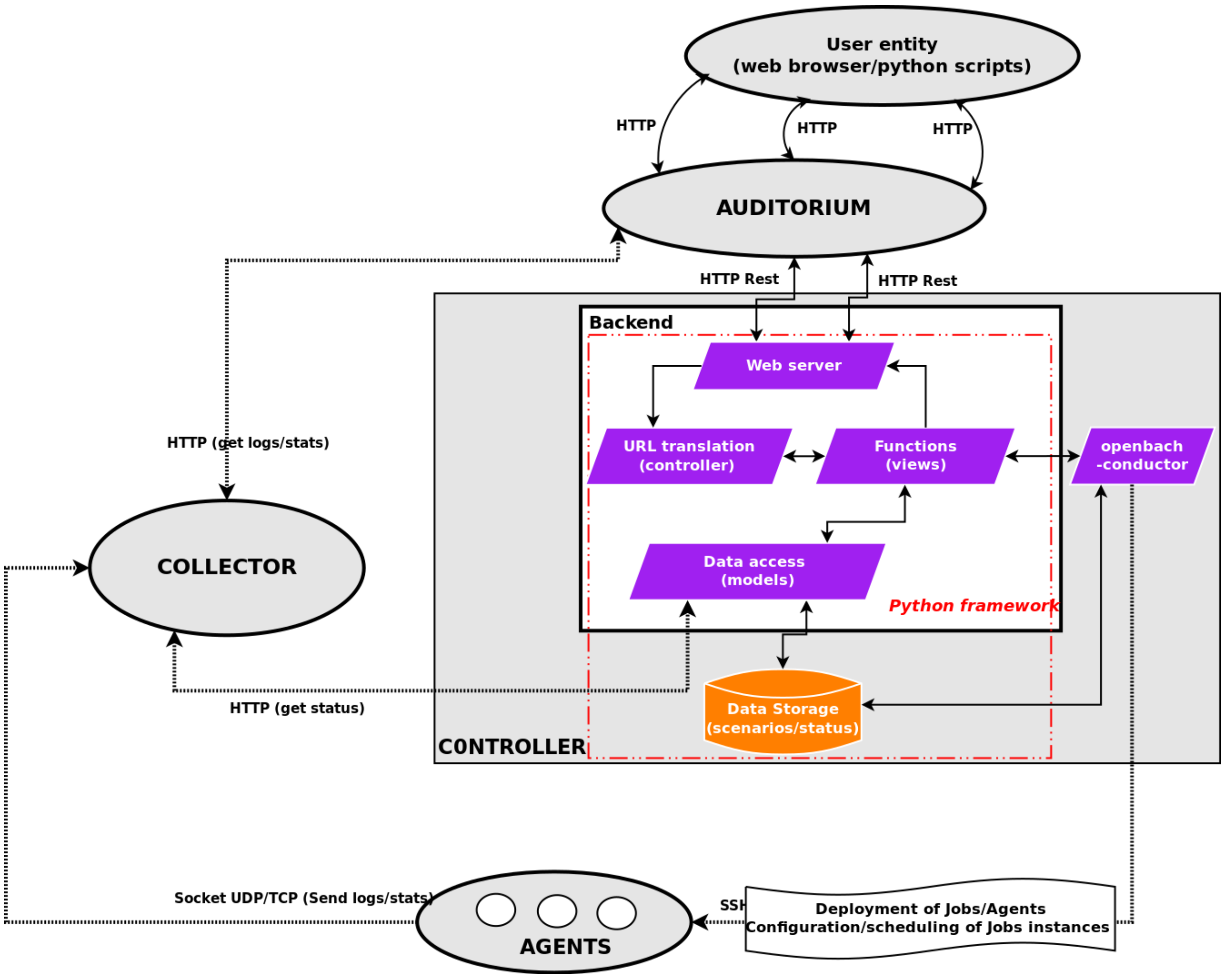


Figure 10. Controller design: Backend and interfaces

4.2.1. Backend

The backend design shall follow the Model-View-Controller (MVC) architectural pattern (as represented Functional blocks per component) since it allows a proper separation between the user-interface and the substance of the application.

In Functional blocks per component, we can observe that a webserver (e.g. Apache or Nginx) shall be set up in front of the MVC pattern in order to handle the user requests (from frontend) before passing those requests that require application logic.

The controller (of the MVC architecture) shall be in charge of receiving inputs and data from user and convert them to commands for the views. The model shall be in charge of managing and accessing the database and the view shall contain the ways to set, compute or manipulate information in order to send an output representation of required data.

In summary, the controller (of the MVC architecture) receives an action and data from the webserver (pushed by the user). It then sends the data to the correct view (i.e. function), depending on the request. The view works with the model to get the appropriate data under objects format and handles these objects in order to perform the required actions and create an output (response) to the user.

The views are the way to execute the openbach actions, which are implemented in the openbach-conductor. Through these functions, the backend views shall be able to:

- authenticate users;
- install and remove Agents and Jobs to/from the benchmark;
- install and remove Jobs on the Agents;
- list the available Agents and the available Jobs per Agent;
- create/modify/delete Projects and Scenarios;
- configure/launch/stop Scenario Instances;
- list the available Projects, Scenario and Scenario instances and their status;
- send commands to schedule/start/stop Jobs instances to the corresponding Agents;
- list the scheduled/started Job instances and their status.
- create/modify/delete Entities for a Project;
- associate/modify/remove an Agent to/from an Entity.

4.2.2. Ansible for communication Controller-Agent

The installation of an Agent or a Job requires the transmission of files (scripts, daemon files, configuration files, etc.), the installation of dependencies (python, apt-get, software, etc.) and other needs such as the installation of a NTP client for synchronizing the network entity. There are several off-the-shelf frameworks available in open-source allowing for application deployment and/or configuration management (Puppet, Chef, Ansible, …). The Ansible solution has been retained because it is a simple and flexible tool that gives the ability to automate common tasks, deploy applications and launch commands in different hosts from a centralized entity (in our case the OpenBACH Controller). In particular, Ansible implements the following features:

- Ansible is open source and written in Python, which harmonizes with the philosophy of OpenBACH of implementing the Agent and the Jobs in Python.
- A scripting system based on YAML syntax, which is easily readable and with a very low learning curve.
- Everything is done via files called "playbook" (YAML syntax). The tasks written in the playbook call the Ansible modules (similar to libraries) with different arguments (e.g. call the "apt-get" module with the option "build-dependencies" and the name of the package).
- Ansible is only installed in the Controller. The distant hosts do not need any software requirements/dependencies to be controlled, except for a SSH access (with the keys for authentication) and Python.
- When playbook is executed, Ansible connects to the various entities to deploy configuration and start tasks. Thanks to the modules, Ansible also ensures that any services that are supposed to work/run are correctly running, that a software is installed (e.g. apt-get install packages), that a task has been performed (i.e. idempotent concept) and that all configuration files are up to date. The last one is one of the strong points of Ansible.

4.2.3. Openbach-conductor

The Backend shall rely on a new functional item, a daemon identified as the openbach-conductor, allowing to defer the logic of executing the requested actions. Each time the backend receive a valid request from a Frontend, it forward this request and its associated data to the openbach-conductor. The conductor is then responsible to answer this request and perform the action; possibly forwarding it to an Agent.

It must be highlighted that though the openbach-conductor shall be able to process most of the actions itself, all actions related to Jobs on Agents (schedule a Job instance, request its status, …) cannot be executed by the openbach-conductor. Instead, the order shall be sent to the required Agent through a TCP socket and the success of the operation shall be specified by the Agent using the same socket connection.

The openbach-conductor shall use threads to schedule long-running actions, such as installing a new Agent, whenever applicable to ensure a fast HTTP response to the Frontend. A protocol to retrieve the status of threaded actions shall be implemented.

4.2.4. MVC

4.2.4.1. MVC: data access

The model shall handle one database that belongs to the backend, to save user information, agents status (running or not), a jobs list per Agent, job instances status, scenarios (and scenario instances) information and status, etc. Some of these information are potentially continuously modified (i.e. job instances status). For updating the status information, the Controller shall implement an action (see next section) that when requested polls the Agent.

Finally, the backend database shall implement different user profile types (see section XX).

4.2.4.2. MVC: views

The views of the Controller are the entry points for the Frontends and, even though they are handled in the backend, their real implementation of the is available in the openbach-conductor. These functions are summarized in Figure 11 and detailed below (the input JSON contents highlighted in bold are the required ones, the other ones are optional). They are classified in 8 main groups depending on the object/component they are bound to, i.e. the Agents, the Jobs, the Job instances, the Project, the Scenarios or the Scenario instances.

In the tables below, we have added a column in order to show if the actions also are available as openbach-function.

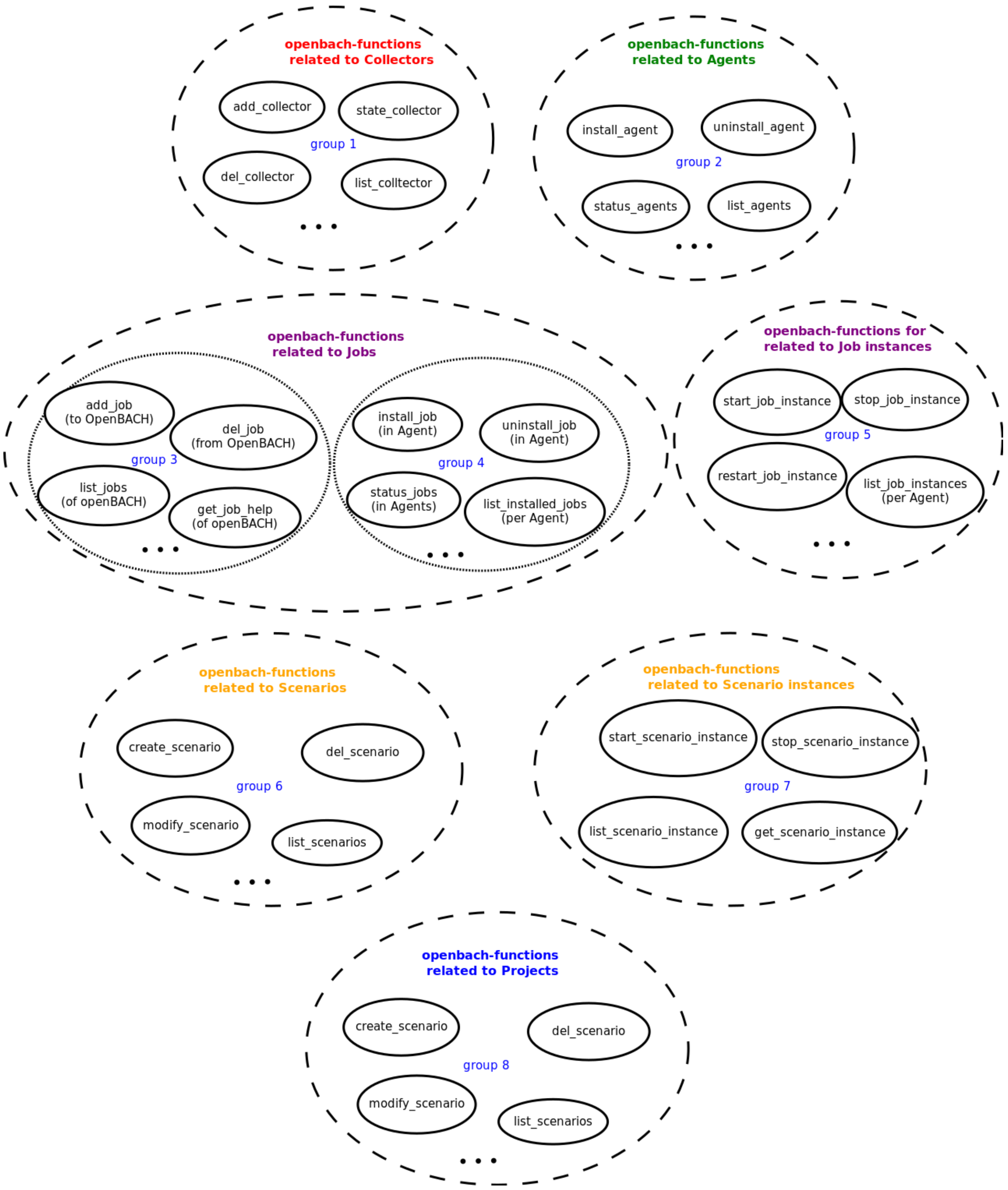


Figure 11. Openbach-actions classified by categories

First the group 1 of OpenBACH actions allowing to manage the Collectors:

Table 3. group 1

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
add_collector	POST	*/collector*	**address**, **name**, username, password, logs_port, logs_query_port, cluster_name, stats_port, stats_query_port, database_name, database_precision, broadcast_mode, broadcast_port	Add a new Collector (and install an Agent on it)	no
modify_collector	PUT	*/collector/***address**	logs_port, stats_port	Modify the Collector (and all the associated Agents)	no
del_collector	DELETE	*/collector/***address**		Remove a Collector	no
get_collector	GET	*/collector/***address**		Return the informations of this Collector	no
modify_collector	PUT	*/collector/***address**	logs_port, logs_query_port, cluster_name, stats_port, stats_query_port, database_name, database_precision, broadcast_mode, broadcast_port	Change the constants associated to this Collector	no
change_collector_address	POST	*/collector/***address**	**address**	Change the address of the machine the Collector was installed on.	no

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
list_collectors	GET	*/collector*		Return the list of all available Collectors	no
state_collector	GET	*/collector/***address***/state*		Return the status of the last commands on the Collector	no

Second the group 2 of OpenBACH actions allowing to manage the Agents:

Table 4. group 2

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
install_agent	POST	*/agent*	**name**, **address**, **collector_ip**, username, password	Install OpenBACH Agent in a network entity (identified by IP address) and add the Agent information to the Controller database.	no
uninstall_agent	DELETE	*/agent/***address**		Uninstall OpenBACH Agent from a network entity and delete the Agent information from the Controller database.	no
list_agents	GET	*/agent*	update	Return the list of Agents, if update is present, this function contact the Agents to retrieve the last information status.	no

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
infos_agent	GET	*/agent/***address**	update	Return the informations about an Agent. If update is pesent, this function contact the Agent to retrieve the last information status.	no
log_severity_agents	POST	*/agent/***address**	**action='log_severity'**, **severity**, local_severity	Change the default log severity of Jobs installed on the Agent.	no
assign_collector	POST	*/agent/***address**	**collector_ip**	Assign this Collector to the Agent	no
state_agent	GET	*/agent/***address***/state*		Return the status of the last commands on the Agent	no

Then group 3 of OpenBACH actions allowing to add/delete a Job to/from the list of available Jobs to install:

Table 5. group 3

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
add_job	POST	*/job*	**name**, path, file	Add a Job to the Jobs list. If no optional argument is provided, install the job from the stable jobs of the openbach-extra repository. If path is present, find the definition files of the job at this path on the Controller. If file is present, it must be a tarball containing the definition files of the job.	no
install_jobs	POST	*/job*	**action='install'**, **names**, **addresses**, severity, local_severity	Install the named Jobs on the specified Agents	no
uninstall_jobs	POST	*/job*	**action='uninstall'**, **names**, **addresses**	Uninstall the named Jobs from the specified Agents	no
add_new_job	POST	*/job*	**name**, **tar_file**	Add a Job to the Jobs list (with the sources in the tar file)	no
del_job	DELETE	*/job/***job_name**		Delete a Job from the Jobs list	no

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
list_jobs	GET	*/job*	(external), (address, update), (string_to_search, ratio)	Return the Jobs list. If external is present return the list of stable jobs in the openbach-extra repository. If address is present, list the Jobs installed on the specified Agent, possibly contacting the Agent to get the latest list if update is specified. If string_to_search is present, filter the jobs in the returned list to those containing the string to search in their description or keywords; possibly using the provided ratio as threshold word dectection.	no
get_job_infos	GET	*/job/***job_name**	type=json	Return the description of the Job.	no
get_job_stats	GET	*/job/***job_name**	**type=stats**	Return the statistics produced by a Job.	no
get_job_help	GET	*/job/***job_name**	**type=help**	Return the help of the Job	no
get_job_keywords	GET	*/job/***job_name**	**type=keywords**	Return the keywords of the Job	no

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
log_severity_job	POST	*/job/***job_name**	**action**='**log_severity**', **addresses**, **severity**, local_severity	Change the default log severity of this Job on the specified Agents.	no
stat_policy_job	POST	*/job/***job_name**	**action**='**stat_policy**', **addresses**, stat_name, storage, broadcast	Change the default statistics policy of this Job on the specified Agents.	no
state_job	GET	*/job/***name***/state*		Return the status of the last commands about the Job	no

Then the group 4 of OpenBACH actions allowing to handle files on Agents:

Table 6. group 4

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
push_file	POST	*/file*	**path**, **agent_ip**, file, local_path	Push a file on the Agent. Either file or local_path must be provided. If file is provided, its content is copied at path on the selected Agent. If local_path is provided, the content of the file present locally on the Controller at this path is copied at path on the selected Agent.	no

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
state_push_file	GET	*/file/state*	**filename**, **path**, **agent_ip**	Return the status of the push of a file on the Agent.	no

The group 5 of OpenBACH-actions allowing to manage a Job instance in a network entity (or Agent):

Table 7. group 5

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
start_job_instance	POST	*/job_instance*	**action='start'**, **agent_ip**, **job_name**, **instance_args**, date, interval	Start a Job instance of the name Job on the selected Agent.	yes
stop_job_instances	POST	*/job_instance*	**action='stop'**, **job_instance_ids**, date	Stop one or more job instances using their instance id.	yes
stop_job_instance	POST	*/job_instance/***id**	**action='stop'**, date	Stop the selected Job instance.	no
restart_job_instance	POST	*/job_instance/***id**	**action='restart'**, **instance_args**, date, interval	Stop the selected instance and restart it with the provided arguments. If instance_args is empty, the new Job instance will reuse the old arguments.	yes

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
list_job_instances	GET	*/job_instance*	**address** (but can be multiple), update	Return the list of the Job instances for the selected Agents. If update is present, the Agents are contacted prior to answering so the status of the Job instances are updated based on their answer.	no
status_job_instance	GET	*/job_instance/***id**	update	Return the information of a Job Instance. If update is present, the Agent hosting this Job instance is contacted prior to answering so the status of the Job instance is updated based on its answer.	no
state_job_instance	GET	*/job_instance/***id** */state*		Return the state of the commands on the Job_Instance	no
kill_all	POST	*/job_instance*	**action=kill**, date	Stop all the scenario instances and job instances.	no

The group 6 of OpenBACH actions allowing to manage a Scenario:

Table 8. group 6

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
create_scenario	POST	*/scenario or /project/***project_name***/scenario*	**scenario_json**	Create a Scenario, optionally associated to a Project. The scenario_json is not the name of a parameter: the body of the POST message should be the JSON content representing the Scenario.	no
del_scenario	DELETE	*/scenario/***name** *or /project/***project_name***/scenario/***name**		Delete a Scenario associated to the given Project.	no
modify_scenario	PUT	*/scenario/***name** *or /project/***project_name***/scenario/***name**	**scenario_json**	Replace the json of the scenario identifed by the given name. The scenario_json is not the name of a parameter: the body of the PUT message should be the JSON content representing the Scenario.	no
get_scenario	GET	*/scenario/***name** *or /project/***project_name***/scenario/***name**		Return the JSON of the scenario identified by the given name. Optionally filtered by Project.	no

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
list_scenarios	GET	*/scenario or /project/***project_name***/scenario*		List all available Scenarios for the selected Project. If no Project is used, list all Scenarios that are not associated to a Project.	no

The group 7 of OpenBACH actions allowing to manage Scenario instances:

Table 9. group 7

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
start_scenario_instance	POST	*/scenario/***scenario_name***/scenario_instance or /project/***project_name***/scenario/***scenario_name***/scenario_instance*	arguments, date	Start a Scenario instance associated to the given Project.	yes
stop_scenario_instance	POST	*/scenario_instance/***scenario_instance_id**	date	Stop a scenario instance.	yes
list_scenario_instances	GET	*/scenario_instance or /scenario/***scenario_name***/scenario_instance or /project/***project_name***/scenario/***scenario_name***/scenario_instance or /project/***project_name***/scenario_instance*		List all the scenario instances. Optionally filtered by scenario_name or project_name.	no

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
del_scenario_instance	DELETE	*/scenario_instance/***scenario_instance_id**		Remove the data associated to the Scenario instance from the Controller database.	no
get_scenario_instance	GET	*/scenario_instance/***scenario_instance_id**		Return the infos of the scenario instance	no
export_scenario_instance	GET	*/scenario_instance/***scenario_instance_id***/csv*		Export the data of a Scenario instance in a CSV file and return this file as an HTTP response.	no

The group 8 of OpenBACH actions allowing to manage projects:

Table 10. group 8

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
add_project	POST	*/project*	**project_json**	Add a new Project. The project_json is not the name of a parameter: the body of the POST message should be the JSON content representing the Project.	no

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
modify_project	PUT	*/project/***project_name**	**project_json**	Modify an existing Project. The project_json is not the name of a parameter: the body of the PUT message should be the JSON content representing the Project.	no
del_project	DELETE	*/project/***project_name**		Delete a Project.	no
get_project	GET	*/project/***project_name**		Retrieve the JSON of a selected Project.	no
list_projects	GET	*/project/*		Get all Projects	no
refresh_topology_project	POST	*/project/***project_name**	networks	Refresh a topology for a Project. If networks is present, update the informations associated to the network topology of the Project instead.	no
create_entity	POST	*/project/***project_name***/entity*	**entity_json**	Create a new Entity associated to the Project. The entity_json is not the name of a parameter: the body of the POST message should be the JSON content representing the Entity.	no

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
modify_entity	PUT	*/project/***project_name***/entity/***entity_name**	**entity_json**	Modify the selected Entity from the given Project. The entity_json is not the name of a parameter: the body of the PUT message should be the JSON content representing the Entity.	no
delete_entity	DELETE	*/project/***project_name***/entity/***entity_name**		Remove the named Entity from the Project	no
get_entity	GET	*/project/***project_name***/entity/***entity_name**		Retrieve the JSON of the selected Entity.	no
list_entity	GET	*/project/***project_name***/entity*		List all Entities of the Project	no

And finally, the group 9 of OpenBACH actions for miscelaneous actions:

Table 11. group 9

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
statistics_names	GET	*/statistic/***project_name**		List the names of statistics generated by Jobs instances of the selected Project.	no
statistics_names_and_suffixes	GET	*/statistic/***job_instance_id**		List the names of statistics and suffixes generated by the selected Job instance.	no
statistics_origin	GET	*/statistic/***job_instance_id**	**origin**	Retrieve the first timestamp associated to the selected Job instance.	no

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
statistics_values	GET	*/statistic/***job_instance_id**	**name**	Retrieve the values of the named statistic for the selected Job instance.	no
statistics_comparison	GET	*/statistic/***job_instance_id**	**name**, **comparative**	Retrieve mean and variance of the values of the named statistic for the selected Job instance.	no
statistics_histogram	GET	*/statistic/***job_instance_id**	**name**, **histogram**	Retrieve a CDF/PDF-ready representation of the values of the named statistic for the selected Job instance.	no
get_version	GET	*/version/*		Return the installed OpenBACH version	no
get_logs	GET	*/logs/*	level, delay	Return the list of orphaned logs from the Collectors	no
list_users	GET	*/login/users/*		Return the list of registered users	no
modify_users	PUT	*/login/users/*	permissions	Update the permissions of registered users	no
delete_users	DELETE	*/login/users/*	usernames	Remove the named users from the Controller database	no

Action	Method	URL	Input contents (JSON or Query sting)	Description	Openbach-function
create_user	POST	*/login*	**action='create'**, **login**, **password**, email, first_name, last_name	Create a new User into the Controller database	no
modify_user	PUT	*/login*	**login**, password, email, first_name, last_name	Modify profile of connected user	no
get_user	GET	*/login*		Return profile of connected user	no
authenticate_user	POST	*/login*	**login**, **password**	Authenticate a user using the provided credentials	no
deauthenticate_user	DELETE	*/login*		Deauthenticate connected user	no
get_user	GET	*/login*		Return profile of connected user	no
authenticate_user	POST	*/login*	**login**, **password**	Authenticate a user using the provided credentials	no
deauthenticate_user	DELETE	*/login*		Deauthenticate connected user	no

4.2.5. Scenario format (JSON)

The scenario backbone (in JSON) is described as follows:

```
{
  "name": "Ping",  ①
  "description": "First scenario (for test)", ②
  "arguments": { ③
     "duration": "duration of pings"
  },
  "constants": { ④
    "agentA": "172.20.42.167",
    "agentB": "172.20.42.90"
  },
  "openbach_functions": [ ⑤
    {
      "start_job_instance": { ⑥
        "agent_ip": "$agentA", ⑦
        "ping": { ⑧
          "destination_ip": "$agentB", ⑨
          "duration": [ ⑩
            "$duration"
          ]
        },
        "offset": 5 ⑪
      },
      "wait": { ⑫
        "time": 0, ⑬
        "launched_ids": [], ⑭
        "finished_ids": [] ⑮
      }
    },
    {
      "start_job_instance": { ⑯
        "agent_ip": "$agentB",
        "ping": {
          "destination_ip": "$agentA",
          "duration": [
             "$duration"
          ]
        },
        "offset": 0
      },
      "wait": {
        "time": 10,  ⑰
        "launched_ids": [],
        "finished_ids": [0] ⑱
      }
    }
  ]
}
```

① "name": the name of the scenario

② "description": a description of the scenario.

③ "arguments": a list of arguments. An argument owns a name and a description.

④ "constants": a list of constants. A constant owns a name and a value.

⑤ "openbach_functions": a list of openbach-functions.

⑥ the name of the openbach-function ("start_job_instance"). Each openbach-function has different elements (see later)

⑦ "agent_ip": an argument of the start_job_instance openbach-function: the agent where the job instance should be scheduled/launched

⑧ "ping": the name of the job to start

⑨ "destination_ip": an argument of the ping job

⑩ "duration": an argument of the ping job

⑪ "offset": the openbach-function will be launched a time "offset" after the beginning of the scenario.

⑫ "wait": a structure used by the conductor to wait for a specific action (see below) before launching the current openbach-function.

⑬ "time": the time that the conductor waits before launching the current openbach-function if the conditions below are fulfilled

⑭ "launched_indexes": the id of the openbach-function that should be already "Finished"" before launching the current openbach-function

⑮ "finished_indexes": the id of the job instance that should "Not running" anymore before launching the current openbach-function

The arguments and constants of the scenario can also be used by the openbach-functions by using an "$" followed by the name of the arguments/constant (as in <7> and <9>).

Thus a user could make the scenario arguments dynamic without modifying the scenario itself (only the arguments).

Example of dependency in the scenario: the second openbach-function <16>, will be launched 10 seconds <17> after the first openbach function with id "0" <18>.

4.2.6. Justification of Djando framework

Django is an open-source Python web development framework. First of all, it has been chosen since it is implemented in Python, which allows to harmonize with the philosophy of OpenBACH (the Agent and the Jobs are developed in Python). Among the available Python frameworks, Django is known for offering off-the-shelf functionalities (data access methods, optimized database structures, plugins for interfacing with different applications, profiles management, etc.) allowing to focus on the pure development and the core functionalities required for the backend of OpenBACH.

Django is defined by their creator as a framework that encourages rapid development and clean, pragmatic design. Built by experienced developers, it takes care of much of the hassle of Web development, so you can focus on writing your app without needing to reinvent the wheel.

4.3. Detailed conception of Collector

As it has been previously presented in the overall design of OpenBACH, the Collector component shall be in charge of centralizing the collection of two main groups of data: the statistics/data and the logs.

The Collector shall be able to receive and collect two types of stream messages: logs and stats/metrics. Each type of stream shall be stored in its own database. The way OpenBACH collects the two types of data has been properly distinguished within the chain of functional blocks of Figure 12.

Both collections shall have the same functional scheme: a pure collector represented by a daemon that listens for new messages sent by the Agents, and a proper data base with efficient search mechanisms an access features, where the daemon stores the statistics and logs.

The fact of differentiating between two different streams (and databases), one for logs and another one for stats, is necessary since the nature and the format of each one is very different. For example, logs need a database capable of efficiently indexing and filtering long messages depending on host/job/type/etc, while stats need high precision when time stamping and storing the data.

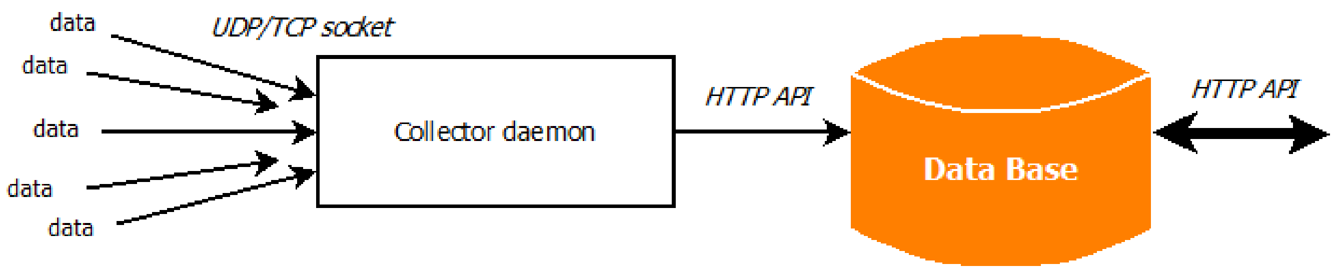


Figure 12. Generic functions of the Collector and interfaces

Regarding the interfaces of communications: the Collector daemon shall listen on a UDP/TCP socket, where all the Agents transmit their respective messages. The daemon shall store the data into a local data base via an HTTP API. Any external access to the data base (e.g. visualize the data in a web interfaces) shall be performed by means of this HTTP API.

The data received can be flagged. The flag can precise if the data should:

- be stored in the database
- and/or broadcasted to the Auditorium. The broadcast is done using a TCP or UDP socket (configurable) on the port 2223.

Table 12. Flag of stats

Stored in DB	Broadcasted	Flag Value
no	no	0

Stored in DB	Broadcasted	Flag Value
yes	no	1
no	yes	2
yes	yes	3

As detailed in the following two sections, off-the-shelf open-source software solutions have been chosen for fulfilling the needs of OpenBACH, and in order to have a robust collecting system at the disposal of OpenBACH. Moreover, this choice allows to focus more effort on the design and the development of an evolutive and robust configuration/control function (one of the critical points of this benchmark).

4.3.1. Logs collection details

Concerning the logs, the collector daemon function is performed by Logstash and the database role is carried out by Elasticsearch.

Logstash is an open-source data collection (under Apache 2 license), and a data transportation pipeline. It allows to efficiently process a growing list of logs, events and unstructured data sources for distribution into a variety of outputs, including the one used herein, an Elasticsearch data base. It is capable of normalizing different data formats by means filters.

Thus, once Logstash collects a log, it sends it to ElasticSearch, a database developed by the same creators of Logstash. The main features of Elasticsearch are:

- It has an indexing engine allowing fast search of data.
- Real-time analytics of the stored data
- It is API driven by a simple Restful API using JSON over HTTP. Log search is performed by this means.
- The requests/queries are returned in common text formats like JSON.
- It is available under Apache 2 open-source license.

Below, it is shown an example of the way logs can be exported from ElasticSearch via the HTTP API (check Elasticsearch manuals for more information). In the example, two filters are used for:

- exporting the logs within a 10 seconds time range, and
- returning only log-type-one logs lines

```
curl -XGET http://localhost:9200/playground/equipment/1?pretty
{
"_source": "message",
"filter": {"type": {"value":"log-type-one"}},
"query": {"range": {"@timestamp" : { "gte":"2015-02-20T12:02:00.632Z", "lt": "2015-02-
20T12:02:00.632Z||+10s"}}}
}
```

4.3.2. Statistics collection details

In the case of the statistics collection, we take profit of InfluxDB as a database, an open-source platform for data collection and storage. We use Logstash here too as the collecting daemon. Logstash is capable of listening on a UDP/TCP socket from the Agents messages (on the port 2222), and redirects the collected data to InfluxDB using an HTTP API. Otherwise, the Agent would have had to insert the data directly into the database (via HTTP), which would have made the Agents dependant on the type of database.

InfluxDB is capable of handling data time series with high precision (1ms if necessary) when the constraints of performance and availability are strong.

The external access to the InfluxDB data storage is also realized by means of this HTTP API.

Below, it is shown an example of writing and querying formats to be used when interacting with InfluxDB database via the HTTP API (check InfluxDB manuals for more information):

- Writing data: a POST shall be sent to the database (e.g. name mydb). The data consists of the measurement "cpu_load_short", the tag keys host and region with the tag values "server01" and "us-west", the field key value with a field value of "0.64", and the Unix Timestamp "1434055562000000000".

```
curl -i -XPOST 'http://localhost:8086/write?db=mydb' --data-binary
'cpu_load_short,host=server01,region=us-west value=0.64 1434055562000000000'
```

- Querying data: to perform a query, a GET request shall be sent. It shall set the URL parameter "db" as the target database, and set the URL parameter "q" as your query. The example allows to query the same data was written in the POST example.

```
curl -G 'http://localhost:8086/query?pretty=true' --data-urlencode "db=mydb" --data
-urlencode "q=SELECT value FROM cpu_load_short WHERE region='us-west'"
```

InfluxDB is released under the open-source MIT License.

4.4. Detailed conception of Agent

The Agent component shall implement two main parts according to the main functionalities of OpenBACH, a Control-Agent for configuring and controlling the Agent, and the Collect-Agent for everything related to statistics and logs collection. These two main parts are represented in Figure 13 as the two grey boxes.

As it has been previously explained, controlling the Control-Agent shall be done through TCP sockets from the Controller. The Control-Agent shall be in charge of scheduling, executing, checking and stopping the Jobs instances (green box) available in the network entity. As it has been previously defined, a job can be thought of as a number of individual tasks, i.e. start a traffic generator, start collecting a new data/statistics, start a service, etc.

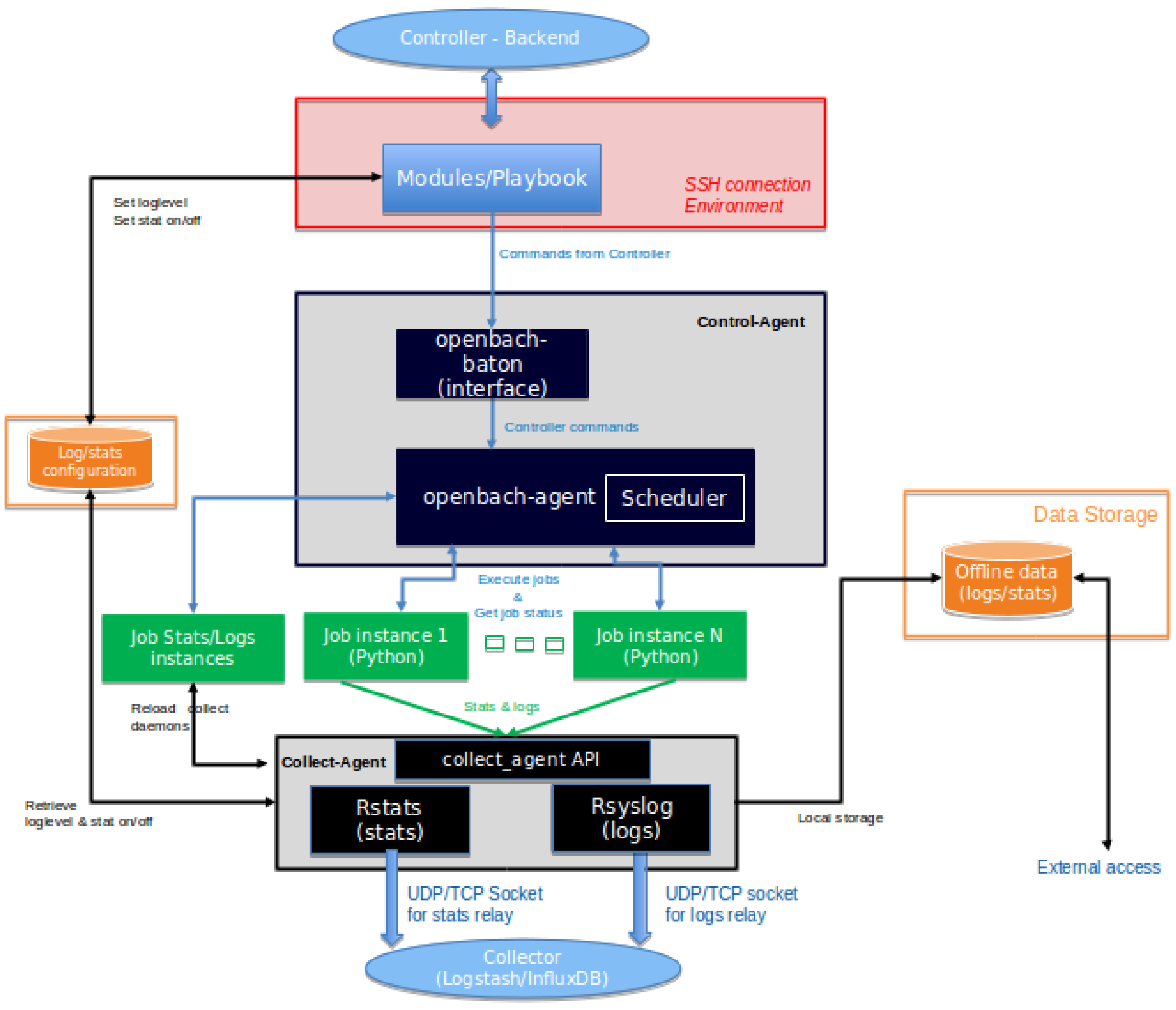


Figure 13. Detailed design of OpenBACH Agent including its interfaces

4.4.1. The Control-Agent part

The Control-Agent shall implement:

- A daemon for centralizing the tasks/jobs control ("openbach-agent" in Figure 13),
- a scheduler (integrated in the daemon "openbach-agent" and based on the Python library "apscheduler") for launching/scheduling the tasks of the daemon.
- a socket server listening on port 1112 for TCP connection so the Agent can listen for and respond to the orders of the Controller.

Below, the main features of the Agent are described:

- The Agent shall be based on a request-to-do policy, i.e. it shall perform tasks only when the Controller asks for.

- Within the command, the Agent may receive start/stop date-time information from the Controller, so that it will know when to execute the "agent-function" associated to.
- Depending on the command type, other options can be used as described below.
- The Agent shall manage the scheduler locally, so that it will be able to control the whole execution/status of the agent-actions.
- The Agent scheduler shall be able to execute the agent-actions with one millisecond accuracy.

4.4.1.1. Agent actions

The agent-actions are a group of actions implemented in the "openbach-agent" that allow performing different tasks regarding the OpenBACH control. These actions are directly related to the "openbach-functions/actions", since as it has been explained, in some cases, these functions need to perform actions/tasks in the Agents side (groups 3 and 4 shown in Figure 11), and the agent-actions are their way to do it.

Table 13 shows the mapping between the openbach-function/action of the Controller and the agent actions implemented in the Agent:

Table 13. Mapping between "openbach-functions/actions" (implemented in the Controller) and the agent-actions (implemented in the openbach-agent)

Openbach-function / action (in Controller) and group	agent-action/s (in Agent)	Objective of agent-action
install_job / 3	AddJobAgent	To inform the agent about a new installed job.
uninstall_job / 3	DelJobAgent	To inform the agent about an uninstalled job.
status_job_instance / 3	StatusJobsInstanceAgent	To send the status of installed jobs to the Collector.
start_job_instance / 5	StartJobInstanceAgent **(calls schedule_job_instance(myfunc="launch_job"))**	To start a job instance in the Agent
start_job_instance / 5	StartJobInstanceAgentId **(calls schedule_job_instance(myfunc="launch_job"))**	To start a job instance in the Agent without providing its instance ID
stop_job_instance / 5	StopJobInstanceAgent **(calls schedule_job_instance_stop(myfunc="stop_job"))**	To stop a job instance in the Agent
restart_job_instance / 5	RestartJobInstanceAgent	To restart a job instance in the Agent
list_installed_jobs / 3	StatusJobsAgent	To retrieve the list
	RestartAgent	To ask the Agent to stop its scheduled jobs and restart anew

Openbach-function / action (in Controller) and group	agent-action/s (in Agent)	Objective of agent-action
	CheckConnection	To verify the availability of a connection between the Agent and the Controller.

Next, it is detailed the different commands that the Control-Agent shall accept from the Controller component:

```
'AddJobAgent': job_name
'DelJobAgent': job_name
'StatusJobsInstanceAgent': job_name, job_id
'StartJobInstanceAgent': job_name, job_id, scenario_id, scenario_owner_id, date or
interval, start value
                Optional arguments may follow (arguments of the Job)
'RestartJobInstanceAgent': job_name, job_id, scenario_id, scenario_owner_id, date or
interval, start value
                Optional arguments may follow (arguments of the Job)
'StopJobInstanceAgent': job_name, job_id, date, date value
'StatusJobsAgent': no arguments
'RestartAgent': no arguments
'CheckConnection': no arguments
```

A configuration file for each job shall be implemented. This configuration file shall be used for verification purposes (e.g. check arguments/parameters/options accepted by the job) and making a job persistent (once it has been installed). The configuration file format shall include 4 sections (general information, the os requirements, the accepted arguments and the to be produced statistics):

```
---
general:
  name:            fping
  description: >
      This Job executes the fping ...
  job_version:     0.1
  keywords:        [ping, fping, rate, rtt, round, trip, time]
  persistent:      true ①

os:
  linux:
    requirements:  'Ubuntu 14.04/16.04'
    command:       '/opt/openbach-jobs/fping/fping.py'  ②
    command_stop:

  windows:
    requirements:  'Windows 2010'
    command:       '...'
    command_stop:

arguments:  ③
  required:
    - name:        destination_ip
      type:        'ip'
      count:       1
      description: >
          The destination ip of the fping
  optional:
    - name:        count
      type:        'int'
      count:       1
      flag:        '-c'
      description: >
          Stop after sending count ECHO_REQUEST packets. Default is 3.
    - name:        interval
      type:        'int'
      count:       1
      flag:        '-i'
      description: >
          Wait interval seconds between sending each packet.

statistics:  ④
    - name:        rtt
      description: >
          The Round trip time of ICMP packets.
      frequency:   'every *count x interval* sent packets or every *duration* time'
```

① The persistent variable should be a Boolean. It indicates if the job shall run on background or if it will only execute some tasks and finish.

② Command to be executed by the "openbach-agent" daemon on the agent when starting the job instance. (i.e. the path to the job script)

③ Accepted "required" and "optional" arguments

④ Produced statistics

When the Agent crashes or if it is restarted, the job configuration files help the Agent to know its own jobs before crashing/restarting.

Finally, it should be highlighted that the way the Agent has been designed would allow a user to control each Agent without a Controller, in other words, the current design would allow to bypass the Controller component if an advanced user needs to do so (see the debug section in OpenBACH wiki for more information).

4.4.2. The Collect-Agent part

The Collect-Agent shall implement two different client for collecting statistics and logs. The collection and forward of logs shall be performed by Rsyslog (open-source tool) and the collection and forward of stats/metrics shall be performed by the rstats client.

Rstats is a home-made program that collects stats and sends them to the Collector. Its principle is similar to the one of statsd (a simple daemon for stats aggregation) but modified in order to fulfill the OpenBACH requirements (in terms of accuracy, performance, etc.)

Two jobs (admin_jobs) shall be dedicated to control the collecting daemons (as shown in the figure): the Job "rsyslog" and Job "rstats" which shall allow to start/stop/restart/reload the rsyslog and rstats daemons, as shown later.

Regarding the logs, data and statistics to be collected, the Job instances shall be in charge of sending the logs/stats to the two daemons of the Collect-Agent (i.e. Rstats and Rsyslog). For that, the "collect agent API" shall be imported in the jobs script to be able use different methods (register_collect, send_log, send_stat, reload_stat, remove_stat, …) allowing to send the stats from the Job instance to the Collect-Agent daemons, which will forward the stats/logs to the Collector component via UDP/TCP sockets.

The collect-agent API allows to transparently manage logs and stats (independently of the clients rsyslog/rstats)

4.4.3. Rsyslog

Rsyslog shall be used in the Agents to handle the logs of the different running Jobs. It shall then forward the log messages to the Collector via a UDP/TCP socket. The configuration parameters to be used for rsyslog per Job shall be:

- Collector IP Address
- Logstash port: 10514 (default port)
- Local log severity level (to locally store in the network entity)

- Remote log severity level (to send to the collector)
- Job Name
- Scenario ID and job instance ID

Thus the Controller (after a user request) can specify the severity level that the Agents will use for both sending the logs to the Collector and locally store them in the network entity. The way these parameters are modified is explained at the end of this section.

The log messages (string format), shall be handled by a Python "Rsyslog API". The number and types of severity levels are chosen among those ones defined for Syslog standard messages, it is proposed to use the following ones:

Table 14. OpenBACH Log level

Value	Severity	Keyword
1	Error	syslog.LOG_ERR
2	Warning	syslog.LOG_WARNING
3	Informational	syslog.LOG_INFO
4	Debug	syslog.LOG_DEBUG

4.4.4. Rstats

Rstats has the same role as Rsyslog but focused on statistics collection and relay. Rstats shall fulfill the following requirements:

- Aggregate the statistics/metrics sent from the available jobs.
- Time stamp each collected statistics with one millisecond accuracy.
- Relay the statistics to the Collector, and allow to activate/deactivate this option for each statistic.
- Add a flag to the data, so the collector knows if it has to store and/or broadcast the received data
- Locally store all statistics.

The flag can be :

- 0 for no storage and no broadcast
- 1 for storage and no broadcast
- 2 for no storage and broadcast
- 3 for storage and broadcast

If the flag is 0, Rstats only stores locally the statistics and does not send the data to the collector.

The deactivation/activation of a statistic shall be realized by means of the following configuration file (one configuration file per statistic):

```
[default]
storage=true
broadcast=false
```

For example, in this configuration file, the statistics are send to the collector with a flag 1. The collector only stores the statistics in InfluxDB.

Rstats communicates with the Collector on an TCP or UDP socket on the port 2222.

4.4.5. Collect-agent API / How to use

Herein, we show an example on how to use the collect-agent API in a Job script:

```
import collect_agent    ①
conffile = "/opt/openbach-jobs/job_name/job_name_rstats_filter.conf"
success = collect_agent.register_collect(conffile)    ②

collect_agent.send_log(syslog.LOG_ERR, "ERROR: %s" % exception)  ③

statistics = {'rtt': rtt_data}
collect_agent.send_stat(timestamp, **statistics) ④
```

① import the API

② register the job instance to collect_agent

③ send a log

④ send a stat of type "rtt" and value "rtt_data" with a timestamp

4.4.5.1. Log severity level and activation/deactivation of stats

This section aims at detailing the way the OpenBACH Agent modifies the loglevel severity and activates/deactivates the stats:

- Step 1: After a user request asking for a new modification, the controller sends (using an Ansible playbook and a SSH connection) a new configuration file (for the aimed Job/Jobs)
- and the command allowing to reload the job "Rstats" (for stat activation/deactivation) or restart the job ‘Rsyslog" (for a log severity level modification).
- Step 2: The file is stored in the directory used by Rsyslog and/or Rstats clients.
- Step 3: Rsyslog and Rstats clients are restarted/reloaded in order to take the new configuration for logs and statistics.

4.5. Interfaces

The interfaces between all the components, the databases, the GUI, and the different blocks (representing different functionalities) is one of the keys to design reliable and robust communications protocols/APIs between all of them.

As it can be seen in Figure 14, where the main interfaces are displayed and listed, HTTP shall be used for communication between most of the elements, mainly in the case of user-to-frontend interfaces, or for frontend-to-backend interfaces (e.g. for web services), as it is a mature technology and it is very well considered among the community. Even the access to the different databases (InfluxDB and ElasticSearch) shall be carried out by means of HTTP API, which allows easy data portability, and fast query/request of data, etc.

Sockets (well known by his efficiency and simplicity of implementation) shall be mainly used for log/statistics transmission between Agent and Collector daemons, and between Jobs instances and Agents. For example, the Agent obtains the logs/statistics from jobs instances by means of local UNIX sockets, and transmits them to the collector by means of UDP/TCP sockets.

For local communication, where no data is transmitted, the elements shall communicate with simple bash/script commands (i.e. for execute/launch a task/process). That is the case of, for example, the Agent (controlling part)-to-job interface, or the Agent (controlling part)-to-(collecting part) interface.

Finally, the communication between the Controller (mainly the openbach-conductor) and the Agents shall be performed via SSH communications (using Ansible), or TCP sockets.

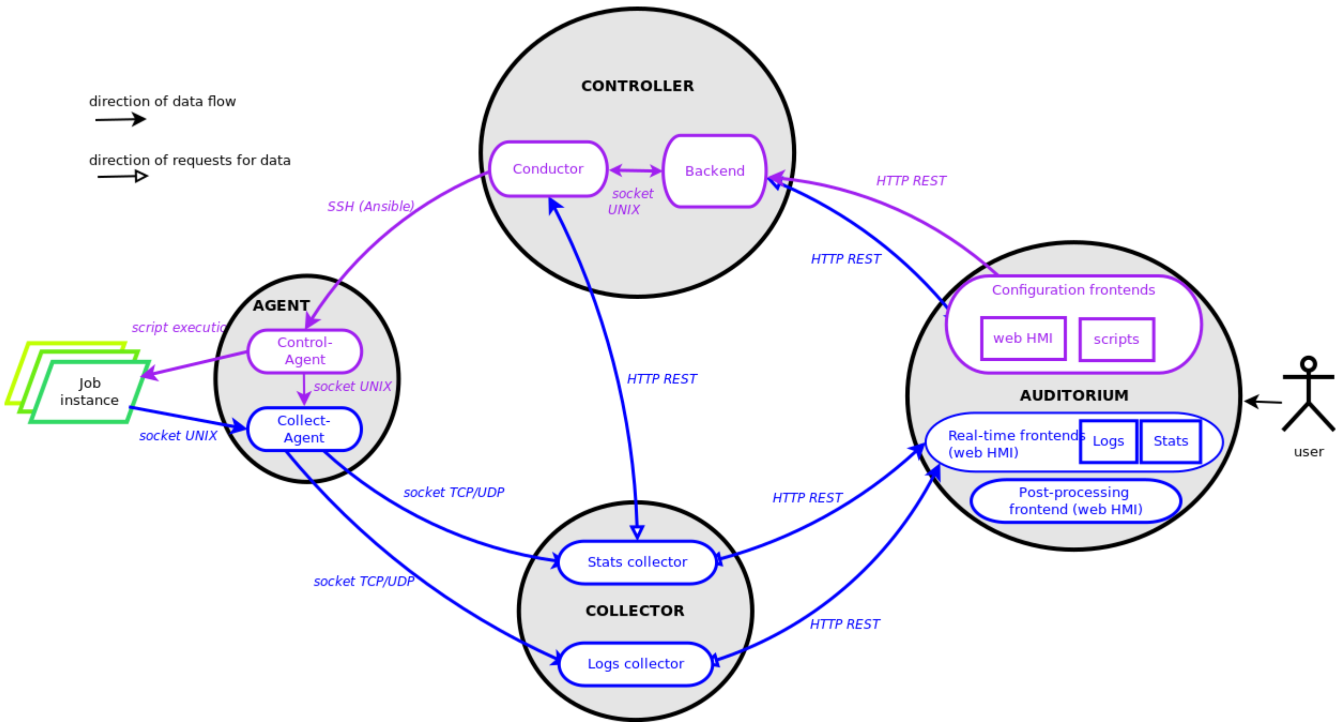


Figure 14. Basic overall design of OpenBACH components

4.5.1. Detailed Controller-Collector-Auditorium interfaces

An overall architecture of the auditorium, the controller and the collector and their interfaces is shown in Figure 15, where we can observe that the main streams of information between these components are those related to the writing/querying of data to/from the Collector databases (both logs DB and stats DB).

Indeed, once the Collector stores the logs and statistics in their databases, the Controller and the

Auditorium shall be capable of pulling this data for visualization and post-processing. Therefore:

- The frontends for displaying the logs and statistics shall use an HTTP API provided by the stats/logs databases for getting the data to be displayed.
- The Controller backend shall be able to query information stored in the database regarding job instances statistics or orphaned logs by means of a proposed HTTP API.

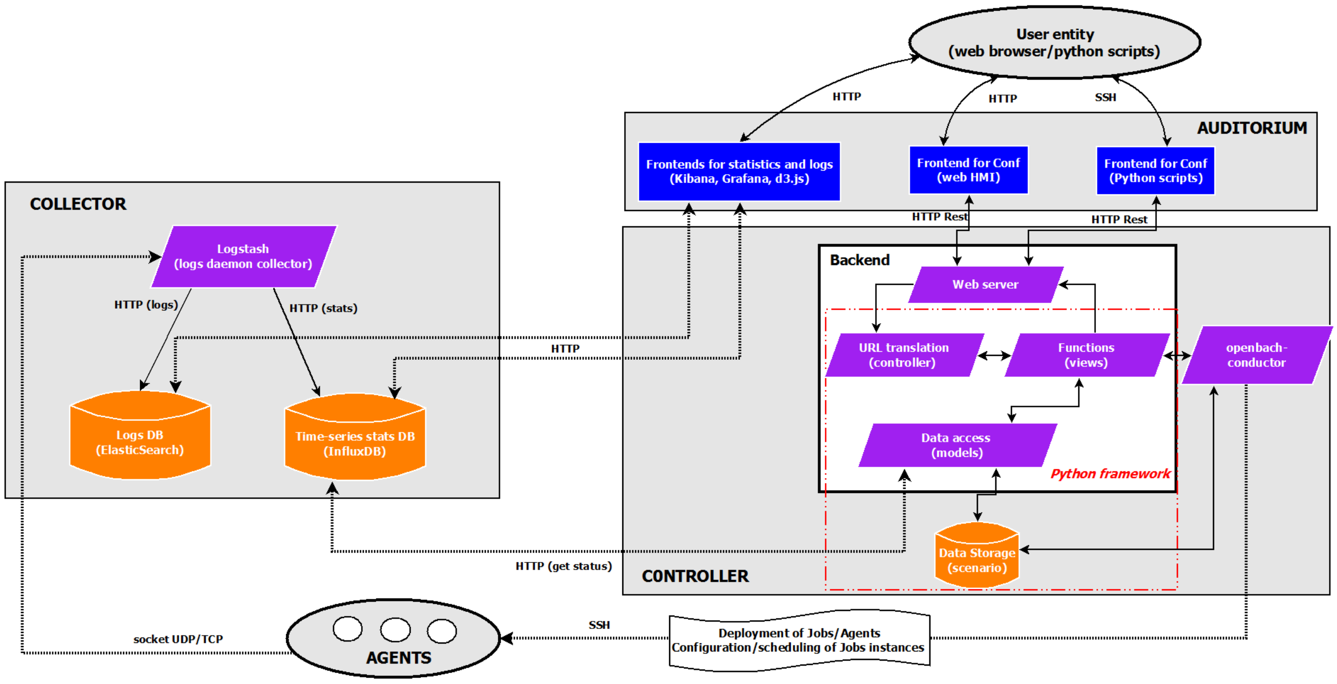


Figure 15. Auditorium, Controller and collector interfaces design

4.6. Post-processing

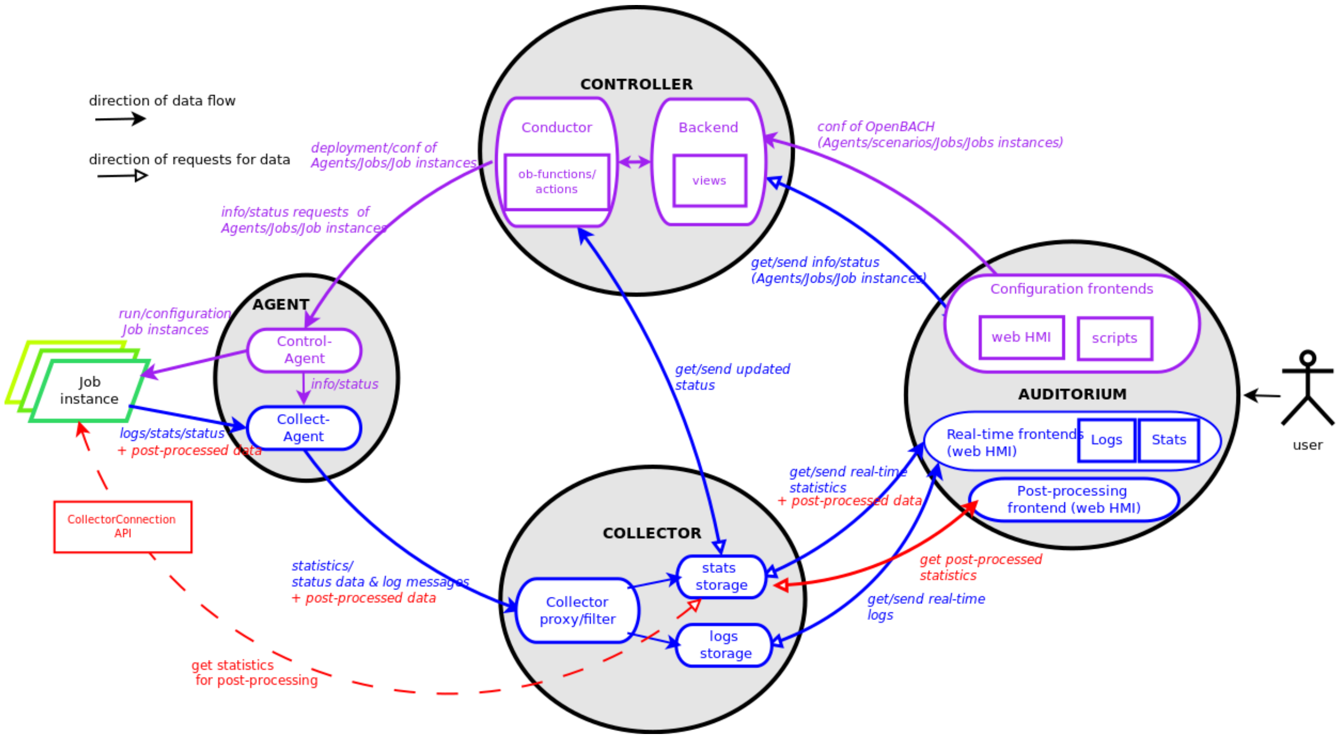
4.6.1. Import/export

As it has been detailed in the OpenBACH design, the Jobs are the way to execute the post-processing tasks allowing to perform dedicated calculations of the collected statistics.

A variant of the functional scheme of OpenBACH that is used for performing operations on the collected data via the post-processing jobs is shown in Figure 16 (highlighted in red), where:

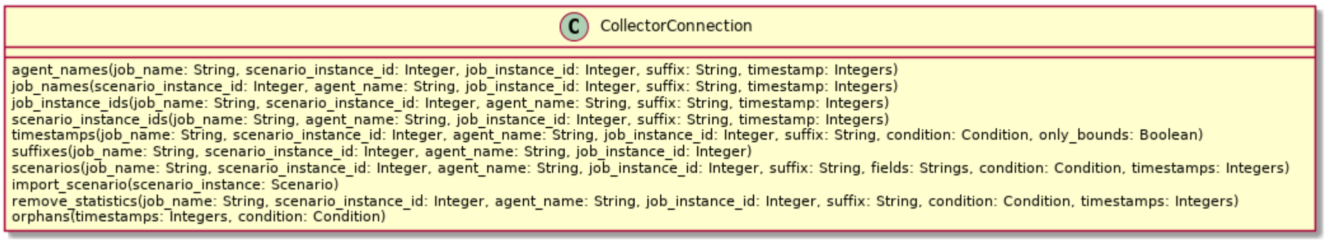
- After a user choses to launch a post-processing job (the same way any other Job is launched)
- The Job instance shall pull the required data from the statistics/logs database (InfluxDB and/or ElasticSearch) of the Collector (via the HTTP API). Then it shall perform the calculations and push the new data the same way a Job instance sends data to the Collector (i.e. via collect-agent: rstats). In that case, the Job (script) shall contain a module to access the database.

The module "CollectorConnection" (`from data_access import CollectorConnection`) has been implemented in order to be capable of exporting data from InfluxDB and ElasticSearch. It contains different functions allowing to access and export data from the Collector. See Figure 17 for a detailed view of the module functions.



direction of data flow
direction of requests for data
CONTROLLER
Conductor
ob-functions/ actions
Backend
views
deployment/conf of Agents/Jobs/Job instances
info/status requests of Agents/Jobs/Job instances
conf of OpenBACH (Agents/scenarios/Jobs/Jobs instances)
get/send info/status (Agents/Jobs/Job instances)
AGENT
Control-Agent
info/status
Collect-Agent
run/configuration Job instances
Job instance
logs/stats/status + post-processed data
CollectorConnection API
get/send updated status
Configuration frontends
web HMI
scripts
AUDITORIUM
Real-time frontends (web HMI)
Logs
Stats
Post-processing frontend (web HMI)
user
get/send real-time statistics + post-processed data
get post-processed statistics
get/send real-time logs
COLLECTOR
stats storage
Collector proxy/filter
logs storage
statistics/ status data & log messages + post-processed data
get statistics for post-processing

Figure 16. Post-processing pull (import)/push(export)



CollectorConnection

agent_names(job_name: String, scenario_instance_id: Integer, job_instance_id: Integer, suffix: String, timestamp: Integers)
job_names(scenario_instance_id: Integer, agent_name: String, job_instance_id: Integer, suffix: String, timestamp: Integers)
job_instance_ids(job_name: String, scenario_instance_id: Integer, agent_name: String, suffix: String, timestamp: Integers)
scenario_instance_ids(job_name: String, agent_name: String, job_instance_id: Integer, suffix: String, timestamp: Integers)
timestamps(job_name: String, scenario_instance_id: Integer, agent_name: String, job_instance_id: Integer, suffix: String, condition: Condition, only_bounds: Boolean)
suffixes(job_name: String, scenario_instance_id: Integer, agent_name: String, job_instance_id: Integer)
scenarios(job_name: String, scenario_instance_id: Integer, agent_name: String, job_instance_id: Integer, suffix: String, fields: Strings, condition: Condition, timestamps: Integers)
import_scenario(scenario_instance: Scenario)
remove_statistics(job_name: String, scenario_instance_id: Integer, agent_name: String, job_instance_id: Integer, suffix: String, condition: Condition, timestamps: Integers)
orphans(timestamps: Integers, condition: Condition)

Figure 17. Class Diagram of the data access (export of data) for logs and stats

Two constraints must be taken into account for correctly pushing the post-processed data into the Collector:

- The post-processed data and the original data shall not have the same name, in order to avoid deleting the original data.
- The post-processed data shall be time stamped: sometimes with the same timestamp of the original data, but it could also be time stamped with a different time (depending on the user needs).

the developer of the post-processing job must take into account these constraints.

4.6.2. Post-processing jobs

The benchmark shall include post-processing jobs allowing to compute the variance, the CDF, the interval of confidence and the average values over a time window.

These jobs shall allow to extract data from the InfluxDB database, compute the required post-processed values and export them into InfluxDB.

4.6.3. Post-processing on the Auditorium

As seen in the group 9 of the OpenBACH actions, the Controller is able to compute some statistics on the values generated by a Job. These values can be retrieved in the Frontend using the "Graphical Analysis" tab on a Project.

4.7. Display of data collection

The objective of this section is to first remind the full data collection chain of OpenBACH, including the collection carried out by the jobs and the centralization of the data in the collector. Secondly, we aim at showing the details regarding the considered display options for the different types of data collected in OpenBACH. We aim at presenting the requirements and the design regarding the data display frontends (real-time data, real-time logs and offline data)

4.7.1. Real-time logs

As it has been detailed previously, the Collect-Agent daemon (Rsyslog) is in charge of collecting the different logs sent by the running job instances in a network entity. Those logs are then relayed to the Collector (via UDP/TCP sockets), which stores them into the chosen Logs database (i.e. Elasticsearch).

Regarding the logs display, OpenBACH shall offer a web interface (via Firefox/Google Chrome web browsers) for visualizing the collected logs on real-time.

The Log messages displayed shall at least contain the following information:

- Time/date of log message collection
- Log level
- ID of the network entity (e.g. hostname)
- Name of the Job sending the log message
- Scenario ID and job instance ID (if they are generated by a job instance)
- The message

Moreover, the logs web interface shall propose tools allowing to perform:

- Logs research
- Logs filtering (e.g. filters for host machine, IP, job, log level, etc.)
- Different auto refresh intervals, from 5 seconds to several hours.
- Calculation of number of statistics per applied filter, per time window.

Kibana has been chosen as frontend for the logs web interface. It is an open-source data visualization platform that allows a user to interact with the collected data, organize and plot different graphics and create your own logs dashboards. It is able to use the HTTP Restful API to query logs from Elasticsearch.

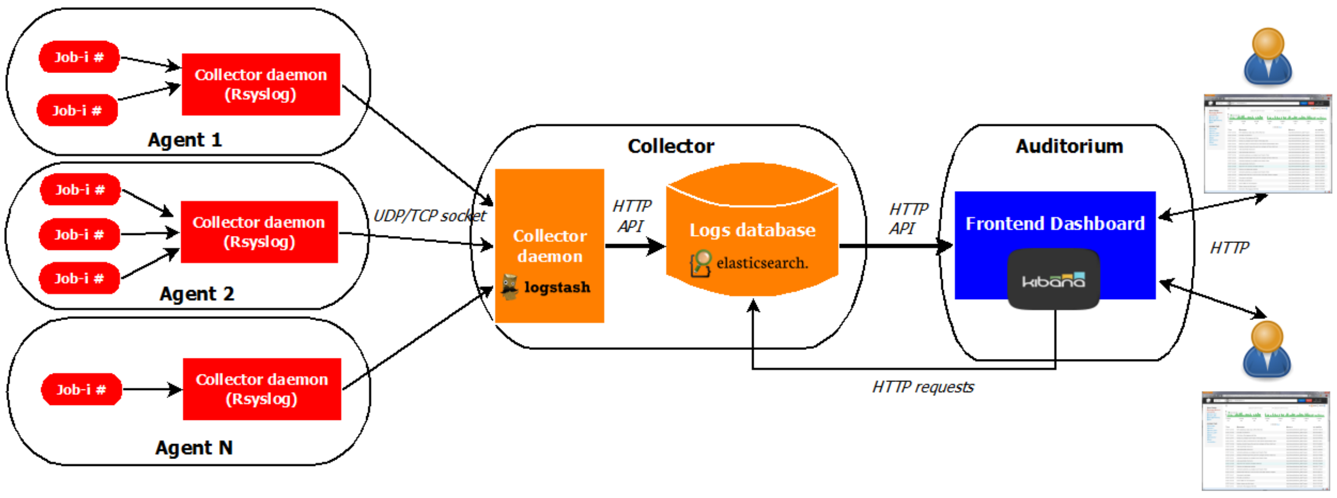


Figure 18. Collection and display of log messages.

4.7.2. Real-time statistics

Concerning the collection of statistics, the Collect-Agent daemon (Rstats) is in charge of collecting the different stats sent by the running jobs instances in a network entity. Those stats are then relayed to the Collector (via UDP/TCP sockets), which stores them into the chosen Stats database (i.e. InfluxDB).

Regarding the stats display, OpenBACH shall offer a web interface (via Firefox/Google Chrome web browsers) for visualizing the collected stats on real-time.

The statistics name shown in the web interface shall be able to be chosen depending on:

- The statistic name (and Job name)
- The ID of the network entity (e.g. hostname)
- The time/date of data sample
- Scenario ID and job instance ID
- The data

Moreover, the stats web interface shall propose tools allowing to perform:

- Statistics research per host and per job instance.
- Simple calculation such as maximum/minimum/average values.
- Different auto refresh intervals, from 5 seconds to several hours.
- Snapshot of the graphics (in order to share them or use them in documents).

Grafana has been chosen as frontend for the stats web interface. It is an open-source dashboard for data display that allows a user to visualize and interact with the collected data, organize and plot different types of graphics and create your own dashboards. It is able to use the HTTP API to query the statistics from InfluxDB data base.

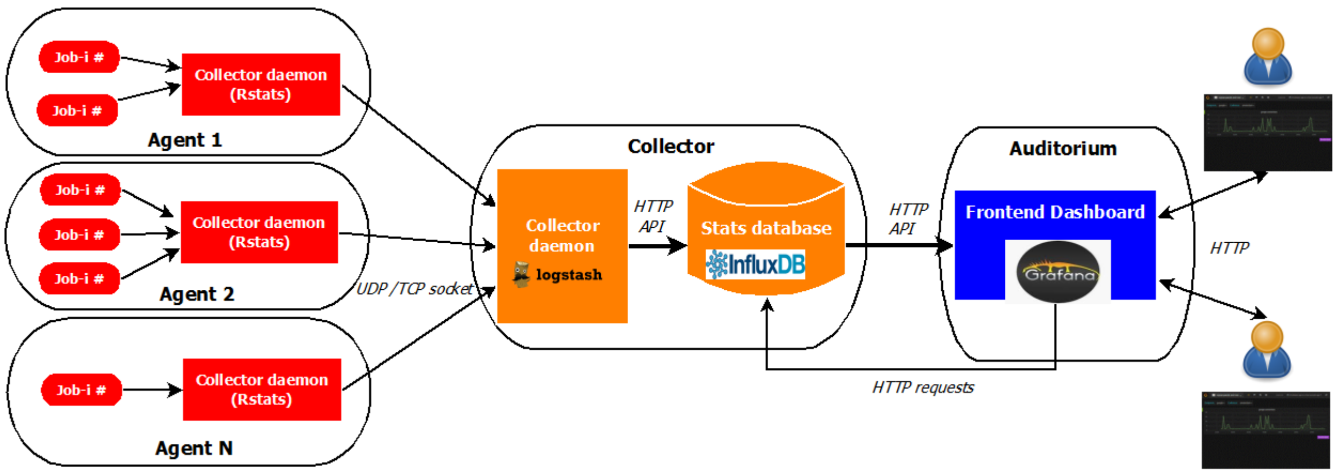


Figure 19. Collection and display of real-time statistics.

4.7.3. Post-processing data statistics

Regarding the offline display, OpenBACH shall offer a web interface (via Firefox/Google Chrome web browsers) for visualizing the post-processed metrics and other offline statisics.

The offline web interface shall propose tools allowing to perform:

- Advanced manipulation of graphics
- Snapshot of the graphics (in order to share them or use them in documents).

These elements have been implemented in the Auditorium Frontend as a "Graphical Analysis" tab in each project.