



Horizon 2020 Program  
Dynamic countering of cyber-attacks  
SU-ICT-2018



Cyber security 4.0: Protecting the Industrial Internet of Things

**D2.2: Deep learning breakthroughs and security-aware dynamic offloading mechanisms<sup>†</sup>**

**Abstract:**

This deliverable is a demonstration deliverable of the components related to Tasks T2.2 and T2.3. This is a brief report describing the operation, the development of the components so far and the initial results planned for this stage of the project.

Contractual Date of Delivery	31/05/2020
Actual Date of Delivery	26/05/2020
Deliverable Security Class	Public
Editor	Georgia Sakellari (UOG)
Contributors	UOG (Georgia Sakellari, William Oliff), UNSPMF (Miloš Savić, Milan Lukić), FORTH (Giorgos Tsirantonakis), IFAG (Antonio Javier Cabrera Gutierrez), ITML (Eleftheria Marini, George Bravos), VIP(Dragan Danilović, Zoran Gajica)
Quality Assurance	AEGIS (Efstathios Dimakos), UP1PS (Jacques Robin)

---

<sup>†</sup> The research leading to these results has received funding from the European Union's Horizon 2020 research and innovation programme under grant agreement No 833828.

### The *C4IIoT* Consortium

FOUNDATION FOR RESEARCH AND TECHNOLOGY HELLAS	FORTH	Coordinator	EL
CENTRO RICERCHЕ FIAT SCPA	CRF	Principal Contractor	IT
INFINEON TECHNOLOGIES AG	IFAG	Principal Contractor	DE
THALES SIX GTS FRANCE SAS	TSG	Principal Contractor	FR
HEWLETT PACKARD ITALIANA SRL	HPE	Principal Contractor	IT
COMMISSARIAT A L ENERGIE ATOMIQUE ET AUX ENERGIES ALTERNATIVES	CEA	Principal Contractor	FR
IBM ISRAEL - SCIENCE AND TECHNOLOGY LTD	IBM	Principal Contractor	IL
AEGIS IT RESEARCH UG	AEGIS	Principal Contractor	DE
UNIVERSITE PARIS I PANTHEON-SORBONNE	UP1PS	Principal Contractor	FR
INFORMATION TECHNOLOGY FOR MARKET LEADERSHIP	ITML	Principal Contractor	EL
SPHYNX TECHNOLOGY SOLUTIONS AG	STS	Principal Contractor	CH
UNIVERSITY OF NOVI SAD FACULTY OF SCIENCES	UNSPMF	Principal Contractor	SRB
UNIVERSITY OF GREENWICH	UOG	Principal Contractor	UK
VIP MOBILE D.O.O.	VIP	Principal Contractor	SRB

## Document Revisions & Quality Assurance

### Internal Reviewers

1. AEGIS
2. UP1PS

### Revisions

Version	Date	By	Overview
0.1	26/03/2020	UOG (Georgia Sakellari)	Initial ToC
0.2	27/03/2020	ITML	Input from ITML
0.3	10/04/2020	UNSPMF	Input from UNSPMF
0.4	15/04/2020	FORTH	Input from FORTH
0.5	24/04/2020	UOG	Input from UOG
0.6	07/05/2020	FORTH, ITML	Input from FORTH and ITML
0.7	08/05/2020	IFAG, VIP	Input from IFAG and VIP
0.8	10/05/2020	UOG	Draft prepared for internal review
0.9	18/05/2020	AEGIS, UP1PS	Draft after internal review
0.95	23/05/2020	All contributors	Reviewers' comments addressed by partners
1.0	26/05/2020	UOG	Final report

## Table of Contents

<b>LIST OF FIGURES .....</b>	<b>5</b>
<b>LIST OF ABBREVIATIONS.....</b>	<b>6</b>
<b>EXECUTIVE SUMMARY .....</b>	<b>7</b>
<b>1 INTRODUCTION .....</b>	<b>8</b>
<b>2 DEEP LEARNING TRAINED MODELS DEPLOYED AT THE EDGE .....</b>	<b>9</b>
2.1 BEHAVIORAL ANALYSIS AND COGNITIVE SECURITY (BACS) TOOL .....	9
2.2 TRAFFIC ANALYSIS TOOL.....	15
2.3 MACHINE LEARNING ANOMALY DETECTION AT THE EDGE – 3ACEs TOOL .....	18
2.4 SECURE HARDWARE SUPPORT TOOLS .....	20
<b>3 SECURITY-AWARE DYNAMIC OFFLOADING DECISION MECHANISM .....</b>	<b>22</b>
3.1 MULTI-CRITERIA DECISION SUPPORT MECHANISM FOR IoT OFFLOADING (MEDICI).....	22
3.1.1 <i>MEDICI Operation</i> .....	22
3.1.2 <i>MEDICI service Implementation</i> .....	22
3.1.3 <i>MEDICI Example Run - Handling a Request and Response</i> .....	24
<b>4 CONCLUSIONS.....</b>	<b>27</b>
<b>5 REFERENCES .....</b>	<b>28</b>

## List of Figures

Figure 2-1: Demo showing how to train, examine and save BACS autoencoders .....	10
Figure 2-2: Demo showing how to load and use previously learned BACS autoencoders .....	11
Figure 2-3: Demo showing how to start a BACS federated learning server (this program is executed on a field gateway) .....	12
Figure 2-4: Demo showing how to train and use federated BACS autoencoders .....	13
Figure 2-5: Incrementally trained federated BACS autoencoder .....	14
Figure 2-6: Concurrently trained federated BACS autoencoder .....	14
Figure 2-7: The signature of the BACS lightweight autoencoders inference function with the definition of data structures used in its implementation .....	15
Figure 2-8: DPI tool attack detection .....	16
Figure 2-9: DPI tool sample from signatures .....	17
Figure 2-10: DPI tool sample from flows.....	17
Figure 2-11: DPI tool local attack detection.....	17
Figure 2-12: 3ACEs' technical architecture .....	19
Figure 2-13: 3ACEs' data interface – definition of the column represents the class .....	19
Figure 2-14: 3ACEs' results' interface .....	20
Figure 2-15: Output of a successful process.....	21
Figure 3-1: High Level MEDICI Service Architecture .....	23
Figure 3-2: MEDICI Agent Services.....	24
Figure 3-3: Demo of external client interacting with MEDICI Service .....	24
Figure 3-4: Demo of MEDICI Service verbose output .....	25

## List of Abbreviations

<b>BACS</b>	Behavioral Analysis and Cognitive Security
<b>FG</b>	Field Gateway
<b>MEDICI</b>	Multi-critEria DecIsion support meChanism for IoT offloading
<b>ML</b>	Machine Learning
<b>MVP</b>	Minimum Viable Product\
<b>NB-IoT</b>	Narrowband IoT
<b>TPM</b>	Trusted Platform Modules
<b>PCR</b>	Platform Configuration Registers

## **Executive Summary**

This deliverable is a demonstrator deliverable. This report is a brief description of the operation and implementation of the individual components described in tasks T2.2 "Deep learning trained models deployed at the edge" and T2.3 "Security-aware dynamic offloading decision mechanism", up to month 12 of the project. These components are then integrated as part of the Minimum Viable Product (MVP) of deliverable D4.2, where we explain how they interact.

# 1 Introduction

Anomaly detection (also known as outlier detection) is the identification of rare or unusual (unexpected) data points. Anomaly detection approaches based on machine learning techniques can be either unsupervised or supervised. The main difference is in the presence of labels in the training data indicating whether a data point corresponds to an anomaly. Anomaly detection algorithms can be also utilized for novelty detection where a machine learning model is trained on data that is not polluted by outliers (i.e., in this case training datasets contain only “normal” data points. In the C4IIoT context, those are data points reflecting normal IIoT device and/or network behavior). The C4IIoT framework will utilize advanced anomaly detection techniques to detect anomalies in (1) sensory data acquired by industrial IoT devices and (2) network traffic flows within industrial IoT systems. The identification of anomalous and malicious behavior, based on both unsupervised and supervised anomaly detection models, will be present at all three C4IIoT layers implying that the corresponding software components will be deployed not only at C4IIoT edge nodes, but also on field gateways and in cloud.

Computation offloading is one of the primary technological enablers of the Internet of Things (IoT), and subsequently Industrial IoT (IIoT), as it helps address individual devices’ resource restrictions. In the past, offloading would always utilise remote cloud infrastructures, but the increasing size of IIoT data traffic and the realtime response requirements of modern and future IIoT applications have led to the adoption of the edge computing paradigm, where the data is processed at the edge of the network, e.g. network gateways, which are closer to the edge devices. The decision as to whether cloud or field gateway resources will be utilised is typically taken at the design stage based on the type of the IoT device. Yet, the conditions that determine the optimality of this decision, such as the time it takes to complete tasks and the real-time condition of the networks, keep changing. For applications such as anomaly detection, the time it takes to execute a task and detect an anomaly is critical. In C4IIoT the offloading mechanism delegates different security tasks to different layers of the infrastructure (field gateway or cloud) according to the trade-off between the time criticality of the action of interest and the computational complexity of processing the task.

In this deliverable we demonstrate our progress regarding deep learning techniques such as the anomaly detection components that are aimed to be deployed at C4IIoT edge nodes and the security aware and multi-criteria dynamic offloading mechanism to be deployed on the field gateway.



## 2 Deep learning trained models deployed at the edge

### 2.1 Behavioral Analysis and Cognitive Security (BACS) Tool

BACS (Behavioral Analysis and Cognitive Security) is one of the C4IIoT components realizing anomaly detection in IoT sensory readings and network traffic flows relying on machine learning techniques, including deep learning. Unsupervised anomaly detection BACS modules detect large, unusual and unexpected deviations from nominal device and network behavior, while supervised anomaly detection BACS modules detect specific security threats for which they were explicitly trained. In this deliverable we will describe BACS modules based on deep learning techniques for C4IIoT edge nodes in the Smart Factory and Logistics 4.0 use cases. A more detailed description of BACS modules for upper C4IIoT layers can be found in deliverable D3.1.

Deep learning BACS modules for unsupervised anomaly detection are based on standalone and federated autoencoders. An autoencoder is a neural network learning a latent (hidden) lower-dimensional representation (an encoding in terms of hidden, latent variables) of training data by reproducing its inputs through latent variables in the hidden layers at the output layer with the smallest possible error that is defined as the difference between the input and output values. Let us assume that device/network behavior is described by a feature vector  $X$  containing  $k$  real-valued features. Those may be values observed at one particular point in time, time series encompassing the last  $k$  observations of a single property, or the last  $p$  observations of a multivariate property having  $k/p$  features. Let  $D$  denote a set of data points described according to the feature vector  $X$  that depicts the normal (nominal) behavior of the device/network (training dataset), let  $A$  be an autoencoder trained on  $D$ , let  $E$  denote the maximal error of  $A$  on  $D$  (the maximal difference between the input and output layer for data points in  $D$ ). Then, a data point  $y$  not contained in  $D$  (a data point that is not present in the training dataset) can be considered as an anomaly if the difference between  $y$  and  $A(y)$  is higher than  $E$  ( $A(y)$  is the output of  $A$  for  $y$ ).

BACS autoencoders for smart factory C4IIoT edge nodes are implemented in Python using the Tensorflow2 library. The implementation of the BACS autoencoders is contained in `TFAutoAD` module which defines one class of the same name. `TFAutoAD` class extends `BaseAD` class from `BACSPY` package which is the base class for all python-based anomaly detection modules in BACS. `TFAutoAD` constructor accepts three parameters:

- `model` – the name of the autoencoder model that will be loaded from the working directory. This parameter is used when instantiating a previously learned `TFAutoAD` model.
- `dataset` – an object of `SWDataset` class which represents a set of multi-variate time series (time series of vectors containing multiple features) constructed from time-ordered data points using the sliding window approach.
- `num_epochs` – the number of epochs (one epoch is one pass through the training dataset when training a neural network).

`TFAutoAD` class defines the following methods:

- `initialize_model_structure(num_features)` initializes the internal structure of an autoencoder (hidden layers) for the given number of features

- `prepare_training_data()` normalizes (scales) feature values in the training data using the standard scaler from the Scikit-learn library. The fitted scaler object is later used to normalize data points before anomaly detection inference.
- `train()` trains the model by minimizing the mean square error (MSE) using the Adam method [5]. This method additionally computes raw anomaly detection scores (MSE values) for all instances in the training set in order to find the maximal error.
- `ad_inference(test_instance)` performs anomaly detection on the given test instance. This method is also called by `is_anomaly(test_instance)` method from `BaseAD` class which returns a boolean indicator (true if an anomaly is detected, false otherwise) and the confidence score.

Figure 2-1 shows a simple program demonstrating how to instantiate `TFAutoAD` class, train, examine and save the model. In this example the model is learned from feature values describing one-day behavior of a robotic arm in the CRF factory. Figure 2-2 presents another program showing how to load and use the previously learned autoencoder.

```
from BACSPY.Dataset import SWDataset
from BACSPY.TFAutoAD import TFAutoAD

# step 1 -- make the dataset from the input file
data = SWDataset("R1.csv", train_fraction = 0.8,\
                 window_length = 5, lat_lon_present = False)

# step 2 -- train the model
model = TFAutoAD(dataset = data, num_epochs = 5)
model.train()

# step 3 -- examine the model
num_anomalies = 0
for t in data.get_test_data():
    anomaly_detected, confidence = model.is_anomaly(t)
    if anomaly_detected:
        print("Anomaly detected, confidence = ", confidence)
        num_anomalies += 1
print("\nValidation on test data, #anomalies = ", num_anomalies)

# step 4 -- save the model
model.save("tf_autoencoder_model")
```

```
svc@svcp:~/BACS_wrk/python/demoTF$ python3 demo1.py
Using TensorFlow backend.
Epoch 1/5
692/692 [=====] - 0s 308us/sample - loss: 7.9158 - mean_squared_error: 1.0483
Epoch 2/5
692/692 [=====] - 0s 61us/sample - loss: 4.1546 - mean_squared_error: 1.0009
Epoch 3/5
692/692 [=====] - 0s 58us/sample - loss: 2.9613 - mean_squared_error: 0.9959
Epoch 4/5
692/692 [=====] - 0s 58us/sample - loss: 2.3433 - mean_squared_error: 0.9963
Epoch 5/5
692/692 [=====] - 0s 62us/sample - loss: 2.0004 - mean_squared_error: 0.9982
692/692 [=====] - 0s 93us/sample - loss: 1.8292 - mean_squared_error: 0.9981

Validation on test data, #anomalies = 0
svc@svcp:~/BACS_wrk/python/demoTF$ █
```

Figure 2-1: Demo showing how to train, examine and save BACS autoencoders

```

from BACSPY.TFAutoAD import TFAutoAD
import numpy as np

# step 1 -- load the model
model = TFAutoAD(model = "tf_autoencoder_model")

# step 2 -- make prediction (inference) on a random time series
rnd_vector = np.random.uniform(low = 0, high = 100, size = 180).tolist()
anomaly_detected, confidence = model.is_anomaly(rnd_vector)
if anomaly_detected:
    print("\nAnomaly detected, conf = ", confidence)

```

```

svc@svcp:~/BACS_wrk/python/demoTF$ python3 demo2.py
Using TensorFlow backend.
Model loaded
Model: "sequential"

```

Layer (type)	Output Shape	Param #
dense (Dense)	(None, 90)	16290
dropout (Dropout)	(None, 90)	0
dense_1 (Dense)	(None, 45)	4095
dropout_1 (Dropout)	(None, 45)	0
dense_2 (Dense)	(None, 90)	4140
dropout_2 (Dropout)	(None, 90)	0
dense_3 (Dense)	(None, 180)	16380
Total params: 40,905		
Trainable params: 40,905		
Non-trainable params: 0		

```

Anomaly detected, conf = 1.0
svc@svcp:~/BACS_wrk/python/demoTF$ █

```

Figure 2-2: Demo showing how to load and use previously learned BACS autoencoders

A standalone autoencoder is trained locally on an edge node considering only local data (data collected at the node). Besides standalone autoencoders, BACS also supports collective training of federated autoencoders. In this setting, a family of homogeneous IoT devices collectively trains an anomaly detection model without exchanging locally collected data points or sending them to a central location. In other words, the model is trained on data partitioned across a cluster of edge nodes. Federated learning brings many benefits including privacy preservation and faster training time. BACS federated learning is realized in two different schemes:

- (1) **Incremental.** A federated learning client downloads the global model from a federated learning server (running on the field gateway), updates it with its local data points and sends the updated model back to the federated learning server. The first connected client initializes the model. The model is locked while it is being updated by a client, i.e. other clients wanting to update the model wait for the currently connected client to finish with its update.

- (2) **Concurrent.** In this case the federated learning server knows in advance all federated learning clients and waits for all of them to connect. In the first round (epoch) all clients train individual models on local data and send the trained models to the server. The server averages received models (by averaging weights of links in neural networks having the same structure) and sends the averaged model back to all clients for the next learning round. The process is repeated for an arbitrary number of learning rounds.

The BACS implementation of federated autoencoders for in Python is given in BACSFED package. This package contains the following modules:

- FEDServerInc – the federated learning server for incrementally learned BACS autoencoders
- FEDServerCon – the federated learning server for concurrently learned BACS autoencoders
- FAEInc – incrementally learned federated BACS autoencoder
- FAECon – concurrently learned federated BACS autoencoder
- util – module containing helper functions to serialize and deserialize BACS autoencoders, the present functions are used by classes defined in the previously mentioned BACSFED modules.

Both FEDServerInc and FEDServerCon classes define a method for handling client requests called `loop_forever` (it should be noticed that the main loop in FEDServerCon is not actually looping forever, but for a given number of epochs; nevertheless this method has the same name as the main loop in FEDServerInc which is looping forever). The constructor of FEDServerInc has two parameters: the name of the model that will be incrementally updated by clients and the port on which the server accepts connections (both parameters have default values). The constructor of FEDServerCon has three parameters: the name of the configuration file containing IP addresses of FAECon clients, the port on which the server accepts connections and the number of epochs for training the model (the last two parameters have default values). A simple program demonstrating how to instantiate FEDServerInc and FEDServerCon classes is shown in Figure 2-3.

```
import sys
from BACSFED.FEDServerCon import FEDServerCon
from BACSFED.FEDServerInc import FEDServerInc

if sys.argv[1] == "incremental":
    print("Starting FED server for incremental learning")
    server = FEDServerInc()
    server.loop_forever()
elif sys.argv[1] == "concurrent":
    print("Starting FED server for concurrent learning")
    server = FEDServerCon("fedservercon.conf", num_epochs = 5)
    server.loop_forever()
else:
    print("Invalid server type")
```

**Figure 2-3: Demo showing how to start a BACS federated learning server (this program is executed on a field gateway)**

Figure 2-4 shows a program demonstrating FAEInc and FAECon classes. Both classes extend TFAutoAD class which means that federated BACS autoencoders are used in the same way as standalone BACS autoencoders. The execution of the program for the incrementally trained model is shown in Figure 2-5, while Figure 2-6 shows the execution in the concurrent learning case.

```
import sys
from BACSPY.Dataset import SWDataset
from BACSFED.FAEInc import FAEInc
from BACSFED.FAECon import FAECon
import numpy as np

# instantiating incrementally learned federated autoencoder
def make_incremental_model(server_addr, local_data):
    return FAEInc(server_address = server_addr,\
                  dataset = local_data, num_epochs = 10)

# instantiating concurrently learned federated autoencoder
def make_concurrent_model(server_addr, local_data):
    return FAECon(server_address = server_addr,\
                  dataset = local_data)

# helper function to evaluate model on given data
def evaluate_model(model, data):
    num_anomalies = 0
    for t in data:
        anomaly, _ = model.is_anomaly(t)
        if anomaly:
            num_anomalies += 1

    print("Num points =", len(data))
    print("Num anomalies =", num_anomalies)

### federated learning client
print("Starting FED learning client")

# create the dataset
print("Train dataset", sys.argv[1])
dataset = SWDataset(sys.argv[1], train_fraction = 0.8,\
                    window_length = 5, lat_lon_present = False)

# train the model
model = None
if sys.argv[2] == "incremental":
    model = make_incremental_model("127.0.1.1", dataset)
else:
    model = make_concurrent_model("127.0.1.1", dataset)

# evaluate the model on test (regular) data
print("\nEvaluating model on test data")
evaluate_model(model, dataset.get_test_data())

# evaluate the model on random data
print("\nEvaluating model on random data")
total_features = 5 * dataset.num_features
random_data = [\
    np.random.uniform(0, 100, total_features).tolist()\
    for _ in range(40)]
evaluate_model(model, random_data)
```

**Figure 2-4: Demo showing how to train and use federated BACS autoencoders**

```

svc@svcp: ~/BACS_wrk/python/demoFED
File Edit View Search Terminal Help
svc@svcp:~/BACS_wrk/python/demoFED$ python3 StartFEDServer.py
Starting FED server for incremental learning
Initializing INC-FL server on port 1210
Loading model...
Model file does not exist, training the model from scratch...
INC-FL server ready for connections...
Accepted connection from ('127.0.0.1', 45302)
Model saved...
Accepted connection from ('127.0.0.1', 45304)
Model saved...
Accepted connection from ('127.0.0.1', 45310)
Model saved...

svc@svcp:~/BACS_wrk/python/demoFED$ python3 DemoFEDClient.py
R1.csv incremental
Using TensorFlow backend.
Starting FED learning client
Train dataset R1.csv
Connecting to FL-INC server...
NULL message received, training model from scratch...
32/692 [>.....] - ETA: 0s - loss: 7.
692/692 [=====] - 0s 93us/sample - l
oss: 1.3097 - mean_squared_error: 0.9994
Sending updated model back to server...

Evaluating model on test data
Num points = 170
Num anomalies = 0

Evaluating model on random data
Num points = 40
Num anomalies = 40
svc@svcp:~/BACS_wrk/python/demoFED$

svc@svcp:~/BACS_wrk/python/demoFED$ python3 DemoFEDClient.py
R2.csv incremental
Using TensorFlow backend.
Starting FED learning client
Train dataset R2.csv
Connecting to FL-INC server...
Updating model...
32/692 [>.....] - ETA: 0s - loss: 2.
692/692 [=====] - 0s 87us/sample - l
oss: 1.0371 - mean_squared_error: 1.0000
Sending updated model back to server...

Evaluating model on test data
Num points = 170
Num anomalies = 4

Evaluating model on random data
Num points = 40
Num anomalies = 40
svc@svcp:~/BACS_wrk/python/demoFED$

svc@svcp:~/BACS_wrk/python/demoFED$ python3 DemoFEDClient.py
R3.csv incremental
Using TensorFlow backend.
Starting FED learning client
Train dataset R3.csv
Connecting to FL-INC server...
Updating model...
32/690 [>.....] - ETA: 0s - loss: 0.
690/690 [=====] - 0s 90us/sample - l
oss: 1.0010 - mean_squared_error: 1.0000
Sending updated model back to server...

Evaluating model on test data
Num points = 170
Num anomalies = 6

Evaluating model on random data
Num points = 40
Num anomalies = 40
svc@svcp:~/BACS_wrk/python/demoFED$

```

Figure 2-5: Incrementally trained federated BACS autoencoder

```

svc@svcp: ~/BACS_wrk/python/demoFED
File Edit View Search Terminal Help
svc@svcp:~/BACS_wrk/python/demoFED$ python3 StartFEDServer.py
Starting FED server for concurrent learning
Initializing CON-FL server on port 1211
CON-FL configured with the following clients
['127.0.0.1', '127.0.0.1', '127.0.0.1']
CON-FL server ready for connections...
Accepted connection from 127.0.0.1
Client approved, waiting for 2 more clients
Accepted connection from 127.0.0.1
Client approved, waiting for 1 more clients
Accepted connection from 127.0.0.1
All clients connected, starting concurrent federated learning
...
Epoch 0 started
Epoch 1 started
Epoch 2 started
Epoch 3 started
Epoch 4 started
Learning finished...
svc@svcp:~/BACS_wrk/python/demoFED$

svc@svcp:~/BACS_wrk/python/demoFED$ python3 DemoFEDClient.py
32/692 [>.....] - ETA: 0s - loss: 6.
692/692 [=====] - 0s 25us/sample - l
oss: 1.0062 - mean_squared_error: 1.0000
Epoch 3
32/692 [>.....] - ETA: 0s - loss: 6.
692/692 [=====] - 0s 34us/sample - l
oss: 1.0028 - mean_squared_error: 1.0000
Epoch 4
32/692 [>.....] - ETA: 0s - loss: 6.
692/692 [=====] - 0s 28us/sample - l
oss: 1.0013 - mean_squared_error: 1.0000
Training finished, computing training scores...

Evaluating model on test data
Num points = 170
Num anomalies = 0

Evaluating model on random data
Num points = 40
Num anomalies = 40
svc@svcp:~/BACS_wrk/python/demoFED$

svc@svcp:~/BACS_wrk/python/demoFED$ python3 DemoFEDClient.py
32/690 [>.....] - ETA: 0s - loss: 0.
690/690 [=====] - 0s 22us/sample - l
oss: 1.0039 - mean_squared_error: 1.0000
Epoch 3
32/690 [>.....] - ETA: 0s - loss: 0.
690/690 [=====] - 0s 21us/sample - l
oss: 1.0016 - mean_squared_error: 1.0000
Epoch 4
32/690 [>.....] - ETA: 0s - loss: 0.
690/690 [=====] - 0s 21us/sample - l
oss: 1.0007 - mean_squared_error: 1.0000
Training finished, computing training scores...

Evaluating model on test data
Num points = 170
Num anomalies = 6

Evaluating model on random data
Num points = 40
Num anomalies = 40
svc@svcp:~/BACS_wrk/python/demoFED$

```

Figure 2-6: Concurrently trained federated BACS autoencoder

For low power, narrowband IoT (NB-IoT) devices, planned in the C4IIoT Logistics 4.0 use case we have implemented inference routines (routines performing anomaly detection for a given data point on a pretrained model) for lightweight BACS autoencoders (lightweight in the sense that those autoencoders have exactly one hidden layer). The implementation is done in C without using any external libraries. The training of lightweight BACS autoencoders is performed offline using a simplified version of TFAutoAD BACS Python class which only trains a model with a predefined structure and exports obtained link weights and data normalization parameters to textual files instead of saving them as a Tensorflow model. Figure 2-7 shows the definition of data structures used in the implementation of `make_inference` function performing anomaly detection. This function is directly integrated in the firmware of C4IIoT NB-IoT devices.

```
// BACS lightweight autoencoder configuration
typedef struct {
    int num_features;
    int num_hidden_nodes;
    double max_MSE; // maximal MSE on the training set
    double std_MSE; // standard deviation of MSE on the training set
    double* f_mean; // mean values of features in the training set
    double* f_std; // standard deviation of features in the training set
} config_t;

// BACS lightweight autoencoder structure
typedef struct {
    // links from the input layer to the hidden layer
    double** w_inp_hid;
    // links from the hidden layer to the output layer
    double** w_hid_out;
} lae_structure_t;

// BACS lightweight autoencoder state
typedef struct {
    double* hidden_activation;
    double* output;
} lae_state_t;

// inference output
typedef struct {
    int is_anomaly; // 1 means anomaly
    double raw_MSE; // raw MSE value
    double score; // confidence score (normalized MSE value)
} inference_result_t;

inference_result_t make_inference(double* input_values);
```

**Figure 2-7: The signature of the BACS lightweight autoencoders inference function with the definition of data structures used in its implementation**

## 2.2 Traffic Analysis Tool

The Traffic Analysis module utilises signatures to detect specific traffic patterns inside network flows. The signatures will be generated during the training stage, where public datasets of malicious traffic from previous research work will be used as ground truth [1] [2].



These datasets contain attacks from IoT environments, like worms and botnets and more general attacks. Our signatures relay on packet metadata (e.g. packet-length, direction, timestamp) in order to be able to be used for matching even if the traffic is encrypted.

Currently, our tool is able to detect portscans and login attempts in SSH and Telnet which are most commonly used in IoT attacks. In addition, we will enrich our signatures dataset from attacks which will be captured by UNSMPF's machine learning anomaly detection module after they have been categorized as true positives manually by a security expert.

The Traffic Analysis module is lightweight and can be deployed in both edge devices and the cloud. It will listen to the network traffic, compare it to our crafted signatures and report any matches to the C4IIoT's visualization module. We plan to add more signatures from public datasets to cover a wide area of divergent threats. In addition, we plan to extend our tool by adding the function of detecting weak encryption keys in C4IIoT's network traffic to deal with misconfigured certificates and deprecated algorithms. Requirements for this module have been set in deliverable D1.3.

In Figure 2-8 we can see an example of our tool listening to the network traffic and detecting a portscan and a telnet login attempt from a remote host. Our tool takes as arguments the name of the network interface it will listen to, in this case "enp0s31f6" and a file containing malicious signatures it will look for inside the network traffic. The file "flows.txt" is being used to keep a record of the packets captured from the network interface. After starting our module, using a remote host as the attacker, two attacks were committed: one portscan using nmap (*Network Mapper*) and a login attempt using telnet. As it can be seen both attacks were detected and reported by our tool. Our tool reports the name of the malicious signature it matches and the source and destination IP addresses.

```
root@tsirant-ThinkPad-L580:~/Desktop/dpi/python-dpi# python3 dpi_c4iiot_forth.py
--file flows.txt --interface enp0s31f6 --signatures signatures.txt
Running live packet capture...
Start sniffing on interface %s enp0s31f6
Sniffing can be aborted via pressing Ctrl-c
Attack Detected:
139.91.76.59
139.91.76.46
portscan2
Attack Detected:
139.91.76.59
139.91.76.46
portscan2
Attack Detected:
139.91.76.59
139.91.76.46
telnetloginattempt2
```

**Figure 2-8: DPI tool attack detection**

In Figure 2-9 we can see a sample of the signatures file which is being used to detect attacks. As we mentioned previously more and more attacks will be added to our dataset since we will keep analysing public malicious datasets and adding them to our database. As it can be seen in the figure for each attack, we keep track of the size of the packet payload and the order they arrive. To add a new signature, we use public malicious datasets, known security tools and manually inspect the packets from the datasets and attacks we conduct. After we find a sequence of packets sizes which is unique to an attack, we test our new signature against large



amounts of benign traffic to reduce the risk of false positives. The use of strict and accurate signatures is critical to the correct use of our module.

[illegible]

**Figure 2-9: DPI tool sample from signatures**

A sample of the traffic collected from our tool can be seen in Figure 2-10. We keep a record of all the traffic in the network interface for up to 1 hour after the last packet in the transmission was sent. Each line represents a different transmission and contains a source and destination IP address, a timestamp of the last packet received and the packet's payload length in the order they arrive.

The Traffic Analysis tool can listen to different interfaces and detect different types of attacks as it can be seen in Figure 2-11. While our module is listening to the local traffic, we use another terminal to execute a local password brute force attack using the known password cracking tool Hydra. Our tool correctly detects, classifies and reports the attack. For sending notifications to other modules about the attack our tool currently supports Syslog [3] and Kafka [4].

[illegible]

**Figure 2-10: DPI tool sample from flows**

```
tsirant@tsirant-ThinkPad-L580:~/Desktop/dpi/python-dpi$ hydra -l root -P passwords.txt ssh://127.0.0.1
Hydra v8.6 (c) 2017 by van Hauser/THC - Please do not use in military or secret service organizations, or for illegal
purposes.

Hydra (http://www.thc.org/thc-hydra) starting at 2020-05-07 11:30:30
[WARNING] Many SSH configurations limit the number of parallel tasks, it is recommended to reduce the tasks: use -t 4
[WARNING] Restorefile (you have 10 seconds to abort... (use option -I to skip waiting)) from a previous session found,
to prevent overwriting, ./hydra.restore
[DATA] max 16 tasks per 1 server, overall 16 tasks, 500 login tries (l:1/p:500), ~32 tries per task
[DATA] attacking ssh://127.0.0.1:22/

root@tsirant-ThinkPad-L580: ~/Desktop/dpi/python-dpi

File Edit View Search Terminal Help

root@tsirant-ThinkPad-L580:~/Desktop/dpi/python-dpi# python3 dpi_c4liot_forth.py --file flows.txt --interface lo --sig
natures signatures.txt
Running live packet capture...
Start sniffing on interface %s lo
Sniffing can be aborted via pressing Ctrl-c
Attack Detected:
127.0.0.1
127.0.0.1
sshbruteforcehydra-local
```

**Figure 2-11: DPI tool local attack detection**

## 2.3 Machine Learning Anomaly Detection at the edge – 3ACEs Tool

Both in Smart Factory and Logistics 4.0 cases, Machine Learning (ML) algorithms will be implemented across all layers; these algorithms, based on the information provided by the offloading mechanism, partition the data and the ML model in question across the layers will trade-off complexity of the current task with time criticality of the reaction times. Among several predictive modelling, and anomaly detection mechanisms, a central mechanism will be based on analysing encrypted network flows.

**3ACEs**, is a data analytics configurable and scalable *ITML* 's solution that embodies Machine Learning and AI technologies in order to provide effective analysis and insights both for Smart Factory and Logistics 4.0 use cases. It constitutes a framework for the development of a predictive analytics system, that deploys multiple advanced ML algorithms in order (i) to understand the complex interplay of information (system status, type of incidents, and measured data) with the various heterogeneous parameters that affect decision-making; (ii) to enhance the capacity to detect (supporting BACS and MEDICI), prevent and mitigate (supporting the C4IIoT Mitigation engine) any issue (iii) to provide real time smart situation awareness at incident time and (iv) to detect incidents and optimise resources (energy, labor etc.).

To that end, 3ACEs comes to complement the services and functionalities offered by BACS and MEDICI in C4IIoT; it can provide (a) data pre-processing functionalities; (b) a set of numerous ML classification and clustering algorithms to be used to support anomaly detection performed by BACS and (c) a set of recommendations, based on the data analyses, that can be used as an additional input for the mitigation procedures of C4IIoT.

End users have direct and easy access to all types of real-time and historical information (recommendations, decision support, thorough system status) and at the same time they can provide to the system feedback to support ML algorithms' training. Figure 2-12 depicts the overall technical architecture of **3ACEs**. The platform consists of three functional high-level blocks, namely the data providers, the data flow container and the analytics container.

Within this task, *ITML* aims to bring ML models (e.g., a deep neural network) from the **3ACES** functionality closer to the edge. More specifically, ML methods that perform detection of complex anomalous and malicious behaviour will be deployed to the edge nodes (e.g. *Raspberry Pi*). This corresponds to updating the models running at edge devices based on analyses carried out in the physically close field gateway and not the cloud. Such modelling shift can significantly reduce time delays, making the ML algorithms more amenable for anomaly detection tasks. The developed ML methods will take as input data streams (e.g., IIoT device data, (encrypted) traffic data) and will give as output clustered or classified data (according to the source data types).

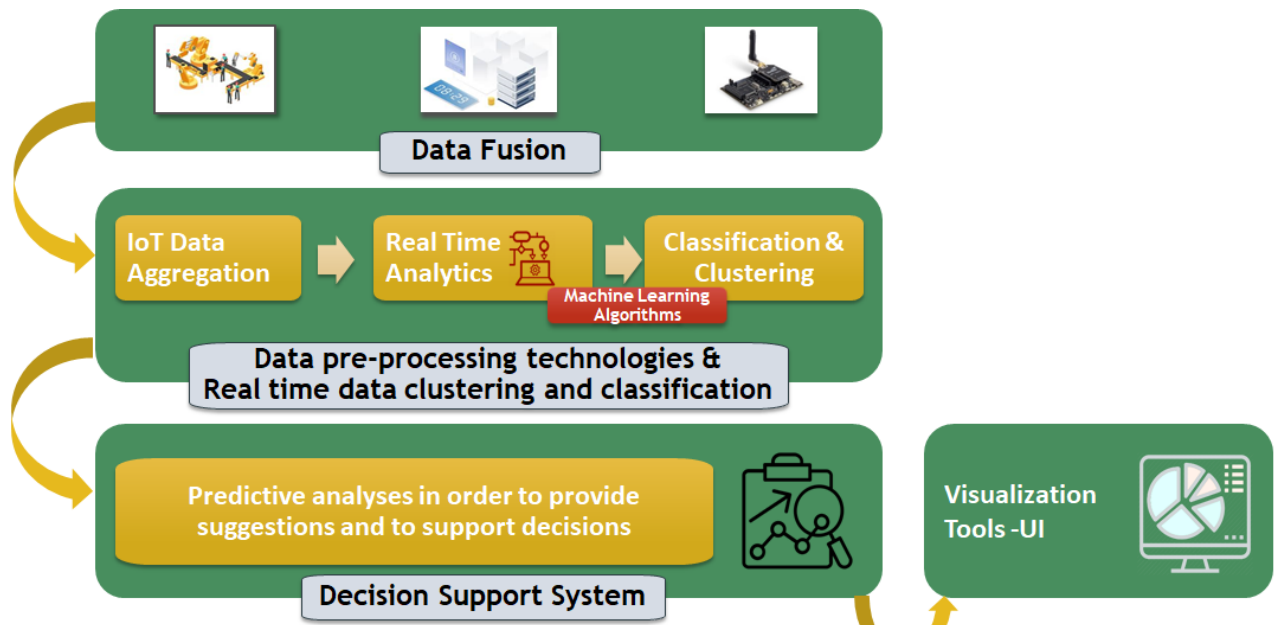


Figure 2-12: 3ACEs' technical architecture

**3ACES** functionality is depicted below demonstrating results' accuracy from several algorithms in a data classification use case.

**3As**
**3ACEs Online Wizard**
☰

Search:

lumn 5	Column 6	Column 7	Column 8	Column 9	Column 10
	K	Ca	Ba	Fe	Type
3	64	877	0	0	'build wind float'
9	57	853	0	0	'vehic wind float'
4	59	843	0	0	'build wind float'
5	0	759	0	0	tableware
6	12	1619	0	24	'build wind non-float'
7	57	879	11	22	'build wind non-float'
	6	893	0	0	'vehic wind float'
5	55	907	0	0	'build wind float'
9	8	907	61	5	headlamps

**2.**

Select the column in the header representing the class in the uploaded dataset

Figure 2-13: 3ACEs' data interface – definition of the column represents the class



Figure 2-14: 3ACEs' results' interface

## 2.4 Secure Hardware Support Tools

In order to run the previously described algorithms safely in the edge node, there must be security also at the hardware level as it offers benefits such as generate security-critical-data and tamper-resistant protection to avoid physical attacks to extract keys. To do this, edge node devices must have a secure element integrated, such as a tamper-resistant hardware platform, capable of securely storing confidential and cryptographic data. This secure element ensures that the data generated in the edge nodes are securely encrypted and security keys are stored in a safe way. In addition, it checks the edge node's integrity to identify manipulation and detect unauthorised changes at the hardware level as described below.

On connected nodes within a network, it is possible to generate attacks aimed at manipulating the software running on them. A principal goal of trusted computing is to provide reliable evidence about the state of software executing on a system. This is the concept called remote

attestation. Remote attestation is a method by which a host (edge node) authenticates its hardware and software configuration. The goal of remote attestation is to enable a remote system to determine the level of trust in the integrity of another system [5]. To carry this out, an attestation protocol is needed.

In many cases, the result of an attestation protocol depends on a complicated mixture of facts that the appraiser can check directly, such as cryptographic signatures, and those that he cannot check, such as the hardware status (number of connected peripherals or a change in some peripheral without notification). Indeed, the importance of the secure elements lies exactly here. Trusted Platform Modules (TPMs) [6] are secure elements on a platform's board. Information, such as measurements of the platform's software state, may be placed into the TPM's Platform Configuration Registers (PCRs). A TPM thus provides cryptographically signed evidence of facts about the state of the platform to which it is attached. The TPM residing on the target may be considered a principal in an attestation protocol.

This ensures that the software implemented on the embedded platforms is reliable and has not been maliciously manipulated so it will perform its function.

This mechanism must be accompanied by another mechanism that should oversee the state of the device at the boot process. This mechanism, called secure boot. When the device starts, the firmware checks the signature of each piece of boot software, for instance, hardware changes can be detected at startup. These changes in the hardware and software impact on the PCRs previously mentioned. These PCRs is used to check the integrity of the system during the boot process. In this case is used to encrypt/decrypt some piece of software, using like the "encrypt/decrypt key" the hash of the PCRs. Figure 2-15 below shows a typical output when the process is successful.

```

root@raspberrypi:/home/pi/Desktop/attached/attached/test# ./decrypt.sh check_integrity.txt text.txt sealed_file
>>> Create a Context
    success
>>> Connect to TPM
    success
>>> GetTPMHandle
    success
>>> Tspi_Context_Connect
    success
>>> Get TPM Policy
    success
>>> Setting Secret
    success
>>> Resetting
    success
>>> Create a Context
    success
>>> Connect to TPM
    success
>>> GetTPMHandle
    success
>>> Tspi_Context_Connect
    success
>>> Get TPM Policy
    success
>>> Create Hash object      success
>>> Hash in the data        success
>>> Get the hashed result    success
>>> Extended the PCR:       success
Enter SRK password:

```

Figure 2-15: Output of a successful process

### 3 Security-aware dynamic offloading decision mechanism

#### 3.1 Multi-criteria Decision support mechanism for IoT offloading (MEDICI)

##### 3.1.1 MEDICI Operation

The C4IIoT security-aware dynamic offloading decision module utilises UOG's Multi-criteria Decision support mechanism for IoT offloading (MEDICI) to decide dynamically whether an anomaly detection task from UNSPMF's BACS component located at the field gateway or the cloud should be triggered, if the confidence level of the BACS detection at the edge device is low.

An IoT task can be processed locally, at the edge or at the cloud. MEDICI takes into account metrics such as the execution time of a task in a device, the time it takes to offload its data and the delays incurred by the network. It can also take into consideration metrics such as the energy consumption of the IoT device. The MEDICI tool takes into consideration predicted processing times, predicted network times and the confidence and the reaction time of the anomaly detection.

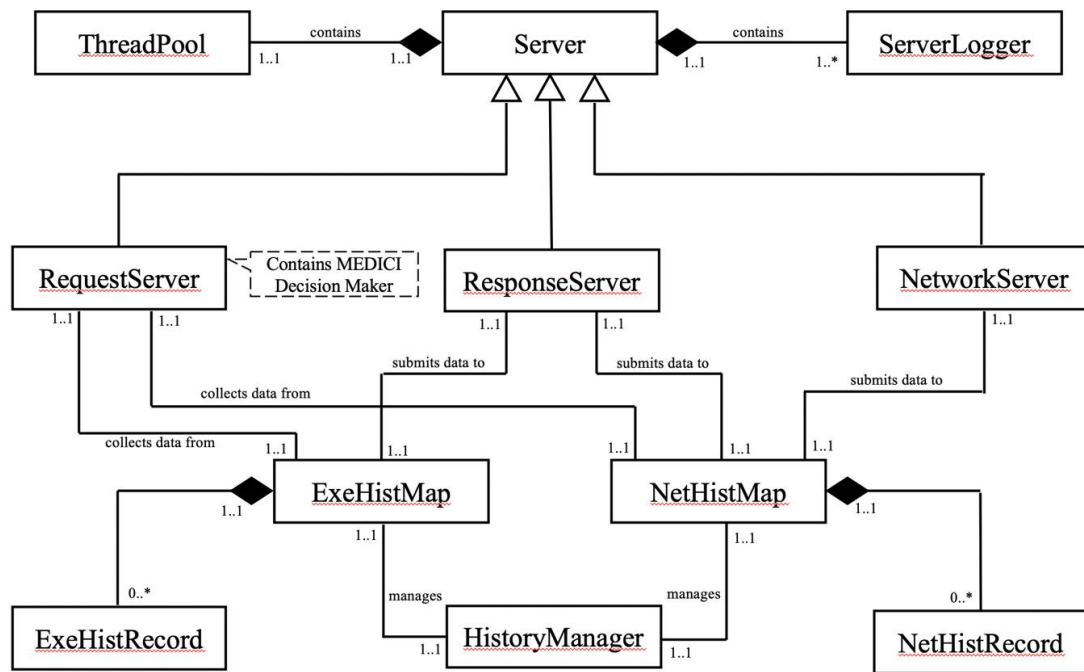
In the following section, we present the implementation of the MEDICI service. In terms of decision making, its modular design allows for flexibility in the choice of algorithms. In the demonstrator available currently, the decision mechanism has the following broad logic: Using lightweight machine learning, it predicts the processing time and networks delays for a task to be executed at a given device based on historical data. An earlier version of this logic has been published in [7].

MEDICI service will be installed on field gateways located in VIP's mobile operator premises for the Logistics 4.0 use case and in CRF's premises for the Smart Factory use case, protected in both cases by a strong security network infrastructure. As an intermediate layer, the field gateway acts as an element with greater computational power than the edge nodes, but also able to perform security-related (e.g. BACS) actions with significantly lower network delay than the cloud. The cloud has typically the highest processing power, but may introduce higher network delay, because it resides further away from the edge nodes than the field gateway. Safe and authorized access to the MEDICI tool is provided using VPN connection between remote site and field gateway.

##### 3.1.2 MEDICI service Implementation

Figure 3-1 showcases a high-level overview MEDICI service which contains three servers called Request, Response, and Network. Each server inherits its behaviour for accepting and establishing connections from the *Server* abstract class, whilst each sub-class provides its own implementation for how connections are handled. This inheritance allows to limit the number of simultaneous connections handled by the service, in a centralised manner, so that we can ensure that the service doesn't exceed any hardware or software limitations of the platform hosting it. This is done by the *ThreadPool* component, which contains a pool of threads to be used by all connections, and thus limiting their number.





**Figure 3-1: High Level MEDICI Service Architecture**

The role of the *RequestServer* is to handle incoming offloading decision requests from clients and respond back with the outcome of the MEDICI decision maker. The reported outcome can be the requesting device and all valid offloading destination located in the field gateway and cloud layers. Additionally, the number of concurrent offloading decision is limited so that system resources utilised by the machine learning algorithms for the purposes making offloading decisions doesn't exceed the available resources of the host environment. Also, during this process, all relevant execution history information and network data stored in the *ExeHistMap* and *NetHistMap* respectively is gathered to inform the decision being made. Note, when gathering network data from the *NetHistMap*, only data related to requesting edge device is used. When gathering a task's execution information (i.e. the amount of time taken in milliseconds of the task to be executed by its destination device) from the *ExeHistMap*, all data, related to the requesting edge device and the possible offloading destinations located in field gateway and cloud layers, is collected. Moreover, the *ExeHistMap* holds all the data concerning of previous task executions, such as, size of the task and task execution time. The *NetHistMap* holds records about each edge devices which can generate offloading requests. Each record holds recent network data from the edge device perspective to all possible offloading locations situated at the Field Gateway and Cloud network layers, such as round-trip latency and current bandwidth between the devices.

The purpose of the *ResponseServer* is to allow the client to provide execution history and networking information related to task after an offloading decision has been made, such as execution time and transmission time to offloading destination. More specifically, the information provided back from clients is then placed into the *ExeHistMap* and *NetHistMap* accordingly, to be used by the MEDICI decision maker and help inform future offloading decisions.

Furthermore, the role of the *NetworkServer* is to collect network-based information from the edge device perspective, when data held in the *NetHistMap* is of an insufficient amount to affectively determine the current network state between an edge device and its possible offloading destinations. To achieve this MEDICI uses agent services installed on devices throughout the C4IIoT network architecture which can either generate offloading decision requests or are a valid offloading destination. As seen in Figure 3-2, *Local Agents* are installed on device located at the edge layer hold a list of possible offloading destinations at the field gateway and cloud layers which have a *Remote Agent*. Then, when triggered by the *NetworkServer*, a *Local Agent* contacts each device within its list individually to collect round-trip latency and network bandwidth information, which is passed back to the *NetworkServer* to update *NetHistReocrd* associated with that particular edge device. Note, *Remote Agents*'s role is purely to facilitate *Local Agents* in collecting the desired network information.

Lastly, the *HistoryManager* is a separate process that removes old and irrelevant records held within *ExeHistMap* and *NetHistMap*, which helps ensuring that only relevant up-to-date information is supplied to the decision maker

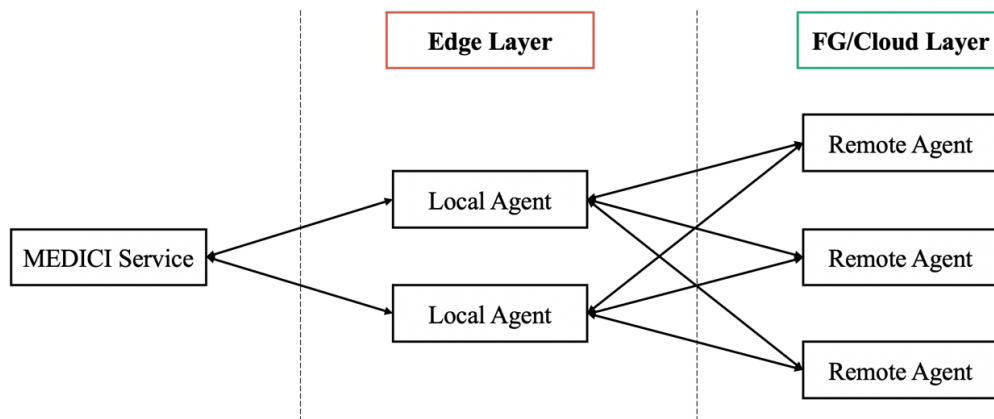


Figure 3-2: MEDICI Agent Services

### 3.1.3 MEDICI Example Run - Handling a Request and Response

Figure 3-3 shows an external client initially generating and sending an offloading decision request to the *RequestServer* of the MEDICI service. Then, the result of the decision is received back, containing the assigned task identifier to track the task and the offloading destination device (where the task should be executed). Lastly, the client reports the execution time for the task to the *ResponseServer* of the MEDICI service, which will update the execution history record, to be used in future offloading decisions requests.

```

willo@test-node-01: ~/MediciService
File Edit View Search Terminal Help
willo@test-node-01:~/MediciService$ java -jar MediciService_TestClient.jar
2020.05.13.15.02.04.835 Sending request to MEDICI request server. App ID = APP1, Input Size = 2343, Output Size = 34343, Task Size = 3434
2020.05.13.15.02.06.070 Data received from MEDICI request server: Task ID = 1589378524854, Destination Device = 192.168.2.11
2020.05.13.15.02.11.071 Sending execution time to MEDICI response server. Execution time = 4.24588
willo@test-node-01:~/MediciService$
  
```

Figure 3-3: Demo of external client interacting with MEDICI Service



Figure 3-4 showcases the verbose output of the *RequestServer* and *ResponseServer* within MEDICI service, which handles and processes the connections and data sent from the external client in Figure 3-3. Note that, each of the server processes within MEDICI has its own *ServerLogger* instance which writes and stores details of connections being established and handled to MEDICI in a logging file, which can then be referred for future analysis and provide details on any errors that may have occurred during the MEDICI's operation. More specifically, each log is in the following format:

Time Stamp	Log Level	Server Name	Log Message
------------	-----------	-------------	-------------

The Time Stamp is the current date and time of that the log was made, based on the system clock machine hosting the MEDICI service. It is in the format of yyyy.MM.dd.hh.mm.ss.SSS. Log Level refers to whether the message is either an error, warning, information, or debug. Server Name refers to which service process the message relates to (e.g., in the case of Figure 3-4, it is the Request and Response for the *RequestServer* and *ResponseServer* respectively). Lastly, the Log Message is the message string of the log entry.

```

Run: Main
2020.05.13.15.01.57.608 INFO Request Server Instance Created
2020.05.13.15.01.57.627 INFO Response Server Instance Created
2020.05.13.15.01.57.628 INFO Response Server Started
2020.05.13.15.01.57.628 INFO Response Server Waiting For Connection
2020.05.13.15.01.57.629 INFO Request Server Started
2020.05.13.15.01.57.629 INFO Request Server Waiting For Connection
2020.05.13.15.02.04.850 INFO Request Server Accepted Connection From: /192.168.2.11:46910
2020.05.13.15.02.04.853 INFO Request Server Waiting For Connection
2020.05.13.15.02.04.855 INFO Request 1589378524854 handling connection 192.168.2.11
2020.05.13.15.02.04.856 INFO Request 1589378524854 read in arguments from 192.168.2.11: appID = APP1, inputSize = 2343, outputSize = 34343, taskSize = 3434
2020.05.13.15.02.04.857 INFO Request 1589378524854 temp directory created successfully
2020.05.13.15.02.04.862 INFO Request 1589378524854 successfully created temp files for medici execution
2020.05.13.15.02.06.074 INFO Request 1589378524854 medici execution complete
2020.05.13.15.02.06.075 INFO Request 1589378524854 temp files and directory successfully deleted
2020.05.13.15.02.06.075 INFO Request 1589378524854 output from medici script: 1589378524854,192.168.2.11
2020.05.13.15.02.06.076 INFO Request 1589378524854 successfully added record to execution history.
1589378524854 requestDeviceID = 192.168.2.11, exeDeviceID = 192.168.2.11, deviceClass = 0, appID = APP1, inputSize = 2343, outputSize = 34343, taskSize = 3434, exeTime = 0.0, valid = false
2020.05.13.15.02.06.076 INFO Request 1589378524854 output successfully sent back to client device 192.168.2.11
2020.05.13.15.02.11.079 INFO Response Server Accepted Connection From: /192.168.2.11:43500
2020.05.13.15.02.11.080 INFO Response Server Waiting For Connection
2020.05.13.15.02.11.085 INFO Response 192.168.2.11 found task identifier 1589378524854
2020.05.13.15.02.11.085 INFO Response 192.168.2.11 data received from client: exe=4.24588
2020.05.13.15.02.11.086 INFO Response 192.168.2.11 found execution time 4.24588 successfully added to task 1589378524854 record.
1589378524854 requestDeviceID = 192.168.2.11, exeDeviceID = 192.168.2.11, deviceClass = 0, appID = APP1, inputSize = 2343, outputSize = 34343, taskSize = 3434, exeTime = 4.24588, valid = true
  
```

Figure 3-4: Demo of MEDICI Service verbose output

Lines 01 to 06 simply state that both the *RequestServer* and *ResponseServer* instances have been created and started successfully and are now listening for connections being made via their assigned ports. Lines 07 to 10 show the connection from the external client in Figure 3-3, establishing a connection to the *RequestServer*, which is assigned a task identifier (in this case 1589378524854), and the required arguments being read from the client. Then, temporary files for the execution history and network history are created to be passed to the MEDICI decision maker, as shown on lines 11 and 12. When creating the execution history file, we only include records from the *ExeHistMap* that relate to the same application identifier supplied from the client, and for any of the possible execution devices (edge, field gateway or cloud). Whereas, when creating the network history file, the network traffic information (i.e. roundtrip latency and bandwidth) from the records held within *NetHistMap* we only include data from the perspective of the requesting client's device. When the temporary files have been created, the MEDICI offloading decision maker is executed, and output is retrieved, as seen on lines 13 and 15 respectively. The output of the decision maker consists of firstly the task identifier, followed by the destination device in which the execution of the task should take place. Also, a new execution history record is added to the *ExeHistMap*, but this is marked as not valid (`valid=false`) as the record does not yet have an execution time and will not yet be used for future offloading decisions, as seen lines 16 and 17. Line 18 states the outcome of the decision being sent back to the requesting external client, as seen in Figure 3-3.

Finally, lines 19 through 24 show a connection being made from the external client in Figure 3-3 to the *ResponseServer* of the MEDICI service. The task identifier and a new execution time successfully read in from the client can be seen on lines 21 and 22 respectively. In lines 23 and 24, the new execution time is successfully added to execution history record of the task, which is then marked as valid (`valid=true`) to allow the record to be used in future offloading decisions.

## 4 Conclusions

This demonstrator deliverable has briefly described the components involved in tasks T2.2 and T2.3 and their implementation up to month 12. Some of these components are also described in other deliverables, such as D3.1. Here, we describe their operation as individual components, while their integration is described in deliverable D4.2 through the MVP description.

## 5 References

- [1] H. Kang, D. H. Ahn, G. M. Lee, J. D. Yoo, K. H. Park and H. K. Kim, “IoT network intrusion dataset,” IEEE Dataport, 2019. [Online]. Available: <http://dx.doi.org/10.21227/q70p-q449>.
- [2] I. Sharafaldin, A. H. Lashkari and A. A. Ghorban, “Toward Generating a New Intrusion Detection Dataset and Intrusion Traffic Characterization,” in *4th International Conference on Information Systems Security and Privacy (ICISSP)*, Portugal, 2018.
- [3] C. Lonvick, “The bsd syslog protocol,” RFC3164, 2001.
- [4] “Apache Kafka,” [Online]. Available: <http://kafka.apache.org..>
- [5] L. Jain and J. VYAS, “Security analysis of remote attestation,” CS259 Project Report, 2008.
- [6] T. C. Group, “Trusted Platform Module (TPM),” [Online]. Available: <https://trustedcomputinggroup.org/work-groups/trusted-platform-module/>.
- [7] A. Jaddoa, G. Sakellari, E. Panaousis, G. Loukas and P. G. Sarigiannidis, “Dynamic decision support for resource offloading in heterogeneous Internet of Things environments,” *Simulation Modelling Practice and Theory*, vol. 101, p. 102019, 2019.