D5.1	
Version	1.0
Author	ZHAW
Dissemination	PU
Date	29-06-2018
Status	FINAL

D5.1 ElasTest Test Support Services v1

Project acronym	ELASTEST
Project title	ElasTest: an elastic platform for testing complex distributed large software systems
Project duration	01-01-2017 to 31-12-2019
Project type	H2020-ICT-2016-1. Software Technologies
Project reference	731535
Project website	http://elastest.eu/
Work package	WP5
WP leader	Andy Edmonds
Deliverable nature	Public
Lead editor	Andy Edmonds
Planned delivery date	30-06-2018
Actual delivery date	29-06-2018
Keywords	Open source software, cloud computing, software engineering, operating systems, computer languages, software design & development, service delivery

Funded by the European Union

License

Contributors

Name	Affiliation
Juan Navarro	URJC
Mica Gallego	URJC
Varun Gowtham	TUB
Sathi Rowshan	TUB
Cesar Sanchez	IMDEA
Felipe Gorostiaga	IMDEA
Pablo Chico de Guzman	IMDEA
Nikolaos Stavros Gavalas	REL
Avinash Sudhodanan	IMDEA
Juan Caballero	IMDEA
Andy Edmonds	ZHAW

Version history

Version	Date	Author(s)	Description of changes
0.1	29.03.2018	Andy Edmonds	Initial draft and outline.
0.2	30.05.2018	All Contributors	Initial section contributions.
0.3	30.05.2018	Andy Edmonds	Editing and formatting.
0.4	31.05.2018	Juan Navarro, Nikolaos Stavros Gavalas, Andy Edmonds	EUS, EBS conclusions, conclusions, section on TSS creation.
0.5	31.05.2018	Andy Edmonds	Section on TSS costing, minor additions to intro, updated API tables to include `/health`
0.6	31.05.2018	Varun Gowtham, Avinash Sudhodanan	Updates to EDS and ESS content
0.7	25.06.2018	Andy Edmonds	Updates resulting from peer review process.
1.0	29.06.2018	Andy Edmonds	Finalisation of deliverable.

Table of Contents

Index of Figures

Index of Tables

Glossary of Acronyms

Acronym	Definition
CI (Continuous Integration)	This refers to the software development practice with that name.
FOSS (Free Open Source Software)	This refers to software released under open source licenses.
IaaS (Infrastructure as a Service), PaaS (Platform as a Service) and SaaS (Software as a Service)	This refers to different models of exposing cloud capabilities and services to third parties.
Instrumentation	This refers to extending the interface exposed by a software system for achieving enhanced controllability (i.e. the ability to modify behaviour and runtime status) and observability (i.e. the ability to infer information about the runtime internal state of the system).
QoS (Quality of Service) and QoE (Quality of Experience)	QoS and QoE refer to non-functional attributes of systems. QoS is related to objective quality metrics such as latency or packet loss. QoE is related to the subjective quality perception of users. In ElasTest, QoS and QoE are particularly important for the characterization of multimedia systems and applications through custom metrics.
SiL (Systems in the Large)	A SiL is a large distributed system exposing applications and services involving complex architectures on highly interconnected and heterogeneous environments. SiLs are typically created interconnecting, scaling and orchestrating different SiS. For example, a complex microservice-architected system deployed in a cloud environment and providing a service with elastic scalability is considered a SiL.
SiS (Systems in the Small)	SiS are systems basing on monolithic (i.e. non-distributed) architectures. For us, a SiS can be seen as a component that provides a specific functional capability to a larger system.
SuT (Software under Test)	This refers to the software that a test is validating. In this project, SuT typically refers to a SiL that is under validation.
TO (Test Orchestration)	The term orchestration typically refers to test orchestration understood as a technique for executing tests in coordination. This should not be confused with cloud orchestration, which is a completely different concept related to the orchestration of systems in a cloud environment.

TORM (Test Orchestration and Recommendation Manager)	Is an ElasTest functional component that abstracts and exposes to testers the capabilities of the ElasTest orchestration and recommendation engines.
T-Job (Testing Job)	We define a T-Job as a monolithic (i.e. single process) program devoted to validating some specific attribute of a system. Current Continuous Integration tools are designed for automating the execution of T-Jobs. T-Jobs may have different flavours such as unit tests, which validate a specific function of a SiS, or integration and system tests, which may validate properties on a SiL as a whole.
TiL (Test in the Large)	A TiL refers to a set of tests that execute in coordination and that are suitable for validating complex functional and/or non-functional properties of a SiL on realistic operational conditions. We understand that a TiL can be created by orchestrating the execution of several T-Job.
Test Support Service (TSS)	A service acquired on-demand and use in support of a T-Job.
ElasTest User impersonation Service (EUS)	One of ElasTest's Test Support Services. See section 4.1.
ElasTest sensor, actuator and Device emulator Service (EDS)	One of ElasTest's Test Support Services. See section 4.2.
ElasTest Monitoring Service (EMS)	One of ElasTest's Test Support Services. See section 4.3.
ElasTest Big data analysis Service (EBS)	One of ElasTest's Test Support Services. See section 4.4.
ElasTest Security check Service (ESS)	One of ElasTest's Test Support Services. See section 4.5.
ElasTest Test and Orchestration Manager (ETM)	Main ElasTest coordinating entity of T-Jobs
Service Oriented Architecture (SOA)	An architectural style used to design distributed service-based applications.
Web Real-Time Communication (WebRTC)	Enables audio and video streams to work in web pages using a direct peer-to-peer paradigm.
Man-in-the-Middle (MitM) attack	A type of security exploit where an interloper is transparently inserted between what should be a one-to-one communication.
Fundamental Modeling Concepts (FMC)	A modelling framework for the description of software-systems.

1 Executive Summary

There is no doubt that the delivery of services to end-users is and has been a huge productivity gain for developers. It is this which is a key aspect when one talks about digitisation of business today. The same should also be provided to testers of software and services. In this deliverable we report the results of designing and implementing Test Support Services that are used within the ElasTest platform. These provide additional functionality to T-Jobs without the owner of the T-Job to be concerned with having to create and manage the implementation of these services. Through the use of orchestration in WP4, these services can also be presented to the T-Job owner in a uniform and contiguous fashion.

In this deliverable, we firstly introduce the work package and its aims, detail the common elements that all services are required to have in order to be used as ElasTest Test Support Services and then for each of the five services in Work Package 5 (WP5) we detail the design and implementation. Those Test Support Services are:

- **ElasTest User impersonation Service (EUS)**: This service enables the impersonation of end-users in their tests through GUI (Graphical User Interface) instrumentation and through mechanisms for QoS (Quality of Service) and QoE (Quality of Experience) evaluation.
- **ElasTest sensor, actuator and Device emulator Service (EDS)**: This service is useful for enabling tests to emulate customized device behaviour at the time of testing IoT (Internet of Things) applications.
- **ElasTest Monitoring Service (EMS)**: This service leverages runtime verification ideas (in turn inspired by formal verification) to represent the system behaviour as sequences of events that can be monitored in universal ways.
- **ElasTest Big data analysis Service (EBS)**: Enables the collection, analysis and visualization of large volumes of logs.
- **ElasTest Security check Service (ESS)**: For security vulnerability checking targeting specifically the problems of the main large scale deployed system.

Ultimately, the core to this deliverable is the software, which is delivered from this technical work package, including common guidelines on how to create a Test Support Service and the architectures and implementations of each of the five Test Support Services.

This deliverable makes references to the following deliverables: D3.1 (mainly for items related to the ElasTest Platform Manager, Service Manager and Monitoring Platform), D4.1 (for items related to the ElasTest Cost Engine) and D6.2 (for items related to the ElasTest Toolbox).

2 Introduction

In this deliverable we describe the work of WP5 up to month 18 of the project ElasTest.

WP5's main objective is to provide and deliver Test Support Services (TSS) that help in the creation of T-Jobs. The functionality of the TSSs is provided as on-demand services. This means that only when the functionality is required, the service/services are instantiated. These services are defined according to the general architecture principles and style specified in D2.3, namely as cloud-native, microservice-based services [TSS1]. Access to the functionality of these services is through well-defined API and UI interfaces. How the API is defined, again follows the D2.3 mandated adoption of OpenAPI[1].

These services are to be ultimately used and consumed by the ElasTest Test and Orchestration Manager (ETM), however they can be used independently of this through the ElasTest Service Manager.

This deliverable is aimed at those wishing to understand how Test Support Services are designed and implemented through:

1. Understanding the common guidelines required to be adhered by each service.
2. Understanding how services can be design and implemented and integrated into the ElasTest service. In this case, the "how" is given by example of the five different services within WP5.

This deliverable is structured in the following way. In section 3, we present the common guidelines that each service should follow from design, through implementation to deployment and delivery. In section 4, we provide the detailed information on the features, architecture and other outputs of each Test Support Service. Finally, in section 5 we conclude the deliverable detailing key follow on development activities for each Test Support Service.

[1] https://www.openapis.org

3 Test Support Service Management

Test Support Services are services used by tests via T-Jobs. They augment the capability of a test by providing some specific features. For example, the ElasTest User Impersonation Service (EUS) provides to tests the ability to control web browsers and emulate a real user using it. Hence, all services should provide some way of programmatic usage of the service, so that it can be configured/used by the test code.

Some services can provide also a graphical interface to be used directly by the tester. For example, EUS provides the web browser inside ElasTest main interface to be managed by the tester if he wants. As another example, the ElasTest Device emulator service could also provide a web interface to control manually the value of a sensor or the state of an interrupter.

The technologies used in WP5 are directly influenced by the technology decisions made in WP3. As such all services are presented for deployment as Docker[2] container images that are composed by Docker's docker-compose[3] technology. The implementation of each service is however at the discretion and preference of the partner that is responsible for the service implementation. It is expected that all services will have a set of tests that can validate the service and how these are implemented is again at the discretion of the implementing partner. As ElasTest supports the approach of Continuous Test, Integration and Deployment (CI/CD), Jenkins[4] is used as the common platform for achieving this and as such, all service must present a Jenkinsfile that details its CI/CD process. Other more ElasTest-specific common aspects are discussed in this section.

3.1.1 Definitions

To aid our discussions related to TSSs, it is helpful to define some basic terms.

- **Service**: The Organization for the Advancement of Structured Information Standards (OASIS) defines service as "a mechanism to enable access to one or more capabilities, where the access is provided using a prescribed interface and is exercised consistent with constraints and policies as specified by the service description" [TSS4]. A service can be uniquely identified by its interface. Its identity is known as its type (i.e. Service Type).
- **Service Instance (SI)**: Is defined as a single instance of a service of a certain service type requested on-demand by a user.
- **Sub-Service (SS)**: Is an integral, internal element of a Service Instance. Such sub-division is typical of microservice- and SOA-based architectures [TSS3].
- **Resource**: Any *physical* or *virtual* component of limited availability within a computer or information management system. Computer resources include means for input, processing, output, communication, and storage. A resource is

[2] https://www.docker.com
[3] https://docs.docker.com/compose/
[4] https://jenkins.io

owned by one or more entities. Services and Sub-Services run upon resources, specifically to ElasTest, mainly virtual.

By extension all of these definitions are applicable to Test Support Services. It is the task of the ElasTest Service Manager (ESM) to manage these entities through a service life cycle process. The ESM is the component responsible for the delivery and management of TSSs (see D3.1 for more details). In general, there are different categories of service types. These can largely be split into the following two types, we define:

- **Atomic**: is category of service that is an indivisible service that executes a particular singular business or technical function. An Atomic Service is not subject to further decomposition and its business or technological function does not justify breakdown to smaller components.
- **Composed**: aggregates/combines services together with orchestration logic. Both Atomic and Composed Services can be used to create further composed services.

Particular to ElasTest there is a category known as **Test Support Services (TSS)**. TSSs are to be ultimately used and consumed by the ElasTest Test and Orchestration Manager in order to provide additional functionality to a T-Job, without the T-Job developer having to implement that functionality. They can be thought of the import of a 3rd party library but delivered as a service. Each TSS is managed by the ESM following a common life cycle model.

3.1.2 TSS Life Cycle

A Test Support Service can be started manually by an ElasTest user or can be started automatically when a T-Job is configured to use a TSS and that T-Job is executed. Each service instance has a life cycle that is defined and managed by the ESM, as defined in D3.1. This life cycle is a simple life cycle and only accounts for the technical realisation of the service. The life cycle includes all phases from the design of the service to the disposal of a service instance. The life cycle is heavily based on the Hurtle orchestrator's [TSS2] life cycle. The phases of the life cycle are as follows:

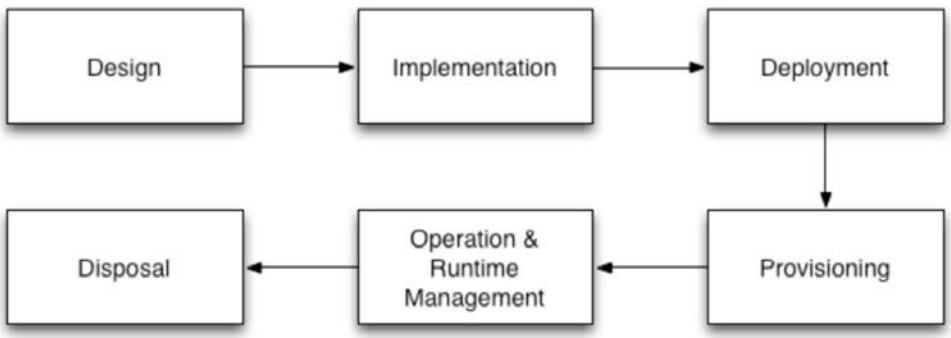

Figure 1 TSS Life Cycle.

- **Design**: Design of the architecture, implementation, deployment, provisioning and operation solutions. This generally a human-oriented activity.

- **Implementation**: Of the designed architecture, functions, interfaces, controllers, APIs, etc. It should be noted here that all TSS APIs are specified using the OpenAPI interface description language.
- **Deployment**: Deployment of the implemented elements, e.g. networks, volumes and containers, etc. Any resource so the service can be used. Access to the service is not available at this phase.
- **Provisioning**: Provisioning of the service environment. Activation of the service such that the user can actually use it. This in the specific case of ElasTest, this means configuring the service with the required parameters. These parameters are supplied by passing environment variables into the container.
    - Once the service is appropriately configured, the owner of the service instance can request access to the service instance through "binding". This part of the provisioning phase is specific to the ESM's API.
- **Operation and Run-Time Management**: In this stage, the service instance is ready and running. Activities such as scaling, reconfiguration of Sub-Services are carried out here. This phase is a service-specific task.
- **Disposal**: Release of all SSs, the service instance itself and virtual resources is carried out here.

3.1.3 TSS Interaction with ElasTest

There are a number of interactions that the TSSs have with other parts of ElasTest. The following subsections describe them in detail.

3.1.3.1 TSS Registration

When ElasTest is started, ETM will register the services in ESM. Services are not started (provisioned) at this time, given that they are created on-demand by the ETM on the behalf of the T-Job owner. The ETM has access to the `elastestservice.json` files located in the root of the repository. The ElasTest Toolbox downloads these files when a new platform container is created and published. For more information about ElasTest Toolbox, please see D6.2.

3.1.3.2 TSSs Used by Tester

When the TSS is started manually by the tester, the following steps are performed:

- The user can request to start a service using the ElasTest GUI whenever they wish.
- Then, ETM will provision the service using ESM API. ESM will use the `docker-compose.yml` content (if using docker-compose) specified as part the service's manifest to start the service.
- When the service is started, ESM provides the IP of every subservice included in the service to the ETM. With these IPs and the endpoints section defined in the `elastestservice.json`, ETM will embed the service web GUIs, if available, in the main ETM GUI using `iframes`. In that way, the ElasTest user can interact with the service. Also, the services can provide other network endpoints to be used by ElasTest users. For example:

- EUS: Provides a WebDriver[5] compatible endpoint to be used to manage tests during test development.
- EDS: Provides an endpoint where sensors, actuators and devices can be managed.
- EMS: Provides a Logstash[6] endpoint to be used to receive beats entries to be processed by event machines.
- EBS: Provides a Spark[7] endpoint to accept job deployment.
- ESS: Provides an endpoint to managed security test runs.

- The ETM can request to delete (deprovision) a service so as not to waste computational resources or incur large charges than are needed.

3.1.3.3 TSSs Used by Tests Inside a T-Job

When the TSS is associated to a T-Job to be used by the tests inside the T-Job, the following steps are performed:

- A T-Job is created in ElasTest by means of ETM (using GUI or using REST API)
- The T-Job specifies what TSSs are going to be used by the test.
- When the user requests the ETM to start the T-Job, the ETM looks at the TSSs that are necessary to execute that T-Job. Then ETM will ask ESM to provision service instances for that T-Job.
- When the services are provisioned, the network endpoints of the subservices will be provided to T-Job using environment variables. These environment variables are created with the pattern:
  `ET_<SERVICE_SHORTNAME>_<SUBSERVICE>_<NAME>_API`
  for HTTP(S) services.
  - `<SUBSERVICE>` won't be included in the main subservice.
  - `<NAME>` won't be included for the first API object of the array.
- All APIs are supposed to be HTTP(S) or WS[8] (s). For example, based on the `elastestservice.json` presented before, the environment variable will be: `ET_EUS_API=http://<IP>:8040/eus/v1`.
- For non-HTTP(S) services, host and port will be provided in different environment variables with the pattern:
  `ET_<SERVICE_SHORTNAME>_<SUBSERVICE>_<NAME>_HOST`
  and:
  `ET_<SERVICE_SHORTNAME>_<SUBSERVICE>_<NAME>_PORT`.
- When a T-Job is finished, then services are deprovisioned by the ETM.

Note that if there are two T-Jobs using the same service, then there will be two service instances executing at the same time. The environment variables[9] that will be available in T-Jobs containers for the Test Support Services included in ElasTest are:

[5] https://www.w3.org/TR/webdriver/
[6] https://www.elastic.co/products/logstash
[7] https://spark.apache.org
[8] Web Socket
[9] These environment variable values are subject to change because they are generated automatically from endpoint section in the `elastestservice.json` file.

- EUS
  - ET_EUS_API
  - ET_EUS_EUSWS_API
- EBS
  - ET_EBS_HOST
  - ET_EBS_PORT
- EMS
  - ET_EMS_API
  - ET_EMS_LSBEATS_HOST
  - ET_EMS_LSBEATS_PORT
  - ET_EMS_ELASTICSEARCH_API
- EDS
  - ET_EDS_API_API
- ESS
  - ET_ESS_API_API

3.1.4 TSS Description

In order for services to be managed by the ESM, they need to be registered with the ESM. The ESM is an ElasTest core component and is started when ElasTest Platform starts. It is responsible for the life cycle management of instantiated services and the service types registered in its catalogue. Once it is started. the ETM component registers all known ElasTest TSSs in ESM. In order to accomplish this registration, the ETM provides the service description of a TSS. This is the definition contained in a file named `elastestservice.json`. In ElasTest, this description file is hosted within the service implementation repository, however, it can be located anywhere the ETM has access to.

3.1.4.1 TSS Descriptor File (`elastestservice.json`)

The `elastestservice.json` is a JSON document and has the following structure:

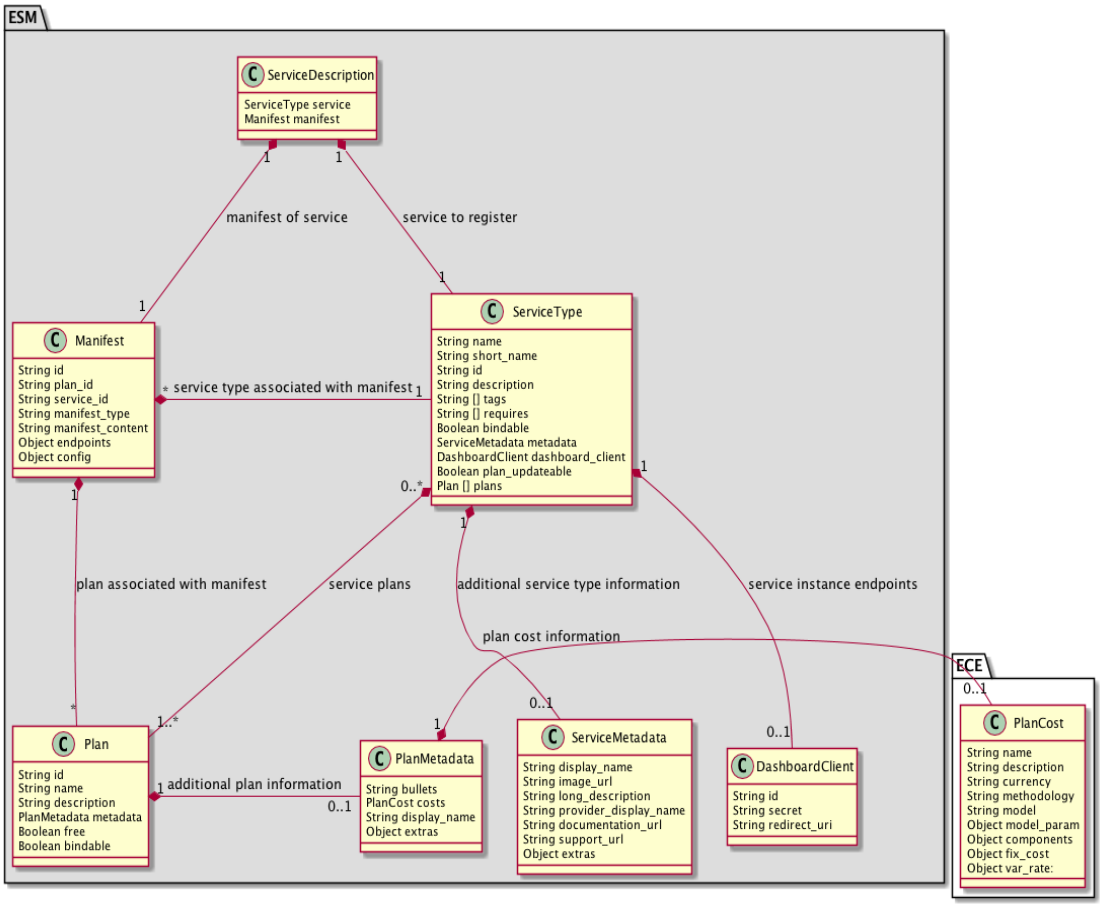

Figure 2 TSS Descriptor File's Document Model

Below is an example of such a service description file.

```
{
   "register": {
      "description": "ElasTest service that provides...",
      "id": "29216b91-497c-43b7-a5c4-6613f13fa0e9",
      "name": "user-emulator-service",
      "bindable": false,
      "plan_updateable": false,
      "plans": [
         {
            "bindable": false,
            "description": "Basic plan for EUS",
            "free": true,
            "id": "b4cfc681-0e28-41f0-b88c-dde69169a256",
            "metadata": {
               "bullets": "basic plan",
               "costs": {
                  "components": {
                  },
                  "description": "On Demand 5 per deployment, 50 per core.",
                  "fix_cost": {
                     "deployment": 5
                  },
                  "name": "On Demand 5 + Charges",
                  "type": "ONDEMAND",
```

```
                  "var_rate": {
                     "cpus": 50,
                     "disk": 1,
                     "memory": 10
                  }
               }
            },
            "name": "EUS plan"
         }
      ],
      "requires": [],
      "tags": [
         "browser",
         "selenium",
         "webdriver",
         "gui automation"
      ]
   },
   "manifest": {
      "id": "2bd62bc2-f768-42d0-8194-562924b494ff",
      "manifest_content": "version: '2.1'\nservices:\n...",
      "manifest_type": "docker-compose",
      "plan_id": "b4cfc681-0e28-41f0-b88c-dde69169a256",
      "service_id": "29216b91-497c-43b7-a5c4-6613f13fa0e9",
      "endpoints": {
         "eus": {
            "description": "W3C WebDriver standard sessions operations",
            "main": true,
            "api": {
               "protocol": "http",
               "port": 8040,
               "path": "/eus/v1/",
               "definition": {
                  "type": "openapi",
                  "path": "/eus/v1/api.yaml"
               },
               "health_path": "/eus/v1/application/health"
            },
            "gui": {
               "protocol": "http",
               "path": "/gui",
               "port": 8041
            }
         }
      }
   }
}
```


The "`register`" property of the file is used as the body of the register operation ("Register a Service") of the ESM. See D3.1 for details of this operation or here for an example of service registration[10].

[10] https://github.com/elastest/elastest-service-manager/blob/master/docs/integration-doc.md#register-a-service

The "manifest" property will be used as the body of the manifest operation ("Register a Manifest for a Service's Plan") of ESM. See D3.1 for details of this operation or here[11] for an example of manifest registration.

The "endpoints" property indicates the endpoints provided by the service. Every object in "endpoints" field should match the service name in the docker-compose. The "api" field is used to define the programmatic endpoints of the service (in every subservice). The "gui" field is used to define the web GUI of the service and/or to enable embedding in the ElasTest GUI.

```
"api": [{
      "protocol": "http",
      "port": 8040,
      "path": "/eus/v1/"
    },
    {
      "name": "ws",
      "protocol": "ws",
      "port": 8041,
      "path": "/eus/"
    }
]
```

3.1.4.2 TSS Context and Configuration Information

When a TSS is provisioned, it is important that the service instance itself knows what the context and configuration is in which it is executed. For that reason, all TSS subservices defined in the manifest (within in the elastestservice.json) will receive the following environment variables from the ETM, to know how to access to other ElasTest components.

- ET_EIM_API
- ET_EDM_API
- ET_EDM_ALLUXIO_API
- ET_EDM_MYSQL_HOST
- ET_EDM_MYSQL_PORT
- ET_EDM_ELASTICSEARCH_API
- ET_EPM_API
- ET_ETM_API
- ET_ETM_LSTCP_HOST
- ET_ETM_LSTCP_PORT
- ET_ETM_LSBEATS_HOST
- ET_ETM_LSBEATS_PORT
- ET_ETM_LSHTTP_API
- ET_ETM_RABBIT_HOST
- ET_ETM_RABBIT_PORT

[11] https://github.com/elastest/elastest-service-manager/blob/master/docs/integration-doc.md#register-a-manifest-for-a-services-plan

- ET_ESM_API
- ET_EMP_API
- ET_EMP_INFLUXDB_API
- ET_EMP_INFLUXDB_HOST
- ET_EMP_INFLUXDB_GRAPHITE_PORT

Other useful environment variable is ET_FILES_PATH. That variable can be used by the TSSs to store files that have to be maintained (associated to the T-Job or ElasTest itself). ETM is responsible to manage files created during T-Job execution and show them in the GUI.

There is also other information that can be useful inside the TSS containers but cannot be injected with environment variables because it is known after the service is provisioned. To avoid that limitation, a REST endpoint is provided to the Test Support Service instance. The URL of that endpoint is codified in the environment variable ET_CONTEXT_API. When that URL is requested with GET method, the following response is returned:

```
{
   "ET_PUBLIC_API": "http://....",
   "ET_PUBLIC_EUSWS_API": "http://....",
   "ET_EMP_INFLUXDB_API": "http://....",
   "ET_EMP_INFLUXDB_HOST": "http://....",
   "ET_EMP_INFLUXDB_GRAPHITE_PORT": "http://....",
   ...
}
```

The important fields are those starting with “ET_PUBLIC_”. These fields have the public URLs that can be used from outside ElasTest (from the browser, for example) to reach the endpoints provided by the service. The rest of the fields are all the environment variables defined previously.

3.1.5 TSS Instance Monitoring for T-Jobs

One of the main features of ElasTest is the ability to monitor T-Jobs and the SuT. That is, the ability to show in ElasTest GUI all log entries and other metrics like CPU utilisation, memory utilisation, etc. of all software artifacts involved in the test execution. For that reason, it is important to also be able to monitor the TSS instances.

Service instances are provisioned as one or more containers using the manifest file. We call "subservices" to every one of these containers. By default, the log of every subservice is sent to the Logstash[12] installed as part of ElasTest Platform. From there, it is recorded in an ElasticSearch[13] database and sent to a RabbitMQ[14] message queue system (to be shown in the GUI). All events (log entries and metrics) are associated with

[12] https://www.elastic.co/products/logstash
[13] https:// www.elastic.co
[14] https://www.rabbitmq.com

the T-Job and with the SuT in such way that all that monitoring information can be analysed at the same time. For example, if a T-Job is using a browser provided by EUS service, then the log generated in that browser is merged with the log generated in the SuT (a web server) and in the test itself.

In addition to the container logs and metrics, an ElasTest service can send monitoring information to Logstash using some of the input plugins it provides: HTTP, Syslog[15] format by TCP, Beats[16], etc. Logstash IP and ports will be available in all containers of the service in the environment variables:

- `ET_LSHTTP_API`
- `ET_LSBEATS_HOST`
- `ET_LSBEATS_PORT`
- `ET_LSTCP_HOST`
- `ET_LSTCP_PORT`

Two more variables are also available for a TSS instance:

- `ET_TJOB_ID`
- `ET_TJOBEXEC_ID`

Alternatively, the current ESM development is on-going to allow specific metrics (of the service developer's want) be sent to the ElasTest Monitoring Platform (EMP).

3.1.6 TSS Health Check

Every TSS implements a fast check to know that the service is operational. This is specific to the implementation, however the basic structure of the returned document should be compliant with Spring Boot health checks[17]. For that, every TSS should provide a REST endpoint to be called by the ESM. The REST endpoint should have the following characteristics:

- **Default path:** "`/health`" at the root of the service. If the health path is different than `/health`, then use the property "`health_path`" in the "`api`" section of the `elastestservice.json` document.
- **Logic:** Implements a simple logic calculation, however the more complete this check can be, the more accurate the status can be (e.g. if a service comprises of a front-end and database, executing some logic within the front-end and querying the liveness of the database connection is useful). The output must be a JSON dictionary/hashmap with the fields: "`status`" having a value of "up" or "down", "`out_of_service`". These are the values used in Spring Boot. Also, custom values can be added, if required and this other information is optional. If other information is to be added, then it should be placed against an attribute named "`context`".

[15] https://en.wikipedia.org/wiki/Syslog
[16] https://www.elastic.co/products/beats
[17] https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-monitoring.html

```
{
  "status": "up", //can also be: down, out_of_service
  "context": {}  //any additional info
}
```

It also can be useful to implement a "`/environment`" endpoint that provides info on the runtime environment that the process was started with. This is optional but can help with operational diagnosis.

3.1.6.1 Registration of Health Check Endpoint

The registration of the health check endpoint should be done as any service endpoint. Please see the following example on registering the endpoint:

```
"api": [{
      "protocol": "http",
      "port": 8040,
      "path": "/eus/v1/"
    },
    {
      "name": "ws",
      "protocol": "ws",
      "port": 8041,
      "path": "/eus/"
    },
    {
      "name": "health",
      "protocol": "http",
      "port": 8444,
      "path": "/health/"
    }
]
```

3.1.7 TSS & Creating New Computational Resources

All services can create new containers during their execution. This allows services to request further resources, allowing them to scale themselves. Currently, each service implementation manages the scaling of itself, this is also reflected in how the ESM's Open Service Broker API is presented. Services have a fixed set of resources depending on the selected plan and so the goal of a service implementation is more to maintain the service level delivered to the user of the service. Dynamic scaling is currently not used in the ESM, rather a plan-based approach is used and therefore static scaling can be achieved by updating the plan associated with a service instance. This work on dynamic scaling will be investigated in upcoming release cycles.

In ElasTest, services can use docker directly using the Docker API or use the ElasTest Platform Manager. For example, the EUS uses the Docker API to create a container for each browser requested. Services that cannot use Docker API directly can use ElasTest Platform Manager API instead. There is an extensive documentation[18] on how to use

[18] https://github.com/elastest/elastest-platform-manager/blob/master/docs/index.md

EPM and further documentation of the EPM can be found in D3.1. The EPM API URL is available for all TSS instances in the environment variable `ET_EPM_API`.

3.1.8 TSS Costing

In order for the ElasTest Cost Engine (ECE) to carry out its function of estimating the complete cost of a T-Job execution, the ECE must be able to retrieve the cost of each Service Type's offered Plan. In order to enable this, the ESM allows to combine the ECE cost model (See D4.1) with the Service Type's Plan[19] description (see ESM's data model in D3.1). An ECE cost model per TSS has been created in collaboration with WP4 and is integrated with each of the TSS's `elastestservice.json` file. Once placed there, the ECE cost model is registered along with the Service itself and this information can be queried through the ESM's service catalogue by the ECE. For providers of TSSs, should they require billing of their service, they must include this information in their `elastestservice.json` file. Below is an example cost model used by the EUS service (starting from line 17).

[19] This is the data structure that is used to describe the cost model of the particular service type.

```
{
  "register": {
    "description": "ElasTest service that provides user impersonation as a service",
    "id": "29216b91-497c-43b7-a5c4-6613f13fa0e9",
    "name": "EUS",
    "short_name": "EUS",
    "bindable": false,
    "plan_updateable": false,
    "plans": [
      {
        "bindable": false,
        "description": "Basic plan for EUS",
        "free": true,
        "id": "b4cfc681-0e28-41f0-b88c-dde69169a256",
        "metadata": {
          "bullets": "basic plan",
          "costs": {
            "description": "cost model for eus",
            "currency": "eur",
            "model": "pay-as-you-go",
            "model_param": {
              "setup_cost": 3.5
            },
            "meter_list": [
              {
                "meter_name": "chrome_browser",
                "meter_type": "counter",
                "unit_cost": 5,
                "unit": "instance-hour"
              },
              {
                "meter_name": "firefox_browser",
                "meter_type": "counter",
                "unit_cost": 2.5,
                "unit": "instance-hour"
              },
              {
                "meter_name": "edge_browser",
                "meter_type": "counter",
                "unit_cost": 2,
                "unit": "instance-hour"
              }
            ]
          }
        },
        "name": "EUS plan"
      }
```


Figure 3 EUS Cost Model Example

3.1.9 TSS Testing

It is important to implement tests for TSSs. All types of tests are important and should take into account the test pyramid[20].

In the case of E2E tests, TSSs should be tested in integration with the rest of the ElasTest platform. At least, two E2E should be implemented in every TSS:

[20] https://martinfowler.com/bliki/TestPyramid.html

- **TSS started by GUI Test:** A Test to use the TSS started manually using ElasTest GUI.
- **TSS started by T-Job Test:** A Test to use the TSS associated to a T-Job started when T-Job is executed.

These tests should simulate the interaction of a user with the ElasTest Web GUI. For this reason, it is recommended to use Selenium technology to implement those tests. TSS developer can use any technology to implement those tests supported by Selenium. In the next sections, we will describe those E2E tests in further more detail.

3.1.9.1 TSS started by GUI Test

This test is implemented with the following characteristics:

- The test is implemented using Selenium technology with any programming language and testing framework.
- The test will receive as environment variable the URL to connect to the ElasTest web GUI.
- Test should perform the following steps:
  - Navigate to Test Support Services section of ElasTest.
  - Select the TSS to be tested.
  - Start the TSS using the GUI.
  - Wait until the TSS instance is available.
  - If the TSS can be used using a GUI:
    - Continue using Selenium to use that GUI embedded in ElasTest GUI. For example, EUS allows the user to start and manage a browser using the GUI.
    - If not: use the TSS instance using the endpoints shown in the GUI. For example, EBS provides a Spark API that can be used from the test itself using the public endpoint. For example, this[21] is a test implemented in Java using Spark.
- The Jenkinsfile[22] should perform the following steps:
  - Start ElasTest using the Toolbox.
  - Wait until ElasTest is started because it can take several minutes.
  - Get ElasTest GUI URL.
  - Execute E2E tests passing the environment variable where ElasTest is accessible.

As an example, the EUS E2E test can be viewed in this source file[23] of its Maven[24] project. Also, to support this example is the EUS Jenkinsfile[25] that starts this test. For more information about the test project please see the documentation[26].

[21] https://blog.matthewrathbone.com/2015/12/28/java-spark-tutorial.html
[22] https://jenkins.io/doc/book/pipeline/jenkinsfile/
[23] https://github.com/elastest/elastest-user-emulator-service/blob/master/e2e-test/src/test/java/io/elastest/eus/test/e2e/EusSupportServiceE2eTest.java
[24] https://maven.apache.org
[25] https://github.com/elastest/elastest-user-emulator-service/blob/master/e2e-test/Jenkinsfile
[26] https://github.com/elastest/elastest-user-emulator-service/blob/aad177c4bb44d720d431479685957a8fa2cee977/e2e-test/docs/index.md

3.1.9.2 TSS started by T-Job Test

This test is implemented similarly to the other test regarding to ElasTest platform start and the Jenkinsfile steps. The only difference is in the steps performed by the test itself. The following are the test steps:

- Create a new project.
- Create a new SuT in the project if necessary.
- Create a new T-Job with the following information.
    - A docker image used to execute the T-Job commands. For example, a docker image with Java, Maven and Git.
    - The list of the commands to execute inside of the image. For example:
        - clone a git repository with one or more tests.
        - execute the tests.
- Select the TSS that is going to be used inside the test.
- Save the T-Job.
- Execute the T-Job.
- Verify that test pass using the services provided by TSS.

An example of such a test can be seen in the EUS E2E test in this source file[27] (a Maven project).

3.1.10 TSS Documentation

A TSS should be documented with two different perspectives: the user documentation and the development documentation. This documentation is split into two new files and is used to create a ElasTest user documentation in the ElasTest web page. All documentation for a TSS must be placed in a folder named '`docs`' in the root of the service's repository.

3.1.10.1 User Documentation

User documentation should include the following:

- **Description**: One paragraph description of the TSS
- **Features**: A complete list of features provided by the TSS in the current version.
- **How to use from GUI**: If the test can be used from the ElasTest GUI, it contains a tutorial on how to use it.
- **How to use from a test**: It contains a tutorial on how to use the TSS from a Java test. Other languages/technologies can also be provided in the tutorial.

This documentation should be written in `/docs/user-docs.md` file of the service's repository.

3.1.10.2 Development Documentation

Development documentation should include these points:

- **Architecture**: Architecture of the component specifying the relation with other ElasTest components

[27] https://github.com/elastest/elastest-user-emulator-service/blob/master/e2e-test/src/test/java/io/elastest/eus/test/e2e/EusTJobE2eTest.java

- **Prepare development environment**: How to prepare the development environment to develop this TSS. For example, what tools are required, in which version, etc.
- **Development procedure**: All information useful to develop the component. For example, how to execute the component in the IDE, how to compile it, etc.
- **Docker images**: Description of the docker images used for the component. If the components can be developed outside ElasTest and need some docker containers to simulate the other ElasTest components, they should be described here.
- **Continuous Integration**: The description of the jobs used in ElasTest CI, how they are used and when they are executed.

This documentation should be written in `/docs/dev-docs.md` file of the service's repository.

3.1.11 TSS Creation

Using the information contained in sections 3.1.1 to 3.1.9 a developer, along with the support of the current set[28] of TSSs creates their own TSS. The key requirements in doing so are listed here.

- The initial step is the design and implementation of the service to be delivered as a TSS. This can be done using whatever design methodology that is appropriate. The architecture of the service can be monolithic all the way through to microservices -based. The key requirement is that the service can be accessed and operated over a network-based protocol (e.g. HTTP(s)).
- The service and subservices themselves can be implemented in any language of choosing.
- It is highly recommended that each service implements a health endpoint as this will allow for the ESM to check and validate that the service instances of that particular service type are operational.
- Services should be tested and documented. Guidelines on what is expected of TSSs can be found in sections 3.1.8 and 3.1.9
- Once a service has been implemented, it must be packaged as container image(s). For ElasTest TSSs they must be packaged as docker containers. Currently it is expected that these container images are placed in a container image registry that is available to the ESM and/or EPM.
- The set of docker containers comprising the service implementation must be described by a service manifest. The service manifest depends on the type of container orchestration management system supported by the ESM. Currently, docker-compose is the only manifest format supported (including the option of deploying using the EPM).
- Once the service manifest is created, the `elastestservice.json` file can then be created according to section 3.1.4. Currently, this descriptor should be hosted at the root of the service's code repository to which the ETM has access.

[28] https://github.com/elastest?utf8=✓&q=service&type=&language=

- Note: should a service be billed to the submitter of a T-Job, then the `elastestservice.json` must include in its service's plan the model used by the ECE (details in D4.1).

A note on networking: Currently, all instances share the same network. This causes the issue of port conflicts both within the docker network and also in exposing host ports. There are currently two proposals to fixing this issue:

- A per instance network is created for the service instance and all containers associated with the service are placed upon this network
  - a. can be done (has been experimentally) and works, except that remapping of port numbers needs to be done by ETM/ESM
- A per tenant docker infrastructure instance is created and only service instances of that tenant are placed here
  - a. requires further integration with the EPM
  - b. requires tested AAA integration in the EPM

- Currently for a service instance to be created it must be placed on a docker-managed network named '`elastest_elastest`'. Below is the docker-compose definition required:

```
networks:
  elastest_elastest:
    external: true
```

Figure 4 Service docker-compose Network Definition.

4 ElasTest Test Support Services

In this section each of the five core ElasTest Test Support Service (TSS) is described. All of these services are available from the main ElasTest repository[29] and are available under the open source Apache 2.0 license[30], unless otherwise specified.

4.1 ElasTest User Impersonation Service

4.1.1 Introduction

The ElasTest User Impersonation Service (EUS) TSS is devoted to providing the appropriate technologies for impersonating users in end-to-end tests. This is achieved by handling GUIs (Graphical User Interfaces) using automation techniques. Currently, EUS provides the ability to impersonate users manipulating web applications. In future releases, it is planned to allow the impersonation of mobile applications too.

Moreover, EUS enables to measure the end-user's perceived quality for standard Web Real-Time Communication (WebRTC) [31] applications, so that testing through the validation of the perceived quality becomes possible. For this, QoS indicators are gathered by EUS. These indicators include traffic metrics such as network latency, network packet loss, network jitter, retransmissions and consumed/estimated bandwidth. In the future, it is planned to offer also QoE metrics. For this, multimedia QoE for audio and video will be analysed using different full-reference algorithms, such as Perceptual Evaluation of Speech Quality (PESQ) for audio or Structural SIMilarity (SSIM) for video to name a few [EUS0].

4.1.2 Features

The list of high-level capabilities provided by EUS at the moment of this writing is the following:

1. Use browsers manually.
2. Drive browsers GUIs in an automated way.
3. Automate and assess WebRTC applications.
4. Measure the end-user's perceived quality by means of QoE and QoS indicators (Planned).
5. Record of browser in automated and manual sessions.

These capabilities are exposed by EUS by means of a REST API. The definition to this REST API has been defined using Open API notation [EUS1]. This specification is available on the EUS GitHub repository[32]. Moreover, this API can be reviewed in a web friendly format in the official ElasTest documentation[33].

[29] https://github.com/elastest
[30] https://opensource.org/licenses/Apache-2.0
[31] https://webrtc.org
[32] https://github.com/elastest/elastest-user-emulator-service/blob/master/api.yaml
[33] https://elastest.io/docs/api/eus/

4.1.3 Baseline Concepts and Technologies

EUS is devoted to providing user impersonation for web browsers GUIs. For this, EUS provides a *Browser as a Service* (BaaS) capability suitable for exposing browser GUIs through an API in a universal way. This service has been built on top of popular technologies such as Selenium [EUS2]. In that way, any testing library can be used with this TSS. In the next versions of this component a new feature will be implemented, the ability to control mobile applications in an automated way. This will be implemented on top of the popular open source automation for mobile applications Appium [EUS3], also based on Selenium.

In order to drive browsers and mobile devices in an automated fashion, EUS has been conceived as an extension of the W3C WebDriver recommendation [EUS8]. This recommendation was based on the so-called JSON Wire Protocol[34], first developed by the Selenium team. This protocol defines a REST API instrumented by means of JSON messages over HTTP. Nowadays, this protocol is being standardized in the WebDriver API by W3C. Therefore, the EUS component provides full compatibility with external browser drivers (e.g. Selenium Grid applications) but enhanced with new capabilities to allow automation for different kinds of GUI applications (including browsers and mobiles in the future) while allowing advance quality assessment (including QoE and QoS metrics).

To manage the browser instances controlled by Selenium, EUS uses Docker containers. Specifically, one Docker container it is instantiated for every controlled browser. The containers used in EUS have been created by the ElasTest project. These images are open and available on Docker Hub[35]. These images are based on the official Selenium Docker containers [EUS4], and also on the containers provided by the Selenoid project (a scalable Selenium Hub implementation in Go language) [EUS5]. Some of the new features added to the ElasTest browser containers are the following:

- noVNC[36] (i.e. VNC client for HTML5 canvas elements) is included in the browser containers, allowing external browsers to be connected directly to the browser container [EUS6].
- Recording video and audio in MP4 format. The recording is currently done directly in the browser container using FFmpeg [EUS7].
- The ability to start and stop the recording of the browser. This allows to record in several time ranges.

All in all, EUS provides a fully SaaS model so that developers do not need to take into consideration problems related to computing resources scheduling, software provisioning or system scaling. For managing the life cycle of containers, EUS is using the service of ElasTest Platform Manager (EPM) core component.

[34] https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol
[35] https://hub.docker.com/u/elastestbrowsers/
[36] http://novnc.com

4.1.4 Component Architecture

The FMC diagram for EUS is shown in Figure 5.

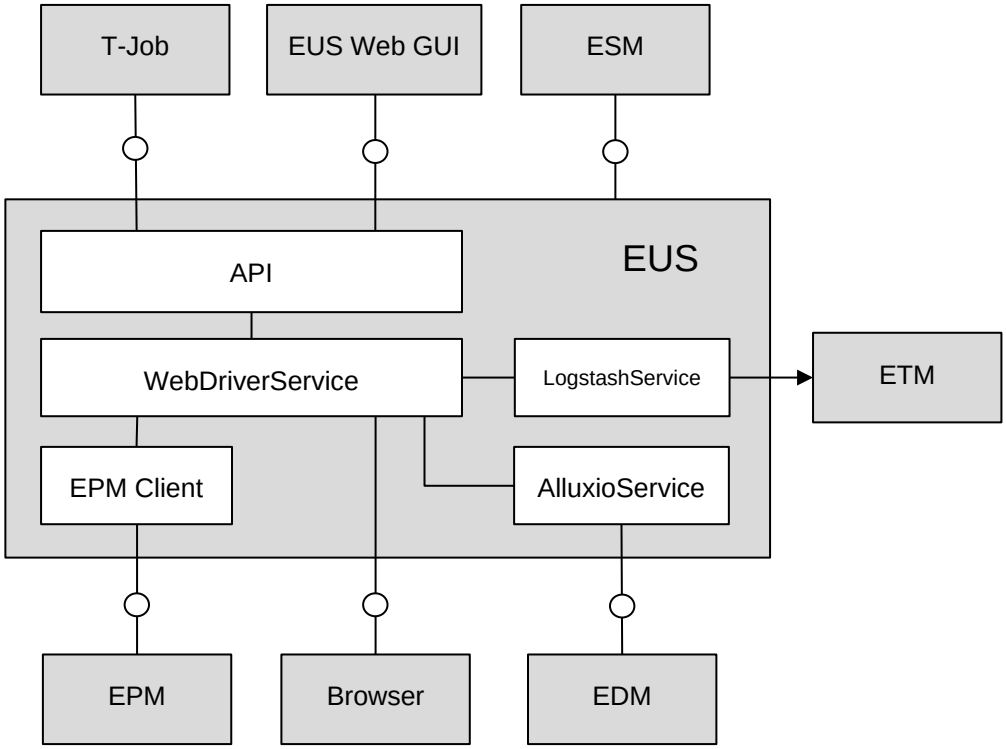

Figure 5 EUS FMC Diagram

The class structure of the EUS is shown in the following UML class diagram.

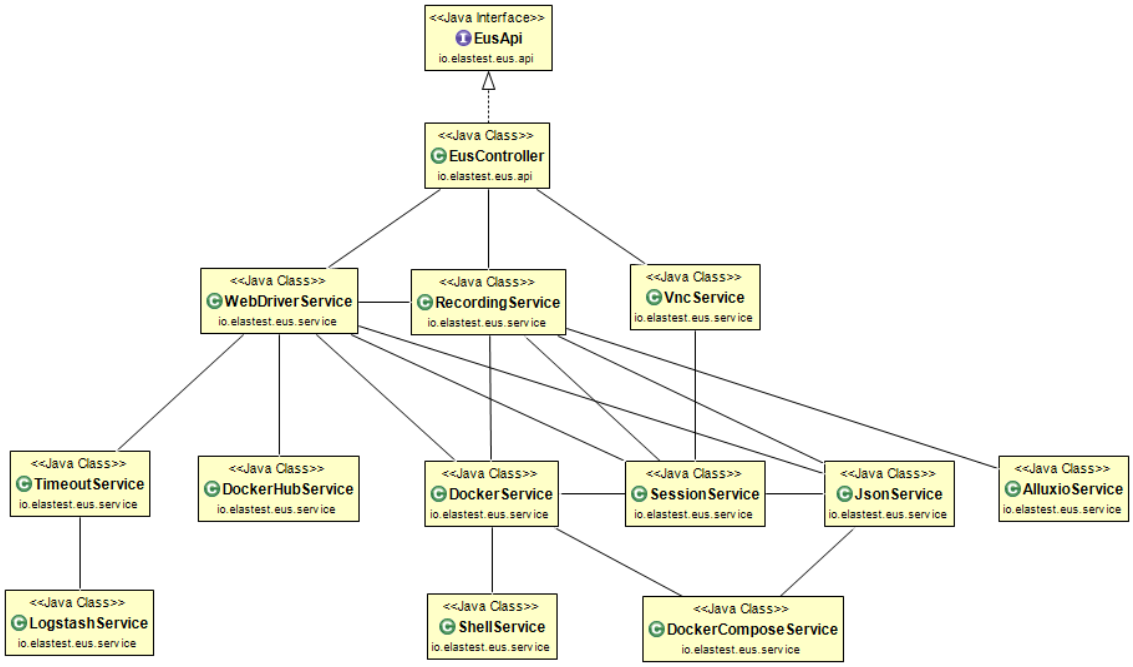

Figure 6 EUS Class diagram

4.1.4.1 Use Case Diagrams

The uses cases for EUS identified at the time of this writing are the following:

1. W3C WebDriver. This use case is aimed to provide full compatibility with the existing W3C WebDriver recommendation, allowing to use EUS by Selenium tests.
2. Basic media evaluation. These operations provide custom features to assess WebRTC applications, such as read audio level of RGB colour of video tags.
3. Event subscription. This set of operations allows to subscribe to Document Object Model (DOM) events (e.g. click, change, and so on).
4. Advance media evaluation. This use case allows to measure advance metrics for WebRTC applications, such as end-to-end latency of audio and video QoE.
5. Remote control. This use case allows to record and share remote sessions of user sessions.
6. WebRTC stats. This use case allows to get complete QoS indicators for WebRTC applications.
7. WebRTC user media. This use case will allow to set specific media content (video and/or audio) for WebRTC communications.

As shown in the following chart, all these operations can be invoked by end-to-end tests, typically executed from ElasTest T-Jobs.

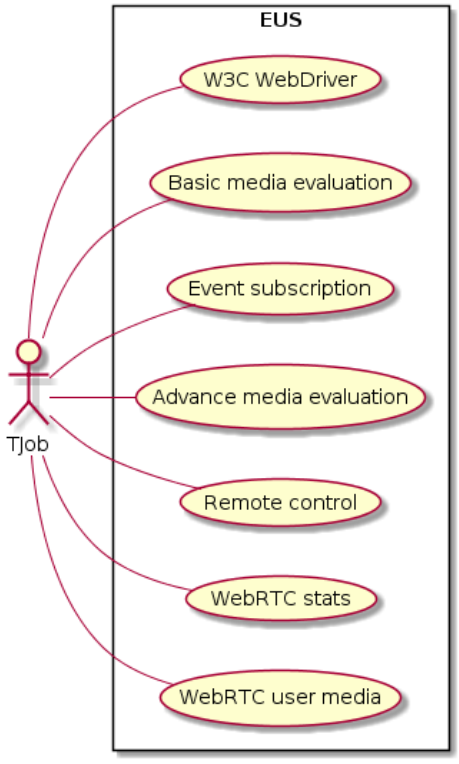

Figure 7 EUS Use Cases

As introduced before, the EUS operations have been specified in OpenAPI format and documented in the ElasTest website. The following table provides a summary of the operations together with the REST endpoints provided by EUS.

Method	URL	Description
1. W3C WebDriver compatibility.		
GET, POST, DELETE	/session/**	W3C WebDriver standard sessions operations
GET	/status	W3C WebDriver standard get status operation
2. Basic media evaluation		
GET	/session/{sessionId}/element/{elementId}/audio	Read the audio level of a given element (audio|video tag)
GET	/session/{sessionId}/element/{elementId}/color	Read the RGB color of the coordinates of a given element
3. Event Subscription		
POST	/session/{sessionId}/element/{elementId}/event	Subscribe to a given event within an element
GET	/session/{sessionId}/event/{subscriptionId}	Read the value of event for a given subscription
DELETE	/session/{sessionId}/event/{subscriptionId}	Remove a subscription
4. Advance media evaluation		
POST	/session/{sessionId}/element/{elementId}/latency	Measure end-to-end latency of a WebRTC session

POST	`/session/{sessionId}/element/{elementId}/quality`	Measure quality (audio|video) of a WebRTC session
5. Remote control		
GET	`/session/{sessionId}/recording`	Get recording
DELETE	`/session/{sessionId}/recording`	Delete recording
GET	`/session/{sessionId}/vnc`	Get VNC session
6. WebRTC stats		
GET	`/session/{sessionId}/stats`	Read the WebRTC stats
7. WebRTC user media		
POST	`/session/{sessionId}/usermedia`	Set user media for WebRTC
8. Service Instance Status		
GET	`/health`	Get the component health status.

Table 1 EUS API Calls

4.1.4.2 Sequence Diagrams

At the time of this writing not all these use cases have been implemented. So far, three out of the total use cases are supported by EUS, namely compatibility with W3C WebDriver, gathering of WebRTC statistics, and remote control. In this section, a sequence diagram per implemented use case is depicted, explaining how EUS interacts with other ElasTest components to support that use case.

The following picture shows the sequence diagram of the use case “W3C WebDriver”. In this sequence diagram two typical W3C WebDriver operations are depicted. First, the creation of a browser session is requested in the execution of a T-Job by means of a “POST `/session`” message. On the reception of this message, EUS requests the creation of a web browser to EPM. The nature of the browser, i.e. the type (Chrome, Firefox, etc) and the version is contained in the body of the origin request. As a result, a

browser is started by EPM (typically using a Docker container), and a unique session identifier (`sessionId`) is returned to the T-Job. This identifier will be used in next requests from the T-Job. At the bottom of the diagram, we can see how the value of `sessionId` is used to terminate the browser session using the command "`DELETE /session/{sessionId}`".

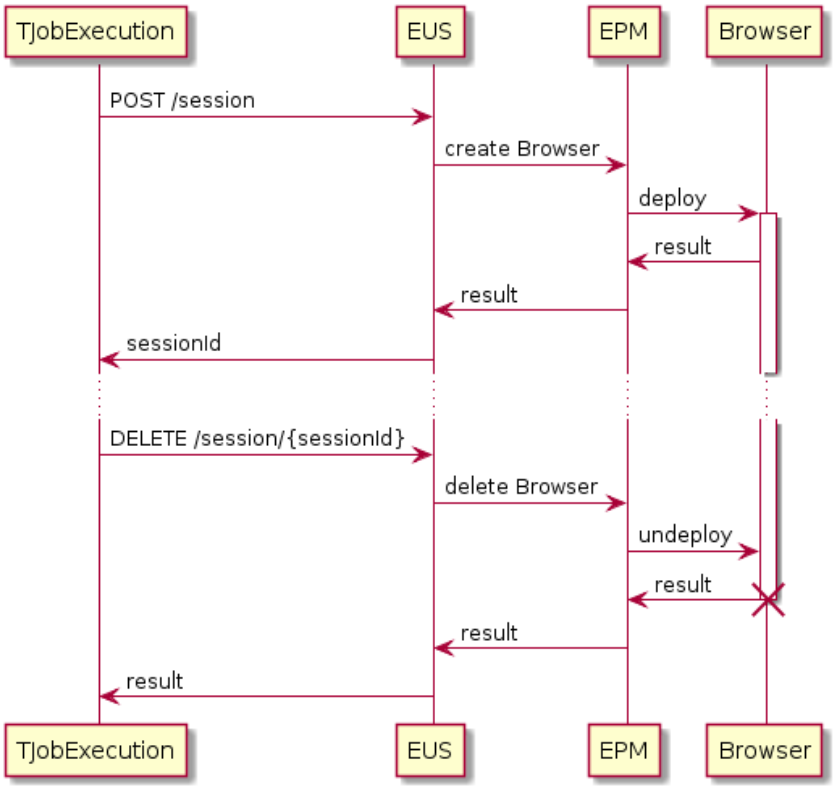

Figure 8 EUS W3C WebDriver

The next use case described with a sequence diagram is related to the capability of gathering WebRTC statistics from previously created web sessions. The command, new in W3C WebDriver and supported out of the box by EUS, is invoked using the command "`GET /session/{sessionId}/stats`". On the reception of that message, EUS will use the native WebRTC support to gather all statistics in the *RTCPeerConnection* objects up and running in the browsers. That data is returned to the T-Job, and also it is sent to the monitoring service (EMS).

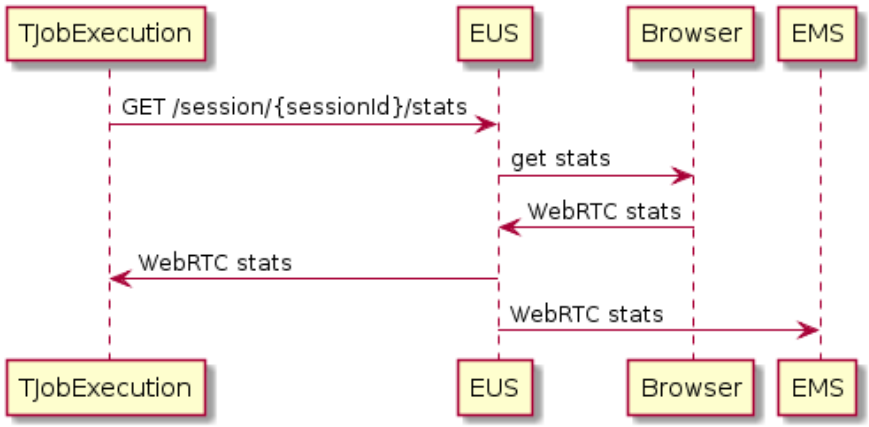

Figure 9 EUS WebRTC Statistics

The last implemented use case so far is related to remote control capabilities. To support these capabilities, EUS implements two additional features out of the box and by default per browser session. On the one hand, browser sessions are recorded automatically. These recordings can be read (and deleted) using custom messages. On the other hand, each browser session is automatically exported using VNC desktop sharing technology. The operation of this capabilities is represented in the following sequence diagram.

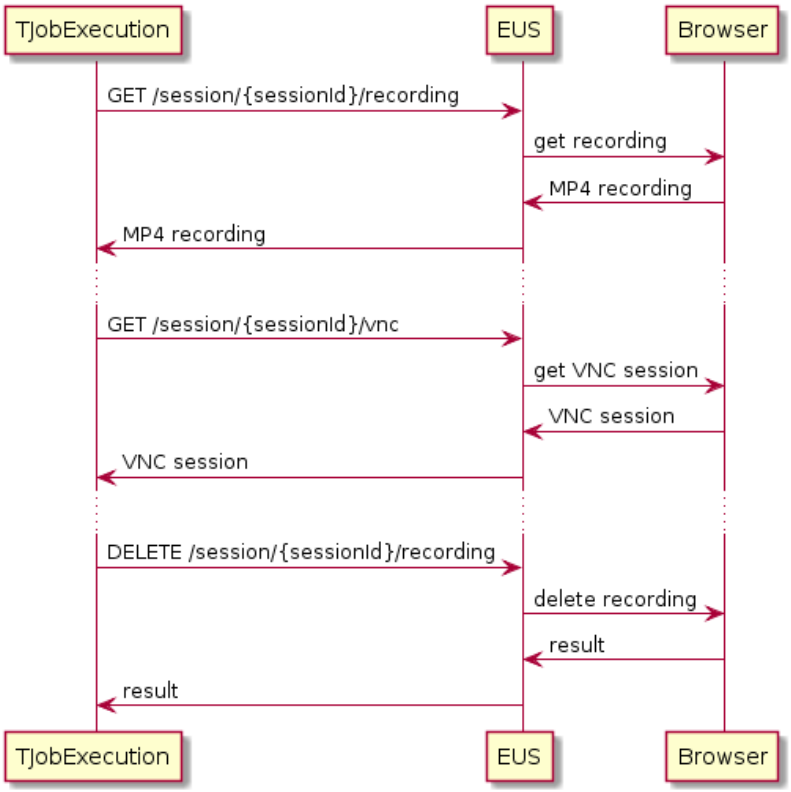

Figure 10 EUS Remote Control

4.1.5 Code Reports

EUS has been implemented as a Spring-Boot application in Java language, and as usual, its source code is open and available on GitHub[37]. The test suite of EUS has been implemented using JUnit 5 [EUS9]. At the time of this writing this suite is composed by 46 tests divided in three categories:

- Unit tests, isolating unit under test using Mockito [EUS10].
- Integration tests, executing different internal components using Spring [EUS11].
- End-to-end tests, using Selenium WebDriver [EUS2] to verify the high-level features of EUS.

The code coverage of this test suite is calculated in each new patch, as long as this new patch does not introduce a new regression (i.e. some test fails). To keep track of the coverage, as usual we use Codecov[38]. As shown in the following chart, the code coverage of EUS remains quite stable (around 75% of line coverage and around 70% of complexity coverage).

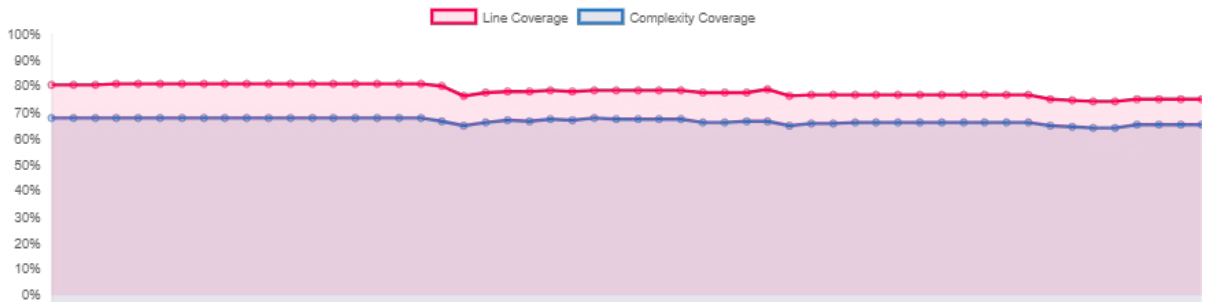

Figure 11 EUS Coverage Chart

4.1.6 Code Links

- The main repository of the EUS is https://github.com/elastest/elastest-user-emulator-service
- The API of the EUS is available at https://elastest.io/docs/api/eus/

4.1.7 Contributions

ElasTest User Impersonation Service contains several innovations and advances in the state of the art that have been published in the following contributions:

- WebRTC Testing: Challenges and Practical Solutions. Boni García, Francisco Gortázar, Luis López-Fernández, Micael Gallego, and Miguel París. IEEE Communications Standards, IEEE, July, 2017.
- WebRTC Testing: State of the Art. Boni García, Micael Gallego, Francisco Gortázar, and Eduardo Jiménez. 12th International Conference on Software Technologies (ICSOFT). SCITEPRESS. Madrid, July 2017.

[37] https://github.com/elastest/elastest-user-emulator-service/tree/master/eus
[38] https://codecov.io/gh/elastest/elastest-user-emulator-service

- Impersonation as a Service in End-to-End Testing. Boni García, Francisco Gortázar, Micael Gallego, and Eduardo Jiménez. 6th International Conference on Model-Driven Engineering and Software Development (MODELWARD), Special Session on domAin specific Model-based AppRoaches to vErificaTion and validaTiOn (AMARETTO). SCITEPRESS. Funchal (Portugal).

In the future, it is expected to include more contributions to this list as EUS development advances and incorporate new features.

4.2 ElasTest Device Emulator Service

4.2.1 Introduction

ElasTest Device Emulator Service (EDS) is a microservice developed in ElasTest as a Test Support Service (TSS), to emulate devices used in the context of Internet of Things (IoT). EDS facilitates rapid prototyping and testing of IoT applications. The emulated devices include sensors, actuators and smart devices which form the basis of IoT applications. Furthermore, EDS can be used to build and test interactive IoT applications. Particularly, in the context of Industry 4.0, the role of Industrial Internet of Things (IIoT) is important in order to automate production environments with the help of cyber physical systems.

The monitoring of industrial shop floor is automated using a cause and effect relationship through the use of sensor as a source of cause and an actuator used to realise the effect. The decision whether to apply an effect based on a set of causes is provided by a logic, which is implemented using a program. The association of the hardware components (such as sensors/actuators) and software components (such as the programmable logic), brings about challenges from defining hardware/software interfaces, to communication and connectivity. The recent advent of fog computing technology has made it possible to distribute sensors/actuators on top of fog nodes which are capable of communicating with other fog nodes to enable Machine to Machine (M2M) communication. The evolution of M2M communication standards such as oneM2M[39] and OPC UA[40] have made it possible to realize IIoT applications using fog nodes.

In order to realize and test IIoT solutions, it is necessary to first procure hardware and software components. To make feasible decisions, it would be helpful if there is a means to test IIoT applications using virtual devices first which is compatible with concepts of M2M communication and fog node architectures. EDS tries to address this challenge by providing emulated devices which communicate using OpenMTC[41], an implementation of the oneM2M standard.

Furthermore, EDS is used in realizing the Industry 4.0 demonstrator in the Work Package 7 (WP7) of ElasTest.

4.2.2 Features

The features of EDS are aimed towards providing a framework to realize Industry 4.0 applications which emulates a smart factory scenario as a whole:

- A means which can provide emulated sensors and emulated actuators which can be initiated in multiplicity based on demand.
- A means which program logic can consume the sensor data and signal actuation.
- A backend which provides the communication capability between sensors, actuators and the application deployed as a SuT in the case of ElasTest.

[39] http://www.onem2m.org
[40] https://opcfoundation.org/about/opc-technologies/opc-ua/
[41] http://www.openmtc.org

Furthermore, a backend is responsible in collecting, analysing activity due to the deployment of the IIoT application.

In the following text, *sensor* refers to emulated sensor and *actuator* refers to emulated actuator, a sensor and actuator is collectively referred to as a *device*, to the level of abstraction that denotes a device to be a fog node.

To support the realization of a smart factory scenario, EDS provides following facilities at various levels of abstraction:

- In the user defined IIoT application (at the level of user):
    - A facility to communicate and register the application with EDS.
    - A facility for user to specify the required sensors and actuators for that application directed to EDS.
    - A facility that deploys the specified sensors and actuators and makes it available to the user where each sensor and actuator is discretely identifiable.
    - A facility for user to consume the data from a discrete sensor selected from the required sensors via a program logic and direct the outcome of such a logic to a discrete actuator selected among the already initiated actuators.
- TSS EDS:
    - is deployed as a minimal component by ESM on requirement (at the level of ElasTest platform):
        - Shall be able to identify the exact user application it is in communication with.
        - Shall include an orchestration feature, which takes incoming requests from demonstrator application about the required sensors and actuators and deploys them, each as a separate device. Thus, each device, by mechanism of mapping or other methods, is identifiable globally by an identifier.
        - Shall track the life cycle of the entity. The entity is born when a requirement is received from the demonstrator application. The entity dies on the exit of the demonstrator application.
        - Shall act as a coordinating entity (gateway/hub) for communication between the application, sensors and actuators.
    - Emulated sensors (at the level of EDS) includes a basic set of sensors commonly found in industry environment used to monitor conditions on the shop floor. They are: Light, Pressure, Movement, Humidity, Temperature and Accelerometer.

In each case, it has to be ascertained what kind of data emulation is required by a given user application and the required sensor, taking into account:

- Suitable range and unit (e.g. normal temperature range is between 20 and 60 degrees centigrade)
- Periodicity of generation
- Aperiodicity/sporadicity of generation
- Jitter and noise
- Probability distribution to be used while generating data
- The provision of a facility, that can communicate the characteristics of data emulation from demonstrator application to the respective sensor.

Emulated actuators normally take in a signal of True or False. Considering this, a minimal actuator can be designed that:

- Takes in signal from the user application.
- Employs the time required for the final actuation to take place after a signal is received.
- The minimal actuator can relay the status of actuation back to the user application.

4.2.3 Baseline Concepts and Technologies

EDS is built using OpenMTC as a middleware. OpenMTC is an implementation of the oneM2M standard. The following text explains basic concepts for OpenMTC. OpenMTC is an open source software, initiated by TU Berlin and Fraunhofer FOKUS. The software was made open source in November 2017 and is available on GitHub[42].

The following terms are relevant in the context of oneM2M and OpenMTC to understand various entities functioning in the implementation of EDS:

- Common Services Entity (CSE): Is an entity which resides in the service layer of oneM2M which provides necessary service functions to realize an application. The OpenMTC gateway is the CSE, in OpenMTC implementation.
- Application Entity (AE): Is an abstraction residing in the CSE representing the user application. Structurally, the AE resides beneath the CSE.
- Container: Is an abstraction in the CSE, which represents a placeholder as the name suggests. In hierarchy, Container resides just under AE.
- Content Instance (CI): Is an abstraction in the CSE, which fills data in the placeholder of Container. In hierarchy, CI resides just under container.

To access the basic facilities of OpenMTC, it is required to at least run the gateway on node which is connected to a network and which is identifiable by an IP address.

[42] https://github.com/OpenMTC/OpenMTC

Consider the following URL:

`http://<Gateway IP>:8000/onem2m`

- Gateway IP: is the IP address of the node where OpenMTC gateway is running.
- 8000 is the service port number used by OpenMTC.
- `onem2m` is the CSE base name.

The URL supports the REST API fully as a consequence of using OpenMTC. It accepts and responds to requests in the form of JSON data format. As the user can nest AE, Container or CI under the CSE base. A desired entity can be reached by using the correct path. For a given path, the children residing under the hierarchy and the parent path are made available in the response for a GET request.

More details about the interaction methods with the gateway can be found in the OpenMTC documentation[43]. From here on, we concentrate on REST API paths starting with the CSE base /onem2m.

A typical path looks like:

`/onem2m/{AE}/{Container}/{ContentInstance}`

It is up to the user to formulate reasonable paths for the his/her application identified by name `{AE}`, under which multiple `{Container}`s can reside, where each `{Container}` can have multiple nested `{Container}`s and so on. For each container, there resides one or more placeholders for data called `{ContentInstance}`. Each path towards an entity is unique because of the unique names given to the entities.

The key concept in building an application is defined below:

- Push data to container: Here an application can push data to a content instance identified by the unique path to the container.
- Subscribe to a container: Here an application can subscribe to a container identified by the unique path to the container. The application is notified and a handler is triggered in the application, when there is data pushed to the subscribed container.

The next subsection explains the implementation which can be understood using the concepts detailed in this subsection.

[43] https://github.com/OpenMTC/OpenMTC/blob/master/doc/openmtc-get-started.md

4.2.4 Component Architecture

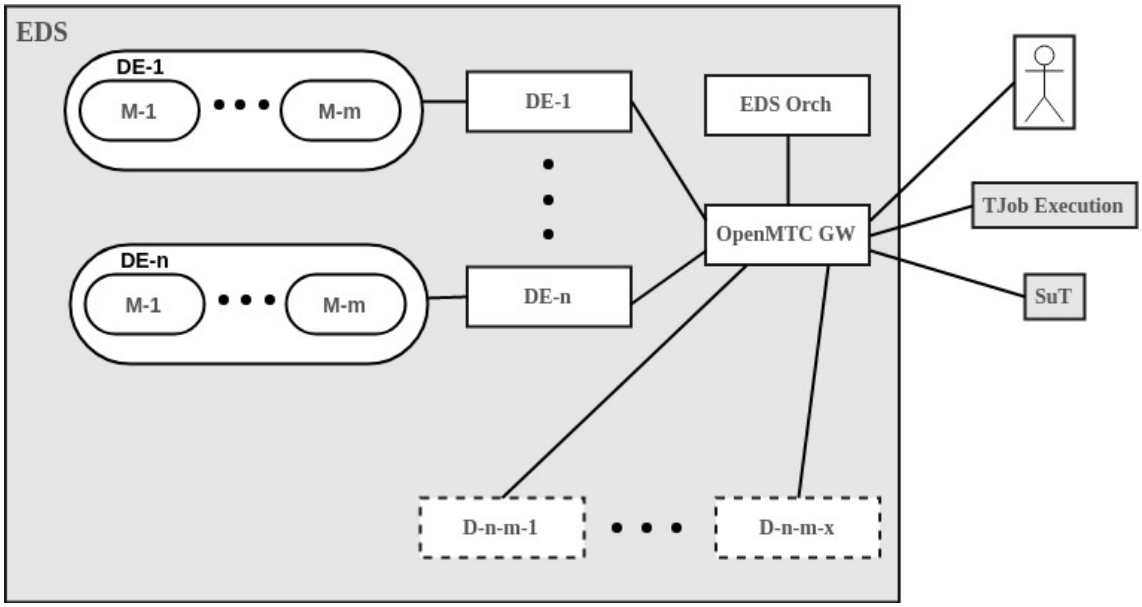

Figure 12 EDS FMC Diagram

The above figure shows the component architecture of EDS. The architecture can be understood by following the descriptions for both internal and external components from the text given below:

Internal components

- OpenMTC GW:
  - This is the CSE defined by OpenMTC, called a gateway (GW) in general. GW is an anchor for communication in EDS. It is the single point of contact for internal and external entities intending to communicate with EDS. It is a main actor and is initialized when EDS is run. It holds all the REST API paths for any entity defined in the context of EDS.
  - Path: `/onem2m/`
- EDS `Orch`:
  - The EDS orchestrator is an AE, which is registered with GW first after the GW is available for communication. The orchestrator is responsible for receiving requests from the user application and taking necessary action by method of first checking if the request is valid and then if valid, the request is processed accordingly. Opens to external world with channels: request, response and status.
  - Path: `/onem2m/EDSOrch/edsorch`
- DE-n:
  - DE stands for device emulator and letter n stands for the type of DE representing a sensor. DE-n is run at start-up by EDS `Orch` for each type of sensor in the manifest of EDS. DE-n is a generic term presented here. The actual name of the DE can be found in EDS repository. The DE on

receiving a request from EDS Orch, is required to process the request and allot the respective device to the requesting application. The nature of the requested device can be obtained from the device model M-m stored in EDS. In response to a given request, the DE on successful installation of a device, attaches to a container under the path of the requesting application and sends the response on its response channel. Opens to external world with channels: request, response and status

    - Path: /onem2m/{DE-n}/

- M-m :
    - These are stored models for each DE, which can provide different behaviour of sensor as requested by user. These are stored in EDS and can be used by an associated DE anytime while EDS is executing.
- D-n-m-x :
    - These are the devices which are established at run time by their respective DEs. The allotted devices are always attached to a user application. In D-n-m-x, D stands for device, n stands for type of DE which it belongs to, m stands for the model being used and x denotes the index of the device from the perspective of EDS Orch. Each device is launched as an individual thread. The thread is responsible for fulfilling the functions of the device based on request provided by the user application. If the device intends to send data, it pushes the data to a container. For example, sensor devices push the sensor values to a container. If the device intends to receive data, it subscribes to a container. For example, an actuator subscribes to a container.
    - Path:
        - If sensor: /onem2m/EDSOrch/{AE}/sensors/{unique_name_of_sensor}
        - If actuator: /onem2m/EDSOrch/{AE}/actuators/{unique_name_of_actuator}

External components

- T-Job Execution: This is the T-Job under execution in ElasTest. The T-Job is able to access or modify the paths established by a running user application.
- SuT: This is the System under Test (SuT), defined by the user. As a result of running the SuT with EDS, one or more new paths are introduced at the GW.
- User: A user, who is able to monitor or introduce test conditions into an established SuT, by accessing or modifying the content instance of a container identified by a path using the REST API.

Following channels are defined for EDS Orch in EDS:

- Request:
  - Path: `/onem2m/EDSOrch/edsorch/request`
  - This is container path, where the user application is required to update the content instance.
  - EDS `Orch` is subscribed to this container.
- Response:
  - Path: `/onem2m/EDSOrch/edsorch/response`
  - This is the container path, where EDS `Orch` updates response to a given request by the user application.
  - The user application is required to subscribe to this path.
- Status:
  - Path: `/onem2m/EDSOrch/edsorch/status`
  - This is the container path, where EDS `Orch` signals whether it is busy with a request or idle.
  - The user application is first required to retrieve the content instance in this container before posting a request.

Similar to the channels described above, the DEs are also provided with request, response and status channels which perform similar functions.

They can be accessed with the following paths:

- `/onem2m/{DE-n}/request`
- `/onem2m/{DE-n}/response`
- `/onem2m/{DE-n}/status`

4.2.4.1 Use Case Diagrams

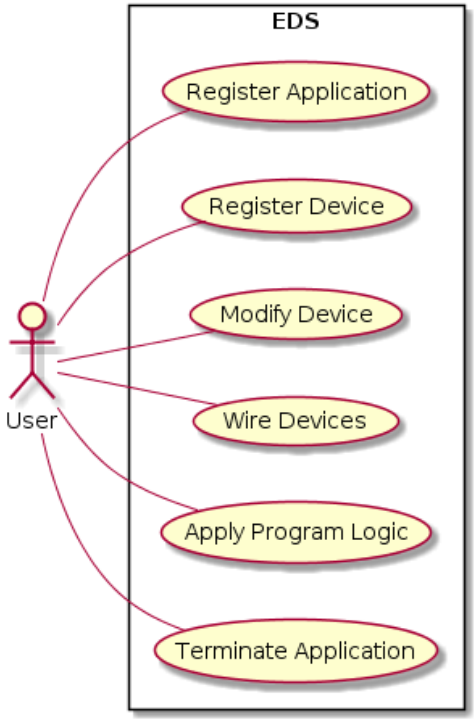

Figure 13 EDS Use Cases

The above image shows the use case diagram for EDS. They can be listed as follows:

1. Register Application: A user application registering with the EDS, in particular with EDS `Orch`.
2. Register Device: A user application, requesting a device from EDS, in particular requested from the EDS `Orch`.
3. Modify Device: A user application or a user, requesting the DE to modify the behaviour of the already available device.
4. Wire Devices: A user application, is able to wire the devices together. This involves:
    a. For any incoming data to the application, the application is required to subscribe to the container from which it expects an input.
    b. For any outgoing data from the application, which the application intends to push into the system, the application is required to push the data to the container to which it intends to push the data as a content instance.
5. Apply Program Logic: This operation is performed on an input received from a user supplied input or an input obtained from a subscribed container. The program logic decides whether to signal or not signal an actuation to the user or the system. In case of if actuation is to be signalled to the system, the application pushes the data to the required container for actuation. Furthermore, such actuation can be time bound, logic bound or even complex that is left to the user's discretion.

6. Terminate Application: User application or user requests to terminate the application. This request is sent to EDS Orch.

Above, use case 4 and 5 are self-explanatory. For the purpose of understanding, use cases 1, 2, 3 and 6 sequence diagrams are presented in the next part of the subsection. These use cases are realised by the following REST API:

Method	URL	Description
1. OpenMTC GW		
GET	/onem2m/**	Get the list of entities available
POST	/onem2m/**	Create AE, Container or CI
PUT	/onem2m/**	Modify a CI
DELETE	/onem2m/**	Delete AE, Container or CI entities
2. EDS Orch		
POST	/onem2m/EDSOrch/edsorch/request	Post a request to EDS Orch
GET	/onem2m/EDSOrch/edsorch/response	Get the response from EDS Orch.
GET	/onem2m/EDSOrch/edsorch/status	Get the status of EDS Orch request channel.
3. Device Emulator (DE)		
POST	/onem2m/{DE name}/request	Post a request to DE
GET	/onem2m/{DE name}/response	DE response from DEl
GET	/onem2m/{DE name}/status	Get the status of DE request channel.
4. Sensor Device		

GET	/onem2m/EDSOrch/{AE name}/sensors/{sensor name}/data	Retrieve the sensor device data
5. Actuator Device		
POST	/onem2m/EDSOrch/{AE name}/actuators/{actuator name}/data_in	Signal an actuator device
GET	/onem2m/EDSOrch/{AE name}/actuators/{actuator name}/data_out	Retrieve the information after the actuator has finished the job
6. Service Instance Status		
GET	/health	Get the component health status.

Table 2 EDS API Calls.

The above table summarizes the REST API for EDS. It should be noted that the OpenMTC provides complete REST API support (i.e. GET, POST, PUT, DELETE) on any path constructed using /onem2m. This is denoted by the first method listed with OpenMTC GW. However, the API type listed in the first column corresponding to the paths listed in each row, indicate the API types which can be used safely in the context of EDS.

Added to the REST API, OpenMTC also provides a low-level SDK with methods which might be more relevant when creating paths for implementation purposes. The methods are listed below:

1. Retrieve.
2. Delete.
3. Create.
4. Notify.
5. Update.

The methods can be used only with the OpenMTC low-level SDK. More information can be found in the OpenMTC documentation[44].

[44] https://github.com/OpenMTC/OpenMTC/blob/master/doc/sdk-client.md

4.2.4.2 Sequence Diagrams

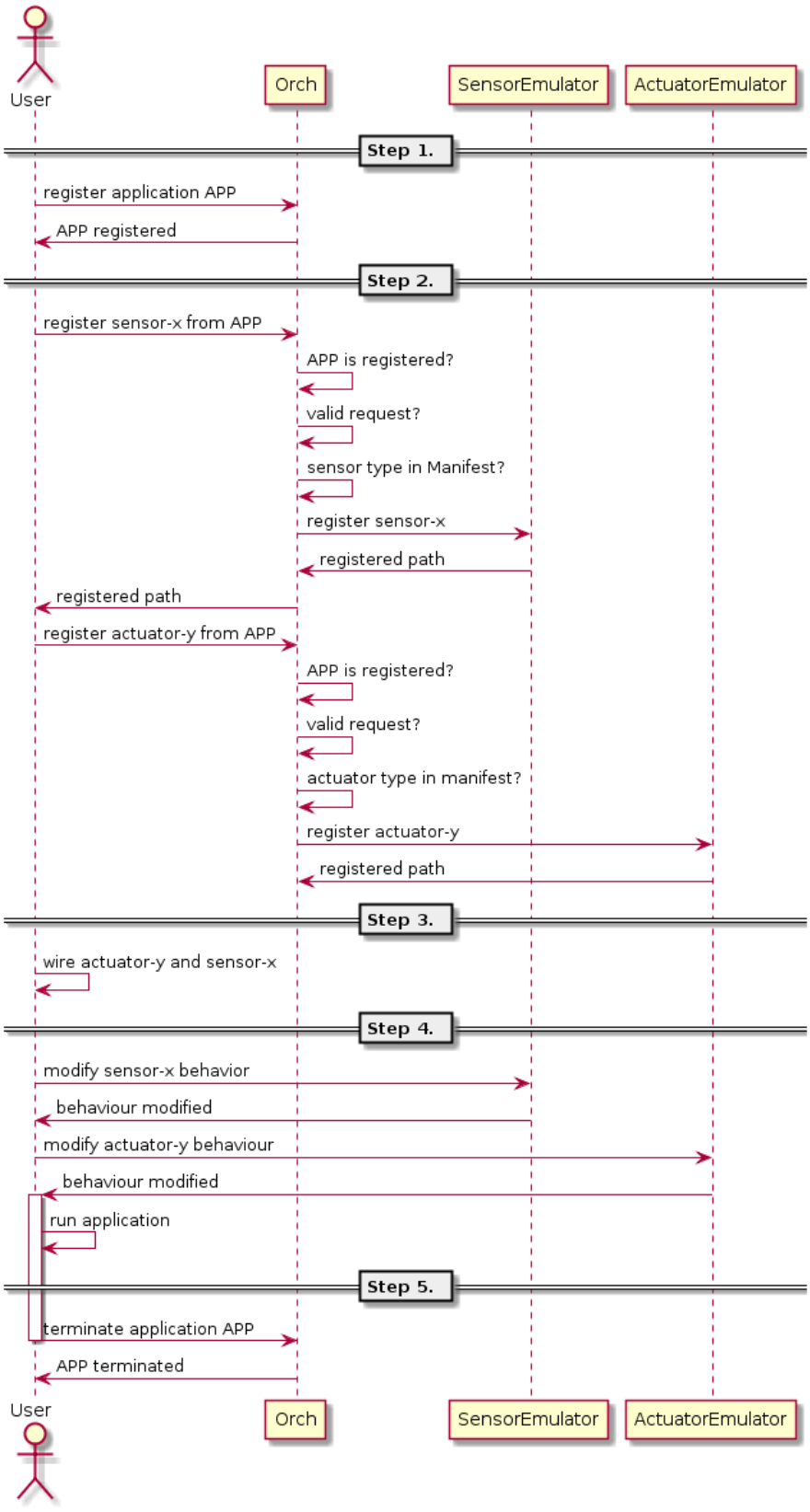

Figure 14 Sequence Diagram of the Life Cycle of a Simple EDS IoT Application

The above shown sequence diagram tries to outline the life cycle of simple IoT application constructed using EDS outlining use cases, 1, 2, 3 and 6 in subsection 4.2.4.1.

The following text details on the Figure 14, the sequence of steps a user follows to interact with EDS to setup a simple IoT application. In this simple IoT application, it is assumed that EDS provides `SensorEmulator` which provides device type sensor and `ActuatorEmulator` which provides device type actuator. The device sensor is assumed to emulate the behaviour of a sensor which can sample for example physical property such as temperature. The device actuator is assumed to emulate a hypothetical actuator which emulates the behaviour of a real-world actuator. The actuator receives a signal to actuate and performs an actuation. In the figure, User could be SuT, T-Job or a regular user able to communicate with EDS using the REST API provided by OpenMTC. In the figure `Orch` stands for EDS `Orch`. Although the main entity to communicate with is the gateway, `Orch` is the entity which receives requests and processes them. Furthermore, device emulators initiated by EDS `Orch`, accept requests to modify the behaviour of the devices that are already alive.

With this information, the sequence diagram could be understood with following sequence of steps:

Step 1. Register the application: User registers with `Orch` the unique application name APP. Once APP is registered with `Orch`, `Orch` maintains a code book for all the applications registered and is able to identify each application and the facilities that the application has been allotted with. This step represents use case 1, mentioned in subsection 4.2.4.1.

Step 2. User requests `Orch` to register a device type sensor with unique name sensor-x. The request is forwarded to the device emulator `SensorEmulator`. The `SensorEmulator` accepts a valid request from `Orch` and registers a device of type sensor with the required model and initiates it with the name sensor-x. Once allotted, the sensor-x's path is forwarded to the `Orch` and `Orch` in turn forwards the response of `SensorEmulator` to the user. The user knowing the path for the sensor-x can now subscribe to that path in the application so that whenever the sensor pushes a value, the user application is notified. In a similar set of events, the device of type actuator is allotted to user by the `ActuatorEmulator`. The actuator with unique name actuator-y is identifiable by a unique path. Here the user in order to trigger the actuator, needs to push data to the container at the actuator's path. Collectively this step summarizes the use case 2 as described in the subsection 4.2.4.1.

Step 3: With the allotted devices, the user can subscribe to the container in the device's path, if the application expects an input; else, the user application can push data to the container in the path, if the application prefers to provide the input to the device. Particularly when the user subscribed to the path of a container path, OpenMTC triggers a handler function. In the handler function, it is possible to implement a program logic to make a decision to provide or not to provide input to a container path. This collective activity is called wiring. This step summarizes the use case 4 and 5 mentioned in subsection 4.2.4.1.

Step 4. If required, the user is free to modify the behaviour of the device that the user application has registered. Modification of behaviour may include tasks such as

switching on/off the device, change the model the device is using etc. These facilities are subject to specific capabilities that are provided by the device emulators. This summarizes the use case 3, mentioned in subsection 4.2.4.1.

Step 5. If the user wishes to terminate the application, this can be done by sending a request for application termination to the `Orch`. This request will be acknowledged by seizing the application and its allotted resources. The termination of the user application does not affect other applications registered with `Orch`. This step summarizes the user case mentioned in subsection 4.2.4.1.

4.2.5 Code Reports

The code coverage was extracted using the EDS unit tests written in Python and ran with nose [45] tests. The code coverage was automated using tox [46]. The code coverage achieved was 40%. This was due to the fact that EDS is composed of applications written on top of OpenMTC framework. The framework in itself is providing various features that the applications might not use to full extent. The code coverage in the case of EDS is tricky to address. EDS is a collection of applications written using OpenMTC using Python. OpenMTC is not present in the EDS repository, rather EDS is installed and shipped in the docker image of EDS. The image is configured such that, on initialization, the application written for EDS in the repository is run in a container where OpenMTC is already installed.

Code coverage of applications where the core facilities of the applications are not present in the repository brings about a risk for misinterpretation of the results. The code coverage of 40% is measured against a machine which has OpenMTC installed and against the EDS GitHub repository of ElasTest. Furthermore, improving code coverage for EDS requires other methods of approach which needs to be planned in the coming releases.

4.2.6 Code Links

- The main repository of EDS is available on GitHub[47].
- The API documentation is available at http://elastest.io/docs/api/eds/

4.2.7 Contributions

With the availability of features with EDS, it is possible to create and test complex applications with ElasTest. Modular applications can be created such they may be interconnected to create more complex applications. This paves way for formulating research problems which can include fault testing and anomaly detection areas.

In the current era, there is an explosion of IoT devices as a result of increase in IoT services [EDS1]. Testing IoT services is expensive because it requires the IoT devices as a pre-requisite. IoT services rely on the data generated and consumed by IoT devices.

[45] https://nose.readthedocs.io
[46] http://tox.readthedocs.io
[47] https://github.com/elastest/elastest-device-emulator-service

IoT services need to ensure that any deviation from the normal behaviour in the data produced is to be detected and dealt with. This topic of understanding the anomaly in the data is called “anomaly detection”. Anomaly detection applies machine learning on top of generated data. EDS can contribute in testing anomaly detection solutions using emulated sensors and actuators [EDS2]. The generation of data and usage could be tracked using the device emulators provided by EDS. The research objectives of EDS can include facilities required for anomaly detection such as changing the behaviour of the devices both in time and type of data generated. Generated data could be modified using the REST API to modify the behaviour of the devices.

It is common understanding that, IoT applications are deployed in multiplicity with variable level of complexity. At the time of designing such applications, the designer is unable to grasp the amount of data that might be generated and plan for a better strategy to avoid faults in the system. A fault for example, could be a sensor or device malfunction. EDS supplements theses test scenarios using the change of behaviour of the device using the REST API. This could provide tester with more opportunities for testing before implementing a live prototype or pilot project.

The above-mentioned publications require a problem to be solved. During the development phase of EDS, the behaviour introduced by devices were kept in mind to define and implement a set of features required in general for anomaly and fault injection testing. In the next half of the project we aim to use the features of EDS to show the impact in anomaly and fault injection testing.

Anomaly detection is another topic in the field of IIoT. This field investigates mechanisms which are used to detect anomaly in machines and the corrective measures that need to be taken if the anomaly is detected. In ElasTest, EDS is capable of provide facilities for anomaly detection and fault testing for IIoT applications.

We intend to extend the features offered by EDS. There is room for improvement in providing the basic type of device emulators. This effort is accompanied by extending the device models of each emulator. Furthermore, a demo application which is capable of displaying the data on a GUI is planned. Providing a wrapper for the OpenMTC application programming interface is planned to be provided to user. This will help user to write applications using basic functionalities of OpenMTC.

4.3 ElasTest Monitoring Service

4.3.1 Introduction

The goal of ElasTest Monitoring Service (EMS) is to provide a monitoring infrastructure that T-Jobs can use for inspecting dynamically executions of a SuT. There are two main aims that a T-Job can pursue with the aid of the EMS:

1. To determine the outcome of a test, where the correctness criteria used depend on temporal aspects of the execution of the SuT. Simple examples include a requirement that the CPU and bandwidth must be below a certain threshold when the SuT is interacted with in a particular way.
2. To guide the test depending on temporal aspects of the execution, that the EMS detects and communicates with the T-Job. A simple example is determining which container is assigned a given task dynamically by the SuT load balancer, so that the T-Job can then interact with the right component to continue the test.

The essential operational element of the EMS is a *monitor*. A monitor observes sequences of events emitted during the execution and collects the necessary information to ultimately detect whether the sequence contains some failure, or some complex sequence of interest. One key aspect of the monitoring technology that we are developing for the EMS is that the monitors are agnostic to the SuT architecture and technology (such as programming language) as monitors are only required to inspect event sequences. Also, the monitors execute in an *outline* fashion (outside the SuT and in parallel with it) and *online* (while the SuT runs).

4.3.2 Features

The key features of the EMS are:

- Ability to dynamically, using the REST API, install monitors using a specification language for describing these monitors. We offer different languages with growing expressivity and cost of execution. Also, the ability to inspect and remove installed monitors.
- An event language based on JSON and the beat ecosystem that allows to express the individual elements of trace observations.
- The ability to receive events using different endpoints (HTTP, TCP) for both logs and metrics, from multiple sources (multiple sources within a SuT, plus TSSs and T-Jobs).
- The ability to connect subscribers to the EMS, which will receive a flow of events that the monitors generate as part of the monitoring process. Each subscriber can select from a variety of technologies to receive the events (RabbitMQ, Elasticsearch, Web Socket).
- Provision of a specific endpoint for the T-Job to receive processed events from the SuT.

Monitoring can also be used to aid in the automation of the assessment of performance and load tests. Testing oracles can be defined based on the outcomes of monitors.

4.3.3 Baseline Concepts and Technologies

The main task of the EMS is to process a stream of events and deliver notifications to subscribers. The description of how to process the input event stream is sent to the EMS using the REST API. This description includes (1) how to classify the events from the input stream into "channels" and (2) the monitors that process the event stream by reading from channels, and potentially delivering events into channels. Subscribers also use the REST API to receive events by choosing a channel and a technology to transport those events.

As a technology for the input events, the EMS uses Logstash[48] as the input adapter layer, which allows to receive events from different sources and using different protocols. The events received are organized and stamped as belonging to internal channels. The rules to deliver events to channels are specified as part of the subscription language according to the rules specified by the deployed Stampers and pass them to the main engine that evaluates the monitor, which is written in Go. Afterwards, newly generated events are stamped by the Stamper rules and sent to output (sink) channels. Then another instance of Logstash is used to send these events to subscribed receivers in different protocols.

For the administration of Monitoring Machines and Stampers, an OpenAPI server was developed using go-swagger[49]. As we detail below, this API is used to install and uninstall monitors and to subscribe clients to receive the desired events as the outcome of the monitors. The API has also facilities to check the health of the component, and to flush (reset) the component to its initial state.

4.3.4 Component Architecture

The following diagram shows the different internal elements that form the internal architecture of the EMS, alongside the external components that interact with the EMS directly.

[48] https://www.elastic.co/products/logstash
[49] https://github.com/go-swagger/go-swagger

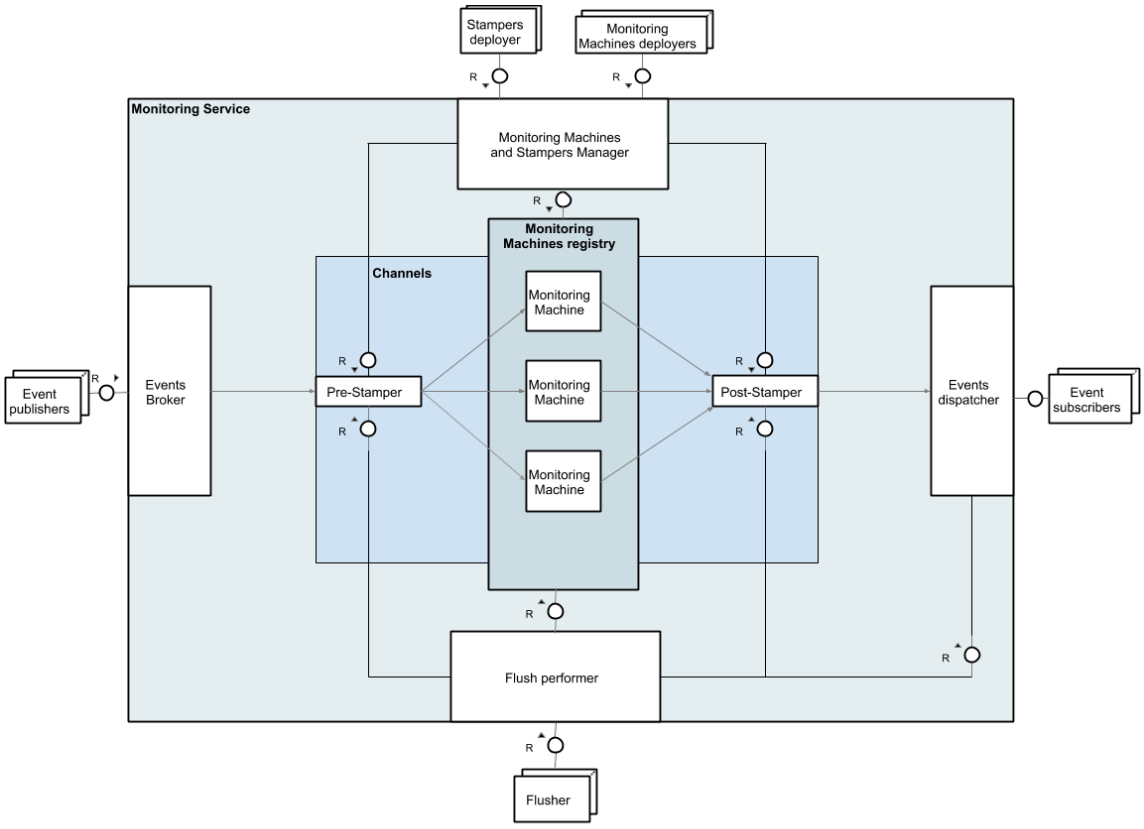

Figure 15 EMS FMC Diagram

We describe of each component shown in the diagram:

- External components
  - Event publishers: these are the external components that send events (logs, metrics, etc) in the form of individual events. Some components that emit these events are:
    - the instrumented SuT so information that may be relevant to a particular test is collected, including logs and metrics.
    - Other ElasTest components in general, who report events to the Monitoring Service. One example are events that mark the beginning and end of different phases of a test, for example mocking an attack or a network or component failure. The purpose of these events is to correlate this phase with the response observed from the SuT. One particular case here is the T-Job itself emitting events to the EMS. The combined stream of events allows to correlate when certain events happen and aid the T-Job in determining the outcome of a test and in guiding the test to better explore a potential bug.
  - Stamper deployer: external components that provide the rules to infer the channels of events. Each event is classified (stamped) according to these rules, where every stamp is called a channel (that is, if an event receives a Stamp, we say that the event is sent to the corresponding channel). One example is a stamper that classifies events as coming from

the SuT, or from a specific component of the SuT, or from the networking load of a specific component of the SuT. Additionally, the Samper deployer also describes rules that describe how to route events between the different channels. These rules simply describe conditions on an event to receive a stamp. For example, the channel #EDS may be configured to receive every event, so the whole test history is recorded, while the channel #Dashboard may be configured to receive only some of the events that are relevant to be displayed for a given test.

- Monitoring Machines deployers: while stampers classify events and allow to route events, the decisions that stamper rules take are simple and only allow the evaluation of each event independently of the history. Monitoring machines, on the other hand, describe temporal and rich temporal and data correlations between the streams of events. The main goal of monitoring machines is to allow TJobs to express -- using the DSL from which the machines are created – the desired sequence of events. The EMS then will evaluate the monitor observe the input stream of events and inform the TJob when the pattern is detected.
- Event subscribers: external components, which are willing to receive events sent over the channels to which they subscribe. In a typical scenario, these include the ETM for dashboarding, the EDS for recording a test and the T-Job for guiding the test dynamically and assessing the outcome of a test.
- Flusher: an external component, which may reset the EMS to its initial state in order to reuse it. The typical scenario here is the ETM that can reuse a TSS after a test for a subsequent test, for better resource utilisation.

- Internal components:
  - Events broker: receives incoming events published by different events publishers (that can use different endpoints) and dispatches them to the stamper. This element is mainly a Logstash instance tailored for the task of receiving these events.
  - Monitoring Machines and Stampers manager: this is a web endpoint developed using swagger and Go, in charge of parsing the DSL languages that describe stampers and monitors, and validating, deploying and undeploying these stampers and monitors in the main engine. Recall that the main engine is the core element of the EMS that evaluates the monitoring machines for each stamped event.
  - Channels: an abstraction used to classify events of the same kind. Channels can be directly specified by the sender or calculated by the stampers rules. Note how there are two phases of stamping: the first phase for incoming events (before the events are processed by the monitors) and the second phase (for newly generated events generated by the monitors).
  - Pre and Post Stamper: these components stamp events according to the definitions deployed by the Stampers deployers.

- Monitoring Machines registry: holds the list of currently deployed Monitoring Machines and updates it accordingly for every insertion and removal of monitors and stampers.
- Monitoring Machines: a “program” (also referred to as a “monitor”) which reads events from certain channels, processes them, and might generate new events as a result. These are stateful entities, so their behaviour may depend on the history of events observed. One example is a machine that tries to observe the behaviour “the bandwidth when no video is played must be below 10% of the bandwidth when the video is played”. Evaluating this behaviour requires observing, computing and remembering average bandwidth and whether videos are being asked to be played. The definition and implementation of the Monitoring Machines is the core concept in the development of the EMS.
- Events dispatcher: the component in charge of feeding events written to output channels to external components that subscribe to these channels. It is mainly a tailored Logstash instance configured to interact appropriately with the supported end-points.
- Flush performer: resets the EMS to its initial state by removing subscribers, stamper and monitoring machines. It is a web endpoint developed using go-swagger.

4.3.4.1 Use Case Diagrams

Broadly speaking, the EMS is used to ease and guide the test of a System Under Test or the test of ElasTest itself.

To accomplish this, the following operations can be invoked during the execution of a T-Job:

1. Event publishing. This is how external components feed input to the ElasTest Monitoring Service, which offers a wide set of endpoints to facilitate the information gathering.
2. Deploy, list and undeploy event Stampers. The T-Job will deploy, update and undeploy rules to stamp and route events through different channels.
3. Deploy, list and undeploy Monitoring Machines. To digest and synthesize information from incoming events, the T-Job will deploy different processors to the EMS. These Monitoring Machines can later be updated or removed.
4. Subscribe to one or many event channels. The T-Job may subscribe itself and other endpoints to receive the events sent to specific channels.
5. Reset the EMS to its initial state. To reuse an EMS after each T-Job, the EMS can be reset to its initial state by removing every deployed Stamper, Monitoring Machine and subscribed endpoint.

As shown in the following chart, all these operations can be invoked by end-to-end tests, typically executed from ElasTest T-Jobs.

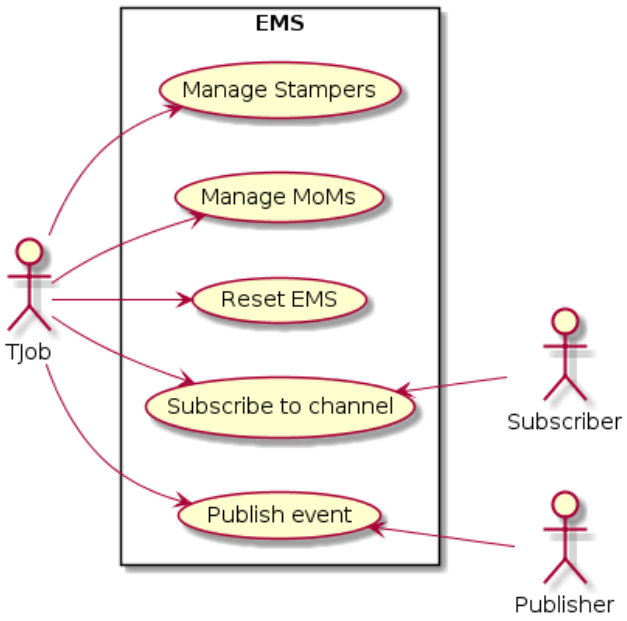

Figure 16 EMS Use Cases

Most of the EMS operations have been specified in OpenAPI and are available as REST methods. The following table provides a summary of such operations. These descriptions can also be found in the ElasTest Monitoring Service API online documentation[50].

Method	URL	Description
1. Event Stamper management		
GET	`/stamper`	Get the list of deployed Stampers.
POST	`/stamper`	Deploy new Stamper.
GET	`/stamper/{StamperId}`	Get the definition of a particular Stamper.
DELETE	`/stamper/{StamperId}`	Undeploy a particular Stamper.
PUT	`/stamper/{StamperId}`	Replace the definition of a particular Stamper.

[50] https://elastest.io/docs/api/ems/

2. Monitor management		
GET	/MonitoringMachine	Get the list of deployed Monitoring Machines.
POST	/MonitoringMachine	Deploy new Monitoring Machine.
GET	/MonitoringMachine/{MoMId}	Get the definition of a particular Monitoring Machine.
DELETE	/MonitoringMachine/{MoMId}	Undeploy a particular Monitoring Machine.
PUT	/MonitoringMachine/{MoMId}	Replace the definition of a particular Monitoring Machine.
3. Event subscription		
POST	/subscriber/elasticsearch	Subscribe an instance of ElasticSearch.
POST	/subscriber/rabbitmq	Subscribe an instance of RabbitMQ.
DELETE	/subscriber/{SubId}	Unsubscribe an endpoint.
4. EMS reset		
POST	/flush	Reset the EMS to its initial state.
5. Service Instance Status		
GET	/health	Get the component health status.

Table 3 EMS API Calls

Currently, the EMS accepts the following endpoints to which the event publishers can send events:

- A TCP endpoint[51] listening on port 5000.
- A TCP endpoint[49] listening on port 5001.
- A Beats endpoint[52] listening on port 5044.
- An HTTP endpoint[53] listening on port 8181.

The list of endpoints available will evolve according to the needs of the rest of the ElasTest components.

4.3.4.2 Sequence Diagrams

We now illustrate using sequence diagrams some of the typical interactions between the components described above:

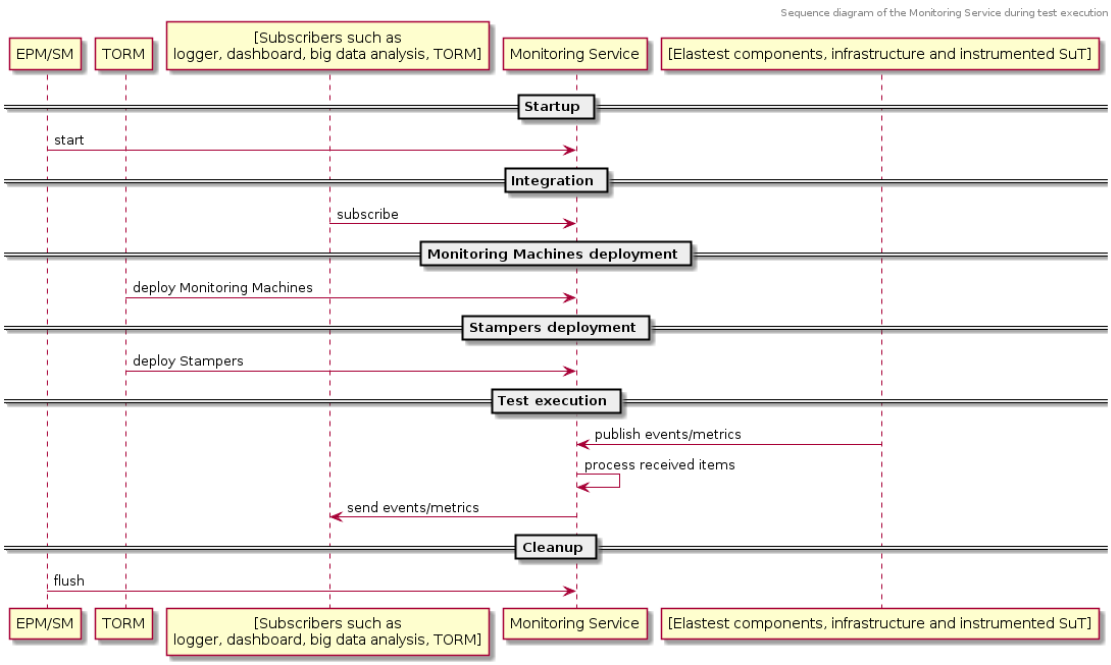

Figure 17 Execution of a Test Sequence Diagram

In this diagram, the EPM/ESM starts the EMS. Then the subscribers indicate the channels they want to listen to, and where (the end-point) and how (the protocol) the output events should be sent to the subscribers. After that, the ETM deploys the Stampers to infer the channel of events and the Monitoring Machines. Then, the test is executed, and publishers start emitting events to the EMS, which in turn processes them using the deployed machines and announcements and sends the outgoing events to the subscribers. For example, the dashboard will display the desired figures and show logs,

[51] http://www.elastic.co/guide/en/logstash/current/plugins-inputs-tcp.html
[52] https://www.elastic.co/guide/en/logstash/current/plugins-inputs-beats.html
[53] https://www.elastic.co/guide/en/logstash/current/plugins-inputs-http.html

the T-Job will receive events with information about how the test is evolving and the EDS will be recording all events. Finally, at the end of the test, the EPM/ESM shall clean up or flush the EMS in order to reset it to its initial state and mark it ready to be reused for another test.

A similar application is shown in the next diagram. In this case, one uses the EMS to debug a component internal to ElasTest. The information that flows through the EMS is now a stream of events sent from the component under debugging (and not from a SuT, as there may not even be a SuT in this scenario). Additionally, information from relevant components is also collected. The engineer debugging the component can review this information and even write programs and monitors to react to this information. This allows to guide the execution to the desired state and to expose the undesired behaviour. The interactions with the EMS are analogous to the previous case.

From the point of view of the EMS, there is no intrinsic difference between the SuT and the infrastructure in which it is deployed (as these are “Event Publishers”). Similarly, there is also no difference between the ElasTest platform and the infrastructure in which it is deployed. For this reason, the EMS can be used to debug ElasTest itself using the same tools as in the testing of third party applications. The main difference lies in the deployed Monitoring Machines, which would focus more on events received from the ElasTest platform, probably ignoring most of those sent by the running “stub application”, which in turn would be specifically designed to stress the ElasTest platform aspects of interest.

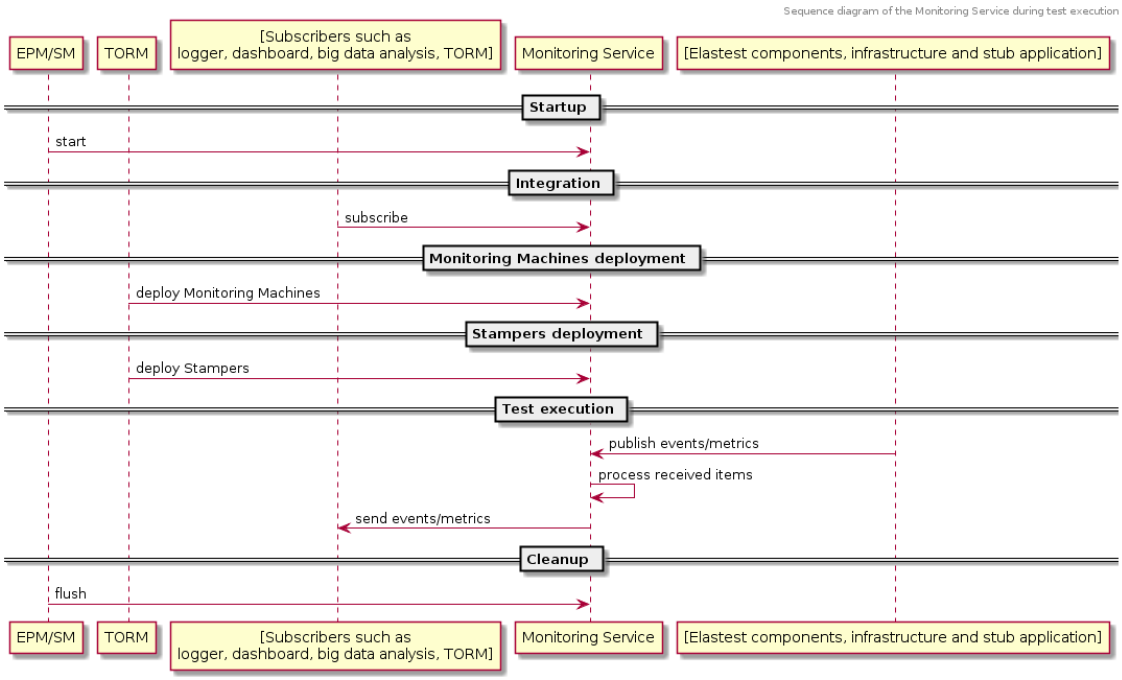

Figure 18 Debugging of the ElasTest Platform Sequence Diagram

We do not show sequence diagrams that illustrate the interaction of each component.

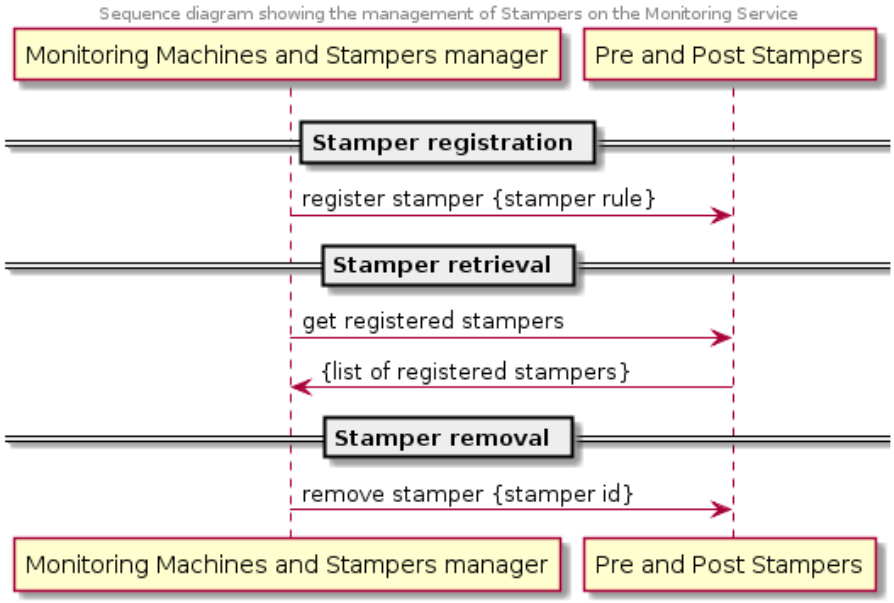

Figure 19 Management of Stampers Sequence Diagram

In the figure above, in its the first interaction, the Monitoring Machines and Stampers manager has already received from the API a request to install a rule and has parsed and checked that the rule is legal. Then, it installs the rule in the Stampers, which immediately start using it to classify incoming events. In the second interaction, the API is used to collect the list of active Stampers. Finally, the last interaction shows how the API can be used to remove undesired rules.

The following sequence diagram shows the analogous interactions of the manager and the Engine in terms of installing, inspecting and removing monitors. The API is used using the analogous entries, and the processes of parsing and validating (though more complex) are completely analogous.

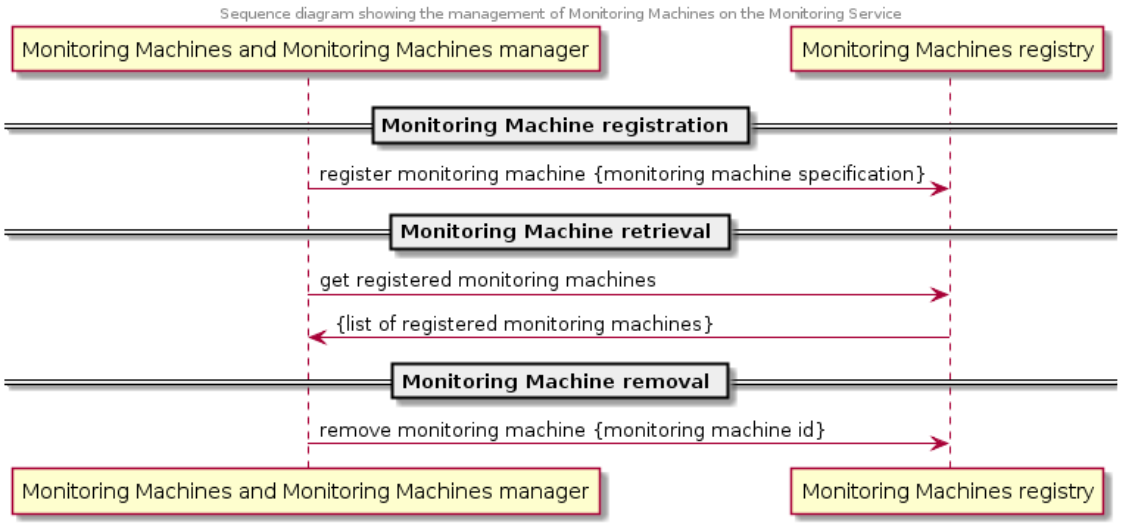

Figure 20 Management of Monitoring Machines Sequence Diagram

The following sequence diagram shows the interactions triggered by an incoming event. This event is received by the Event Broker which, after parsing the JSON, sends it to the Pre-Stamper. This Stamper evaluates the rules and classifies the event by stamping it with the right channel names. Then, the Monitoring Machine evaluates all monitors for which the event is relevant, typically making the monitors change the state. Optionally, the monitors can produce new events which are classified by post-Stampers. Finally, all events (the incoming event and the new events alike) in channels that are relevant for subscribers are sent the corresponding event dispatchers. Note this interaction is a cascade of independent interactions (like a pipeline) and can be executed in parallel. Note also that most of these activities can be parallelized (between different events and between different monitors even for the same event). Note also that, unlike many monitoring tools (for example ElasticSearch or InfluxDB), the events are not saved (but instead the monitors keep an explicit state) and the monitors are activated precisely as a reaction to the arrival of an event, as opposed to periodically evaluated.

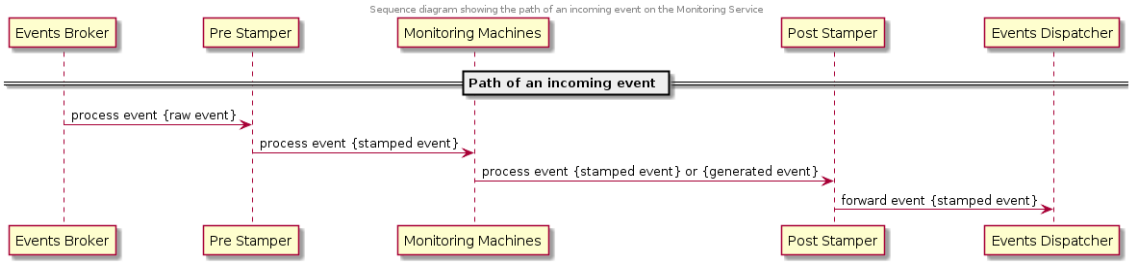

Figure 21 Path of an Incoming Event Sequence Diagram

The following sequence diagram illustrates the activity of resetting the EMS. Here the Flush Performer invokes the Stamper and main engine facilities to remove all engines and their associated state, to move back to the initial state.

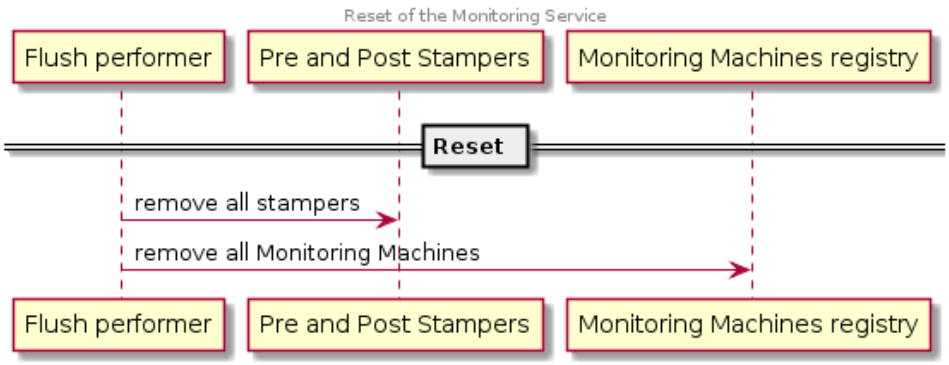

Figure 22 EMS Reset Sequence Diagram

4.3.5 Code Reports

Most of the components of the EMS, including the stampers and the main engine are developed in the Go programming language. The exceptions are third-party software that are configured for the specific needs, namely the Logstash components that serve

as adaptors of incoming and outgoing events. Unit tests are performed using the testing framework of Golang, through the go test[54] command, while the code coverage is calculated using the go cover[55] tool and automatically reported using codecov[56].

External libraries and third-party programs such as Swagger and Logstash are not included in the unit testing.

The current percentage of code covered by unit tests is 41%, even though this figure changes as new tests are added, and the development continues. For example, the figure has decreased recently due to the development of new features, whose unit tests will be developed and included in future releases. The updated code coverage percentage can be checked as a badge in the GitHub repository: https://github.com/elastest/elastest-monitoring-service/

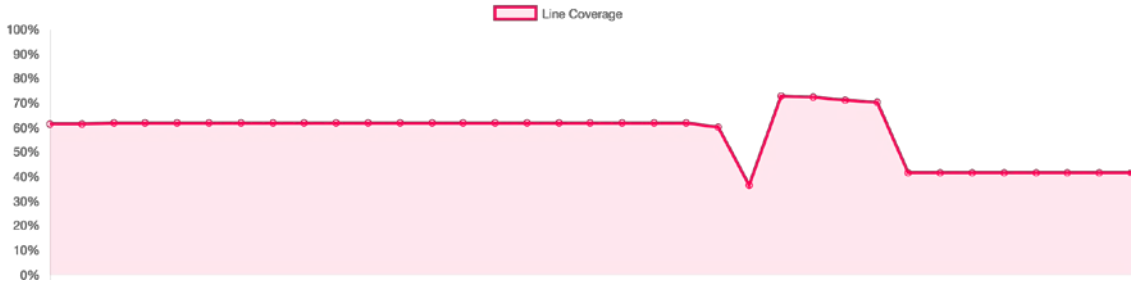

Figure 23 EMS Code Coverage

4.3.6 Code Links

- The main repository of the EMS is https://github.com/elastest/elastest-monitoring-service/
- The API of the EMS is available at http://elastest.io/docs/api/ems/

4.3.7 Contributions

Related to the research results and plans, we plan to:

1. Investigate and design correlation languages to express patterns to observe in the flow of events received by the EMS (coming from the SuT and from ElasTest services) for:
    a. Usability: testers can express the conditions they want to observe to assess the outcome of tests
    b. Testability: testers can express conditions to maximize the likelihood of a test failing or a repeating a failure
    c. Performance: the language can be evaluated in a performant way
2. Evaluate, compare the performance of the languages designed in 1. against a brute-force "store all" and "rich search" infrastructure (e.g. InfluxDB or ElasticSearch).

[54] https://golang.org/cmd/go/#hdr-Test_packages
[55] https://golang.org/cmd/cover/
[56] https://codecov.io/

3. Assess, in synthetic cases (and then in real cases) whether the directed testing enabled by 1.b above can help to hit concurrency bugs in elastic applications. Note that 1.b requires a feedback of events into the T-Job.

This is the central area of research related to the EMS, which involves foundational work on specification languages and evaluation engines for monitoring from runtime verification. We are investigating principles, languages and methods for efficiently evaluating real-time streams and how to provide guarantees in terms of expressivity, parallelism, memory consumption and algorithmic cost of evaluating event streams. We have published a related paper in SAC'18 [EMS1] (April) about monitoring algorithms for asynchronous streams. A new paper is in preparation about a very expressive evaluation engine, with formal correctness proofs and guarantees of resources, that subsumes the SAC'18 work. We will probably submit this work to RV'18 [EMS2].

Future plans include adding features to this language, like parametrization, (we envision a conference submission in early 2019) and a paper on practical empirical evaluation based on ElasTest. We will also prepare a journal version including all main foundational results in a comprehensive way submitted during 2019.

Related problems surface during the development of the project, typically related to the orchestration and resource usage of complex cloud applications like ElasTest. We have two articles that address some of these problems: one short paper in ICWS'18 [EMS3] and one longer paper under review.

4.4 ElasTest Big Data Service

4.4.1 Introduction

The ElasTest Big-Data Service (EBS) is an ElasTest Test Support Service (TSS) that provides an on-demand computing engine based on Apache Spark [EBS0] to be utilized by tests inside ElasTest. The purpose of EBS is to allow tests (T-Jobs) or other components to define their computation requirements using Spark API and use it to perform complex distributed calculations on top of the Spark engine. After completing these calculations, the EBS engine can be safely decommissioned, which allows for a smaller cloud footprint and a potentially dramatic reduction in infrastructure costs.

Currently the component is capable of scaling independently, although the capability of external scaling (i.e. external request) is also possible. It is hence planned for future releases to allow dynamic scaling based on specific test performance requirements.

4.4.2 Features

The current version of ElasTest Big data Service, provides the following features:

- Spark API to launch tasks (using programming language clients / shell).
- Integration with Alluxio in EDM for importing/exporting data to/from HDFS.
- Integration with ElasticSearch in EDM for direct processing of execution logs.

4.4.3 Baseline Concepts and Technologies

The main purpose of EBS is to provide a scalable and disposable parallelized computing engine to any tests or other components that require it. This computing engine is based on Apache Spark, which is a widely adopted distributed processing engine currently available. Spark does not only provide a very fast compute engine, but it can also integrate with a wide variety of data sources, allowing for easier future extensions.

In order for EBS to be disposable, it was designed and provided as a computing engine separate from all data persistence services. This approach allows Spark clusters to be commissioned and decommissioned on demand. A negative side effect of this is that Spark is detached from the Hadoop Distributed File System (HDFS) nodes, which virtually disables data-locality awareness on Spark jobs (a query submitted to the Spark cluster). This, however, is left to the underlying data centre management service (e.g. Kubernetes [EBS1]) to manage, since ElasTest as a whole is distributed in the form of immutable containers. Hence, Spark and Hadoop nodes co-location is managed externally so this should not be considered as a drawback of the chosen architecture. Hadoop was chosen as a data-lake store, in order to provide the most possibly flexible solution in data management. It provides the capability to store raw data in its original format and process it as-is, as well as extend with other technologies (e.g. Spark, Hive) and create a data platform fit for every purpose. Relational uses Hadoop as the core of every provided big-data solution, and we consider it to be the single most stable proposal for the IT industry.

4.4.4 Component Architecture

The following Functional Modelling Diagram depicts the architecture (i.e. all the internal components) of EBS, as well as its interactions with other ElasTest components. In addition, a detailed description of these components and their interactions is provided in order to further clarify the component operation.

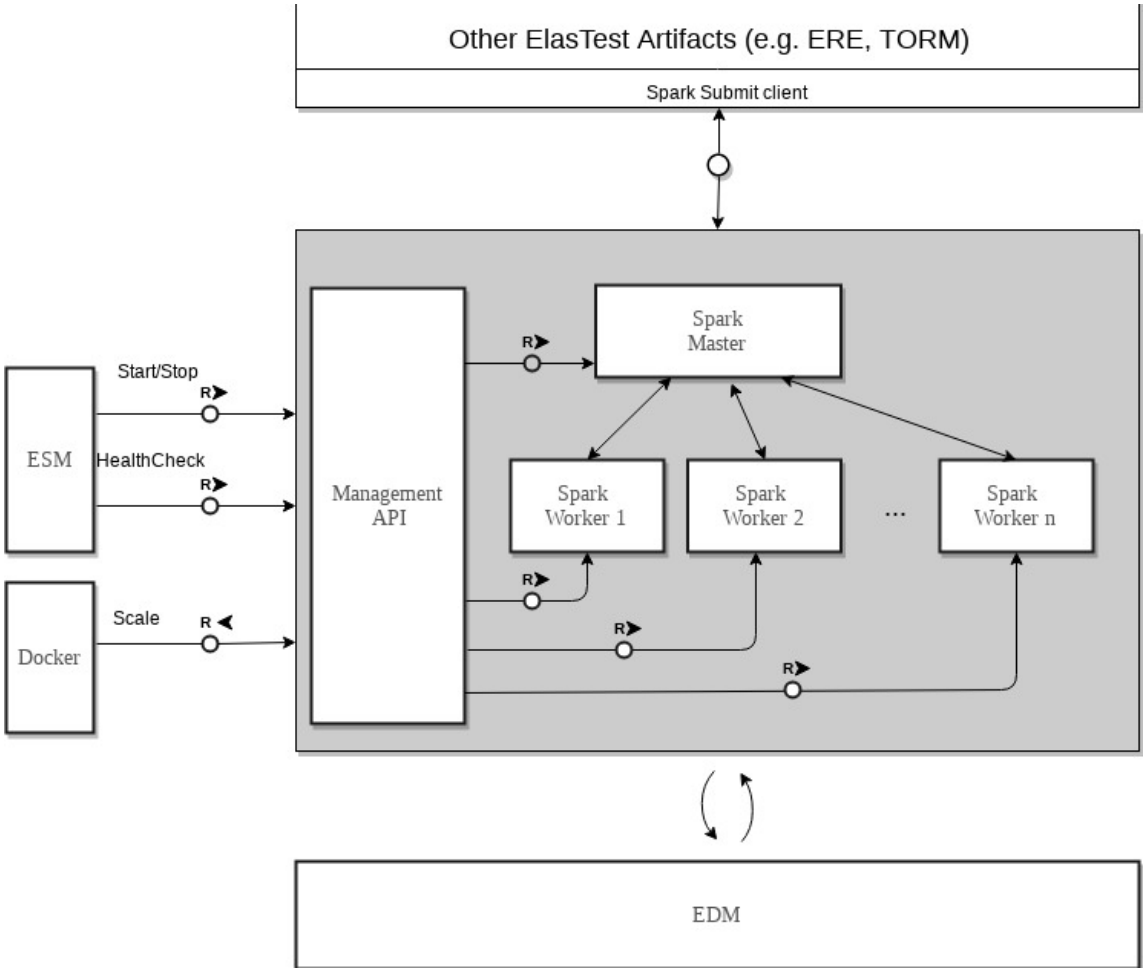

Figure 24 EBS FMC Diagram

Based on the above diagram, a detailed description of each EBS component, as well as the interactions with external components is given:

- Internal Components
  - **Spark Master:** Spark Master is the entry point and central management node of an Apache Spark cluster. Its main role is to maintain a list of alive Worker nodes, split the incoming jobs into distributed tasks and distribute those tasks between the Worker nodes based on several criteria such as data-locality, current workload and available resources. In this specific implementation (stand-alone mode), Spark Master also operates as a job scheduler i.e. it holds the queue of incoming jobs and serves them in a FIFO manner.
    More sophisticated schedulers can also provide job prioritization based on

external criteria. Such implementations were avoided since there is no requirement for job prioritization in the context of ElasTest.

- **Spark Worker:** A Spark Worker node is a simple calculation engine. It simply operates by receiving Tasks from the Spark Master, performing the calculations and returning results to the Spark Master. Spark Workers are generally disposable, in the sense that they can be added/removed to a running cluster during operation, without affecting the Job results. All communication between the Spark cluster and the persistence services (EDM) is done via the Worker nodes, i.e. each node requests the data chunks related to its own Tasks from the corresponding EDM service.
- **Management API:** EBS management API is implemented in Python, using Django [EBS2] as the Framework to implement its REST API. This API is mainly consumed by ESM and is a single-entry point to perform all kinds of operations to the cluster; it performs internal health checks on the whole Spark cluster and replies back to external requests (e.g. ESM) the overall status of the whole service.

- External Components
  - **ESM:** Since EBS is a Test Support Service, it is managed by the Service Manager module (ESM). ESM controls the life cycle of each EBS instance spawned and also holds the health status information of the service. The latter is provided by EBS API and is consumed by ESM for management purposes.
  - **Docker:** This is the Docker API. As every TSS is currently responsible to scale itself, the actual scaling action can be done either via using EPM, or by directly accessing Docker and spinning up the required resources. Currently, EBS is able to scale itself by accessing Docker directly.
  - **Other ElasTest Artifacts:** This is an umbrella term that contains all components of ElasTest that may require a calculation engine for their purposes. As an example, ERE can use Spark to apply data transformations and machine learning algorithms to SuT and T-Job generated data (e.g. logs) and then save the results to EDM for further consumption by ERE internal procedures. Another example is a T-Job (ETM triggered) that could use EBS as a calculation engine to process the T-Job generated logs and end up with a success or failure based on specific patterns found inside those logs.
  - **EDM:** EDM is the ElasTest component that provides all persistence services for the platform. These persistence services (ElasticSearch, HDFS and MySQL) are accessed by EBS as both data sources and data sinks.

4.4.4.1 Use Case Diagrams

As explained, EBS mainly operates as a calculation engine for large amounts of data. In order to utilize EBS during the execution of a T-Job, the submission of a Spark Job to EBS is required. After the Job execution is finished, its exit code can be grabbed and used by

the T-Job and the processed data can be found in EDM. The exact same operation is also true in the case of other ElasTest components using the service.

In addition to the service usage scenarios, the Management API is used by ESM to get a service status. All of the above scenarios are depicted in the following Use Case Diagram.

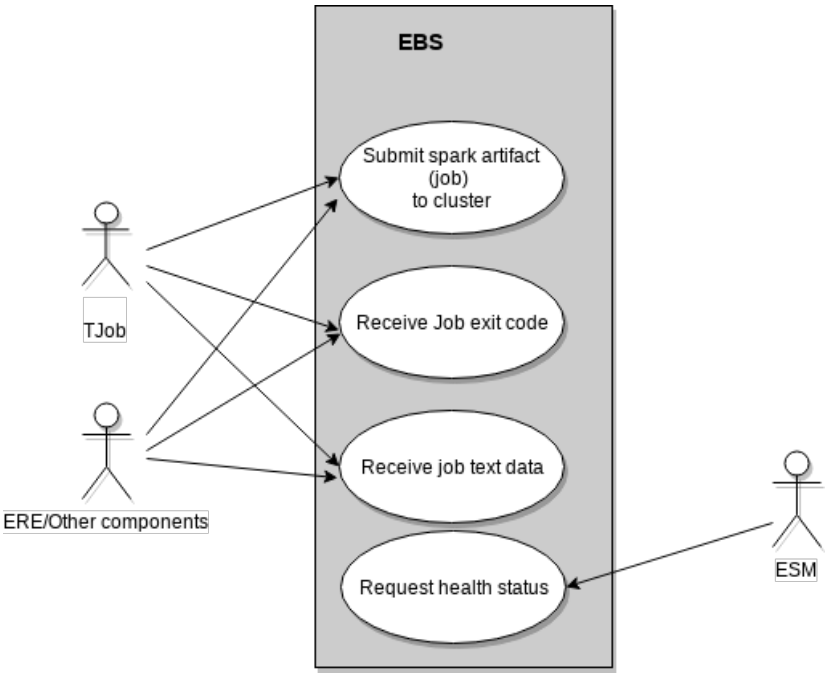

Figure 25 EBS Use Case Diagram

EBS management API methods are defined as REST methods. A Swagger[EBS3] endpoint is also provided. The actual communication with Spark cluster is being done via job submissions.

Method	URL	Description
1. Management API		
GET	/	Swagger UI providing API methods documentation
GET	/environment	Returns detailed information for the whole EBS environment.
GET	/health	Get the component health status.

Table 4 EBS API Calls

Since the communication with the cluster is being done via Spark clients and the Job submission process, there are no actual endpoints for contact with the Spark cluster. In

future releases it is suggested to provide such an API if needed, such as Apache Livy [EBS4].

4.4.4.2 Sequence Diagrams

In this section sequence diagrams are provided, which depict the exact operations performed by EBS and the components that interact directly with it. A thorough description of these actions is provided here as well.

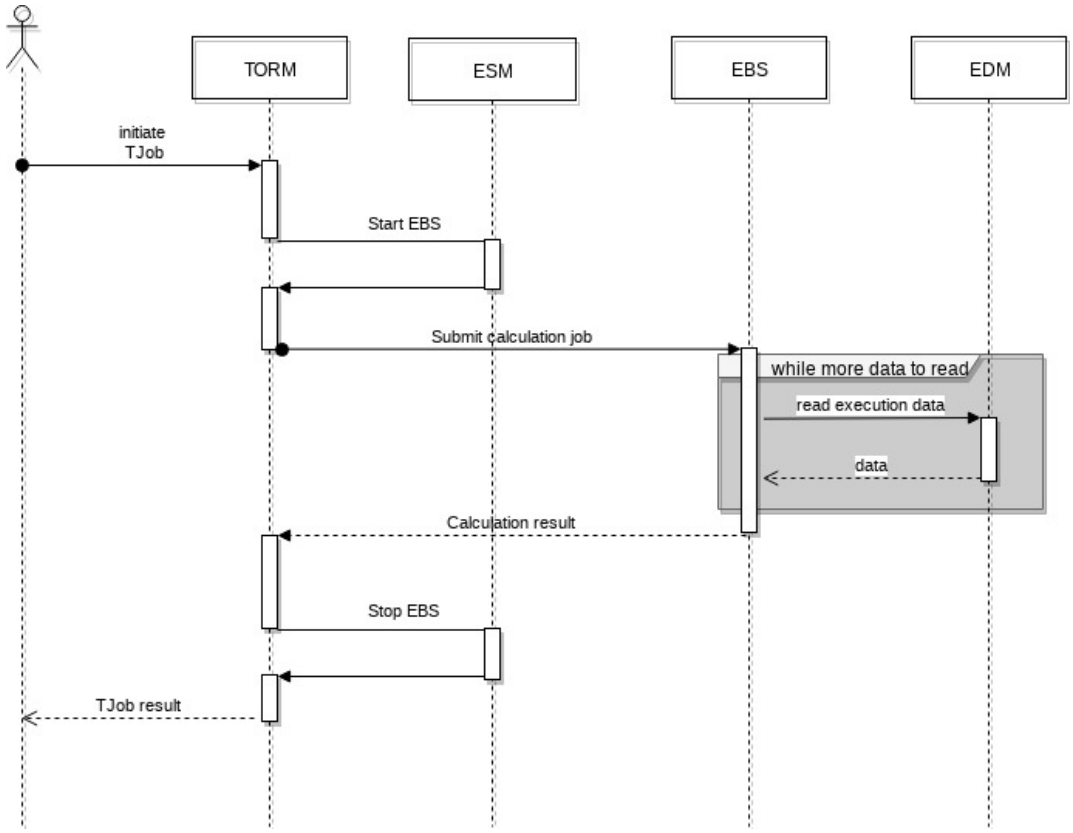

Figure 26 EBS Sequence diagram; Use from a T-Job

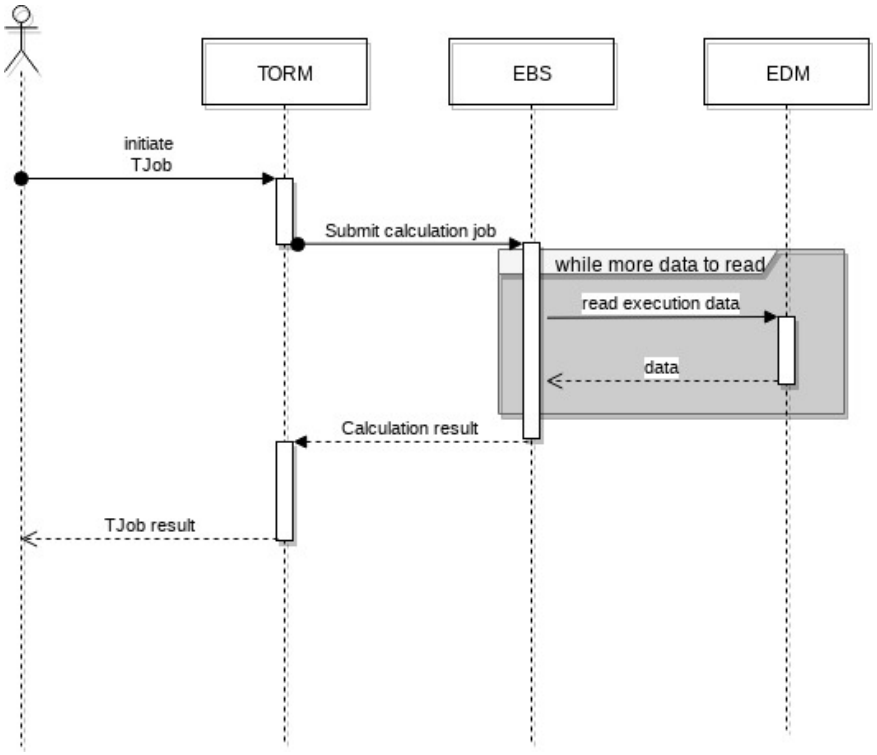

Figure 27 EBS Sequence diagram; Use from a T-Job without ESM

As depicted in the figure above, the actions that take place during a T-Job execution that utilizes EBS are the following:

1. User initiates the T-Job.
2. T-Job requests an EBS instance from ESM.
3. ESM initializes EBS and returns the endpoint to ETM.
4. T-Job (via ETM) performs a spark-submit to EBS.
5. EBS performs the required calculations. During this execution, all available persistence services from EDM (ElasticSearch, HDFS, MySQL) may be used for reading or writing, as per Job configuration.
6. After calculation finishes, the execution stdout and the exit code are returned to ETM.
7. ETM performs a request to ESM, in order to decommission the EBS cluster.
8. After any other remaining tasks, the T-Job returns its result to the user.

Figure 27 above describes the same process, but without an on-demand EBS cluster created during T-Job execution. In this case, the user has requested an EBS cluster to be created by using ETM (the exact workflow is outside the scope of this use case) so the process is exactly the same but skips the steps 2, 3 and 7. The workflow described in this Figure will provide a huge increase in execution speed of a larger group of tests, since there will be no commissioning/decommissioning of clusters between T-Jobs. The drawback of this method is that after all executions have finished, the cluster is still active and consumes resources until the User manually requests a decommission. In this

case the cost of maintaining a cluster in a cloud service will increase depending on EBS total uptime.

4.4.5 Code Reports

The EBS Management API and the component end-to-end tests are written in Python, using several different frameworks. Unit testing is performed by nose framework [EBS5], and code coverage is being reported using codecov. The current coverage report is at 100%, although a more realistic number is around 85%. The difference is due to a set of codecov ignores that are planned to be removed immediately in the next release.

Figure 28 EBS Code Coverage

4.4.6 Code Links

- The EBS code repository can be found on GitHub[57] and is licensed using Apache 2.0.
- Within that repository, there is documentation[58] detailing how to run, use and extend the EBS.
- The API of the EBS can be viewed online here[59].

4.4.7 Contributions

Since EBS is mostly implementation work, our (RELATIONAL) research is targeted at exploiting containerized scalable computing architectures for commercial purposes.

More specifically, demand for deploying applications in external (customer) data centres introduces an increased complexity in describing distributed application prerequisites to infrastructure teams. This complexity is extended by the huge diversity between infrastructure architectures, due to the multitude of selections in that sector; public/private clouds, bare metal systems, distributed operating systems [60] and container orchestration systems [61] are all selections that may be used alone or in different combinations in order to create the best approach for each case.

In this ecosystem, having a scalable distributed computing engine in the form of a single component, able to be deployed in the majority - if not all - of the aforementioned

[57] https://github.com/elastest/elastest-bigdata-service
[58] https://github.com/elastest/elastest-bigdata-service/tree/master/docs
[59] https://elastest.io/docs/api/ebs/
[60] http://mesos.apache.org
[61] Kubernetes and Docker Swarm are two well-known examples.

solutions without maintaining different configurations is a huge key to success for software vendors and integrators. It is therefore our main target to bullet-proof, extend and productize the usage of EBS as a reusable, portable scalable computing engine as a standalone system or a software component of a larger solution.

4.5 ElasTest Security Service

4.5.1 Introduction

The ElasTest Security Service (ESS) is an ElasTest service for security testing cloud-based Web Applications. ESS operates on two different modes: the passive testing mode and the active testing mode. In the passive testing mode, ESS does not interact with the System under Test (SuT). In this mode the HTTP traffic generated by a T-Job associated to the SuT is analysed to identify security vulnerabilities. The second mode is the active testing mode where ESS probes the SuT by generating security tests that mimic the actions of a malicious user on the SuT. Through these two modes of operation, ESS supports the detection of common Web Application security weaknesses and other sophisticated vulnerabilities.

4.5.2 Features

The key features of the ESS are:

- Ability to identify potential security weaknesses in a SuT by:
  - analysing the HTTP traffic generated by a T-Job associated to the SuT
  - probing the SuT by crafting HTTP requests that mimics attacker behaviour
- Ability to detect common Web Application security vulnerabilities such as SQL injection, cross-site scripting etc.
- Ability to detect vulnerabilities in the SuT that facilitates replay attacks and cross-origin attacks
- Generates interactive HTML security test reports for the tester

4.5.3 Baseline Concepts and Technologies

Here we define some key concepts in the ESS.

- **Security Tester:** From the perspective of ESS, the Security Tester is an agent who wants to identify the vulnerabilities contained within an application (i.e., the SuT) via the ESS. The Security Tester can provide T-Jobs that test certain functionalities in the SuT.
- **Security Testing:** From the perspective of ESS, Security Testing is the process of identifying security vulnerabilities in the SuT via T-Jobs and reporting them to the security tester.
- **Security Testing through the ESS:** The security tests performed by the ESS are based on the HTTP traffic generated during the execution of the T-Jobs associated to the test. In order to have visibility of the HTTP traffic of the T-Jobs, the ESS provides a Man-in-the-Middle (MitM) proxy component. For performing a security test using the ESS, the security tester must configure the T-Jobs in such a way that they make all HTTP connections via the ESS MitM. The MitM proxy used by the ESS is based on the OWASP ZAP MitM proxy [ESS1]. When these T-Jobs are executed, all HTTP communication is visible to the ESS via the MitM proxy. The ESS can control and intercept this traffic via the API provided by OWASP ZAP.

The ESS operates in two mode, the passive testing mode and the active testing mode. They are explained below.

1. **Passive Testing mode:** In this mode of operation, the ESS identifies security vulnerabilities in the SuT by simply analysing the HTTP traffic generated upon the execution of a T-Job associated to the SuT. No interaction is required between the ESS and the SuT. An example of a vulnerability that can be identified by the ESS by analysing the HTTP traffic of the SuT is the missing protection of cookies from cross-site-scripting attacks. This protection is enabled by sending a "secure" attribute in the HTTP responses for setting cookies. These insecure cookies can be identified by analysing whether the *secure* attribute is missing in the Set Cookie header in the HTTP responses of the SuT.
2. **Active Testing mode:** In this mode of operation, the ESS identifies security vulnerabilities by probing the SuT. During probing, the ESS sends HTTP requests to the SuT that resemble the actions of a malicious agent. An example of a vulnerability that can be identified by active testing is as follows. Consider the case where the SuT is a shopping cart application that allows shoppers to enter coupon codes to get discounts over the prices of the products. Suppose that there is a vulnerability in this SuT that allows an attacker to enter the same coupon code multiple times to purchase a product for free. This vulnerability can be discovered (via black-box testing) by sending the HTTP request for submitting the coupon code multiple times and checking whether the SuT accepts them.

4.5.4 Component Architecture

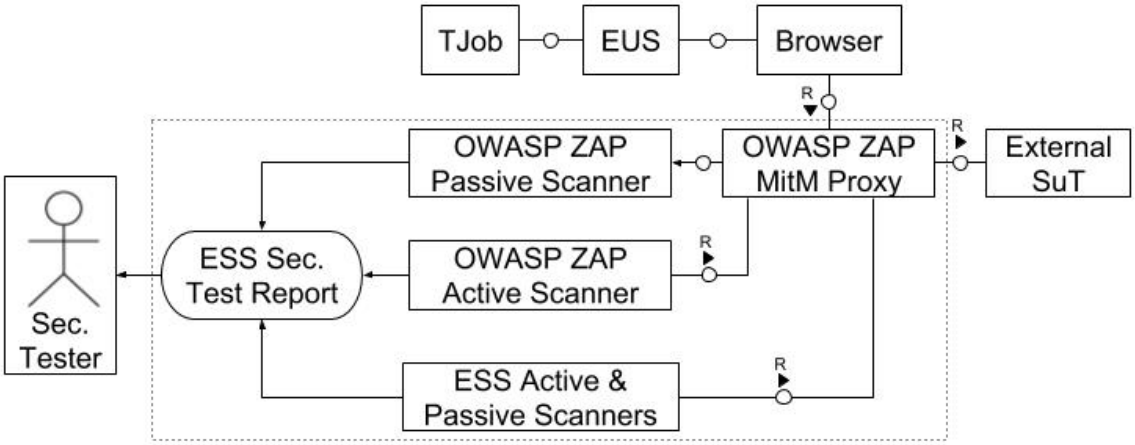

Figure 29 ESS FMC Diagram

The architecture of ESS is shown in the figure above. The ESS components are enclosed within the dotted lines. The explanation is as follows:

1. When a T-Job associated to the SuT is executed, as it is configured to open a Web Browser (via EUS) and proxy its traffic via the ZAP MitM proxy (within ESS), all the HTTP communication is visible to the ESS.
2. The ZAP passive scanner will analyse this HTTP traffic passively to identify security vulnerabilities.

3. The ZAP active scanner will probe the SuT (again via the ZAP MitM proxy) for actively detecting common security vulnerabilities.
4. The active and passive scanners within the ESS will also test the SuT via the ZAP MitM proxy to detect replay attacks and cross-origin vulnerabilities.
5. Finally, all the detected vulnerabilities are stored in a single security test report that can be accessed by the security tester.

4.5.4.1 Use Case Diagrams

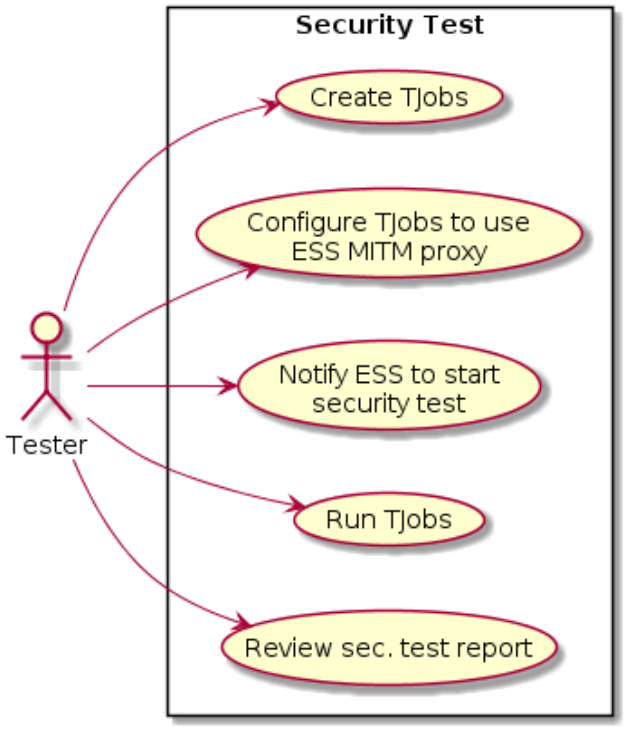

Figure 30 ESS-based Security Test Use Case

The use case diagram of the ESS-based security testing is shown above. The explanation is as follows. The security tester (shown as Tester in the figure) creates T-Jobs and configures them to use the ESS MitM proxy in their HTTP communications. The tester should also have the responsibility to notify the ESS to start the security tests because ESS cannot know by itself when the T-Job has finished executing all the HTTP-related actions. The tester runs the T-Jobs configured with ESS and finally after the ESS generates the security test reports, the tester must review them to identify false positives, come up with mitigations for the true positives etc.

4.5.4.2 Sequence Diagrams

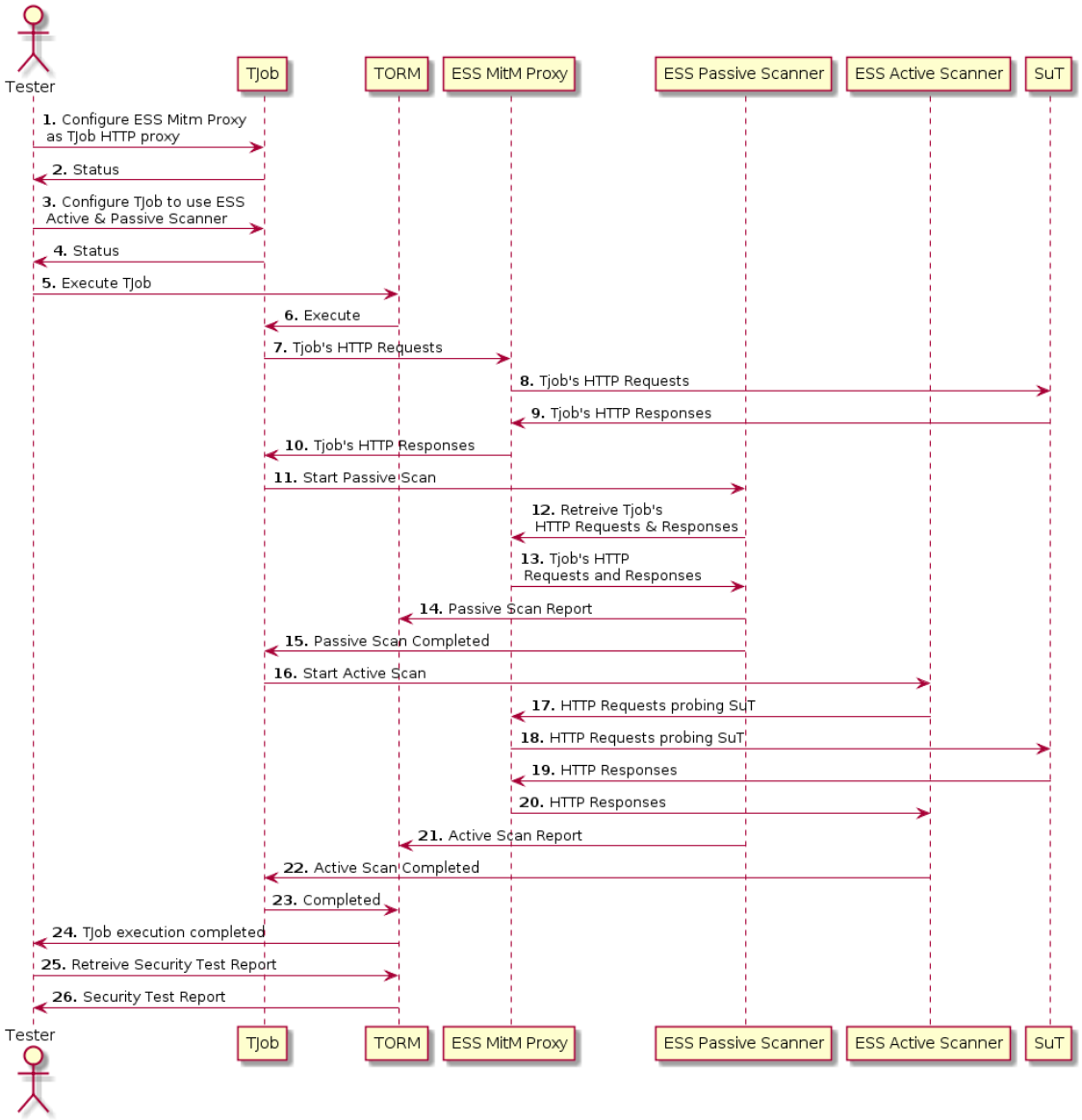

Figure 31 Sequence Diagram of an ESS-based Security Test

The sequence diagram of an ESS-based security test is shown above. The explanation is as follows:

- **Steps 1 to 4:** Before starting the test, the tester must create a T-Job that is configured to make HTTP connections via the ESS MitM proxy and the tester should also configure the T-Job to call the ESS active and passive scanners to perform the security tests.
- **Steps 5 to 10:** After the T-Job has been configured, the tester executes the T-Job via the TORM (ETM). All HTTP communications from the T-Job are logged and proxied via the ESS MitM proxy.

- **Steps 11 to 15:** the T-Job calls the ESS API to start the passive scanner (see Table 5) and the passive scanner collects the HTTP traffic from the ESS MitM for inspection and the passive scan report is sent to ETM.
- **Steps 16 to 22:** the T-Job calls the ESS API to start the active scanner (see Table 5) and the active scanner probes the SuT for vulnerabilities and the active scan report is sent to ETM.
- **Steps 23 to 26:** The end of the execution of the T-Job is notified to both the ETM and the tester and the tester retrieves the reports of the security tests from the ETM.

The following are the API calls of ESS.

Method	URL	Request Parameters	Body	Description
1. API calls for starting, stopping and retrieving the status of the security scan				

POST	`/scan/start`	`time`
		`zapactive`
		`zappassive`
		`essactive`
		`esspassive`
				Start the scan

GET	`/scan/status`			Return the progress of the scan
POST	`/scan/stop`	`consent`		Stop the scan
2. Retrieve the security scan reports				
GET	`/scan/report/html`			Return scan report in HTML format
GET	`/scan/report/json`			Return scan report in JSON format
3. Service Instance Status				

GET	/health	Get the component health status.

Table 5 ESS API Calls

4.5.5 Code Reports

The back-end code of the ESS is written using Python and the front-end is developed using JQuery, JavaScript, HTML and CSS (with Materialize CSS as the basis). End-to-end tests are available for automatically testing whether new changes break the integration with other components. ESS unit tests [62] are written using the Python unit test framework. The details about the latest code coverage are available codecov [63]. As shown in Figure 31, the current code coverage is 28%. This low coverage is mainly due to the recent design changes to ESS. As part of these changes, we have been removing many existing ESS API calls and introducing new ones. Once this process is finished, the new unit tests will be made available and the code coverage will further improve.

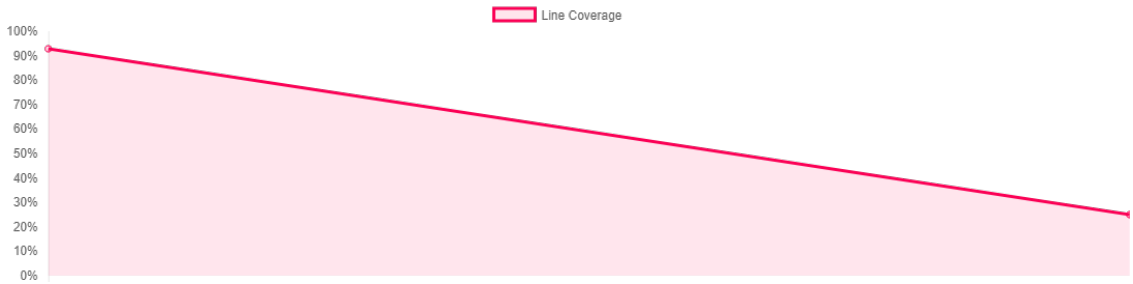

Figure 31 ESS Code Coverage

4.5.6 Code Links

- The link to the ESS code is repository is https://github.com/elastest/elastest-security-service
- The API documentation is available at https://elastest.io/docs/api/ess/

4.5.7 Contributions

Most of the research in the context of ESS is happening within the topic black-box security testing. In particular, currently we are involved in a research for detecting login oracle attacks in Web Applications. The paper is work-in-progress and the target is to submit a paper at the 2019 Network and Distributed Systems Security Symposium[64]. Since it is a prestigious conference, the amount and quality of research results that must be included within the paper is high. Hence, instead of disseminating the work as small papers, we are disseminating the work as a large single paper.

As part of the same research, we are also testing the security of Web Browsers. As we need to test many Web Browsers of different versions, we are planning to use the EUS

[62] https://github.com/elastest/elastest-security-service/blob/master/test_ess.py
[63] https://codecov.io/gh/elastest/elastest-security-service
[64] https://www.ndss-symposium.org

for our experiments. If we find interesting results within the Web Browser domain, we will split the paper into two and disseminate at two different venues.

We found that certain concepts from the domain of program slicing can be used to detect replay attacks in e-commerce applications. With more experimental evidence we are planning to publish a short conference paper on this.

5 Conclusions

A large amount of initial research and implementation has happened in the first half of WP5. This has resulted in an integrated plaform including the support services that adhere to a common design, implementation and deployment approach. Along with this, research activities have taken place including early stage publications. Now with a platform and services that are self-supporting, further feature additions can be added and research can be conducted upon it. Specifically, per service we conclude:

- **EUS**: ElasTest User Impersonation Service enables the impersonation of end-users in their tests through GUI instrumentation. This service provides full compatibility with external browser drivers, but enhanced with extra capabilities, such as event subscription, log gathering, or advance media capabilities for WebRTC applications. This service has been built extending the W3C WebDriver specification, and therefore, popular technologies such as Selenium and Appium are completely compatible with ElasTest. At the moment of this writing, ElasTest is still under development. Therefore, some features are still not available. For instance, the measurement of the end-users' perceived QoE is still ongoing. Measuring QoE is in general a complex topic and this task shall perform the appropriate research activities for evaluating the most suitable way of doing it, which may involve simple mechanisms such as evaluation of response-time from the GUI.

- **EDS**: The ElasTest Device Emulator Service is available with ElasTest as a TSS. In the present release, it is possible to define a T-Job and execute it against SuT. The application running in SuT or T-Job is able to communicate with EDS to request the devices. The applications use the OpenMTC to communicate with EDS. This is planned to be changed for future release. By providing a wrapper function it would be possible to write an application with minimal knowledge of OpenMTC. EDS is integrated with ESM and therefore available to be deployed from T-Jobs defined at the level of TORM. A main of objective of EDS was to enable rapid prototyping of IoT applications with emulated devices, which is currently possible. Furthermore, EDS is one of the free and open source device emulators integrated into ElasTest. EDS brings OpenMTC to facilitate testing of IIoT applications in particular.

- **EMS**: The ElasTest Monitoring Service is available in ElasTest as a test support service. The main functionality of the EMS is to aid in the design of complex tests by providing languages that automate the correlation of streams of events (observations) from the SuT and from the T-Job. The EMS can serve to design simpler and more effective T-Jobs as the outcome of the EMS is a processed stream of events that can determine the outcome of the test directly. Moreover, the EMS can also be used by the T-Job to guide the test online, based on the execution observed. The second functionality is to route some of the events to the ElasTest dashboard, and to the permanent storage to be analysed offline after the testing phase. The EMS is ready to receive events that contain metrics

(from the SuT), logs and special purpose events emitted by the T-Job and other TSS that serve to demark different phases of the test. In its simpler usage, the EMS can be instructed to only filter interesting events, but more sophisticated cases of correlations with temporal meaning can be defined (for example: the bandwidth under an attack must be no larger than 10% more than the bandwidth under normal usage). The domain specific language offered to program the EMS can greatly simplify test designs. During the rest of the project we will be extending the DSL that the EMS offers and validate these extensions against more and more complex tests. Currently, the EMS uses Logstash as an interface to receive events as input, and another instance of Logstash to interface the output events with the T-Job, the dashboard and the EDS. Other technologies are possible, and we will evaluate the usage of resources that Logstash entails and possible remedies to alleviate its high resource consumption (if relevant). The future work includes providing more device emulators into ElasTest as well as making it easy for a new user to use EDS to write IIoT applications and subsequently test them.

- **EBS:** ElasTest Big-data Service is an ElasTest service that provides a computing engine to tests run on ElasTest, as well as other components. This will allow tests to perform complex calculations in big datasets and have a 'green light' approach to the calculation results, thus abstracting some complexity from the test definition. Additionally, EBS can be used to transform and manipulate data stored persistently in ElasTest, providing even more flexibility to the platform. Future functionality will also allow to submit applications using a REST API, in order to completely create a self-contained calculation engine, without any need for client libraries. This in combination with the ability to deploy to any underlying infrastructure, will generate a very interesting piece of software and a very powerful extension of ElasTest when it comes to large scale data processing.

- **ESS:** The ElasTest Security Service provides options for security testing cloud-based Web applications. The main advantage of using ESS over other security testing tools is that it supports the detection of both common Web application weaknesses (e.g., cross-site scripting and SQL injection) and complex vulnerabilities such as cross-origin vulnerabilities and replay vulnerabilities. ESS operates on two different modes: the passive testing mode and active testing mode. In passive testing mode, for identifying security vulnerabilities, ESS depends solely on the HTTP traffic generated by a T-Job. In active testing mode, ESS probes the SuT by mimicking the behaviors of attackers. Currently ESS integrates OWASP ZAP (a prominent, open-source penetration testing tool) for detecting common Web application weaknesses and detects four different cookie-based vulnerabilities. For the future releases of ESS, we plan to add support for detecting complex cross-origin attacks (e.g., cross-site script inclusion) and replay attacks. The research outcomes of these efforts will be published at top security conferences.

6 Appendix

6.1 References

[TSS1] J. Gilbert, Cloud Native Development Patterns and Best Practices. Packt Publishers, 2018.

[TSS2] B. Sousa, L. Cordeiro, P. Simoes, A. Edmonds, S. Ruiz, G. A. Carella, M. Corici, N. Nikaein, A. S. Gomes, E. Schiller, T. Braun, and T. M. Bohnert, "Toward a Fully Cloudified Mobile Network Infrastructure," IEEE Trans. Netw. Serv. Manage., vol. 13, no. 3, pp. 547–563, Aug. 2016.

[TSS3] Erl, Thomas, Service-Oriented Architecture: Concepts, Technology, and Design. Service Oriented Computing Series, 2005, Prentice Hall. ISBN 0-13-142898-5.

[TSS4] OASIS. (2012). Reference Architecture Foundation for Service Oriented Architecture (Version 1.0. utg.).

[EUS0] Möller, S. and Raake, A. eds., 2014. Quality of experience: advanced concepts, applications and methods. Springer.

[EUS1] OpenAPI initiative. https://www.openapis.org/

[EUS2] Selenium framework. https://www.seleniumhq.org/

[EUS3] Appium framework. http://appium.io/

[EUS4] Docker-Selenium. https://github.com/SeleniumHQ/docker-selenium

[EUS5] Selenoid. https://aerokube.com/selenoid/latest/

[EUS6] noVNC (VNC client using HTML5). http://novnc.com/

[EUS7] FFmpeg. https://www.ffmpeg.org/

[EUS8] W3C WebDriver Recommendation. https://www.w3.org/TR/webdriver/

[EUS9] JUnit 5. https://junit.org/junit5/docs/current/user-guide/

[EUS10] Mockito framework. http://site.mockito.org/

[EUS11] Spring framework. https://spring.io/

[ESS1] OWASP ZAP
https://www.owasp.org/index.php/OWASP_Zed_Attack_Proxy_Project

[EBS0] https://spark.apache.org

[EBS1] https://kubernetes.io

[EBS2] https://www.djangoproject.com

[EBS3] https://swagger.io

[EBS4] https://livy.incubator.apache.org

[EBS5] http://nose.readthedocs.io/en/latest/

[EMS1] https://www.sigapp.org/sac/sac2018/

[EMS2] https://rv2018.isp.uni-luebeck.de

[EMS3] http://www.icws.org/2018/

[EDS1] Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016 (URL: https://www.gartner.com/newsroom/id/3598917) accessed:25.06.2018

[EDS2] Brady, Shane & Hava, Adriana & Perry, P & Murphy, John & Magoni, Damien & Portillo, Omar. (2017). Towards an Emulated IoT Test Environment for Anomaly Detection using NEMU. 10.1109/GIOTS.2017.8016222.