D7.3	
Version	1.0
Author	CNR
Dissemination	PU
Date	30-12-2019
Status	FINAL

D7.3 Public demonstrator artifacts

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	WP7
WP leader	Antonia Bertolino (CNR)
Deliverable nature	Other
Lead editor	Antonia Bertolino, Eda Marchetti (CNR)
Planned delivery date	31-12-2019
Actual delivery date	30-12-2019
Keywords	Open source software, cloud computing, software engineering, operating systems, computer languages, software design & development

Funded by the European Union

License

Contributors

Name	Affiliation
Zahoor Ahmed	TUB
Antonia Bertolino	CNR
Antonello Calabró	CNR
Varun Gowtham	TUB
Eda Marchetti	CNR
Guiomar Tuñón	Naeva Tec
Marisol Prieto	ATOS WORLDLINE

Version history

Version	Date	Author(s)	Description of changes
0.1	30/09/2019	CNR	Proposed structure of document
0.2	09/10/2019	CNR	More detailed TOC
0.3	02/12/2019	Naeva Tec	Inputs for FullTeaching demonstrators
0.4	09/12/2019	TUB	Inputs for TUB demonstrators
0.5	14/12/2019	CNR	Editing and revision
0.6	16/12/2019	Naeva Tec	More inputs for FullTeaching demonstrators
0.7	16/12/2019	CNR	More editing
0.8	19/12/2019	ATOS WL	Internal review
1.0	30/12/2019	CNR	Final revision

Table of contents

List of figures

Glossary of acronyms

Acronym	Definition
API	Application Programming Interface
AWS	Amazon Web Services
CCS	Comparative Case Study
CI	Continuous Integration
DoA	Description of Action
EBS	ElasTest Big data Service
ECE	ElasTest Cost Engine
EDM	ElasTest Data Manager
EDS	ElasTest Device Emulator Service
EMS	ElasTest Monitoring Service
EOE	ElasTest Orchestration Engine
ERE	ElasTest Recommendation Engine
ESM	ElasTest Service Manager
ESS	ElasTest Security Service
ETM	ElasTest Tests Manager
EUS	ElasTest User Impersonation Service
GUI	Graphical User Interface
GDPR	General Data Protection Regulation
IaaS	Infrastructure as a Service
IIoT	Industrial Internet of Things
JSON	JavaScript Object Notation
MBaas	Mobility Backend as a Service
npm	Node Package Manager
OIF	OpenIoTFog
POC	Proof of Concept
QE	Quasi Experiment
QoE	Quality of Experience
REST	REpresentational State Transfer
SiL	System in the Large
SPA	Single Page Application
SUT	System Under Test
TE	Test Engine
TiL	Test in the Large

TJob	Testing Job
TSS	Test Support Service
WP	Work Package
WE	With ElasTest
WO	Without ElasTest

1. Executive summary

This deliverable belongs to the WP7 workpackage that performs the validation of the ElasTest platform. In particular, the methodology and the metrics defined for the validation, as well as the results observed, are described in detail in the companion deliverable D7.2. This document describes some of the demonstrators developed in the project and used for the validation studies.

As explained in the project DoA, the platform targets different applications domains. For this reason, the validation covered several demonstrators belonging to different domains, namely:

- telecommunication infrastructures and networks (Task 7.2),
- WWW and mobile applications (Task 7.3),
- smart environments and Internet of Things (Task 7.4), and
- multimedia communication (Task 7.5).

For the first two domains in the list, the adopted demonstrators are private and are described in a companion deliverable D7.4 that is private in nature.

This specific document is public and reports the description of the public demonstrators artifacts corresponding to the third and fourth domains, provided respectively by the TUB and the Naeva Tec partners.

As described in D7.2, we conducted on such demonstrators several empirical studies, including Quasi-Experiments and Comparative Case Studies. In the remainder of this document, for each of the two partners we present the provided artifacts for the respective studies. Specifically, for each artifact we provide: i) a description of the functionalities and architectures; ii) (when applicable) a pointer to the repository where the application can be found; iii) an illustration of the test strategy followed, as well as of the developed test suites, and finally iv) a genuine feedback by the partner on perceived advantages/difficulties in using ElasTest.

Chapter 2 includes the contribution by Naeva Tec, spanning over the FullTeaching application, used for the QE study, and the OpenVidu application, used in a QE specifically conceived for the QoE validation study.

Chapter 3 includes the contribution by TUB, which used an IIoT application on top of OpenMTC middleware for the QE studies, and the EU project EMBERS for the CCS.

2. Naeva Tec artifacts

2.1. The Full Teaching demonstrator

FullTeaching is a proof of concept of a Web application for making online lessons easy for students and teachers. FullTeaching makes use of different OpenVidu components. OpenVidu provides capabilities for Real-Time multimedia communication (in video-conferences) with special requirements of asymmetric communication, where a teacher must have certain level of administration features that students do not have.

Figure 1: FullTeaching web site

Whereas FullTeaching is a full and working software system, it is still considered here as a proof of concept, as it may need some rework in order to be used in high demanding production environments. For instance, scenarios in which more than 20 users in the same "class" may be active concurrently and more than 4 sessions may also be running at the same time. Also currently the login functionality is very simplistic and the personalization options are quite basic. More importantly, to comply with the GDPR, disclaimer and privacy policies have still to be addressed. Therefore, at the moment it is valid for demonstrating features and validating processes and tools, but it is not prepared to run in production environments.

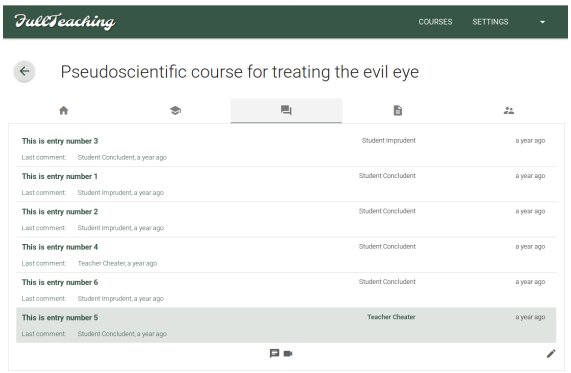


Figure 2: Pseudo course

2.1.1. Features & Requirements

FullTeaching covers the two traditional involved roles of a teacher and a student. These are well defined and delimited as follows:

- **Teacher:** This is the responsible for creation, update and deletion of courses and resources. The teacher is also the manager in a video-conference (or video-lesson); they will manage the access and the actions made available for students during the video-lesson.
- **Students:** They are the consumers of the contents and will be able to access both the resources from the courses they are enrolled in, and the video-conferences. During the video-conferences in order to actively participate they

should “raise the hand” to participate as publishers: e.g., when the teacher asks a question, a student may raise his/her hand and the teacher can allow them to join as presenter/co-presenter in the session until the teacher decides to stop him/her.

2.1.1.1. Communication mechanisms

The FullTeaching application is designed to support different ways of communication that may occur naturally during the learning process:

- Asynchronous communication:
  - Documents: lessons, exercises, tutorials, references, etc.
  - Forums and video forums.
- Synchronous communication:
  - Written: chats.
  - Multimedia: video/audio conferences (one to one or multi-conferences)

2.1.1.1.1. Asynchronous communication: forums

Activation / De-activation

The FullTeaching forums are linked to each of the courses registered in the application. The teacher that manages each course can activate or deactivate it, so he/she and the students can leave comments. When a forum is deactivated all the messages that were previously written won’t be deleted, they will just become inaccessible. In this way, if the forum is reactivated all the comments can be shown again.

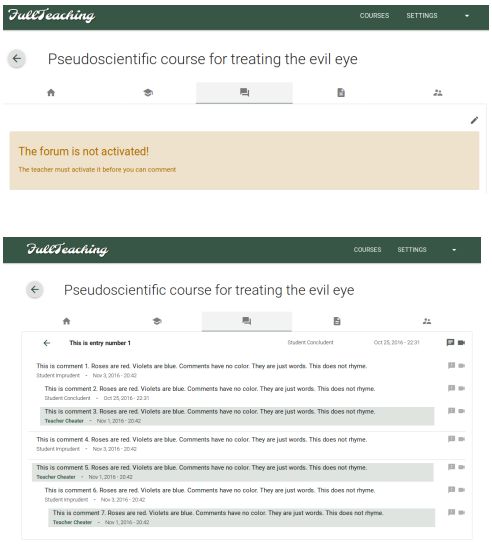


Figure 3: Forum activation and de-activation

Messages (Entries and comments)

The messages will be sorted as a tree hierarchy. Each conversation linked to the course (also called entry) will appear in the main list of the forums tab. Each of these entries may have nested comments that will be shown when the list item is accessed.

Teacher comments are highlighted, to help students to find answers and requests from the teacher.

In the FullTeaching forums it is possible not only to find written comments or entries, but also video comments, entries and screen records.

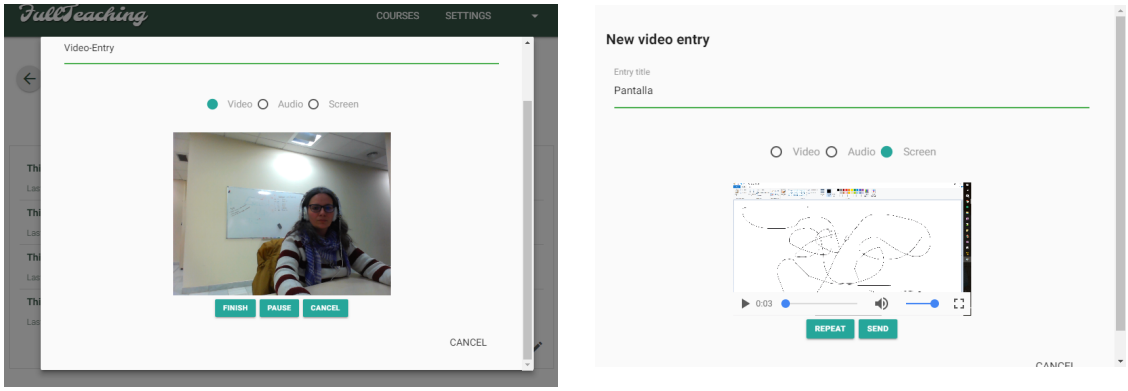


Figure 4: Video entries

2.1.1.1.2. Real-time communication (Synchronous): video and chat sessions

Chat

The video sessions allow the communication also through an embedded chat between all the participants. This chat it is a must for all video-conference applications, to communicate any incidence, make little annotations and share resources. In this chat window the connection and disconnection events will also appear.

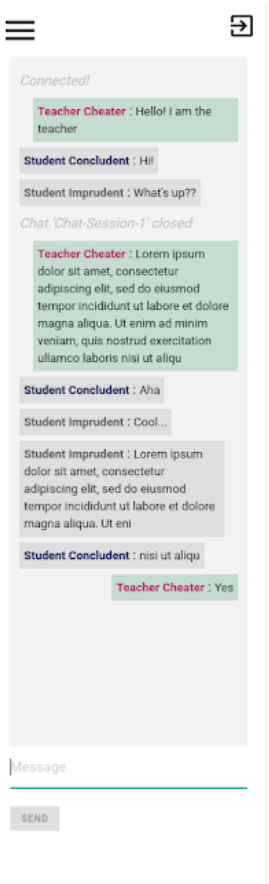


Figure 5: Chat example

Video-conference / video-session

The kind of video-conference proposed for FullTeaching is mainly a one-to-many, i.e. by default only the teacher publishes media while all the students are the recipients. But while this is the main functionality, there is also the option where a student can request the publisher role, and the teacher then may approve his/her intervention. In this case both media flows between student and teacher will be published.

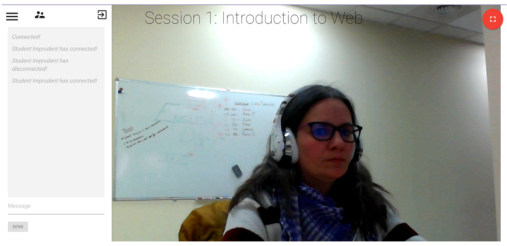
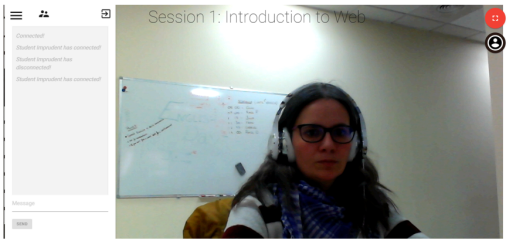
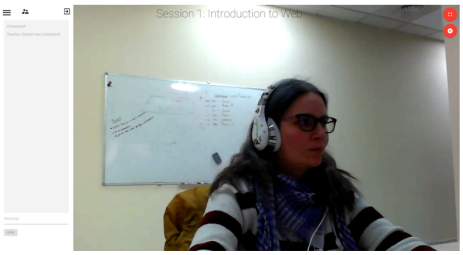
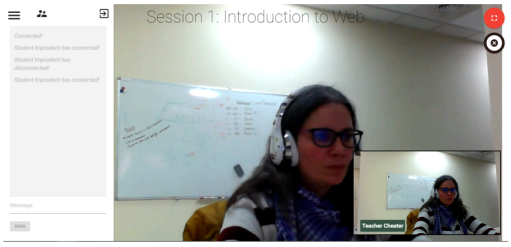
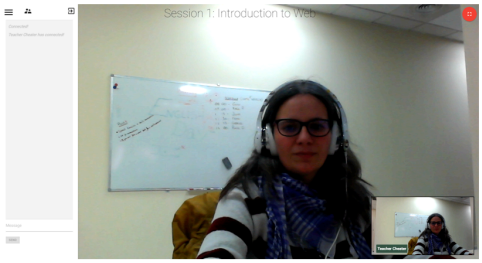
Teacher screen	Student screen
1. Start video session	
	 The student has a button to request intervention.
2. The student request intervention	
 In the teacher screen a button to grant the intervention appears.	 In the student screen a button for cancelling the request appears.
3. Intervention is approved by the user	
 During the intervention the teacher has a button to finish the intervention	

Figure 6: A video session example

2.1.1.2. Requirements

The requirements implemented in the FullTeaching demonstrator are the following ones:

2.1.1.2.1. General Requirements

Requirement ID	Description
GR01	All the users will see all the courses to which they participate
GR02	All the users will be able to check their personal data
GR03	Al the users will be able to modify their profiles -name/nick, email, password, picture, etc…-
GR04	All the users in a course will be able to:
GR04a	See all the resources associated to the course –forum, files, sessions, …-
GR04b	If the forum enables to participate in the forum
GR04c	Access to the scheduled video sessions

2.1.1.2.2. Teacher Requirements

Requirement ID	Description
TR01	The teacher will be able to create new courses with a name and a picture
TR02	The teacher will be able to create and schedule a new video session with name, description and date
TR03	The teacher will be able to activate or deactivate the course forum
TR04	The teacher will be able to publish entries and comments in the forum
TR05	The teacher will be able to upload, update and delete files in the course
TR06	The teacher will be able to add students to the course
TR07	The teacher will be able to initiate a video session
TR07a	The teacher will be the first publisher (only his/her video and

	audio stream will be shown)
TR07b	The teacher will be able to manage the session and the students interventions.
TR07c	The teacher will be able to grant and revoke publisher privileges to students
TR07d	The teacher will be able to mute his/her own audio
TR07e	The teacher will be able to stop sending video
TR07f	The teacher will be able to comment in the chat associated to the video session

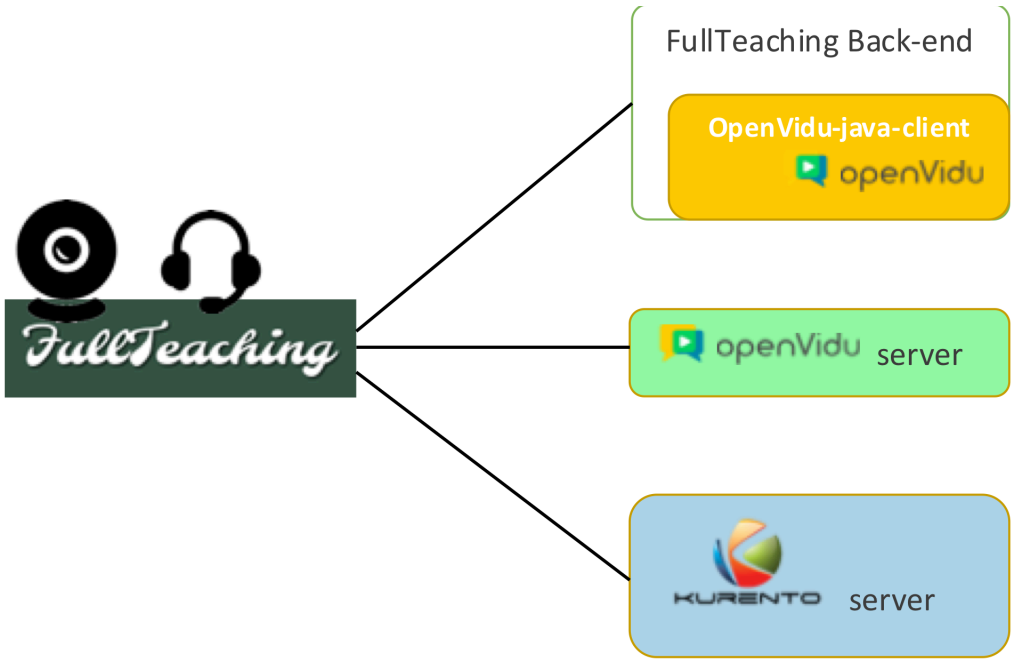
2.1.1.2.3. Student Requirements

Requirement ID	Description
SR01	The student will be able to interact with the courses in his/her dashboard
SR02	The student will be able to see the list of the resources (files) of any course he/she participates in
SR03	The student will be able to publish entries and comments in any enabled forum of any course where he/she is participating
SR04	The student will be able to join video sessions from any course where he/she is participating
SR04a	The student will receive video and audio from the teacher
SR04b	The student will be able to block audio and/or video from the teacher
SR04c	The student will be able to write in the chat associated to the current video-session
SR04d	The student will be able to request be publisher (raise his/her hand)
SR04e	The student will be able to publish his/her audio and video to the rest of the participants as soon as the teacher approves his/her intervention
SR04f	The student will be able to cancel his/her request to be publisher at any time

2.1.2. Architecture

- **Front End:**
  - **FullTeaching Frontend:** Refers to the client module of FullTeaching. It provides users with the entry point to the platform. It can be executed in any kind of supported browser (Chrome, Firefox, Opera and latest version of Edge). It consists in a SPA – Single Page Application – developed with Angular. It communicates with the FullTeaching backend through HTTP requests.

  - **openvidu-browser:** It is the connection interface between the FullTeaching front-end and the OpenVidu Server. It provides an API that simplifies the development of the client application for all applications that make use of OpenVidu. The openvidu-browser is available as a npm package, so it is quite easy to integrate in Angular applications.



- **Figure 7: OpenVidu architecture**

- **Backend:**
  - **FullTeaching Backend:** It is the module that contains the business logic of FullTeaching, including: user management, life cycle of WebRTC sessions, data entities -courses, lessons, forums, files, etc.- It is also the component in charge of managing access control and security. Technically this module is developed combining Java, Spring and Hibernate.

- **openvidu-java-client:** it is a Java library that enables the FullTeching backend to secure communications. It accepts different security parameters for the video-conference sessions. It provides an extra security layer.
- **openvidu-server:** it is the component that manages the video-conference sessions and interacts directly with the media-server Kurento. It handles low level WebRTC communication requirements such as ICE candidates, reconnections, connections, etc.
- **Kurento Media Server:** It is a standalone media-server that provides transmission, management and processing of the media flows in the lower level. As said before it is managed by the OpenVidu-server in a seamless way for the user and the developer.
- **FullTeaching DB:** simple and minimal MySQL DB where all the persistent information is stored.

2.1.3. Resources

The FullTeaching version used for the ElasTest project is available in GitHub in the ElasTest organisation (here: https://github.com/elastest/full-teaching). It contains 3 branches: the master, where code is updated from external repositories in which other people are updating and maintaining FullTeaching; one branch for the ElasTest testing, and the third one for the no-ElasTest testing, in order to maintain a differentiation and a policy of no contamination for the Quasi-Experiment proposed.

An instance of Full Teaching is available under request in the ElasTest CI environment, in order to provide an example for component e2e tests and for validation of the application itself.

2.1.4. Full Teaching test suite

ElasTest provides a really attractive service in order to test FullTeaching and similar web application that also provide real-time communications. The ElasTest functionalities that mainly benefit the testing of FullTeaching are:

- Test Orchestration
- Multiconfiguration
- Test comparison
- Browser recording
- WebRTC browser stats
- Log analysing
- Security testing

Even when many of these features may be available in other tools, they are never available all together into one platform, as for ElasTest, and specific features like the WebRTC browser stats are currently not provided by any other testing tool. When a

tester needs to retrieve them and use them in the context of a testing session, he/she must implement them tailoring them just for the specific test or retrieve them manually from the browser.

In the end, the testing of web applications with realTime communications before ElasTest was very challenging and test automation for WebRTC communications was too costly, so in the end, what was usually done is manual test for a limited set of users, or scenarios.

The tests suites generated in order to validate ElasTest are available in the same repositories as the code; in the folder called e2e-test [https://github.com/elastest/full-teaching/tree/master/e2e-test] you may find the test code for either With-ElasTest and WithOut-ElasTest in the specific branch.

The test strategy that was created to be implemented by both, With-ElasTest and WithOut-ElasTest, was the one defined by Naeva Tec developers as follows:

2.1.4.1. Test Strategy

2.1.4.1.1. Testing types

The tests that will cover the application will be of the following types:

- Unitary tests
- Integration tests
- Performance tests
- Acceptance tests:
  - A compilation of the previous tests that will check if the release can be live or not. Some of the test if failed can be omitted and the release can be launched, and later those failing functionalities corrected or implemented.

2.1.4.1.2. Testing frameworks

All the tests of each of the blocks should be automated and a report should be released for each execution.
The tests will be developed with the following tools:

- **Unitary:**
  - **Angular:** Karma + Jasmine
  - **Java:** jUnit (&maven)
  - This test will execute in the build phase of each release.

- **Integration Test:**
  - Integration with OpenVidu
  - Integration with BDD

- **Performance Test:**
  - Consumed resources
  - Time of Response

- **Acceptance Test:**
  - End 2 End functional
  - End 2 End integrity

2.1.4.2. Test Objectives

2.1.4.2.1. Global scope of testing

The following chart shows all the testing scopes to be tested in optimal conditions.

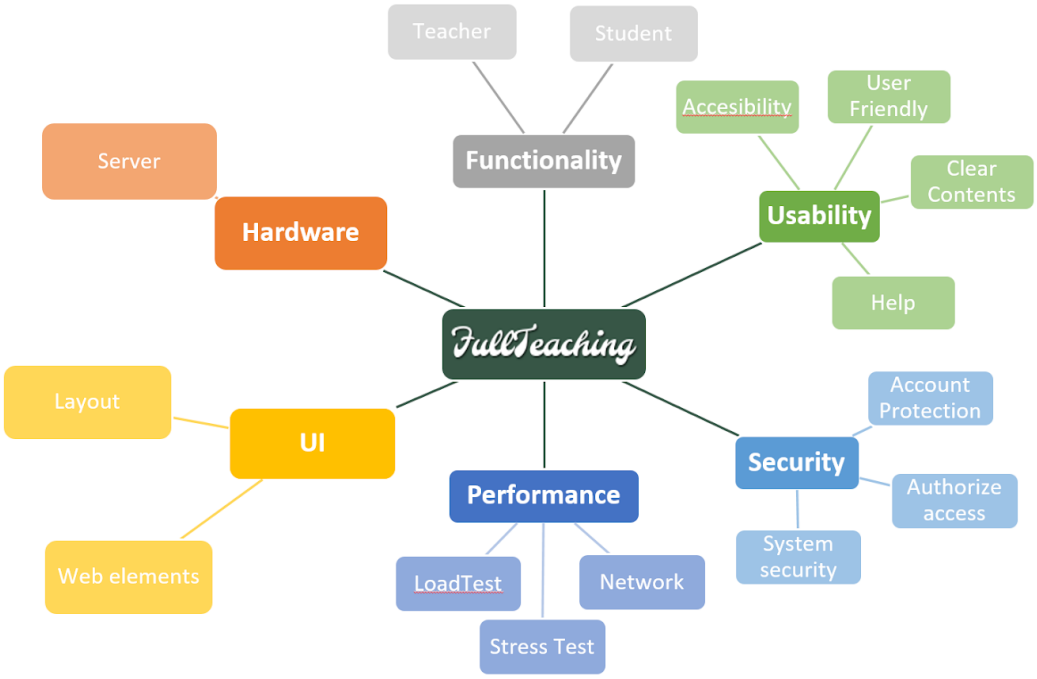


Figure 8: FullTeaching testing scopes

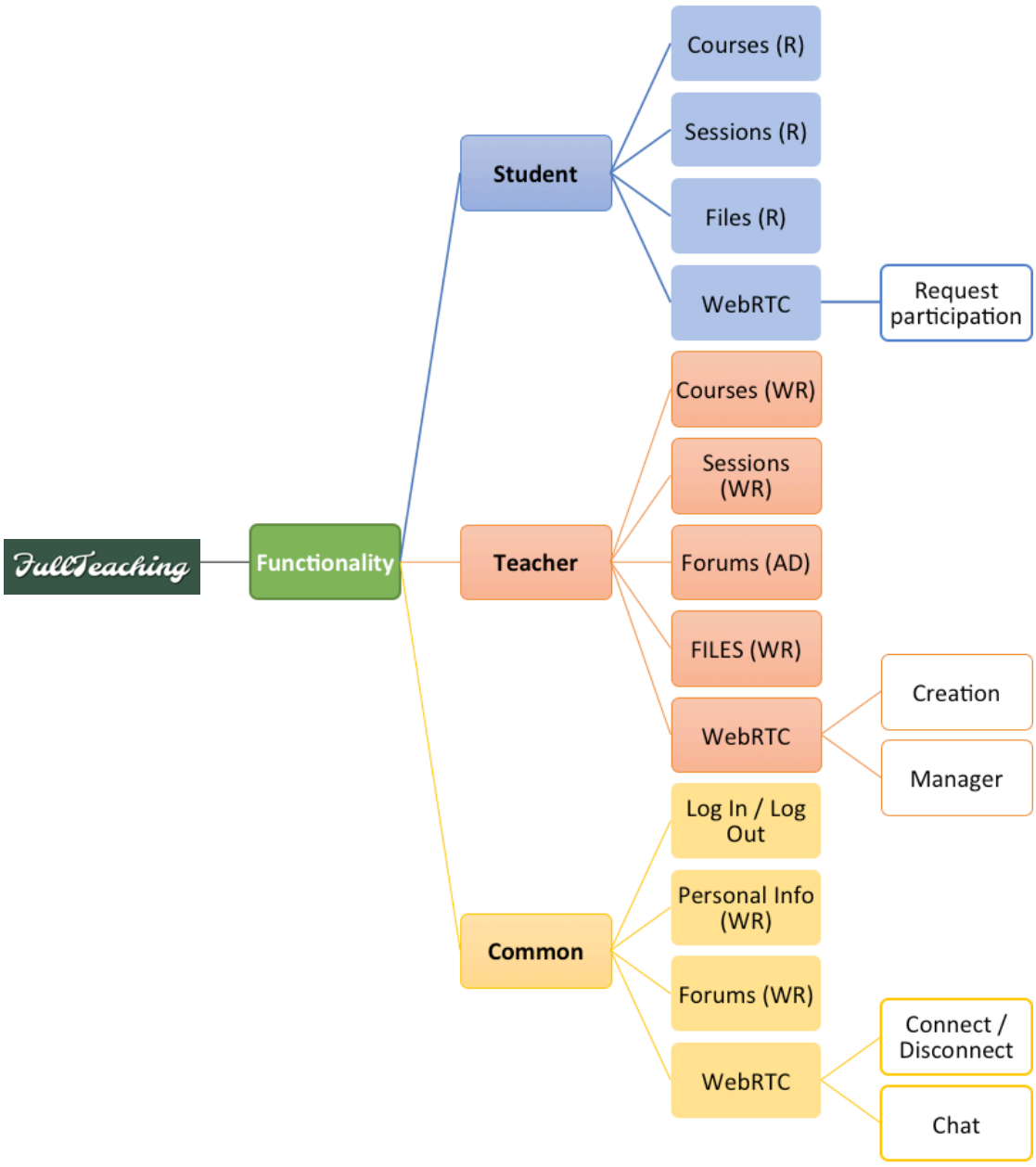


Figure 9: Illustration of test objectives

Figure 9 shows the complete testing objectives set. The priority of each of the branches will be evaluated on following sections and test coverage will be adapted to the priority taking into account time of test development, time of execution, general complexity, and availability of tools, etc.

2.1.5. Full Teaching testing demo

For Unitary and Integration tests in both WE and WO branches, the tests are executed through a mvn command.

In the case of the **end-to-end**, the tests executed consist on:

1. Launching Chrome with the required flags. In case of the WO the Selenium code in the test orchestrator will launch every client node; on the other hand, in WE it is ElasTest that launches and provides the browsers.
2. Loading the FullTeaching web to load in the browsers and start the tests.

WithOut ElasTest the whole process implies adopting lot of tools, by which the tester must look for the results and then look for the application logs. The test results and the tests logs are in the Jenkins sever (in our case), so he/she must go to the application server and in case it was not running specifically for those tests he/she must narrow the logs to the time frame of each test. With ElasTest not only he/she has all the logs and the results in the same place, but also he/she can check the videos of the execution.

In the case of the **security** tests the WE and WO branches are completely different. In WO we had to find and run the OWASP tool deployed with Docker and configure it to test FullTeaching. Even through the tutorials we have not been able to execute and find the results of all the security tests provided by OWASP. With ElasTest this was much easier as ElasTest simply executes the tests and provides straight forward information.

2.2. The OpenVidu demonstrator

While we were running the second round of the QE, in order to be able to reach a better comprehension of the QoE we have created an additional demonstrator that consisted just in an OpenVidu server and a simple JavaScript application that allows to generate OpenVidu sessions and start consuming and publishing to all the users.

As the communication features in FullTeaching are directly provided by the OpenVidu server and the Kurento server, and the QoE of the video call is mainly affected by the quality provided by them, this mini demonstrator allows us to test the QoE of OpenVidu and Kurento without the overhead of a full web application with complex features on top.

2.2.1. Features & Requirements.

This mini application only has a form to set the needed properties to connect to an OpenVidu server:

- OpenVidu public url

- OpenVidu Secret
- User Id
- Session Id

With these parameters the application is capable of connecting to a pre-existing session or generate a new one in the provided OpenVidu server.

The main requirements are that the application should accept any number of concurrent users and sessions, so we can overload OpenVidu and evaluate the degradation of QoE parameters.

2.2.2. Architecture

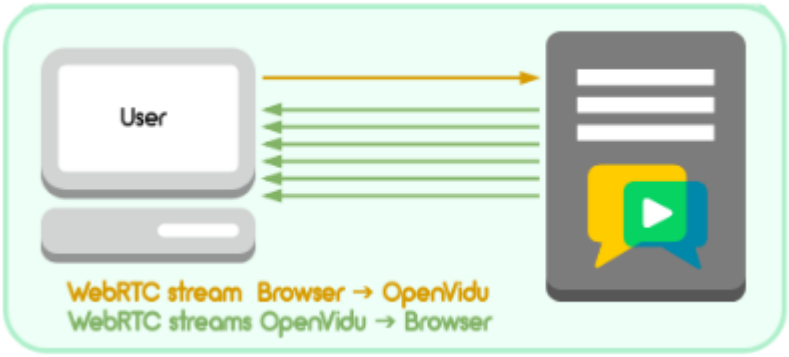


Figure 10: OpenVidu architecture

The architecture of this mini demo is simple and plain as there is one OpenVidu server –with a Kurento media server behind- and a front end with JS that connects to the OpenVidu and shows all the streams that are being published to OpenVidu.

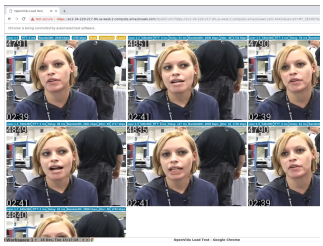


Figure 11: Print screen from OpenVidu session

2.2.3. Resources

The tests specifically developed to be tested Without-ElasTest can be found here: https://github.com/naevatec/openvidu-elastest-qe

The tests specifically developed to be tested With-ElasTest can be found here:

https://github.com/elastest/codeurjc-qe-openvidu

In both repositories the mini-demonstrator can be found.

2.2.4. Mini-demonstrator test suite

2.2.4.1. Test Strategy

2.2.4.1.1. Testing Types

In this mini-demonstrator we will only focus in the load tests and more specifically how the load affects the QoE.

2.2.4.1.2. Load Test Process

We will be testing 3 different environments of OpenVidu, i.e. different deployments of the OpenVidu and the mini-demonstrator.

For each environment:

- Every participant will be connecting from a single browser. Every browser will be launched in its own Docker container with fixed resource configuration (available RAM, number of cores and bandwidth)
- Every browser will be a Chrome instance launched with the following options: allow-file-access-from-files, use-file-for-fake-video-capture=fakevideo.y4m, use-file-for-fake-audio-capture=fakeaudio.wav, window-size=1980,1280
- OpenVidu will be deployed in a dedicated EC2 machine. Every OpenVidu session (and therefore every dockerized browser) will be connecting to this same instance
- Each video will have a resolution of 540×360 pixels, 30 fps
- Each browser will be responsible of obtaining the necessary token to connect to its specific test session (URL will contain as parameters the secret, the session identifier and the ip where to perform REST operations, so the JavaScript code can get the token)
- Client HTML/JS code will show up 1 local video and 6 remotes videos (for 7 users per session), including WebRTC stats for all of them

2.2.5. Full Teaching testing demo

In both With ElasTest and WithOut ElasTest the test follows the same path.

1. Launch Chrome with the required flags (Selenium code in the test orchestrator will launch every client node)
2. Wait for the testing web application to load. This static web app will be hosted in the same AWS EC2 machine as OpenVidu Server.
3. Wait for the browser to connect to the session in OpenVidu Server (connectionCreated event)
4. Wait for the local video to be playing (videoPlaying event)
5. Wait for each one of the remote videos to be playing (videoPlaying event)

With ElasTest at this point gathering the statistics and making decisions is quite simple and needs little human effort. The human just need to compare the different executions on ElasTest and select the best configuration.

On the other hand, without ElasTest the tester may need to develop just for this use case a whole set of additional tailor-made tools that will be collecting and printing the metrics. Then manually they need to do the comparison between executions. When talking of a limited time to make this comparison the number of executions that can be run and compared is very limited. Considering the time needed just to develop and set in place the additional tailor-made tools, this will never be done in real development projects. In this study we have developed it just for the purpose of measuring and comparing with ElasTest branch the same type of data.

2.3. The banking CCS demonstrator

In order to provide feedback and data for the CCS Naeva Tec has selected a POC that showcases the OpenVidu technology applied to Bank OnBoarding. Banks are deploying into their processes ways of allow users to hire banks services fully remote, for this one of the methodologies implies assisted or non-assisted recorded video calls, that should have specific conditions where both conditions and client agreement should be recorded together under certain security parameters.

The POC is based in a one-to-one recorded video-conference, in which a bank agent reads and provides the information about the product and the client accepts the conditions read by the agent, also in the same video a valid ID should be shown. The recorded video-conference is proof of the contract.

As this application is owned by the client, its detailed description is not included in this deliverable. Moreover, this POC has been developed in parallel to the ElasTest platform so there is no coherent baseline data for comparison. The first complete test suite has been prepared to be run already with ElasTest, as Naeva Tec aimed at validating internally the usage of ElasTest platform for this new type of projects. However, even though no quantitative comparison could be performed, in the reporting of Naeva Tec qualitative feedback we also leveraged their experience within this study.

2.4. Naeva Tec feedback

To evaluate ElasTest we have taken into account our experience during the project, with the QE and with our own experience while trying ElasTest in other company projects (CCS - Naeva Tec Bank Onboarding), and experience prior to the project in testing without ElasTest. We absolutely resolve that ElasTest improves the testing process in many measurable factors such as the TTM, because it reduces the time of the testing phase and the bug detection by a considerable amount. Also we infer that as the testing process is more efficient, more tests can be automated, so in the medium and long term more tests would be executed for each release as those automated test will be used in the future to avoid regressions.

Specifically regarding testing real-time communications prior to ElasTest we had minimum test for that applications, we focussed in functionalities and then, on demand, we could try to execute QoE tests usually manually. With our experience with ElasTest we foresee to automate QoE tests for all real-time communications applications, because we have already seen that the time of developing them and execute them in many cases is less than what it takes execute them manually.

3. TUB artifacts

3.1. The IIoT demonstrators

A central use-case in the TUB demonstrators is to show the device emulation capability available in ElasTest through EDS by constructing custom IIoT applications, with the intent of realizing the application concept with minimum or no hardware, thus reducing development costs. Furthermore, by using ElasTest, we aim to evaluate the task of orchestrating a set of targeted tests on the application.

We used OpenMTC[1], a middleware that is the reference implementation of oneM2M[2] Machine to Machine (M2M) communication standard. With OpenMTC, it is possible to have information models constructed on a centralized gateway. Next, the so-called oneM2M containers act as hierarchical placeholders for data, aiding in creation of the information model. Provided an application can reach the oneM2M gateway, the high level use cases can be summarized as follows:

1. An application can perform create, read, update and delete actions on containers and place it on a hierarchical structure in the information model. The containers can be reached by REST API, through a suitably constructed path based on the hierarchy.
2. An application can push data to a preferred container identified by the container's REST path.
3. An application can subscribe to a preferred container through its REST path and get notified when data gets pushed into the container.
4. The oneM2M applications are thus governed by either timer or subscriber notification events for which an application can respond suitably, by performing an action, which can result in a local operation or further push necessary data a container.

It is important to note that TUB provides a Test Support Service (TSS) called ElasTest Device Emulator Service (EDS) that is involved in deploying emulated sensors or actuators on demand as needed by the application implemented in the demonstrator. During Release 8 of ElasTest, EDS already encompasses 3 applications that were part of OIF. Furthermore, EDS is available as a full-fledged application of OpenMTC container, including 3 different sensors and a simple actuator. The demonstrator is thus able to get data from the sensors, apply logic to the data and flag the actuator based on the logic. One can imagine the System in Large as composed of several applications that can be tested using EDS.

We explain the TUB demonstrators in two parts. The first part includes the application used for the quasi-experiments inside the project, while the second includes the demonstrator used for the Comparative Case Study (CCS), which was carried out on an

[1] OpenMTC, http://www.openmtc.org/
[2] oneM2M, http://onem2m.org/

external application. A common aspect in both scenarios is that the ElasTest Device Emulator Service (EDS) has been employed for device emulation.

3.2. Description of application used in QE

For this vertical demonstrator, TUB has chosen OpenIoTFog (OIF), a set of Industrial Internet of Things (IIoT) applications that involve sensors and actuators commonly found on the industry shopfloor that are deployed on fog/edge nodes. OpenMTC lays at the core of OIF, enabling Machine to Machine (M2M) communication between applications and is used as a middleware.

OpenMTC is an implementation of the oneM2M standard, which is used as a middleware by OIF. OpenMTC is offered as open source software and is currently in beta release. A user can clone the software, write an application and test it using OpenMTC. In the context of ElasTest, a demonstrator application implemented by a user becomes a System under Test (SUT) and since the OpenMTC is used to implement such an application, OpenMTC becomes part of SUT. TJobs can cover tests belonging to specifics of the implemented demonstrator application as well as OpenMTC. Along this reasoning, TUB proposes to use test specifications of oneM2M to come up with TJobs concerning the development of OpenMTC. The advantage of testing OpenMTC in this regard is that it can cover test cases for a wide range of possible future demonstrator applications. This can provide a good view on the proposed EDS architecture and demonstrator applications for validation in terms of metrics provided by quasi experiment.

For the purpose of the second round of quasi experiment, TUB thus used OpenMTC as a middleware to develop and test an IIoT application based on the concept of control loop, using the EDS.

3.2.1. IIoT architecture

An IIoT application consists of sensor, logic and actuator. In the control loop, a temperature sensing application is run with which we would like to flag an alarm if temperature goes above 50 degrees centigrade. For this a temperature sensor is needed which feeds data in periodic intervals, say 1 second, to the logic. The logic decides if an actuation is needed by checking whether the temperature provided by sensor is greater than 50 degrees. If greater than 50 degrees an actuating signal is sent to actuator which may be an alarm. In the above-mentioned example of temperature monitoring, the temperature sensor and the actuator are provided by EDS, while the logic is implemented by the demonstrator, collectively acting as a System under Test (SUT). Furthermore another Test Support Service (TSS) of ElasTest, called ElasTest Monitoring Service (EMS) was used along with OpenMTC as a middleware to construct TJobs.

To provide context, an explanation of simple IIoT application is provided first. Then we elaborate the approach taken in With ElasTest (WE) and Without ElasTest (WO) branches to build the application.

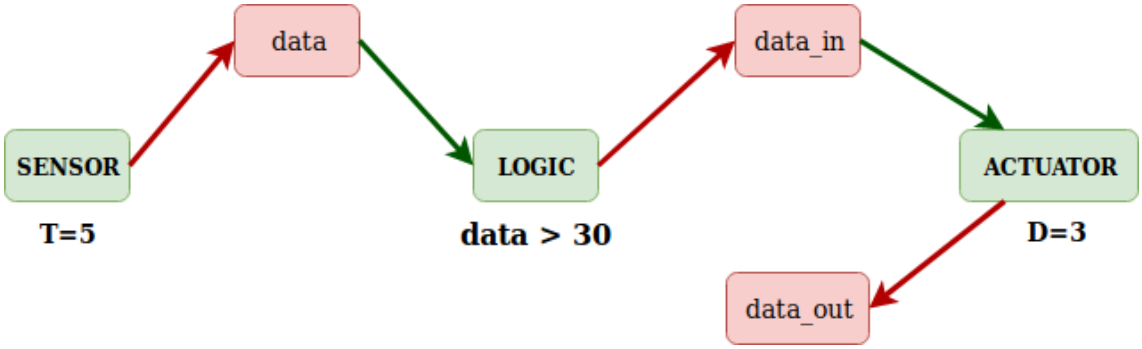


Figure 12: Simple IIoT application

Figure 12 depicts a simple IIoT application composed of a temperature sensor, a logic and an actuator, which are denoted by green boxes representing OpenMTC applications. The red boxes denote OpenMTC containers. The functionality of the application can be understood as follows:

1. **Temperature sensor**: Is an application that creates the data container. Produces a value every 5 seconds and pushes it to the data container.
2. **Actuator**: Is an application that creates two containers, data_in and data_out and subscribes to data_in container which acts as the input for the actuator. The function of the actuator is to wait for a delay of 3 seconds once actuated, before pushing the received data to data_out. The delay parameter models the time required for actuator to complete the task.
3. **Control Logic**: The logic is an application that subscribes to the data container that acts as an input. When sensor pushes to data container, the logic is notified and it invokes a handler where user logic is applied. If the logic is satisfied, the logic pushes the data received from sensor to the data_in container of the actuator.

3.2.2. IIoT test environment

The above-mentioned application was used as SUT in WE and WO branches of QE. The two implementations of the testing environment are described below:

- In case of WO, the SUT was composed of 3 different OpenMTC applications, one each for temperature sensor, actuator and control logic. These applications are provided in a self-documented repository[3].
- In case of WE, the SUT was composed of 3 different OpenMTC applications, with logic implemented natively on the SUT, and EDS providing temperature and simple actuator as emulated devices to the SUT. The architecture of the SUT inside ElasTest can be found in the Figure 13.

[3] OpenMTC example applications, https://github.com/elastest/elastest-device-emulator-service/tree/master/demo/examples

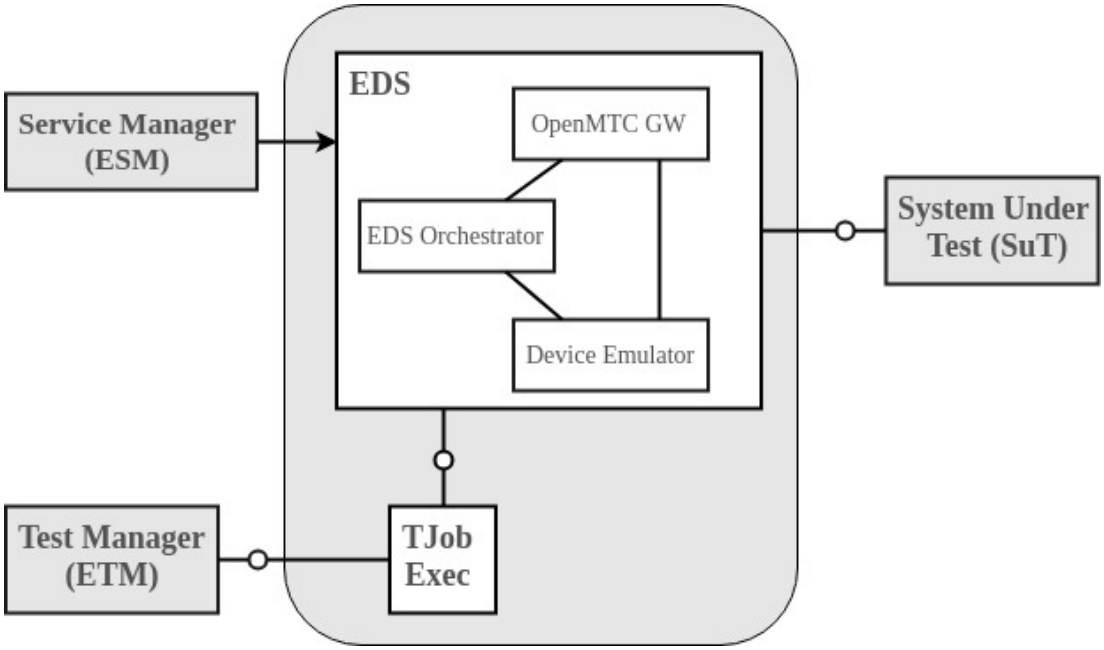


Figure 13: SUT collective architecture in ElasTest using EDS

The workflow to establish the SUT in WE branch of QE is presented as a sequence diagram in Figure 14.

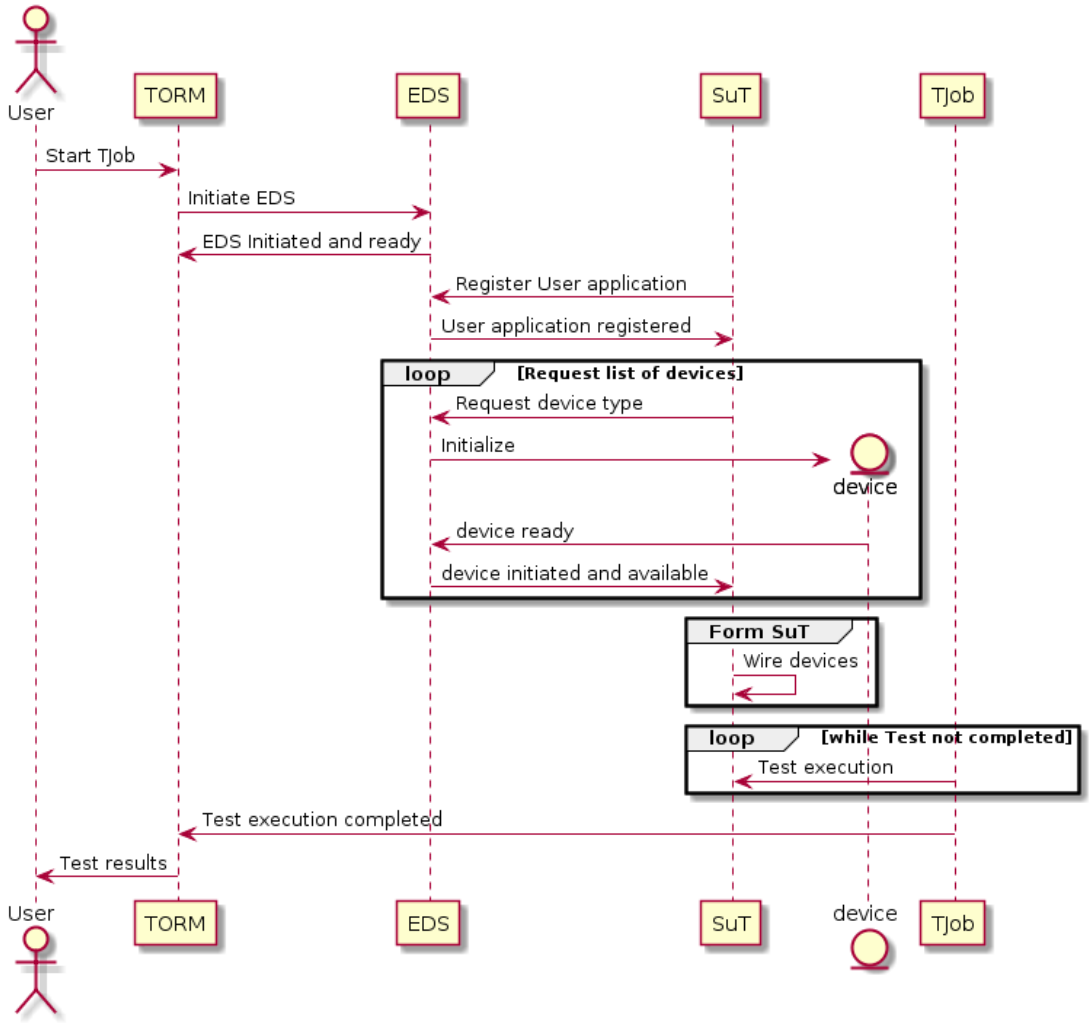


Figure 14: WE SUT application sequence diagram

3.3. Description of application used in CCS

For the CCS TUB used EMBERS[4], a Horizon 2020 project funded from the EU. EMBERS is a smart city project, the tests were performed on a smart city platform, called the Mobility Backend as a Service (MBaas). For the purpose of CCS, different kinds of configuration were used for "WE" case. These configurations include different software for several tests, i.e. for connection testing between the broker and the clients. The tools include Docker images, shell scrips and test suites. In case of ElasTest high level manual and automated SDK testing were performed to create TJobs, which is the job to run the SUT and initiate testing.

The data used for validation have been collected from EMBERS in two different stages:

- Without ElasTest (WO)
- With ElasTest (WE)

[4] https://embers.city/

3.3.1. EMBERS architecture

As EMBERS is a smart city project, the tests were performed on a smart city mobility platform, called the Mobility Backend as a Service (MBaaS). Figure 15 shows the overall architecture to understand the study.

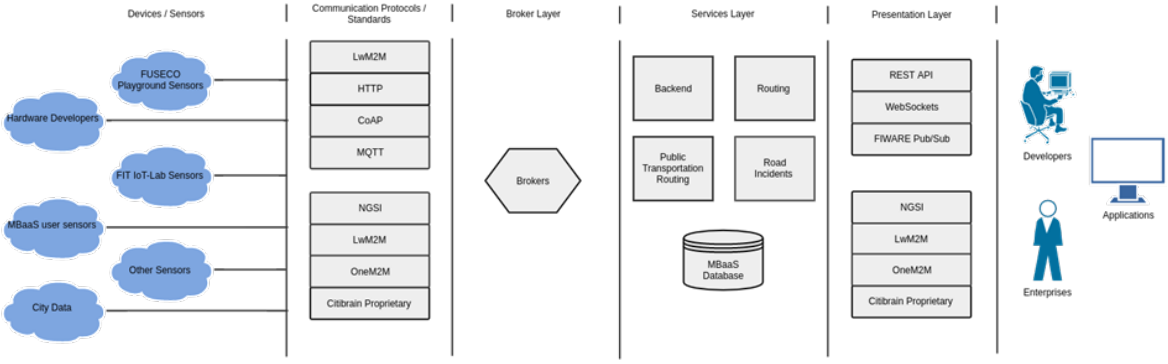


Figure 15: Architecture of EMBERS

The left side of Figure 15 illustrates the link between the physical devices, open data portals or any data source that integrate the data into the MBaaS. These data are being sent to the MBaaS via one of the supported protocols and standards. After the data are sent to the platform, they pass through the broker layer, which analyses the incoming streams and forwards the data to their destination regardless of their standard or protocol. Lastly, the MBaaS processes these data and present them to the end user, via REST APIs or real-time channels. In short, the right side of Figure 15 shows possible connections through the Northbound API while the left side depicts the possible connections to the Southbound API.

Northbound API testing is structured around the idea of how many concurrent users the API can handle. This will help to battle-test and identify bottlenecks in the API implementation and the source code before allowing access to real users. For this purpose, different load-testing tools were analyzed i.e. Apache JMeter[5], smartmeter[6], Tsung[7] and Locust[8].

The southbound API tests are structured around 3 main themes: 1) scaling and load testing the system in order to find - and push - the limits, 2) basic performance comparison between protocols (http, https, mqtt etc.), and 3) real-life long-running use-cases. These tests also include the testing and comparison of Meshblu broker and openMTC backend.

The tests that were performed with ElasTest were only on the southbound part, while 'without ElasTest' part was done on both Northbound and Southbound APIs. For this

[5] https://jmeter.apache.org
[6] https://www.smartmeter.io/
[7] https://github.com/processone/tsung
[8] https://docs.locust.io/en/latest/what-is-locust.html

reason, a simple comparison cannot be possible. However, the results can be analyzed by a thorough examination, especially the time parameters.

3.4. IIoT test suites

IIoT is a field including diverse types of devices that follow various standards. To this end, industrial communication has evolved in identifying and drawing a boundary between devices that are normally represented as nodes, and industrial M2M communication help in not only transporting information but also help in constructing information models. A challenge that arises in large-scale applications is a deadlock between implementation and testing, where procurement of the required hardware is the first step. The deadlock in implementation cannot be solved until the application is validated. This situation was overcome previously by method of simulation or emulation of devices such that applications can be constructed using the emulated devices and tested/validated before investing resources into building an actual application.

ElasTest helps to address such challenge by:

1. Providing emulated devices using EDS which is offered as a TSS within ElasTest.
2. Leveraging the test orchestration facilities as an easier, generic and more adaptable framework for testing IIoT applications.
3. Solving the industrial communication challenge by following an established M2M communication standard called oneM2M in the form of a reference implementation called OpenMTC.

In this section we are going to discuss on usage of ElasTest in testing applications related to QE (Section 3.4.1) and CCS (Section 3.4.2).

3.4.1. Testing in QE

In the first round of QE, TUB used the oneM2M test specifications to test OpenMTC as a reference implementation. This has been documented in D7.1 deliverable at the completion of the first round, therefore in this document we focus on testing methods for second round of QE.

Section 3.2 provides an overview on the SUT used for WE and WO branches of QE. As a prerequisite for developers of both WE and WO we defined a set of 4 test suites depending on the following attributes of the SUT:

1. Sensor Behavior: This suite tests if emulated sensor is behaving accordingly. The suite can be further broken down into two tests:
    a. Sensor Trigger Time: This test checks if there was data generated every 5 seconds by the emulated sensor. If the time difference between two successive timestamps of data generated is greater than 6 seconds, then the test fails.
    b. Sensor Data: This test checks if the data type generated by the emulated sensor is of floating point number, if the data is of any other

type including NULL then the test fails. This test ensures that the device node is generating the right type of data suitable for consumption by the logic application.

2. Actuator Trigger: This test suite contains a single test. When the emulated sensor generates data, this test verifies if in a previous period actuator was supposed to trigger, if yes, did it or not trigger. This test helps to ascertain discrepancies in application logic behavior.
3. Actuator Data Behavior: This test suite contains a single test. When the actuator signals completion signal via actuator's output container, this test verifies if the actuator produces the same data as given to the input of the actuator by the application logic. This test ascertains if there are no discrepancies in signals given from the application logic.
4. Actuator Time Behavior: This test suite contains 3 tests, which are:
    a. Actuator Trigger Expected: This test ensures that the actuator does not misfire unexpectedly. On the event of data generation from the sensor, the application logic decides if the actuator is supposed to trigger. In the event of actuation signal received from data_out container of the actuator, the test checks if the actuator trigger was expected. This test helps in ascertaining the correct behavior of the application logic also provides room for introducing faults for testing robustness.
    b. Actuator Not Late: This test checks if the actuator took a longer time to react. This test helps to ascertain if actuator is fast enough to accept and act on signals from application logic, considering speed of data generation from the emulated sensor.
    c. Actuator Behavior Correct: This test helps to ascertain if the actuator took a reasonable time to complete actuation procedure once the actuation signal was given by the application logic.

The test suites provide a general idea on the tests to be implemented. The details and specifics of the implementation were left to the WO and WE testers. However, developers of both branches agreed on using Python as the main language for programming and unittest framework of Python [9] for testing purposes. The implementation of SUT differs between WO and WE in the following ways:

1. In case of WO:
    a. The OpenMTC gateway was run separately.
    b. The temperature sensor, application logic and actuator were each run as standalone OpenMTC applications.
    c. The test suites were also run as standalone OpenMTC applications.
    d. Developers had to manually start each application and collect logs.
2. In case of WE:
    a. The OpenMTC gateway was provided as part of EDS.

[9] Python Unit testing framework, https://docs.python.org/3/library/unittest.html

b. A single OpenMTC application was now sufficient to request required devices from EDS, wire them together and apply application logic.
c. With the assistance from TSS, ElasTest Monitoring Service (EMS), it was possible to send events from SUT. This significantly sped up test development because the developer did not have to concentrate on actual paths of the containers.
d. At the TJob, it was possible to construct tests based on screening events received from SUT.
e. The WE branch leveraged the test orchestration and log analysis facilities of ElasTest in addition to using EDS and EMS.
f. The tests can be seen documented online[10]. The tests can be executed in a similar fashion how the SUT application was executed.

The following text gives a brief description of insights gained during collecting data for metrics required as part of validation experiments.

3.4.1.1. Time to Market

In both cases of WE and WO, time to market was the time difference between the start of the QE and end of QE. Due to the complexity involved in developing the SUT, the individual test applications and efforts in managing and orchestrating the applications manually proved to be costly in time for WO.

3.4.1.2. Scalability

For this metric, testers had to test 3 SUTs in parallel in both branches. It was found out the WE is more scalable because of the requirement of less number resources as compared to WO. In other words, testing WE completed testing the same number of SUTs with less number of tests and hence lesser efforts.

3.4.1.3. Robustness

For this metric, we introduced a random fault inside the SUT and TJob each. The faults were introduced by an external personal, such that the location and type of introduced faults were unknown to both branches. When the fault occurred, WE branch was successful in locating the fault. Thanks to the log analyzer facility, it was possible for the developer to locate the fault easily. For the developer in WO branch, it was difficult to go through localized log files and search for the fault.

3.4.2. Testing in CCS

For the purpose of CCS, different kinds of configurations were used for 'WE' case. These configurations include different software for several tests, i.e. for connection testing between the broker and the clients. The tools include Docker images, shell scripts, test suites. In case of ElasTest, high level manual and automated SDK testing

[10] QE tests, https://github.com/varungowtham/eds

were performed to create TJobs, which is the job to run the system under test (SUT) and initiate the testing.

3.4.2.1. Time to Market

Time to market for the WO stage starts from the beginning of the project until the 18th month. This stage involved eight developers and four testers in total. The developers and testers worked on different components as well as in the integration of parts i.e. North and Southbound Integrations. There were 13 test cases in total, for example, Connection testing, Load testing for traffic lights, parking and air pollution scenarios. Most of the devices under tests were the physical sensors that collect the data from the environment i.e. temperature, light, etc. Locust and JMeter were the tools that were used for load testing of Northbound API testing, while some scripts along with some other software were used for Southbound API testing. During this 18 months' time, most of the time (around 70%) was spent on individual component development, testing, and analysis.

Unlike TTM for WO, WE stage does not start from the beginning of the project. It starts from the month 19th until 24th. In this stage two developers were involved who were responsible for continuous integration of the developed components as per agile methods. Two testers were responsible for continuous testing and analysis of the components and the system as a whole. Of course, along with ElasTest, some other tools were also used for validation. There were five test cases in total and all the tests were executed on the southbound API of MBaaS. Most of the devices under tests were the emulated sensors that send the data to MBaaS. During this phase, around 50% of the time was spent on components development and testing. While the analysis and end-end testing utilized the 33% and 17% of the total time respectively.

By comparing the data from both branches, we can see that time to market is decreased especially in unit and component testing.

3.4.2.2. Productivity

Considering the productivity in WO, the total number of test cases implemented were 13. This includes around 10k lines of code approximately. The total number of defects of different sorts were eight and all of them were debugged. As far as WE case is concerned, 5 test cases consisting around 4k lines of code were tested in the span of six months. As the number of developers and testers were also limited to 2 and 2 respectively, we could see that the productivity has increased a lot. Lines of code and test cases per unit increased from 555 & 0.7 (WO) to 666 and 0.8 (WE), even though the personnel effort reduces to 1/3.

3.4.2.3. Maintenance

By analyzing the gathered data, there were around 8 bugs found by using tools other than ElasTest. All of the bugs were fixed, however it took around 120 hours of time in total. This time includes the detection, analysis and the removal of faults. By using

ElasTest, we could detect only two bugs that were fixed in 10 hours. By considering all factors i.e. total span of time, number of test cases, the number of defects found and the time spent on fixing the bugs, we can conclude that maintenance requires much less effort by using ElasTest comparing to other tools used in these cases.

3.5. IIoT testing demo and tutorial

For a generic tutorial on testing using EDS, the reader is pointed to the reference documentation of the component[11]. Furthermore, EDS can be re-purposed and enhanced to operate more devices. The source code of EDS is available on the GitHub repository of the project[12].

3.6. TUB feedback

With the conclusion of the validation experiment and CCS, TUB has observed the following aspects:

1. With the availability of EDS, it possible to request and wire sensors through an OpenMTC application. Apart from this, EDS runs an OpenMTC gateway natively. EDS further promotes testing, using emulated devices of large scale applications with lower effort in implementation as well as in testing.
2. Test orchestration is the most effective tool. ElasTest takes care of initialization, management and graceful termination of SUTs and TJobs. This helps users in maintaining a clean state of test environment before and after the execution of tests. This feature proved to be very important for the demonstrator.
3. EDS offers generic emulated devices that can be customized according to the requirements of the user. The generic emulated devices from EDS supports reusability across test cases, by changing the configuration of the devices used in the application.
4. The log analyzer facility is another useful feature in ElasTest, which can significantly reduce the time required in locating test failures.
5. The ability to use multiple TSS helps in combining tests. In the validation experiments, TUB combined the facilities of EMS with EDS that simplified testing process significantly.

[11] EDS reference documentation, https://elastest.io/docs/test-services/eds/
[12] EDS GitHub repository, https://github.com/elastest/elastest-device-emulator-service