Eclipse IoT-Testware Documentation

Eclipse IoT-Testware Team

Sep 15, 2021

Contents

1 Quickstart Guide **3**
- 1.1 Preparation . . . 3
- 1.2 Installation . . . 3

2 Installation **5**
- 2.1 Target Environment . . . 5
- 2.2 Dashboard . . . 5
- 2.3 Dockerized Command Line . . . 6
- 2.4 Manual Installation . . . 8

3 Protocol Conformance Testing **19**
- 3.1 General . . . 19
- 3.2 Test Suite Structure . . . 20
- 3.3 Test Configurations . . . 20
- 3.4 Test Purpose Catalogues . . . 20
- 3.5 IoT-Testware Test Suites . . . 21

4 ETSI and Eclipse Foundation **31**

5 Dashboard **33**
- 5.1 Introduction . . . 33
- 5.2 Overview . . . 33
- 5.3 Backend . . . 34
- 5.4 Frontend . . . 34
- 5.5 Integration . . . 34

6 Fuzzing **37**

7 Smart Fuzzing Proxy **39**
- 7.1 Installation . . . 40
- 7.2 General Concept . . . 41
- 7.3 Identify the SUT . . . 42
- 7.4 Identify Input fields . . . 42
- 7.5 Choose a Test Data Generator . . . 48
- 7.6 Running a Fuzzing Session . . . 48
- 7.7 Analyzing Fuzzing Logging . . . 48

8 About Eclipse IoT-Testware **51**
8.1 IoT-Testware Team . . . 51
8.2 Objective . . . 51
8.3 Conformance Test Methodology and Framework . . . 51
8.4 Implementation . . . 52

9 Glossary **55**

10 References **57**
10.1 2017 . . . 57
10.2 2018 . . . 57
10.3 2019 . . . 57

11 License **59**

12 MQTT Specification **63**

13 CoAP RFC 7252 **65**

14 Indices and tables **67**

Bibliography **69**

Index **71**

Great that you considering the Eclipse IoT-Testware to test your implementations. We would like to guide you to start with the test suites.

CHAPTER 1

Quickstart Guide

The easiest way to get started is to use Docker. The provided Dockerfile hides the complexity of TTCN-3 from the user. Therefore the user has more time to test his systems.

1.1 Preparation

- Make sure you have a working Git installation
- Make sure you have a working Docker installation
- (optional) Set the following environment variables

```
TESTWARE=iot_testware
```

1.2 Installation

1. Clone the IoT-Testware Dashboard project

```
git clone https://github.com/eclipse/iottestware.dashboard
cd iottestware.dashboard
```

2. Build the Docker image

```
docker build -t $TESTWARE .
```

3. Start the Docker container

```
docker run --network=host -ti $TESTWARE
```

4. Call the Dashboard in your browser from `https://localhost:3001`

CHAPTER 2

Installation

2.1 Target Environment

It is possible to install and run (parts of) the IoT-Testware in at least three different ways:

- *Dashboard*
- *CLI*
- *Manual*

For each of the mentioned possibilities exists different requirements for the environment to be set up. Please consider your favorite way of using the IoT-Testware and make sure you fullfill all requirements.

2.2 Dashboard

2.2.1 Preparation

- Make sure you have a working Git installation
- Make sure you have a working Docker installation
- (optional) Set the following environment variables

```
TESTWARE=iot_testware
```

2.2.2 Installation

1. Clone the IoT-Testware Dashboard project

```
git clone https://github.com/eclipse/iottestware.dashboard
cd iottestware.dashboard
```

2. Build the Docker image

```
docker build -t $TESTWARE .
```

3. Start the Docker container

```
docker run --network=host -ti $TESTWARE
```

4. Call the Dashboard in your browser from `https://localhost:3001`

2.3 Dockerized Command Line

If you are already familiar with concepts and the CLI of Eclipse Titan than this way is a light-weight solution to run the Testware without webserver and dashboard.

2.3.1 Preparation

- Make sure you have a working Git installation
- Make sure you have a working Docker installation

2.3.2 Installation

1. Clone the main IoT-Testware project

```
git clone https://github.com/eclipse/iottestware
cd iottestware
```

2. Build the Docker image

```
docker build -t TESTWARE .
```

3. Start the Docker container

```
docker run -ti TESTWARE /bin/bash
```

2.3.3 Run Test Campaigns

The IoT-Testware Docker image ships currently two test suites for MQTT and CoAP. We will show you quickly how to configure and run the test suites.

Starting test suites with TITAN

```
ttcn3_start [-ip host_ip_address] executable [file.cfg] {module_name[.testcase_name]}
```

Looks quite easy: in order to start a test campaign TITAN requires us to provide an **executable** test suite. As we want also be able to provide different kinds of configurations, we also need to provide a **.cfg** file. Fortunately, we already have all the components in Docker. Let's see how we can run some MQTT conformance tests against your System Under Test (SUT).

1. Change directory to the MQTT playground and make yourself familiar with the provided files.

```
cd /home/titan/playground/mqtt; ls
```

By default, the public MQTT Broker `iot.eclipse.org` is preconfigured in the provided configuration file. If you want to change the default configuration follow the next instructions.

1.1 (optional) get started with your own configuration

```
cp BasicConfig.cfg YOUR_CONFIG.cfg
vi YOUR_CONFIG.cfg
```

1.2 (optional) to configure the TS (Test System) for your SUT you can change the `hostName`, `portNumber` and `credentials` definitions.

```
tsp_addresses := {
  {
    id := "mqtt_server",
    hostName := "iot.eclipse.org",
    portNumber := 1883
  }, {
    id := "mqtt_client",
    hostName := "0.0.0.0",
    portNumber := 45679,
    credentials :=
    {
      clientId := "CLIENT_ID",
      username := "USER_NAME",
      password := "PASSWORD",
      topicName := "your/mqtt/topic/name"
    }
  }
}
```

1.3 (optional) put together your own test campaing by choosing test cases from the `[EXECUTE]` section.

Note: The configuration file can contain any white space characters. There are three ways to put comments in the file: you can use the C comment delimiters (i.e. /* and */). Additionally, characters beginning with # or // are treated as comments until the end of line.

```
[EXECUTE]
### CONNECT Control Packet
MQTT_TestCases.TC_MQTT_BROKER_CONNECT_001    # <- will execute
MQTT_TestCases.TC_MQTT_BROKER_CONNECT_002    # <- will execute
#MQTT_TestCases.TC_MQTT_BROKER_CONNECT_003   # <- won't execute
...
```

2.1 Run the whole test campaign given in `YOUR_CONFIG.cfg`

```
ttcn3_start iottestware.mqtt YOUR_CONFIG.cfg
```

2.2 (alternative) Run a single test case `TC_MQTT_BROKER_CONNECT_001` from `MQTT_TestCases`

```
ttcn3_start iottestware.mqtt YOUR_CONFIG.cfg MQTT_TestCases.TC_MQTT_BROKER_
↪CONNECT_001
```

2.4 Manual Installation

The IoT-Testware is composed of several test suites from different repositories with once again several dependencies to the **Eclipse Titan** runtime. Hence, the process for a native installation can become quite complex. However, a native installation is very helpful for development. Therefore we provide additional 'flavours' of installation for development purposes.

- We highly recommend to consider the Titan installation guide to set up Titan properly (check the supported OS ons the side beforehand).
- If you are not using one of the supported OS, we recommend to set up a virtual machine or use Titan with Docker.

2.4.1 install.py

This documentation helps you to understand and use the install.py script.

Contents

- *install.py*
  - *Prerequisites*
  - *help command*
  - *protocol command*
  - *build command*
  - *path command*
  - *executable command*
  - *verbose command*

Prerequisites

- Latest version of
- Latest version of . For installation details please consult the official

help command

You can refer to the help command if you don't know how to continue at any point. Simply run one the following command:

```
python install.py -h
python install.py --help
```

The result gives you a first idea, what is possible with the script. The output looks like this:

-h	show this help message and exit
-p PROTO	available protocol test suites are {mqtt, coap, opcua} sets a protocol that will be cloned together with its dependencies
-b	build the project and create a Makefile

--path PATH	specify optionally your root directory, where all dependencies will be stored
-e NAME	set the name of the executable that will be generated
-v	progress status output is verbose

protocol command

The protocol command is the a **mandatory** flag. It will scan your working directory to check whether all dependencies are met. In case there is something missing, the script will download the missing dependencies automatically. The command demands a parameter representing the interesting protocol.

Let's assume you would like to run tests against a CoAP implementation. Run one of the following command to get the CoAP conformance test suite and all it's dependencies:

```
python install.py -p coap
python install.py --protocol coap
```

Use the same procedure for any available protocol.

build command

This optional command can be used to build the IoT-Tesware. It builds a *Makefile* first and creates an executable afterwards. You can only build one protocol at a time. It is determined by the *protocol command*. To build the CoAP test suite for example, run one of the following commands:

```
python install.py -p coap -b
python install.py -p coap --build
```

path command

When you run the install script, it creates a folder structure under `~/Titan` by default. This is your base directory where the IoT-Testware and all it's dependencies are stored:

IoT-Testware You find the test suites for the protocols you have chosen via the `protocol` command. It creates a folder for every protocol separately in the form of `iottestware.<PROTOCOL>`

Libraries Collection of libraries needed for the specific IoT-Testware protocol.

ProtocolModules The protocol modules, provided in , are included inside this directory. The subset of protocol modules are protocol dependent. They define the protocol types.

TestPorts To bridge the gap between the test suite and the system under test (SUT), test ports are needed. They are provided by Eclipse Titan project.

executable command

With this command it is possible to name the executable that is generated when calling the *build command*. In contrast, the install script chose a default name for the executable following the scheme:

`iottestware.<PROTOCOL>`

To set your own name for the resulting executable, let's say "myExecutable" simply run one of this command:

```
python install.py -p coap -e myExecutable
python install.py -p coap --executable_name myExecutable
```

verbose command

If you set this command, the console output will be verbose and give you more information during the process. By default, the output is quite, meaning only important messages are shown. To switch the verbose output on, you add either ‘‘ -v ‘‘ or ‘‘ –verbose ‘‘ to your command like in the following examples:

```
python install.py -p coap --verbose -e myExecutable
python install.py -p coap -b -v
python install.py --verbose --protocol coap
```

2.4.2 Install with Eclipse IDE

These instructions will get you a clean IoT-Testware clone up and running on your local machine for development and testing purposes.

Contents

- *Install with Eclipse IDE*
  - *Prerequisites*
  - *Installing Quickstart*
  - *Set up*
  - *Eclipse IDE*

Prerequisites

- Latest version of **Java**
- Latest version of **Python (v2 or v3)**
- Latest version of **Eclipse Titan**. For installation details please consult the official Eclipse Titan

Installing Quickstart

Note: Make sure all prerequisites are met. As we use Eclipse Titan (natively running under Linux) to compile and execute our test suite, the following instruction won't cover other OS like Windows or MacOS. We recommend installing a Linux derivate in a virtual machine.

Set up

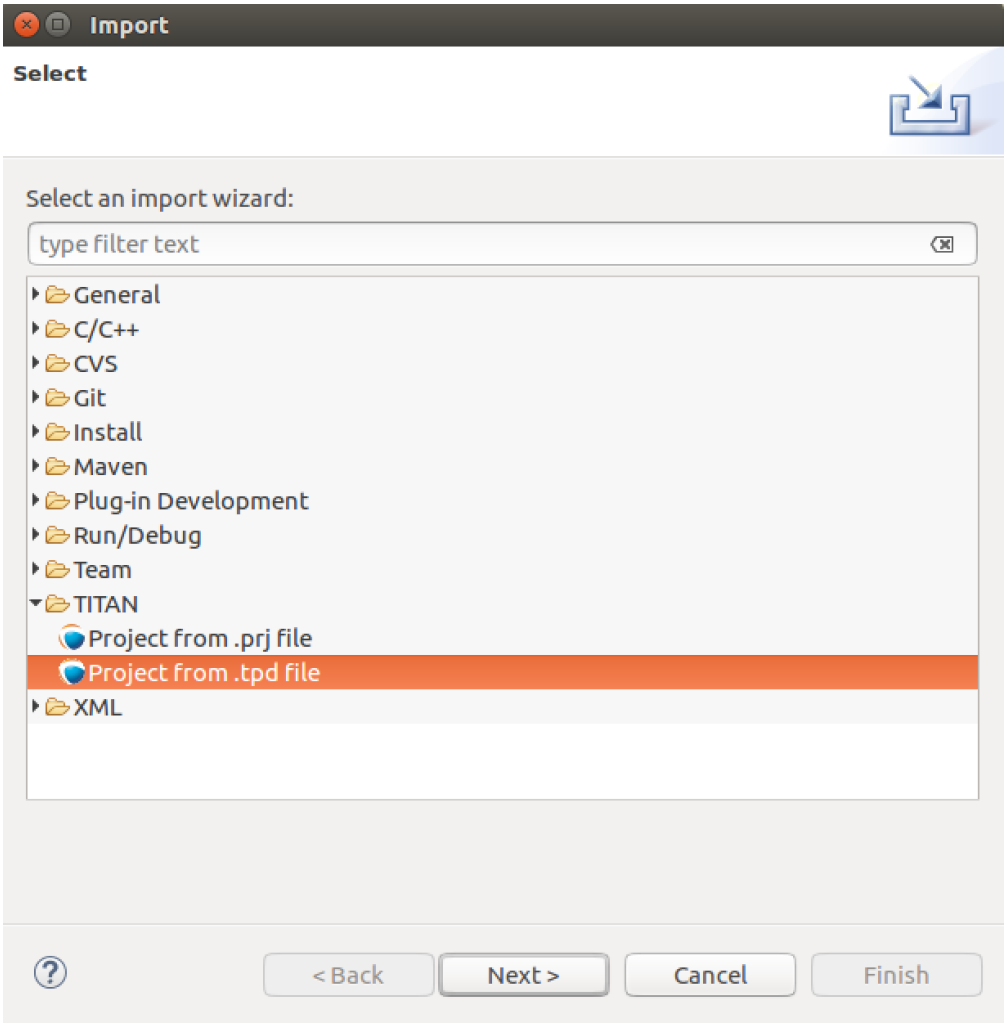
Firstly, you need to get all needed dependencies to run the test suites. To do so, simply run the python script `install.py` with your protocol of choice:

```
python install.py -p <PROTOCOL>
```

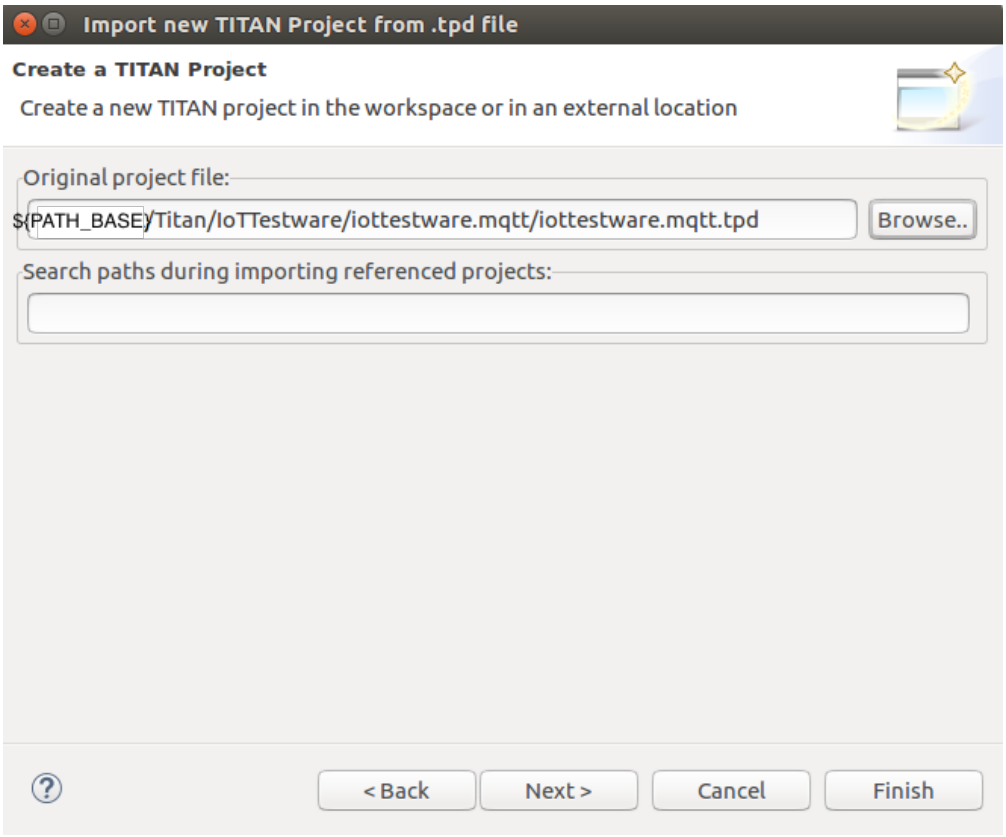
This is the most minimalistic way of getting the dependencies. For a more complete explanation of the installation script please refer to the documentation. With Eclipse Titan you are free to choose to work either from the Command Line (CLI) or from Eclipse IDE. Go ahead and read further instructions of your preferred way of working.

Eclipse IDE

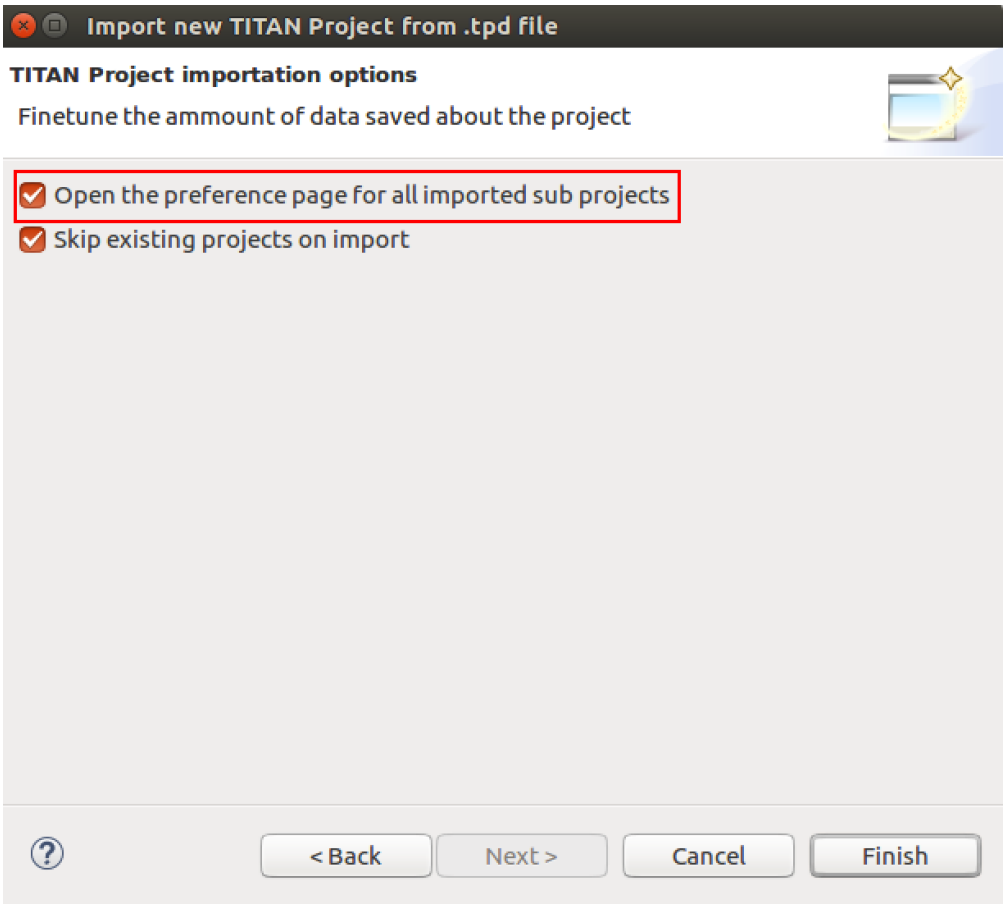
In every of our protocol repositories you find a **iottestware.<PROTOCOL>.tpd** file that you need to import. Open the Titan IDE in your desired workspace and use the import feature `File -> Import -> TITAN -> Project from .tpd file`



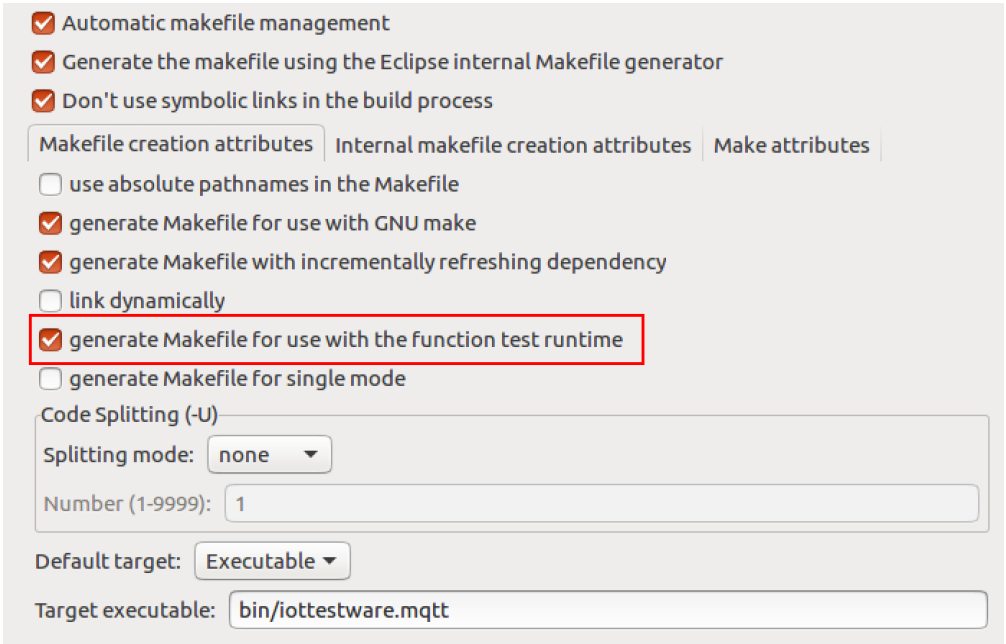
Klick Next and choose the **iottestware.<PROTOCOL>.tpd** from **${PATH_BASE}**



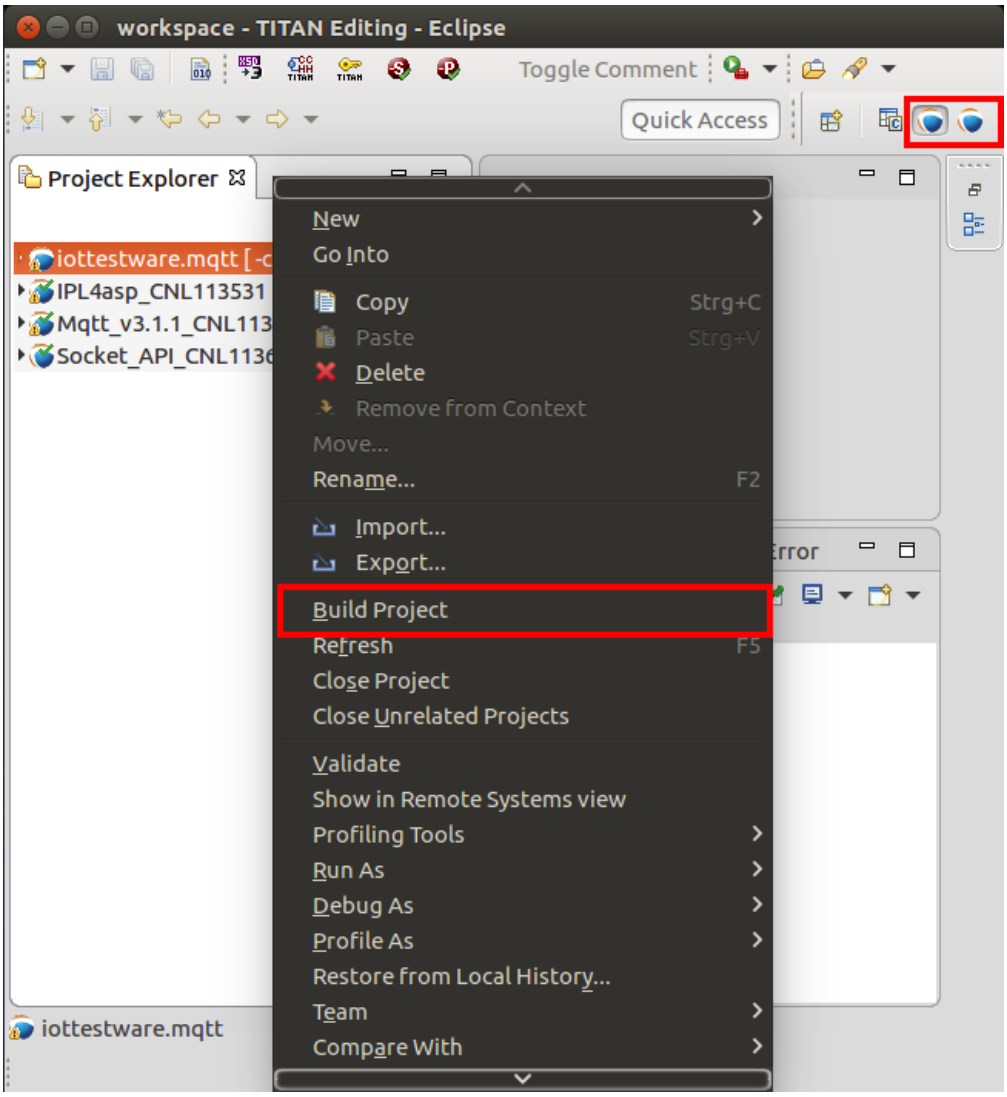
Klick Next and choose importation options.



Klick Finish and the IDE will import all the required Projects and open the properties for each. Make sure each project is configured to **_generate Makefile for use with the function test runtime_**.



Right-click the **iottestware.<PROTOCOL>** project and select **Build Project**.



Note: Make sure you are in the TITAN Editing Perspective, otherwise the Build Project might be not available.

2.4.3 Install with Docker

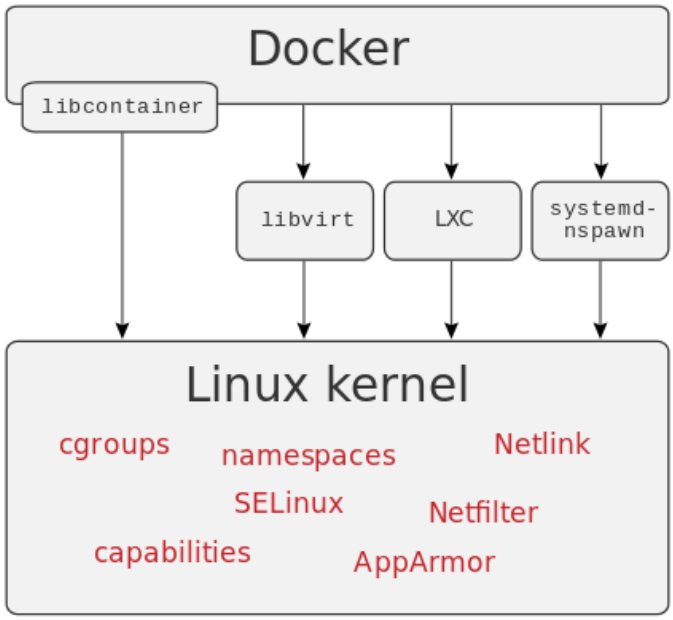
For an easy deployment, you can use the shipped Dockerfile coming with the repository.

Contents

- *Install with Docker*
  - *Preparations*
    - *Additional Docker commands*
  - *Start Docker container*
  - *Run the Dashboard*

Docker is a computer program that performs operating-system-level virtualization. It uses the resource isolation

features of the Linux kernel to allow independent containers to run within a single Linux instance, avoiding the overhead of staring and maintaining virtual machines.



Docker will perform all the heavy lifting of virtualization and running the IoT-Testware including Eclipse Titan. Therefore, an installed and functioning Docker is the only prerequisite.

Note: Although, it is possible to run Docker containers on different operating systems, Docker's host networking driver only works on Linux hosts. Hence, it is recommended to run the Docker containers on a Linux machine.

Preparations

- Make sure you have a working Docker installation
- (optional) Set the following environment variables

```
TW_CONTAINER_NAME=iot_testware
TW_NETWORK_NAME=iottestware_net
TW_SUBNET=172.18.0.0/16
TW_FIXED_IP=172.18.0.4

TW_VOLUME_NAME=testware_volume
TW_VOLUME_PATH=/home/titan/iottestware.webserver/backend/resources/history
```

- (optional) Create separated Docker network

```
docker network create --subnet $TW_SUBNET $TW_NETWORK_NAME
docker network ls
```

- (optional) Create persistend storage and docker volumes

```
docker volume create $TW_VOLUME_NAME
```

- Build the Docker container

```
docker build -t $TW_CONTAINER_NAME .
```

Additional Docker commands

- Stop all running container

```
docker stop $(docker ps -aq)
```

- Delete all containers

```
docker rm $(docker ps -aq)
```

- Delete all images

```
docker rmi $(docker images -q)
```

- Force delete a specific image

```
docker rmi -f <IMAGE_ID>
```

- open second bashwindow

```
docker exec -it <CONTAINER_ID> /bin/bash
```

Start Docker container

Docker offers many options for starting and integrating containers. In this section we will show how the container can be started with persistend storage and how to attach the container to the previously created sub-network. Read the Docker networking overview for more information.

1. Most basic way to start a Docker container *without persistend storage* and using the *host's network interface*

```
docker run --network host $TW_CONTAINER_NAME
```

2. Isolated Docker container which is attached to the *sub-network with a fixed IP* and *without persistend storage*

```
docker run --network $TW_NETWORK_NAME --ip $TW_FIXED_IP $TW_CONTAINER_NAME
```

3. Using *host's network interface* and *with persistend storage*

```
docker run --network host -v $TW_VOLUME:$TW_VOLUME_PATH $TW_CONTAINER_NAME
```

4. Isolated Docker container which is attached to the *sub-network with a fixed IP* and *with persistend storage*

```
docker run --network $TW_NETWORK_NAME --ip $TW_FIXED_IP -v $TW_VOLUME:$TW_
↪VOLUME_PATH $TW_CONTAINER_NAME
```

Run the Dashboard

Note: This step requires that you have used the Dashboard Docker file

Once everything is correctly deployed and started you can access the IoT-Testware Dashboard from your browser. Dependent on you network configuration simply open one of the following URLs in your browser:

- If you used the host network for the container: `https://localhost:3001`
- If you deployed the container with a custom network and given fixed IP: `https://$TW_FIXED_IP:3001`

Useful

CHAPTER 3

Protocol Conformance Testing

Contents

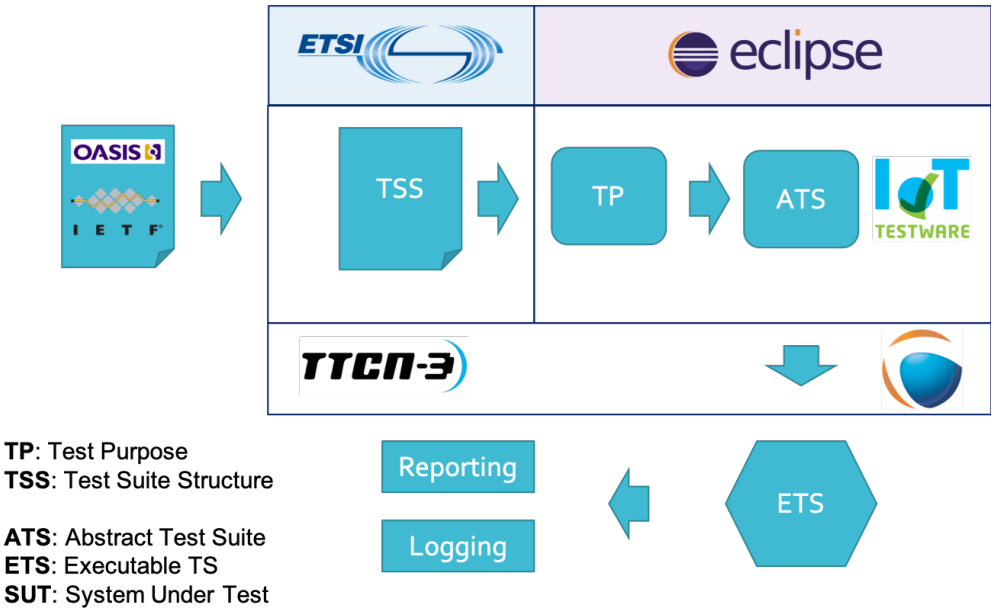
- *Protocol Conformance Testing*
  - *General*
  - *Test Suite Structure*
  - *Test Configurations*
  - *Test Purpose Catalogues*
  - *IoT-Testware Test Suites*

3.1 General

The purpose of conformance testing is to determine to what extent a single implementation of a particular standard conforms to the individual requirements of that standard. Please find additional and more detailed information about conformance testing at ETSI's *Center for Testing & Interoperability*

The ISO (International Organization for Standardization) standard for the methodology of conformance testing (ISO/IEC 9646-1 and ISO/IEC 9646-2) as well as the ETSI (European Telecommunications Standards Institute) rules for conformance testing (ETSI ETS 300 406) are used as a basis for the test methodology.

To implement this methodology we require several intermediary artefacts. Those single artefacts break down the whole complexity of conformance testing into smaller pieces, each with a specific perspective on the problem.



3.2 Test Suite Structure

In the first step we define a TSS (Test Suite Structure) for a specific IUT (Implementation Under Test).

The TSS reflects the coverage of the reference specification by the TS (Test System): it is a synopsis of "which tests are performed on which aspects of the reference specification". The conformance requirements and the ICS (Implementation Conformance Statement) proforma of the base specification are an essential source of cross-reference to check that the coverage of the test suite specified by the TSS&TP (Test Suite Structure & Test Purposes) is acceptable.

3.3 Test Configurations

TODO: Why do we need Test configurations?

3.4 Test Purpose Catalogues

A TP (Test Purpose) (*Test Purpose*) is a formal description of a test case. A formal description in the form of a TP offers a possibility of describing the purpose of a test without having the later technical implementation in mind. Following the TSS the tester is supported in systematically covering the complete IUT specification.

The listing below shows a simple MQTT (MQ Telemetry Transport) TP specified in TDL-TO (Test Description Language - Structured Test Objective Specification).

```
Test Purpose {
  TP Id TP_MQTT_Broker_CONNECT_001

  Test objective
  "The IUT MUST close the network connection if fixed header flags in CONNECT Control␣
↪Packet are invalid"
  Reference
```

(continues on next page)

(continued from previous page)

```
    "[MQTT-2.2.2-1], [MQTT-2.2.2-2], [MQTT-3.1.4-1], [MQTT-3.2.2-6]"
  PICS Selection PICS_BROKER_BASIC

  Expected behaviour
  ensure that {
    when {
        the IUT entity receives a CONNECT message containing
        header_flags indicating value '1111'B;
    }
    then {
        the IUT entity closes the TCP_CONNECTION
    }
  }
}
```

The exemple below shows a simplified tabular representation for the TP.

TP-ID	TP_MQTT_BROKER_CONNECT_01
Selection	PICS_Broker
Summary	The IUT MUST close the network connection if…
Reference	[MQTT-2.2.2-1], [MQTT-2.2.2-2]
Expected bahaviour	
initial condition statement	
ensure that statement	

3.5 IoT-Testware Test Suites

This steps focuses on a technical implementation of the TPs. We use TTCN-3 and Eclipse Titan to implement each TP into a TC (Test Case) and orchestrate to executable test suites.

3.5.1 MQTT Test Suite

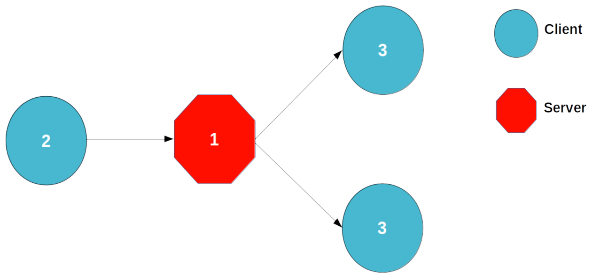
Contents

- *MQTT Test Suite*
  - *MQTT Protocol*
  - *Test Configurations*
  - *Test Purposes*
  - *Test System*
  - *Test Cases*
  - *Test Case Functions*
  - *Translation Port*
  - *Generic Functions*
  - *Test Templates*

- *Default Behaviours*

MQTT Protocol

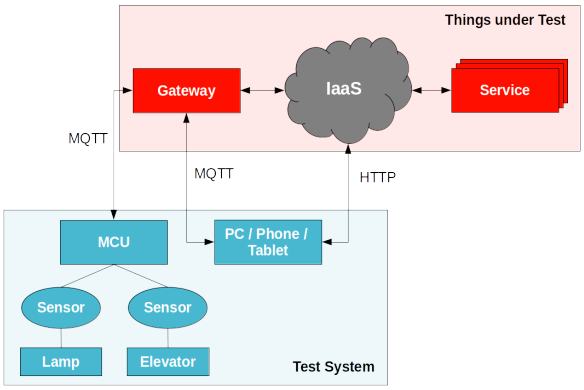
A very brief summary of MQTT from the FAQ MQTT stands for MQ Telemetry Transport. It is a publish/subscribe, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks.



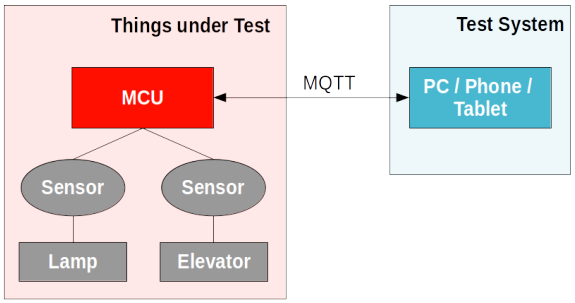
Note: We provide an *annotated version* of the official MQTT specification which can be directly referenced (e.g. and)

Test Configurations

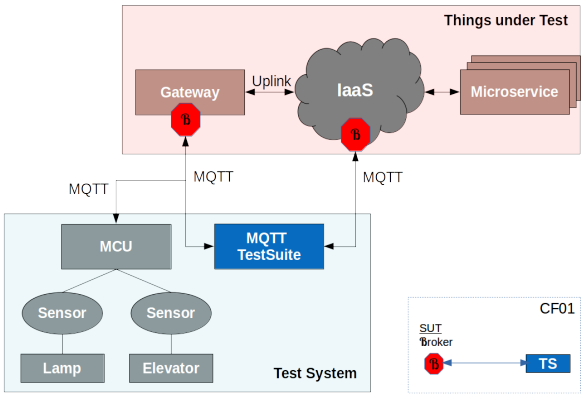
From a general and abstract perspective MQTT has two basic architectures for testing. This architecture directly reflects the choice of your SUT (System Under Test). We will call the first architecture *Broker Testing*. A MQTT Broker is the SUT as shown in the figure blow:



The second major architecture we will call *Client Testing* as now, the Client is in focus as the SUT.



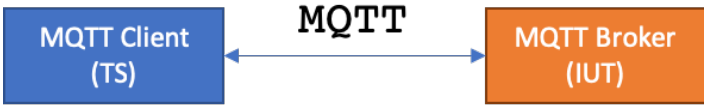
Now we can start to extract different configurations from the test architectures. The image below depicts the step of retrieving test configurations from the architecture:



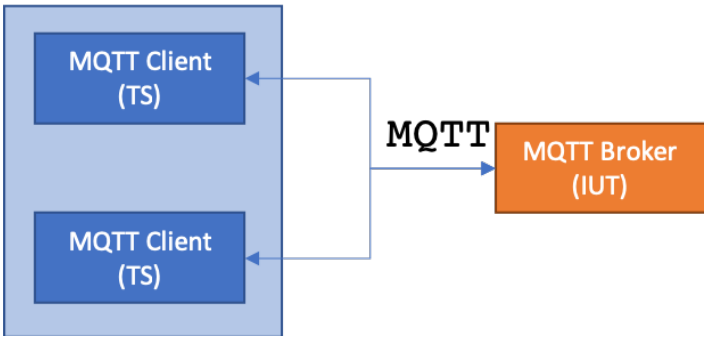
The output of this exemplary step is a test configuration (`CF01`) where the Broker is the SUT and the TS takes the role of a MQTT Client.

The MQTT test suite uses four test configurations in order to cover the different test scenarios. In these configurations, the tester simulates one or several MQTT clients or brokers implementing the MQTT protocol.

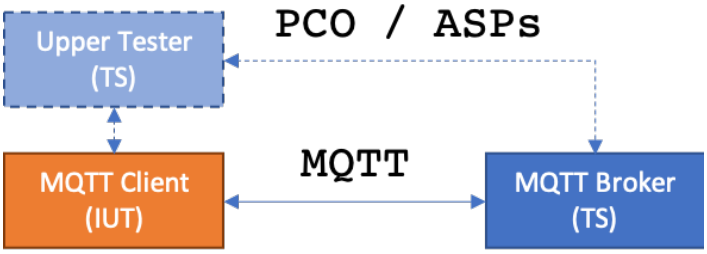
ID:	MQTT_Conf_01
Description:	The MQTT Broker is the IUT and the TS takes the role of a MQTT Client



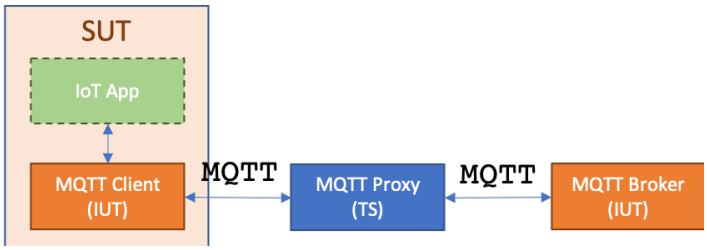
ID:	MQTT_Conf_02
Description:	The MQTT Broker is the IUT and the TS takes the role of multiple MQTT Clients.



ID:	MQTT_Conf_03
Description:	The MQTT Client is the IUT and the TS takes the role of a MQTT Broker. For this configuration an optional UT (Upper Tester) might be required.



ID:	MQTT_Conf_04
Description:	As well the MQTT Broker as the MQTT Client, each is a IUT in this configuration. The part of the UT from the previous configuration is here replaced by a concrete application.



Test Purposes

TODO: link to .tplan2 from GitHub and .pdf from ETSI

```
1   Test Purpose {
2     TP Id TP_MQTT_Broker_CONNECT_001
3
4     Test objective
5     "The IUT MUST close the network connection if fixed header flags in CONNECT␣
  ↪Control Packet are invalid"
6     Reference
7       "[MQTT-2.2.2-1], [MQTT-2.2.2-2], [MQTT-3.1.4-1], [MQTT-3.2.2-6]"
8     PICS Selection PICS_BROKER_BASIC
```

(continues on next page)

(continued from previous page)

```
    Expected behaviour
    ensure that {
      when {
          the IUT entity receives a CONNECT message containing
          header_flags indicating value '1111'B;
      }
      then {
          the IUT entity closes the TCP_CONNECTION
      }
    }
  }
```

Test System

TODO: describe Test System -> TTCN-3 code

Test Cases

You can find all MQTT Tests on GitHub.

We will examine the procedure of a single TC in order to get the understanding of the code structure.

The code block below shows the TTCN-3 implementation of the TC `TC_MQTT_BROKER_CONNECT_01` for the cohesive TP `TP_MQTT_BROKER_CONNECT_01`

```
/*
* @purpose The IUT MUST close the network connection if fixed header flags in CONNECT␣
↪Control Packet are invalid
*
* @reference [MQTT-2.2.2-2], [MQTT-3.1.4-1], [MQTT-3.2.2-6]
*/
testcase TC_MQTT_BROKER_CONNECT_001() runs on MQTT_Client
{
  if(f_init("mqtt_client", "mqtt_server"))
  {
    f_TC_MQTT_BROKER_CONNECT_001();
  }
  f_cleanUp();
}
```

Let's have a deeper look into the details of a TC. The first block comment contains only two TTCN-3 documentation tags but these give us a direct connection between a TC, a TP, and the MQTT specification.

```
/*
* @purpose The IUT MUST close the network connection if fixed header flags in CONNECT␣
↪Control Packet are invalid
*
* @reference [MQTT-2.2.2-2], [MQTT-3.1.4-1], [MQTT-3.2.2-6]
*/
```

The signature of a TTCN-3 TC contains many information which help us to reflect the Test Architecture.

```
testcase TC_MQTT_BROKER_CONNECT_001() runs on MQTT_Client
```

We have a distinct name for the TC which can be easy mapped to it's cohesive TP. *(see in the MQTT \|tp\| catalogue for TP_MQTT_BROKER_CONNECT_001)* The signature tells us also, that this TC will be executed on a Client (`runs on MQTT_Client`).

The body of the TC is used to initialize the test configuration and start the TC behaviour which is wrapped into a single function.

```
  if(f_init("mqtt_client", "mqtt_server"))
  {
    f_TC_MQTT_BROKER_CONNECT_001();
  }
  f_cleanUp();
```

Test Case Functions

How does a TC function look like?

```
function f_TC_MQTT_BROKER_CONNECT_001() runs on MQTT_Client
{
  var UTF8EncodedString v_clientId := f_getClientId();

  var template MQTT_v3_1_1_Message v_conMsg := t_connect_flags(p_client_id := v_
↪clientId, p_flags := '1111'B);
  f_send(valueof(v_conMsg));

  if(f_receiveNetworkClosedEvent())
  {
    setverdict(pass, "IUT closed the Network Connection correctly");
  }
  else
  {
    setverdict(fail, "IUT MUST close the Network Connection");
    f_disconnectMqtt();
  }
}
```

While syntactically this function might appear confusing, though the semantic behind is quite powerful and simple. Let's go through the implemented TC and the according TP.

Translation Port

TODO: Explain translation port

Generic Functions

TODO: describe MQTT_Functions module

Test Templates

TODO: describe templates

Default Behaviours

TODO: describe the default behaviours

See also:

MQTT Interop Test Day in Burlingame, CA - March 17, 2014 The goal was to have as many different MQTT client and server implementations participate in interoperability testing to validate the implementation of the upcoming OASIS MQTT standard.

MQTT Interop Test Day in Ottawa, Canada – April 8, 2014 MQTT Test Day Demonstrates Successful Interoperability for the Internet of Things

MQTT Interop Test Day in Burlingame, CA - March 9, 2015 The goal was to have as many different MQTT client and server implementations participate in interoperability testing to validate the implementation of the OASIS MQTT 3.1.1 specification.

3.5.2 CoAP Test Suite

Contents

- *CoAP Test Suite*
  - *CoAP Protocol*
  - *Test Configurations*
  - *Test Purposes*
  - *Test System*
  - *Test Cases*
  - *Test Case Functions*
  - *Translation Port*
  - *Generic Functions*
  - *Test Templates*
  - *Default Behaviours*

CoAP Protocol

CoAP (Constrained Application Protocol) is a specialized Internet Application Protocol for constrained devices, as defined in R
The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks. The nodes often have 8-bit microcontrollers with small amounts of ROM and RAM, while constrained networks such as IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs) often have high packet error rates and a typical throughput of 10s of kbit/s. The protocol is designed for machine- to-machine (M2M) applications such as smart energy and building automation.

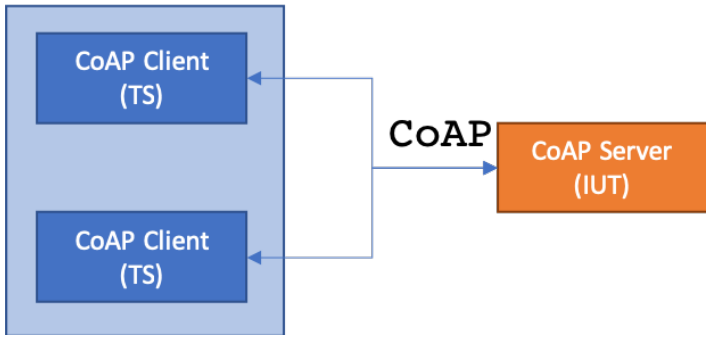
Note: We provide an *annotated version* of the official RFC which can be directly referenced (e.g. or)

Test Configurations

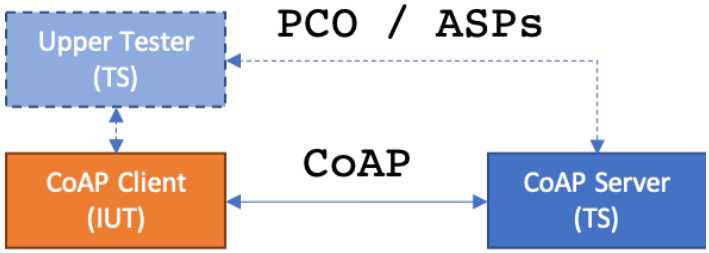
The concrete CoAP test configurations are listed below:

ID:	CoAP_Conf_01
Description:	The CoAP Server is the IUT and the TS takes the role of a CoAP Client

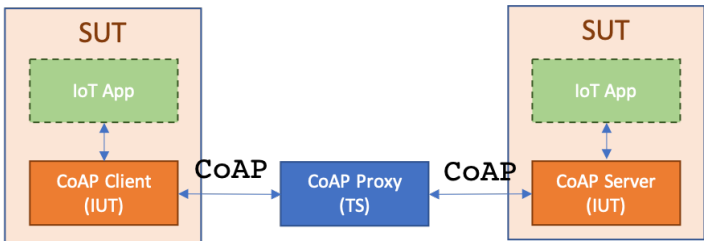
ID:	CoAP_Conf_02
Description:	The CoAP Server is the IUT and the TS takes the role of multiple CoAP Clients.



ID:	CoAP_Conf_03
Description:	The CoAP Client is the IUT and the TS takes the role of a CoAP Server. For this configuration an optional UT might be required.



ID:	CoAP_Conf_04
Description:	As well the CoAP Server as the CoAP Client, each is a IUT in this configuration. The part of the UT from the previous configuration is here replaced by a concrete application.



Test Purposes

TODO: link to .tplan2 from GitHub and .pdf from ETSI

```
Test Purpose {
   TP Id TP_CoAP_MessageFormat_Header_Version_001

   Test objective
   "The IUT is responding on a correctly set version number."

   Reference
   "RFC7252#section-3", "https://tools.ietf.org/html/rfc7252#section-3",
   "RFC7252#section-4.1", "https://tools.ietf.org/html/rfc7252#section-4.1",
   "RFC7252#section-4.2 (b)", "https://tools.ietf.org/html/rfc7252#section-4.2"

   PICS Selection PIC_Server

   Expected behaviour
   ensure that {
      when {
         the IUT entity receives a request message containing
            version indicating value 1,
            msg_type indicating value 0, //Confirmable
            token_length indicating value 0,
            code indicating value 0.00, //Empty Message
            msg_id corresponding to MSG_ID1;
      } then {
         the IUT entity sends a response message containing
            version indicating value 1,
            msg_type indicating value 3, //Reset
            token_length indicating value 0,
            code indicating value 0.00, //Empty Message
            msg_id corresponding to MSG_ID1;
         or the client entity times_out //from section 4.2 (b)
      }
   }
}
```

Test System

TODO: describe Test System -> TTCN-3 code

Test Cases

You can find all CoAP Tests on GitHub.

```
testcase TC_COAP_SERVER_001() runs on MTC_CT
{
  map(self:p, system:p);
  f_TC_COAP_SERVER_001();
  unmap(self:p, system:p);
}
```

Test Case Functions

TODO: describe a CoAP Test Case Function

```
function f_TC_COAP_SERVER_001() runs on MTC_CT
{
  f_sendMessage(m_coapPingMessage);
  f_receiveMessage(m_coapEmptyMessage);
}
```

Translation Port

TODO: Explain translation port

Generic Functions

TODO: describe MQTT_Functions module

Test Templates

TODO: describe templates

Default Behaviours

TODO: describe the default behaviours

See also:

CoAP Plugtests 1: Guide ETSI CTI Plugtests Guide (First Draft V0.0.16 2012-03) for achieving interoperability

CoAP Plugtests 1: Report The 1st CoAP Plugtest was held from 24 to 25 March 2012 in Paris, France and was co-located with IETF#83. This event was jointly organized by ETSI, IPSO Alliance and the FP7 Probe-IT project1.

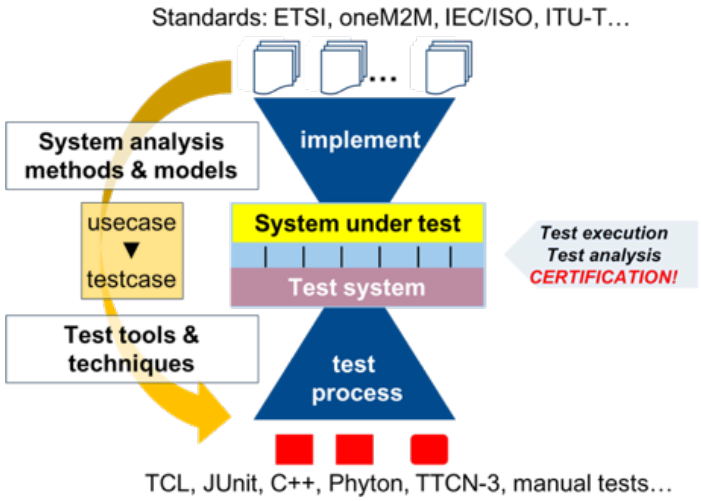
3.5.3 OPC-UA Test Suite

CHAPTER 4

ETSI and Eclipse Foundation

Test projects currently cover MQTT, CoAP, Lo-RaWAN and foundational security IoT-Profile of IEC 62443-4-2. The work of ETSI MTS-TST is correlated with the IoT-Testware which is hosted by the Eclipse Foundation. The technical contributions from the Eclipse members are coordinated by several dedicated Eclipse committers. The work includes Test purposes in TDL (primarily TDL-TO which is an extension of TDL for *Structured Test Objective Specification*) but also TTCN-3 test code developments that is important for test campaign execution in the test labs. In particular, ETSI members from MTS-TST control the test purposes developments and are responsible for the utilization of the resulting TP definitions for the ETSI working items and technical specifications. This approach allows to get input from active developers from the Eclipse community and a fast implementation of the target test suites for the interested industry but also support a faster develop-ment of ETSI specifications.

The illustrates an overview regarding two development procedures and its relationships: (a) the defini-tion and imple-mentation of the target system (under test) that needs to address subjects, assets and require-ments as well as threats, policies and assumptions, and (b) the test development including test architecture, test purposes and test suite structure. System and test engi-neers need to derive the test implementation to be executed and analysed, e.g. for certification purposes.



CHAPTER 5

Dashboard

Contents

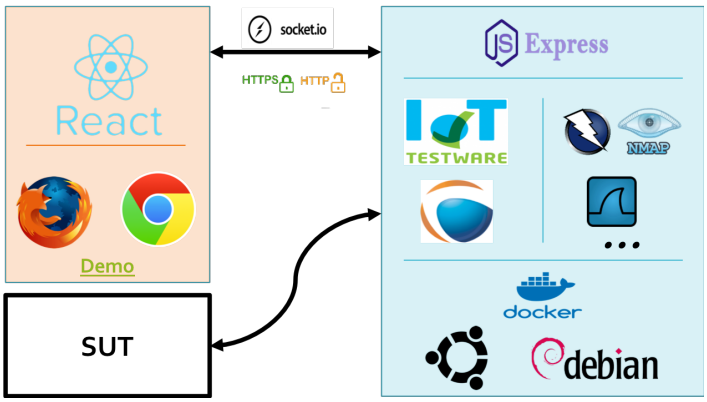
- *Dashboard*
  - *Introduction*
  - *Overview*
  - *Backend*
  - *Frontend*
  - *Integration*

5.1 Introduction

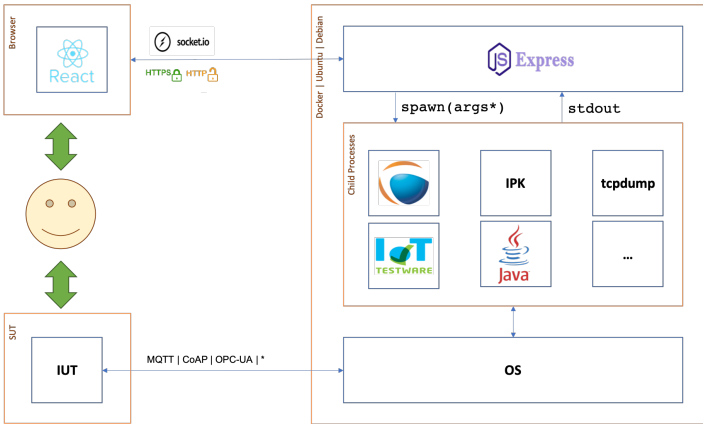
The **Eclipse IoT-Testware Dashboard** is a collection of several tools and test suites which makes it extremely flexible and powerful. Firstly the *Backend* which runs in background and provides the core functionality. Secondly, the *Frontend*, a thin ReactJS Application which provides a convenient user interface. And last but not least, the IoT-Testware with the test suites itself.

5.2 Overview

The first picture gives a brief overview about the basic idea.



Building upon the overview, the next picture is intended to give a more detailed view on the whole system.



5.3 Backend

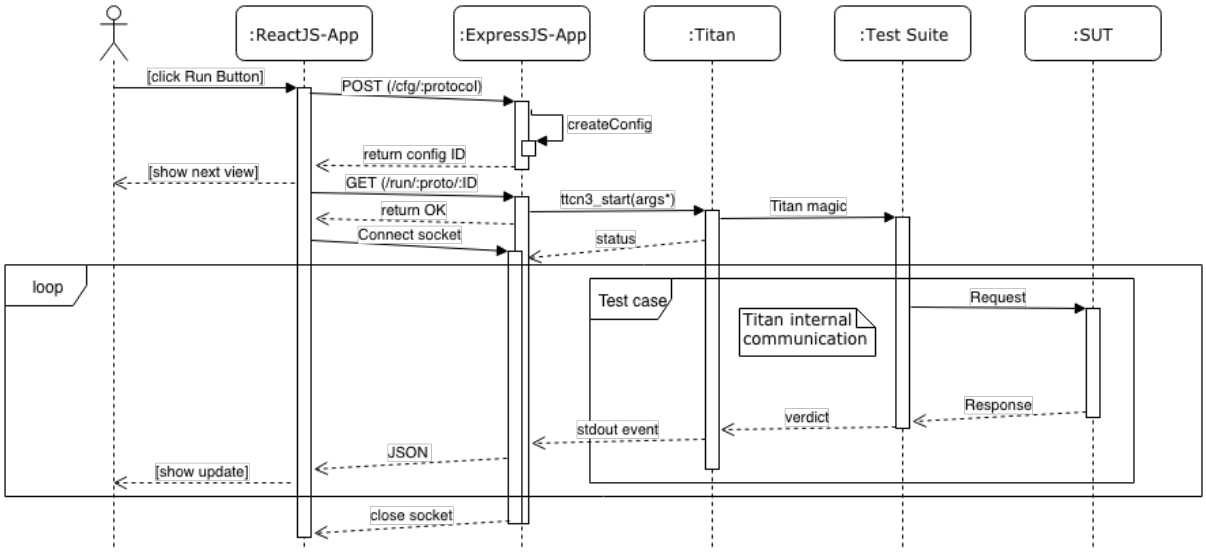
The backend component is a node.js application which is intended to abstract the “low-level” handling of testing tools and their configurations.

5.4 Frontend

The frontend component is a JavaScript application which serves as a user interface to the backend component. Through the nature of the architecture the frontend is intended to be a thin client.

5.5 Integration

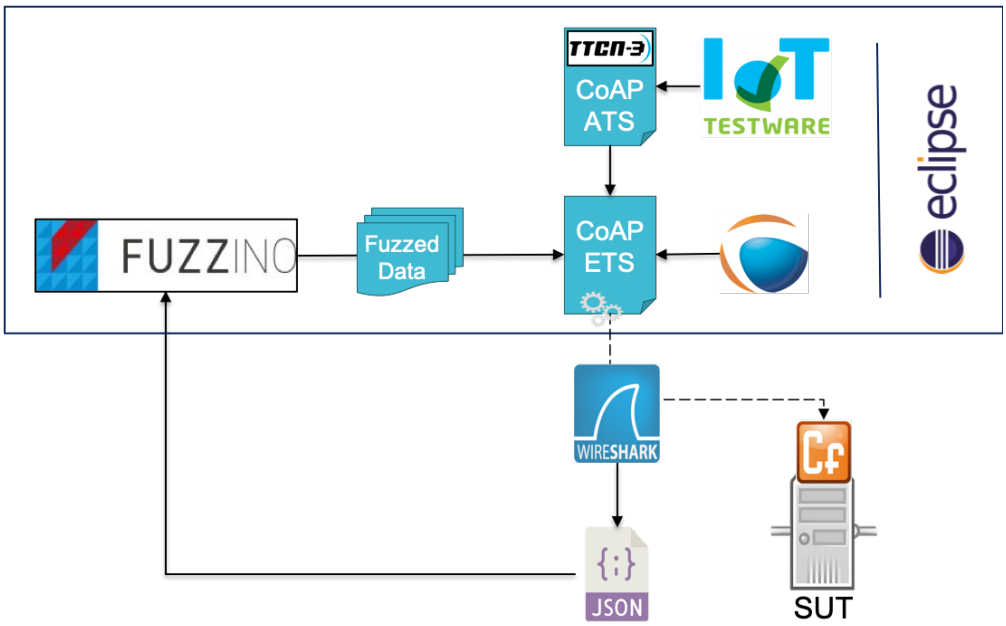
The sequence diagram below gives an overview about the interaction of involved components, starting from the user and ending with the SUT.


:ReactJS-App
:ExpressJS-App
:Titan
:Test Suite
:SUT
[click Run Button]
POST (/cfg/:protocol)
createConfig
return config ID
[show next view]
GET (/run/:proto/:ID
ttcn3_start(args*)
Titan magic
return OK
Connect socket
status
loop
Test case
Titan internal communication
Request
Response
verdict
stdout event
JSON
[show update]
close socket

CHAPTER 6

Fuzzing

TODO: What is fuzzing and how do we make use of fuzzing?



CHAPTER 7

Smart Fuzzing Proxy

Contents

- *Smart Fuzzing Proxy*
  - *Installation*
    - *Prerequisite*
    - *Set-up a Virtual Environment*
    - *Manual Set-up of a Virtual Environment*
    - *Quickstart*
  - *General Concept*
  - *Identify the SUT*
  - *Identify Input fields*
    - *Protocol Module*
      - *Protocol Message Examples*
    - *Fuzzdata Generators*
    - *Fuzzing Mutators*
      - *Unary Operators*
      - *Binary Operators*
    - *Filters*
      - *Filter Direction*
      - *Filter Definition*
    - *Rules*

        * *Checking the configuration*
    - *Choose a Test Data Generator*
    - *Running a Fuzzing Session*
    - *Analyzing Fuzzing Logging*

7.1 Installation

7.1.1 Prerequisite

Make sure the following prerequisites are met.

- Python (at least version 3, 3.7 is recommended)
- pip
- python3-venv

7.1.2 Set-up a Virtual Environment

The Fuzzing Proxy requires a set of external libraries which must be installed beforehand. To get you started as easy as possible, a helping Monkey (based on the Infinite monkey theorem) is provided to assist you with your fuzzing endeavor. Provided that your system meets the prerequisites, instruct the Monkey to setup a virtual environment for you:

```
./monkey setup
```

This will create a Python virtual environment in ./venv directory.

7.1.3 Manual Set-up of a Virtual Environment

The previously introduced Bash script provides an easy way to set up a virtual environment and installation of all required dependencies. However, the script might fail or you just might want to set up the virtual environment on your own. In this case the following steps will be required.

1. Create a virtual environment with `python3 -m venv {VENV_DIRECTORY}`
2. Activate the virtual environment with `source {VENV_DIRECTORY}/bin/activate`
3. Install `pipenv` for your virtual environment with `pip install pipenv`
4. Install all required libraries with `pipenv install`
    1. if installation fails with a `FileNotFoundError` probably `pdflatex` is missing as discussed on GitHub
    2. *Quickfix:* simply install LaTeX with e.g. `apt install texlive-latex-base` and repeat step 4

Note: If you choose to create your {VENV_DIRECTORY} other than ./venv the Monkey won't be any longer able to find your virtual environment. To overcome this issue you will either need change the directory in the `monkey` script or simply replace `monkey` with `python3 proxy.py` in the following examples.

7.1.4 Quickstart

To start the Fuzzing Proxy against `iot.eclipse.org:1883` with a sample configuration execute the following command:

```
./monkey fuzzing -l 1883 -r iot.eclipse.org -c resources/samples/mqtt_01.json
```

This will start the Fuzzing Proxy which will proxy the MQTT traffic from your local machine to `iot.eclipse.org` and vice versa.

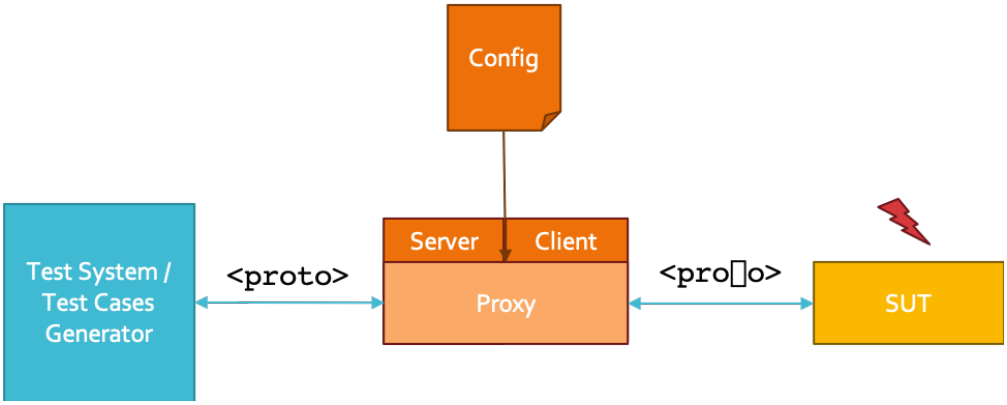
You can further explore the functionalities provided by the Fuzzing Proxy by using the built-in help.

```
./monkey --help
./monkey {command} --help
```

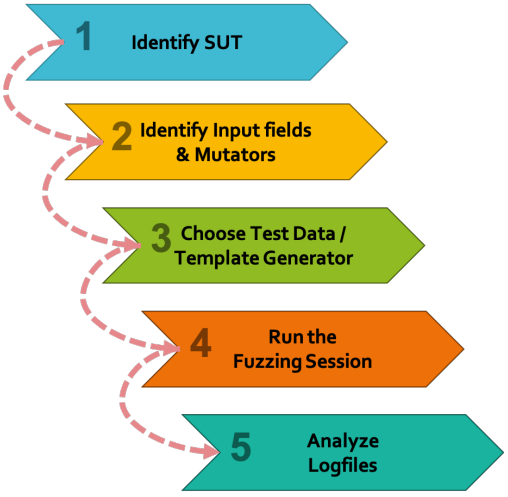
Important: Do not fuzz against production environments or systems other than your ones without permission.

7.2 General Concept

The Fuzzing Proxy is a **MITM** (Man-in-the-middle) **Fuzzer** which is capable of proxying the network traffic between to systems and altering this traffic on behalf of predefined rules. The Fuzzing Proxy does not generate any message on it's own. To trigger the fuzzing you need to provide (more or less) valid templates.



In the next sections we will show you how to use the Fuzzing Proxy by applying a *5 Steps Fuzzing Workflow* as shown in the following graphic.



7.3 Identify the SUT

Identifying the SUT (System Under Test) is the first step in the workflow.

Note: Needs to be further documented.

7.4 Identify Input fields

Identifying input fields and corresponding mutators for your fuzzing session is probably the most challenging part in the whole workflow. In this section we will demonstrate how interesting input fields can be chosen and corresponding mutators defined in the configuration file.

The configuration file provided to the Fuzzing Proxy contains abstract fuzzing instructions which are used at runtime for manipulation of proxied messages. This section will explain the basic concepts behind the configuration file and show you how to build a fuzzing scenario.

The configuration file is a plain JSON file following a specific schema. Below each single configuration block is described in detail.

7.4.1 Protocol Module

First, we will start with the `protocolModule` block which simply defines some basic information about the fuzzed protocol.

```
{
  "protocolModule": {
    "protocol": "MQTT",
    "encoding": "utf-8",
    "payload": "json" }
}
```

1. `protocol` obviously, defines which protocol will be used.
2. `encoding` defines how strings within the protocol fields should be handled.

3. `payload` defines how to handle payloads within messages. Currently, only `json` and `raw` are possible.

You can simply ask the Monkey to tell you the supported protocols:

```
./monkey protocols
```

Protocol Message Examples

For the next steps we will require some insights into the protocol chosen in the protocol module. Especially, the field namings and message structures provided by Scapy (which is used to decode and encode the messages) are required in the next steps. To get familiar with Scapy's representation of protocol messages, you can use some provided message examples.

Again, simply ask the Monkey to give you some examples for your chosen protocol:

```
./monkey samples mqtt
```

7.4.2 Fuzzdata Generators

The Fuzzing Proxy is capable of using multiple fuzzdata generators. Currently, only a basic random generator is implemented. However, the integration of different kinds of more sophisticated fuzz-data generators like Fuzzino might be implemented in the future.

```
{
  "generators": [
  { "id": "g_rand_uniform" },
  { "id": "g_rand_uniform_2" },
  { "id": "g_with_seed", "seed": 123 } ]
}
```

For now, each basic generator requires only an unique `id` which we will use later on to reference the generators. Additionallly, each generator can be initialised with a predefined `seed` for deterministic results.

Note: Generator IDs are enforced to start with `g_`

7.4.3 Fuzzing Mutators

Now things start getting interesting. Fuzzing Mutators are one of the basic concepts of the Fuzzing Proxy. The following block shows a set of different mutators, which will be explained in detail afterwards.

```
{
  "mutators": [
  { "id": "m_xor_protoname_fixed", "field": "protoname", "binary": "XOR", "hex":
↪"0xA5A5" },
  { "id": "m_inc_protolevel", "field": "protolevel", "unary": "INCR" },
  { "id": "m_inc_clientid_len", "field": "clientIdlen", "unary": "INCR" },
  { "id": "m_replace_clientid", "field": "clientId",  "binary": "SET", "generator":
↪"g_with_seed" },
  { "id": "m_replace_username", "field": "username",  "binary": "SET", "generator":
↪"g_rand_uniform_2" },
  { "id": "m_invert_username_flag", "field": "usernameflag", "unary": "NOT" },
  { "id": "m_invert_flags_dup", "field": "DUP", "unary": "NOT" },
  { "id": "m_invert_flags_qos", "field": "QOS", "unary": "NOT" }]
}
```

Note: Mutator IDs are enforced to start with `m_`

Generally, each mutator requires an unique `id` and is always bound to a specific message `field` with an operation. For now, two kinds of operations exist: `unary` and `binary` operations.

Unary Operators

Unary Operators, as the name implies, are unary with respect to the number of parameters which they expect. An unary operator expects only one single parameter. It takes the value of the specified field (as the one and only parameter) and applies a fuzzing operation on it. The following example of an `increment operator` is an unary operation without any additional parameter:

```
{ "id": "m_inc_protolevel", "field": "protolevel", "unary": "INCR" }
```

Or, if you want to think about the mutator in a more functional way:

```
function increment(value) => { return (value + 1) }

protolevel := increment(protolevel)
```

The class of Unary Operators has the following concrete operators:

- `INCR` which simply increments the given value by one
- `DECR` which simply decrements the given value by one
- `NOT` which simply inverts the given value

Binary Operators

Binary Operators on the other hand, take two parameters, the value of the specified field and either a fixed value or a generator as the second parameter. The following example of a `xor operator` is a binary operator with a fixed value:

```
{ "id": "m_xor_protoname_fixed", "field": "protoname", "binary": "XOR", "hex": "0xA5A5
↪" }
```

This mutator will take value of the field `protoname` and apply the `XOR` operator with a fixed hex value of `0xA5A5`. Or, if you want to think about the mutator in a more functional way:

```
function xor(value, mask) => { return (value ^ mask) }

protoname := xor(protoname, 0xA5A5)
```

Additionally, Binary Operators can be also provided with generators which generate dynamically (random) values. The following example of a `set operator` is a binary operator with a dynamically generated value:

```
{ "id": "m_replace_clientid", "field": "clientId",  "binary": "SET", "generator": "g_
↪with_seed" }
```

This mutator will take the value of the field `clientId` and appyl the `SET` operator with a dynamically generated value generated by the generator with the ID `g_with_seed`. Or, if you want to think about the mutator in a more functional way:

```
function set(value, generator) => { return generator.rand(typeof(value)) }

clientId := set(clientId, g_with_seed)
```

All Binary Operators can either be used with a fixed and predefined `hex`-value or with a `generator`. The example below is also perfectly valid:

```
{ "id": "m_replace_clientid", "field": "clientId",  "binary": "SET", "hex": "0xA5A5" }
{ "id": "m_xor_protoname_fixed", "field": "protoname", "binary": "XOR", "generator":
↪"g_with_seed" }
```

In a functional representation, these would look as follows:

```
function set(field, hex_mask) => { return hex_mask) }
function xor(field, generator) => { return (field ^ generator.rand(typeof(field)) }

clientId := set(clientId, 0xA5A5)
protoname := xor(protoname, g_with_seed)
```

The class of Binary Operators has the following concrete operators:

- `XOR` which applies an **xor** operation with the given second parameter on the given value
- `AND` which applies an **and** operation with the given second parameter on the given value
- `OR` which applies an **or** operation with the given second parameter on the given value
- `SET` which applies an **set** operation with the given second parameter on the given value

7.4.4 Filters

After mutators, filters are the second building blocks on the path of building fuzzing rules. These fuzzing filters are conceptually very similar to Wireshark's DisplayFilters and serve pretty much the same purpose. As one might want to intercept more complex protocol behaviours, altering each single message would be a bad idea. The concept of filters allows the user to pick only specific messages for fuzzing, while other message not matching any filter are simply passed through without being fuzzed. The given example below contains two filters which will be explained in more detail afterwards:

```
{
  "filters": [
  {
    "id": "f_all_connect",
    "description": "This filter matches all connect packets",
    "direction": "Request",
    "filter": { "field": "type", "op": "-eq", "value": 1 }
  }, {
    "id": "f_publish_subscribe",
    "descritpion": "Matches all publish or subscribe or unsubscribe packets",
    "direction": "All",
    "left": {
      "left": { "filter": { "field": "type", "op": "-eq", "value": 3 } },
      "op": "OR",
      "right": { "filter": { "field": "type", "op": "-eq", "value": 8 } } },
    "op": "OR",
    "right": { "filter": { "field": "type", "op": "-eq", "value": 10 } }
  } ]
}
```

Note: Filter IDs are enforced to start with `f_`

The first filter with the ID `f_all_connect` is a quite simple one and more or less self explanatory. However, the filter demonstrates the basic concept. While `id` and `description` are quite obvious, the `direction` and `filter` fields require a little more explanation.

Filter Direction

The `direction` field defines the direction of the filter. Valid directions are `Request`, `Response` and `All`. Filters with `"direction": "Request"` will be only applied on messages passing through the Fuzzing Proxy from the client to the server. On the other hand, filters with `"direction": "Response"` will only look for responses coming from the server. And obviously, `"direction": "All"` will look for both directions.

Filter Definition

The filter field is actually the part which defines the filtering criteria. The following example filter matches all packets of the `type = 1` (the CONNECT Control Packet in case of MQTT):

```
{ "filter": { "field": "type", "op": "-eq", "value": 1 } }
```

Conceptually, this filter can be thought of as follows:

But filters can be more complex and contain multiple filtering criteria as demonstrated by the filter with the ID `f_publish_subscribe`: This filter contains a combination of three simple filters.

```
{
  "left": {
    "left": { "filter": { "field": "type", "op": "-eq", "value": 3 } },
    "op": "OR",
    "right": { "filter": { "field": "type", "op": "-eq", "value": 8 } } },
  "op": "OR",
  "right": { "filter": { "field": "type", "op": "-eq", "value": 10 } }
}
```

Conceptually, this filter can be thought of as follows:

As one can see, we have two types of operators (`op`) in filters. The comparison operators are used within concrete `filter` definitions as shown below:

Valid comparison operators are:

- `-eq` which stands for ==
- `-ne` which stands for !=
- `-gt` which stands for >
- `-lt` which stands for <
- `-ge` which stands for >=
- `-le` which stands for <=

The logical operators, on the other hand, are used to combine multiple single filter logically.

Valid logical operators are: `OR`, and `AND`

7.4.5 Rules

Finally, the Rules Engine can be build by combining mutators and filters to concrete fuzzing rules. The given example below contains two rules which will be explained in more detail afterwards:

```
{
  "rules": [
  {
    "match": "f_all_connect",
    "mutators": [ "m_inc_clientid_len", "m_xor_protoname_fixed", "m_inc_protolevel" ]
  }, {
    "match": "f_publish_subscribe",
    "distribution": {
      "model": "multinomial",
      "seed": 12345,
      "nxp": 10
      "items": [
        { "strength": 11, "mutators": [ "m_invert_flags_retain", "m_inc_flags_qos",
↪"m_or_topic" ] },
        { "strength": 12, "mutators": [ "m_invert_flags_retain", "m_xor_topic", "m_
↪inc_pl_length" ] },
        { "strength": 13, "mutators": [ "m_inc_pl_length" ] },
        { "strength": 14, "mutators": [ ] }
      ] }
  } ]
}
```

The first rule is a quite simple one and contains two fields. The `match` field references a matching filter previously defined in *Filters* and a set of `mutators` previously defined in *Fuzzing Mutators*. You can think of the first rule as follows: *Once the matching filter with the ID* `f_all_connect` *matches a message, apply all three given mutators with the IDs* `m_inc_clientid_len`, `m_xor_protoname_fixed` *and* `m_inc_protolevel` *to this message.*

The second rule is a little more complex, though more flexible and powerful with regards to it's capabilities to manipulate protocol messages. Once again, the `match` field references a matching filter. However, this time the rule does not contain a simple fixed list of `mutators` but rather a `distribution` of several lists. Let's have a closer look at the distribution block. The `model` field defines the kind of the distribution which will be used to choose one of the `items`. Currently, `multinomial` distribution is the only one possible. (The concrete implementation uses the numpy.random.multinomial module.) The distribution can also be configured with a `seed` and `nxp` (Number of experiments). However, these fields are optional and can be omitted. The `items` list represents all the possibilities the distribution can choose from. Each set of mutators within a distribution is not necessarily equally weighted. That means, you can favor specific sets of mutators and disfavor others via the `strength` field. Thereby applies, the higher the strength, the higher the probability to be chosen. In concrete, in the example above the distribution has four items with an overall strength of $11+12+13+14=50$. That means, the probability to get the empty list of mutators (the last one, which in fact won't manipulate any fields at all) is $p_4 = \frac{14}{50} = 0.28$ whereas the probability to get the first list is only $p_1 = \frac{11}{50} = 0.22$ and so on.

Warning: The total number of rules/filters should be kept as small as possible. In worst case scenarios each single filter must be checked for each message. Therefore, messages which do not match any filter (or the last one) need to be checked agains each single filter. The filter mechanism has a $O(n)$ time complexity. This might lead to performance issues on large sets of filters.

Note: The total number of items within a distribution has a $O(1)$ time complexity, which means, you can theoretically have as many items as you need without influencing the runtime performance.

7.4.6 Checking the configuration

A fuzzing configuration can be become quite complex and confusing which might lead to errors at runtime. To ensure that your configuration is valid, the `validate` command comes handy. Once you have finished your configuration, ask the Monkey to validate your configuration:

```
./monkey validate {CONFIGURATION}
```

Where `{CONFIGURATION}` can be either a relative or absolute path to your configuration file.

7.5 Choose a Test Data Generator

As testcase generator you can either use a simple client of your choice or use an according IoT-Testware conformance test suite.

7.6 Running a Fuzzing Session

Once you have build and validated your configuration you are ready to start fuzzing. The Fuzzing Proxy can help you to start the Fuzzing Proxy:

```
./monkey fuzzing -l {LISTEN_PORT} -r {REMOTE_ADDRESS} -c {CONFIGURATION}
```

Where `{LISTEN_PORT}` is the port on which the fuzzing proxy will listen for incoming requests and forward to `{REMOTE_ADDRESS}:{LISTEN_PORT}` with considering the rules given in `{CONFIGURATION}`. The following sketch demonstrates the following fuzzing setup: `0.0.0.0:{LISTEN_PORT} <-> {REMOTE_ADDRESS}:{LISTEN_PORT}`

If you need to further configure the ports and interfaces you can provide parameters as follows:

```
./monkey fuzzing -l {LOCAL_LISTEN_PORT} -r {REMOTE_ADDRESS} -p {REMOTE_LISTEN_
↪PORT} -c {CONFIGURATION}
```

This will result in the following fuzzing setup: `0.0.0.0:{LOCAL_LISTEN_PORT} <-> {REMOTE_ADDRESS}:{REMOTE_LISTEN_PORT}`

To stop the Fuzzing Proxy press `Ctrl + C`

7.7 Analyzing Fuzzing Logging

Once you have finished a fuzzing session, you will find within the `logs` folder a newly created folder with a timestamped naming e.g. `20190822_13_38_24` (which is in fact the datetime when the Fuzzing Proxy was started). Inside of this folder you will find at least the two log files `fuzzing_operations.log` and `proxy_traffic.log` which can be used to analyze the fuzzing session. In general, the `proxy_traffic.log` contains all events happened, mainly incoming and outgoing messages and network events. On the other hand, the `fuzzing_operations.log` contains all events which were executed by the fuzzer during the reception and the forwarding of each single messages.

Let's have a closer look at an exemplary log file of a MQTT session. First we will look into the `proxy_traffic.log` and go through the snipped line by line:

In line 1 we see a network event (TCP connection) coming from a client (`127.0.0.1:58636`) and on line 2 the client sent immediatelly a message over this connection. This message is a MQTT Connect Control Packet triggered by the client to request a (MQTT) connection to the server. If you like to look into the MQTT message, ask the Monkey to decode the bytestream for you:

```
./monkey decode -p mqtt -o '101c00044d5154540402003c00104d5154545f46585f436c69656e745f32'O
```

So far so good, everything went fine so far and the fuzzer didn't intercept. In line 3 we can observe how the Fuzzing Proxy proxies the original network event (TCP connection) to the SUT (`192.168.56.101:1883`) and forwards the original, though manipulated, MQTT Connect Control Packet. Finally, the SUT reacts with an error handling routine by simply closing the network connection due to a protocol violation. So, what happend inside the Fuzzing Proxy and what was the reason for the protocol violation? First, we might take a closer look at the incoming and outgoing MQTT packets:

Well, didn't change that much. Let's consult the `fuzzing_operations.log` to figure out what happened in between. There we will find a line like the following one:

First of all, we can acknowledge this fuzzing operation took place in between the reception and forwarding of the MQTT packet by comparing the timestamps. Second, by following the `[f_all_connect.3]` hint, we can clearly traceback to applied rule, matching filter and set of mutators. And finally, with `[NOT(DUP)=0->1; NOT(QOS)=0->1]` we get a summary of the complete set of mutators applied to this message.

In the following code block we can see the corresponding rule and the marked item:

```
{
  "match": "f_all_connect",
  "distribution": {
    "model": "multinomial",
    "seed": 3456,
    "nxp": 10,
    "items": [
      { "strength": 1, "mutators": [ "m_xor_protoname_fixed", "m_inc_protolevel" ] },
      { "strength": 12, "mutators": [ "m_replace_username", "m_invert_username_flag"␣
↪] },
      { "strength": 15, "mutators": [ "m_inc_clientid_len", "m_replace_clientid" ] },
      { "strength": 7, "mutators": [ "m_invert_flags_dup", "m_invert_flags_qos" ] },
      { "strength": 15, "mutators": [ ] }
    ] }
}
```

By having this information, we can now even further precise the protocol violation.

However, it should be noted, that this example is not a perfect fuzzing example as the broker responded perfectly conformant according to the specification without crashing or exposing any weaknesses. But this simple example should only serve as an illustration of the capabilities of the fuzzer and demonstrate the workflow.

CHAPTER 8

About Eclipse IoT-Testware

8.1 IoT-Testware Team

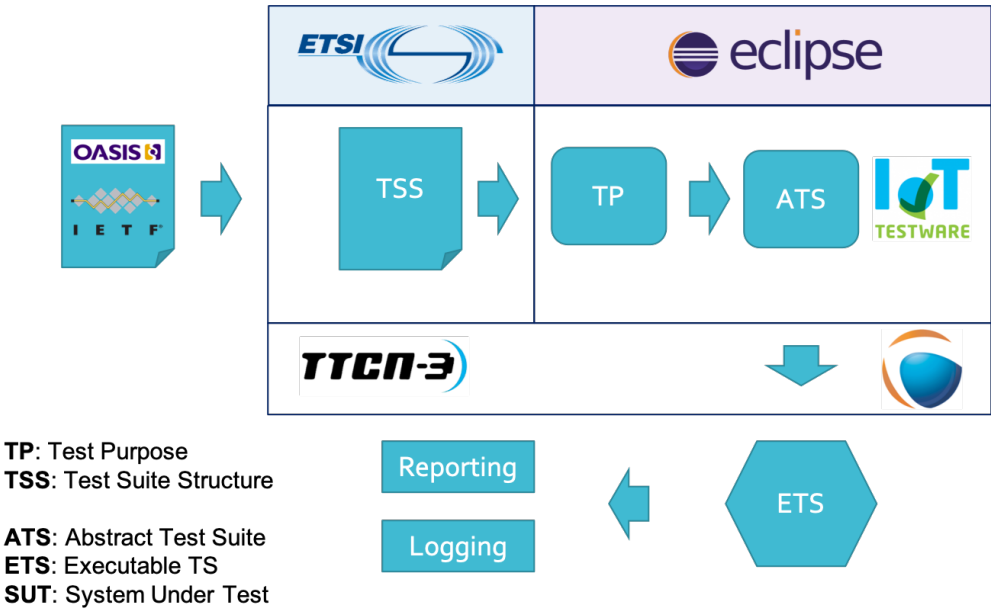
Who's involved

8.2 Objective

As stated in [MVQ18] communication protocols for the IoT are currently in a immature state and offer different kinds of attack vectors. We believe… *TODO!*

8.3 Conformance Test Methodology and Framework

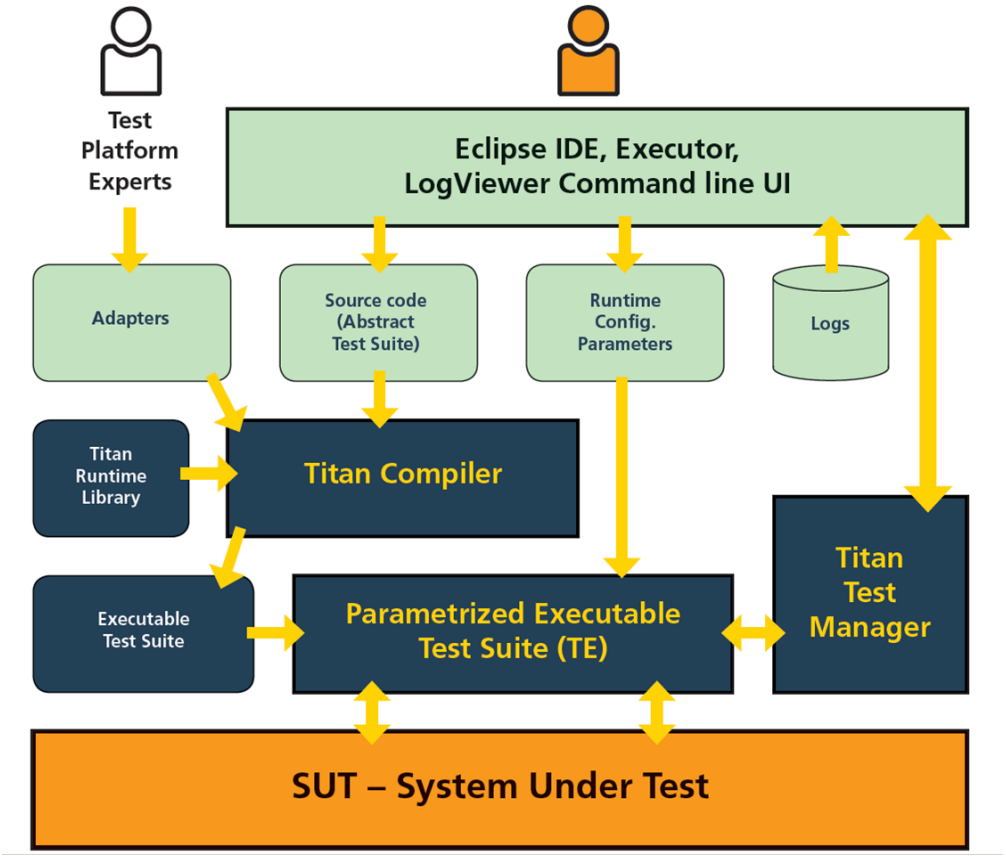
The IoT-Testware test suites will have a well-defined test suite structure (TSS) and a set of protocol implementation conformance statements (PICS) as well as protocol implementation extra information for testing (PIXIT). The work will follow the standardized approach as defined in ISO "Conformance Test Methodology and Framework" ISO 9646 and the best practices as described by ETSI White Paper No 3 "Achieving Technical Interoperability – the ETSI Approach".



8.4 Implementation

The Eclipse IoT-Testware project provides standardized Abstract Test Suite (ATS) for popular IoT protocols. For the implementation of the ATS for CoAP and MQTT we apply ETSI Test Methodology which is well-proven in standardizing and testing of telecommunication systems.

Such an ATS contains of several parts which are required to implement the Conformance Test Methodology and Framework. But ATS, as the name says, are abstract, which means we need a system which executes the ATS. Just like Java code requires the JVM to be executed, an ATS requires in our case a TTCN-3 runtime. As our TTCN-3 runtime we have chosen Eclipse Titan which can compile and run our ATS.



The Executable Test Suite (ETS) is, as the name states, is a test suite under execution, just like running Java code.

CHAPTER 9

Glossary

ASP Abstract Service Primitive

ATS Abstract Test Suite

CoAP The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks.

ETS Executable Test Suite

ETSI The European Telecommunications Standards Institute is an independent, not-for-profit, standardization organization in the telecommunications industry (equipment makers and network operators) in Europe, headquartered in Sophia-Antipolis, France, with worldwide projection.

IoT see *Internet of Things*

Internet of Things The internet of things, or IoT, is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction.

IUT Implementation Under Test

MQTT The MQ Telemetry Transport is a machine-to-machine (M2M) / *Internet of Things* connectivity protocol. It was designed as an extremely lightweight publish/subscribe messaging transport. It is useful for connections with remote locations where a small code footprint is required and/or network bandwidth is at a premium.

PCO Point of Control and Observation

PDU Protocol Data Unit

SUT System Under Test

Test Case A test case is...

Test Purpose A test purpose is...

TC see Test Case

TDL Test Description Language (TDL) is a new language for the specification of test descriptions and the presentation of test execution results.

Thing The Thing in the context of IoT is an entity which is connected to the IoT and consumes or provides digital services.

TP Test Purpose

TS Test System

TSS Test Suite Structure

TTCN-3 Testing and Test Control Notation version 3 is a standardized, modular testing language specifically designed for testing communication systems.

UT Upper Tester

CHAPTER 10

References

10.1 2017

[HKR17] [KKR17] [RennochAxel17] [SKRW17] [SR17]

10.2 2018

[Sch18] [RH18] [RHK18]

10.3 2019

[RK19] [RK19]

CHAPTER 11

License

```
Eclipse Public License - v 1.0

THE ACCOMPANYING PROGRAM IS PROVIDED UNDER THE TERMS OF THIS ECLIPSE PUBLIC
LICENSE ("AGREEMENT"). ANY USE, REPRODUCTION OR DISTRIBUTION OF THE PROGRAM
CONSTITUTES RECIPIENT'S ACCEPTANCE OF THIS AGREEMENT.

1. DEFINITIONS

"Contribution" means:

a) in the case of the initial Contributor, the initial code and documentation
   distributed under this Agreement, and
b) in the case of each subsequent Contributor:
    i) changes to the Program, and
   ii) additions to the Program;

   where such changes and/or additions to the Program originate from and are
   distributed by that particular Contributor. A Contribution 'originates'
   from a Contributor if it was added to the Program by such Contributor
   itself or anyone acting on such Contributor's behalf. Contributions do not
   include additions to the Program which: (i) are separate modules of
   software distributed in conjunction with the Program under their own
   license agreement, and (ii) are not derivative works of the Program.

"Contributor" means any person or entity that distributes the Program.

"Licensed Patents" mean patent claims licensable by a Contributor which are
necessarily infringed by the use or sale of its Contribution alone or when
combined with the Program.

"Program" means the Contributions distributed in accordance with this
Agreement.

"Recipient" means anyone who receives the Program under this Agreement,
```

(continues on next page)

(continued from previous page)

```
including all Contributors.

2. GRANT OF RIGHTS
  a) Subject to the terms of this Agreement, each Contributor hereby grants
     Recipient a non-exclusive, worldwide, royalty-free copyright license to
     reproduce, prepare derivative works of, publicly display, publicly
     perform, distribute and sublicense the Contribution of such Contributor,
     if any, and such derivative works, in source code and object code form.
  b) Subject to the terms of this Agreement, each Contributor hereby grants
     Recipient a non-exclusive, worldwide, royalty-free patent license under
     Licensed Patents to make, use, sell, offer to sell, import and otherwise
     transfer the Contribution of such Contributor, if any, in source code and
     object code form. This patent license shall apply to the combination of
     the Contribution and the Program if, at the time the Contribution is
     added by the Contributor, such addition of the Contribution causes such
     combination to be covered by the Licensed Patents. The patent license
     shall not apply to any other combinations which include the Contribution.
     No hardware per se is licensed hereunder.
  c) Recipient understands that although each Contributor grants the licenses
     to its Contributions set forth herein, no assurances are provided by any
     Contributor that the Program does not infringe the patent or other
     intellectual property rights of any other entity. Each Contributor
     disclaims any liability to Recipient for claims brought by any other
     entity based on infringement of intellectual property rights or
     otherwise. As a condition to exercising the rights and licenses granted
     hereunder, each Recipient hereby assumes sole responsibility to secure
     any other intellectual property rights needed, if any. For example, if a
     third party patent license is required to allow Recipient to distribute
     the Program, it is Recipient's responsibility to acquire that license
     before distributing the Program.
  d) Each Contributor represents that to its knowledge it has sufficient
     copyright rights in its Contribution, if any, to grant the copyright
     license set forth in this Agreement.

3. REQUIREMENTS

A Contributor may choose to distribute the Program in object code form under
its own license agreement, provided that:

  a) it complies with the terms and conditions of this Agreement; and
  b) its license agreement:
      i) effectively disclaims on behalf of all Contributors all warranties
         and conditions, express and implied, including warranties or
         conditions of title and non-infringement, and implied warranties or
         conditions of merchantability and fitness for a particular purpose;
     ii) effectively excludes on behalf of all Contributors all liability for
         damages, including direct, indirect, special, incidental and
         consequential damages, such as lost profits;
    iii) states that any provisions which differ from this Agreement are
         offered by that Contributor alone and not by any other party; and
     iv) states that source code for the Program is available from such
         Contributor, and informs licensees how to obtain it in a reasonable
         manner on or through a medium customarily used for software exchange.

When the Program is made available in source code form:

  a) it must be made available under this Agreement; and
```

(continues on next page)

(continued from previous page)

```
  b) a copy of this Agreement must be included with each copy of the Program.
     Contributors may not remove or alter any copyright notices contained
     within the Program.

Each Contributor must identify itself as the originator of its Contribution,
if
any, in a manner that reasonably allows subsequent Recipients to identify the
originator of the Contribution.

4. COMMERCIAL DISTRIBUTION

Commercial distributors of software may accept certain responsibilities with
respect to end users, business partners and the like. While this license is
intended to facilitate the commercial use of the Program, the Contributor who
includes the Program in a commercial product offering should do so in a manner
which does not create potential liability for other Contributors. Therefore,
if a Contributor includes the Program in a commercial product offering, such
Contributor ("Commercial Contributor") hereby agrees to defend and indemnify
every other Contributor ("Indemnified Contributor") against any losses,
damages and costs (collectively "Losses") arising from claims, lawsuits and
other legal actions brought by a third party against the Indemnified
Contributor to the extent caused by the acts or omissions of such Commercial
Contributor in connection with its distribution of the Program in a commercial
product offering. The obligations in this section do not apply to any claims
or Losses relating to any actual or alleged intellectual property
infringement. In order to qualify, an Indemnified Contributor must:
a) promptly notify the Commercial Contributor in writing of such claim, and
b) allow the Commercial Contributor to control, and cooperate with the
Commercial Contributor in, the defense and any related settlement
negotiations. The Indemnified Contributor may participate in any such claim at
its own expense.

For example, a Contributor might include the Program in a commercial product
offering, Product X. That Contributor is then a Commercial Contributor. If
that Commercial Contributor then makes performance claims, or offers
warranties related to Product X, those performance claims and warranties are
such Commercial Contributor's responsibility alone. Under this section, the
Commercial Contributor would have to defend claims against the other
Contributors related to those performance claims and warranties, and if a
court requires any other Contributor to pay any damages as a result, the
Commercial Contributor must pay those damages.

5. NO WARRANTY

EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, THE PROGRAM IS PROVIDED ON AN
"AS IS" BASIS, WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, EITHER EXPRESS OR
IMPLIED INCLUDING, WITHOUT LIMITATION, ANY WARRANTIES OR CONDITIONS OF TITLE,
NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Each
Recipient is solely responsible for determining the appropriateness of using
and distributing the Program and assumes all risks associated with its
exercise of rights under this Agreement , including but not limited to the
risks and costs of program errors, compliance with applicable laws, damage to
or loss of data, programs or equipment, and unavailability or interruption of
operations.

6. DISCLAIMER OF LIABILITY
```

(continues on next page)

(continued from previous page)

```
EXCEPT AS EXPRESSLY SET FORTH IN THIS AGREEMENT, NEITHER RECIPIENT NOR ANY
CONTRIBUTORS SHALL HAVE ANY LIABILITY FOR ANY DIRECT, INDIRECT, INCIDENTAL,
SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING WITHOUT LIMITATION
LOST PROFITS), HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
ARISING IN ANY WAY OUT OF THE USE OR DISTRIBUTION OF THE PROGRAM OR THE
EXERCISE OF ANY RIGHTS GRANTED HEREUNDER, EVEN IF ADVISED OF THE POSSIBILITY
OF SUCH DAMAGES.

7. GENERAL

If any provision of this Agreement is invalid or unenforceable under
applicable law, it shall not affect the validity or enforceability of the
remainder of the terms of this Agreement, and without further action by the
parties hereto, such provision shall be reformed to the minimum extent
necessary to make such provision valid and enforceable.

If Recipient institutes patent litigation against any entity (including a
cross-claim or counterclaim in a lawsuit) alleging that the Program itself
(excluding combinations of the Program with other software or hardware)
infringes such Recipient's patent(s), then such Recipient's rights granted
under Section 2(b) shall terminate as of the date such litigation is filed.

All Recipient's rights under this Agreement shall terminate if it fails to
comply with any of the material terms or conditions of this Agreement and does
not cure such failure in a reasonable period of time after becoming aware of
such noncompliance. If all Recipient's rights under this Agreement terminate,
Recipient agrees to cease use and distribution of the Program as soon as
reasonably practicable. However, Recipient's obligations under this Agreement
and any licenses granted by Recipient relating to the Program shall continue
and survive.

Everyone is permitted to copy and distribute copies of this Agreement, but in
order to avoid inconsistency the Agreement is copyrighted and may only be
modified in the following manner. The Agreement Steward reserves the right to
publish new versions (including revisions) of this Agreement from time to
time. No one other than the Agreement Steward has the right to modify this
Agreement. The Eclipse Foundation is the initial Agreement Steward. The
Eclipse Foundation may assign the responsibility to serve as the Agreement
Steward to a suitable separate entity. Each new version of the Agreement will
be given a distinguishing version number. The Program (including
Contributions) may always be distributed subject to the version of the
Agreement under which it was received. In addition, after a new version of the
Agreement is published, Contributor may elect to distribute the Program
(including its Contributions) under the new version. Except as expressly
stated in Sections 2(a) and 2(b) above, Recipient receives no rights or
licenses to the intellectual property of any Contributor under this Agreement,
whether expressly, by implication, estoppel or otherwise. All rights in the
Program not expressly granted under this Agreement are reserved.

This Agreement is governed by the laws of the State of New York and the
intellectual property laws of the United States of America. No party to this
Agreement will bring a legal action under this Agreement more than one year
after the cause of action arose. Each party waives its rights to a jury trial in
any resulting litigation.
```

CHAPTER 12

MQTT Specification

CHAPTER 13

CoAP RFC 7252

CHAPTER 14

Indices and tables

- genindex
- modindex
- search

Bibliography

[HKR17] Sascha Hackel, Dorian Knobloch, and Axel Rennoch. Qualitätsanalyse mit IoT-Testware. url: https://gi.de/fileadmin/FG/TAV/42.TAV/42_GI-TAV_paper_1.pdf, 2017.

[KKR17] Alexander Kaiser, Sascha Kretzschmann, and Axel Rennoch. Eclipse IoT-Testware: Die Open-Source-Testsuite für das MQTT-Protokoll. url: http://www.sigs.de/public/ots/2017/OTS_Testing_2017/Kaiser_OTS_Testing_2017.pdf, 2017.

[MVQ18] Federico Maggi, Rainer Vosseler, and Davide Quarta. The Fragility of Industrial IoT's Data Backbone. *Trend Micro Research White Paper*, pages 65, 2018.

[RH18] Axel Rennoch and Sascha Hackel. Quality analysis with IoT-Testware. url: http://publica.fraunhofer.de/eprints/urn_nbn_de_0011-n-4971018.pdf, 2018.

[RHK18] Axel Rennoch, Sascha Hackel, and Dorian Knobloch. Test Execution Infrastructure for IoT Quality Analysis. url: http://vvass2018.ist.tugraz.at/wp-content/uploads/2018/08/IoT-T_Testware_VVASS.pdf, 2018.

[RK19] Axel Rennoch and Alexander Kaiser. Functional and non-functional testing for the IoT. url: http://publica.fraunhofer.de/eprints/urn_nbn_de_0011-n-5344321.pdf, 2019.

[SR17] Schieferdecker, Ina and Axel Rennoch. IoT Testing and the Eclipse IoT Testware. url: https://paris.utdallas.edu/qrs17/docs/Keynote-Ina-Schieferdecker-slides.pdf, 2017.

[Sch18] Ina Schieferdecker. IoT Testware - Modern Automated Test Solutions for IoT Components and Systems. 2018.

[SKRW17] Ina Schieferdecker, Sascha Kretzschmann, Axel Rennoch, and Michael Wagner. IoT-testware - an eclipse project. url: https://ieeexplore.ieee.org/document/8009903, 2017.

[RennochAxel17] Rennoch, Axel. Test automation for the iot: iot-testware. url: https://www.iotcamp.net/fileadmin/iot-barcamp/IoTcamp2017_FOKUS-Rennoch.pdf, 2017.

Index

A

ASP, **55**
ATS, **55**

C

CoAP, **55**

E

ETS, **55**
ETSI, **55**

I

Internet of Things, **55**
IoT, **55**
IUT, **55**

M

MQTT, **55**

P

PCO, **55**
PDU, **55**

S

SUT, **55**

T

TC, **55**
TDL, **55**
Test Case, **55**
Test Purpose, **55**
Thing, **56**
TP, **56**
TS, **56**
TSS, **56**
TTCN-3, **56**

U

UT, **56**