Alma Mater Studiorum · Università di Bologna

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA-SCIENZA E INGEGNERIA (DISI)

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

TESI DI LAUREA

in

SISTEMI DISTRIBUITI

Distributed Federated Learning in Manufacturer Usage Description (MUD) Deployment Environments

CANDIDATO
Angelo Feraudo

RELATORE
Prof. Paolo Bellavista

CORRELATORI
Prof. Jon Crowcroft
Dr. Poonam Yadav

Anno Accademico 2018/2019
Sessione III

Ringraziamenti

Desidero ringraziare il Professor Paolo Bellavista per la sua disponibilità, la sua guida e per le opportunità datomi fornendo un grande contributo alla mia formazione. Ringrazio il Professor Jon Crowcroft per il suo supporto e per avermi trasmesso la voglia di intraprendere strade del tutto inesplorate. Ringrazio Poonam Yadav per aver condiviso con me idee risultate fondamentali per la parte iniziale e finale di questo lavoro. Ringrazio il mio compagno di ufficio Anwaar Ali per avermi sempre messo di buon umore nei momenti di buio totale. Ringrazio anche le persone del Computer Laboratory dell'Università di Cambridge per avermi accolto con entusiasmo all'interno del loro gruppo.

Grazie ai miei compagni di corso, che mi hanno sempre trasmesso una grande voglia di imparare e con i quali ho condiviso importanti momenti universitari e non.

Grazie ai miei coinquilini di Bologna (Marco, Nicola e Maria) per essere stati presenti anche durante questa ultima fase del mio percorso pur essendo lontani chilometri di distanza.

Infine un enorme grazie va ai miei amici di una vita, e in particolare alla mia famiglia che mi ha permesso di intraprendere questo percorso e mi ha sempre fornito un grandissimo supporto nei momenti più difficili che ho dovuto affrontare.

Contents

Abstract

Il costante avanzamento dei dispositivi Internet of Things (IoT) in diversi ambienti, ha provocato la necessità di nuovi meccanismi di sicurezza e monitoraggio in una rete. Tali dispositvi sono spesso considerati fonti di vulnerabilità sfruttabili da malintenzionati per accedere alla rete o condurre altri attacchi. Questo è dovuto alla natura stessa dei dispositivi, ovvero offrire servizi aventi a che fare con dati sensibili (p.es. videocamere) seppur con risorse molto limitate. Una soluzione in questa direzione, è l'impiego della specifica Manufacturer Usage Description (MUD), che impone al *maufacturer* dei dispositivi di fornire dei file contenenti un particolare pattern di comunicazione che i dispositivi da lui prodotti dovranno adottare. Tuttavia, tale specifica riduce solo parzialmente le suddette vulnerabilità. Infatti, diventa inverosimile definire un pattern di comunicazione per dispositivi IoT aventi un traffico di rete molto generico (p.es. Alexa). Perciò, è di grande interesse studiare un sistema di *anomaly detection* basato su tecniche di machine learning, che riesca a colmare tali vulnerabilità.
In questo lavoro, verranno esplorate tre prototipi di implementazione della specifica MUD, che si concluderà con la scelta di una tra queste. Successivamente, verrà prodotta una Proof-of-Concept uniforme a tale specifica, contenente un'ulteriore entità in grado di fornire maggiore autorità all'amministratore di rete in quest'ambiente. In una seconda fase, verrà analizzata un'architettura distribuita che riesca ad effettuare *learning* di anomalie direttamente sui dispositivi sfruttando il concetto di Federated learning, il che significa garantire la privacy dei dati. L'idea fondamentale di questo lavoro è quindi quella di proporre un'architettura basata su queste due nuove tecnologie, in grado di ridurre al minimo vulnerabilità proprie dei dispositivi IoT in un ambiente distribuito

garantendo il più possibile la privacy dei dati.

Introduction

The huge proliferation of Internet of Things (IoT) devices is an unavoidable reality, which has invaded most of daily environments. In fact, according to the Jupiter Research group [1], the number of connected IoT units will count 38.5 billion by the end of 2020, and due to the advancement of 5G networks it is foreseen to reach more than 64 billion of IoT devices by 2025 [2]. Most of IoT devices are expected to find their way in homes, enterprises, cities and even cars, by making "*smart*" most of the environments around each final user. Thus, the data generated contains most of the times sensitive informations, which, if exposed, could create a potential threat for the user privacy. In fact, although these devices are strong enough to host code, they do not have any security systems. Consequently, they generate new vulnerabilities in a typical network environment, which implies the opportunity for a malicious actor to either access to sensitive data or conduct other attacks, e.g. Distributed Denial of Service (DDOS). Hence, it is important to adopt countermeasures that reduce the attack surface and generate a more safe deployment for final users. In this direction different machine learning based solutions have been proposed to investigate IoT device traffic in order to detect abnormal behaviour in a network [3–5]. A potential solution, which has received a lot of attentions in both industry and academia and which is not machine learning based, is the Manufacturer Usage Description (MUD) specification [6]. It gives the opportunity for the manufacturer to identify each device type and to describe the network communications that the device requires to perform its intended function. The descriptions, called MUD files, consist of whitelist describing the devices' legitimate communication. However, their production can be a laborious process, because of the complex commu-

nication pattern of IoT devices. An example is the definition of communication pattern for IoT devices that behave similarly to general pupose devices (e.g. voice assistants). In fact, they have a large communication pattern range that, even if defined, can still be vulnerable to different attacks. Hence, the idea of supporting MUD with a machine learning algorithm that inspects the network traffic generated by IoT devices [7] is a feasible solution.

In this work three different implementations of MUD deployments have been explored, in order to create the right backgrounds useful to build a MUD compliant environment in local and small business network. Furthermore, the work introduces a new entity that gives more authority to an administrator, in order to define new communication patterns more suitable for the network environment in which MUD is deployed, by exploiting the description concept introduced by MUD. At the end of the work, a distributed architecture that makes MUD deployments ready for the Federated Learning approach is provided. This approach enables IoT devices to collaboratively learn a model while keeping all the training data on the devices, so that properties like privacy, ownership and locality of data are guaranteed. Thus, the aim of this project is to build an easy to deploy distributed architecture that improves security in IoT based environments by using these two technologies. The project will represent the first working architecture where both concepts are adopted in the same environment, with the common goal of **reducing vulnerabilities that afflict the IoT devices**.

The rest of the thesis is organised as follows. Chapter 1 describes goals, motivations and contributions of this work. Chapter 2 provides the technical backgorund needed to build the secure environment described. Chapter 3 outlines the related works in the state-of-art MUD deployments and Federated Learning environments. Chapter 4 defines the architecture infrastructure able to support the two core technologies of this work. Chapter 5 presents the core components of the architecture and their descriptions. For each of them, experiments that give an idea of how the component involved affects the

overall architecture performance are carried out. Furthermore, the chapter provides possible future directions which expand the project capabilities in order to further improve security and efficiency of the distributed architecture provided.

Chapter 1

Overview

The proceeding chapters of the thesis present challenges and solutions of deploy Manufacturer Usage Description(MUD) specification and Federated Learning capabilities in the same network environment, in order to create safe deployment conditions for IoT devices. Thus, the aim of this chapter is to provide the motivations that bring the work to choose these technologies and describe the overall architecture that emerges from them. Nowadays, it is increasingly common to find IoT devices in a home network, which implies to expose the network to new kind of vulnerabilities, as a consequence of their limited capabilities. Therefore, the main purpose of this work is to produce an easy to deploy architecture that improves security in IoT based environments. As a result of the "*easy to*" paradigm, the user interventions must be minimal. In this direction, the MUD standard, which requires few user interventions, has been approved by the academia and the industry as a method to reduce vulnerabilities in home networks. In order to employ it the user needs to act only on the components (e.g. network router) that allow to make the network MUD compliant. Whereas, the IoT devices can become MUD enabled with a simple firmware update, which requires minimal user actions. Hence, MUD has been chosen by this work as a starting point to improve the security in a home network.
The MUD employment allows the IoT devices to communicate only with services specified by the manufacturer, which implies a reduction of vulnerabilities and information exposure. For slightly more complex environments, the work provides a further entity

that increases the network administrator authority in MUD deployments and allows to make all the IoT devices MUD compliant. When the network includes cheap IoT devices for which the communications granted by the manufacturer are designed with the aim of selling data [8], the information exposure may become too high by network standards. Thus, the entity proposed allows an administrator to define new filtering rules that are more suitable for its network, by exploiting the concept of communication pattern description introduced by MUD.

Nevertheless, MUD is not intended to address network authorization of general purpose devices, which even include IoT devices with similar behaviour, as their manufacturer can not predict a specific communication pattern. Additionally, it is not intended to face up with attacks that compromise IoT devices. For example, if a malicious actor compromises a MUD compliant device, he is still able to conduct a Distributed Denial Of Service attack to the servers for which the communciation is granted by the manufacturer. Thus, the specification may be supported by a machine learning algorithm that investigates the network traffic in order to find unexpected behaviours. In this direction, the entity proposed, in addition to provide an administrator access point, can be used to host machine learning algorithms so that the router overloading is reduced. Considering that the informations produced by IoT devices most of the times include sensitive data, the work aims to create an architecture able to learn a model without looking at the data, which means preserving its privacy. Furthermore, such an approach minimises the machine learning entity overloading, as a result of distributing the model learning on edge devices. Thus, the work builds an infrastructure that makes the entity and devices within the network able to support this approach called Federated Learning. The latter provides for "*bringing the code to the data*" instead of "*bringing the data to the code*", by enabling the IoT devices to collaboratively learn a model while keeping all the training data on the devices.

After creating this architecture, the work provides all the evaluations needed to chose the components and to optimise the tradeoff between number of interactions and device overloading coming from the Federated Learning approach. Finally, the work produces

a network deployment including the Federated architecture and MUD in order to take advantage of the benefits of both, and explores some future directions to further optimise security and efficiency of the architecture provided.

In summary, this work **contributes** in giving:

- a detailed analysis of the existing MUD implementations, which helps in finding the most suitable implementation for the type of network used;
- a MUD based access point, which allows the network administrator to actively participate and improve the MUD deployment;
- a distribute architecture that enables **real devices** and network components for the Federated Learning approach;
- a distribute architecture able to support the **Transfer Learning** concept, i.e. (1) pre-training on public data, (2) fine-tuning on sensitive data;
- the first detailed analysis of real devices behaviour involved in a Federated Learning infrastructure;
- a direction to optimise the tradeoff that characterises all the Federated Learning environments;
- the first existing architecture deployable in a network that employs both the approaches with the common goal of providing the best secure environment for IoT devices.

Chapter 2

Technical background

This chapter provides the necessary technical background to understand the design choices of the system produced. In the first two sections, a general description of network traffic and different existing firewalls is provided. The following section describes the Manufacturer Usage Descriptions specification and analyses the features and limits of some implementations [9]. In the last section, a background on Federated Learning and some known infrastructures and frameworks is given.

2.1 Network traffic

The term *network traffic* refers to the amount of data moving across the network at a given point of time. The network packets encapsulate the most of network data, and provide the load in the network. Network traffic is the main component for network traffic **measurement**, **control** and **simulation**. The proper organisation of it, helps in the ensuring the quality of service of a given network.
The baselines of this work rely on the network traffic *control*, in order to manage, prioritise, control or reduce the network traffic. Additionally, to reduce the possible vulnerabilities within a network, new policies that allow or discard the forwarding of network packets are generated and then inserted in the Firewall (section 2.2). In the end, the work provides the network traffic *measurement* to evaluate the performance of the distributed

architecture produced.

2.2 Firewall

The firewalls are the core for the intranet security. They relies on predetermined security rules, which allow to monitor and control the incoming and outgoing network traffic. The idea is to establish a barrier (Fig. 1) between secured and controlled internal networks, which can be defined as trused, and untrusted outside network, such as Internet. Thus, the typical aims are:

- Establish controlled link.
- Protect the trusted network from Internet-based attacks.
- Provide a single choke point.

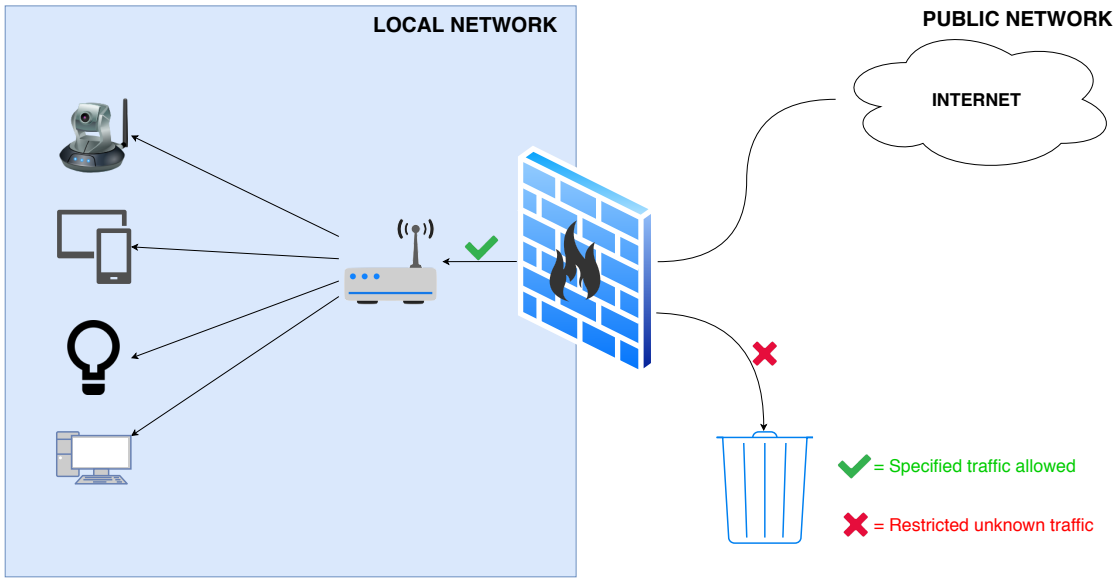


Figure 1: Firewall

By consequence, the Firewall itself should be immune to penetration, by meaning of having a trusted system with a secure operating system.
Typically, there are three general types of Firewall that host rules defining the authorised traffic allowed to pass:

1) **Packet-Filtering Router** applies a set of rules to each incoming IP traffic and then forwards or discards the packet. The packet filtering acts in both the directions and is typically set up as a set of rules based on matches to field in the IP (Internet Protocol) or TCP (Transmission Control Protocol) header.

2) **Application-level gateways** act as relay of application-level traffic. By using this process, it becomes easy to log and audit all the incoming traffic. However, there is an additional processing overhead on each connection.

3) **Circuit-level gateways** can be used as a stand-alone system or a specialised function perfomed by an Application-level Gateway. It relies on stetting up two TCP connection, one inside the intranet and one outside, so the gateway relays the TCP fragments from one connection to the other without examining the contents. The security function consists of determing which connection is allowed.

In addition to the use of a simple configuration of a single system more complex configurations are possible, but they are beyond the aims of this work. In fact, the type of Firewall included in this work is the **Packet Filtering**, because of its **simplicity**, **transparency** to users and **high speed**. Nevertheless, this simple configuration introduces some disadvantages, such as lacking of authentication and difficulty of setting up packet filter rules. The former, makes the Firewall prones to possible authentication attacks such as *IP address spoofing*, which can be solved by using some intranet authentication techniques (e.g. *WiFi Protected Setup*). The latter can influence performance and security of the firewall itself. In fact, according with A. Wool [10], the protection that firewalls provide is only as good as the policy they are configured to implement. Thus, being hard to set up, in the Packet Filtering configuration becomes easy to violate the well-established security guidelines. Always referring to [10], the quality of rules in terms of consistency and redundancy can influence the performance of the Router itself. Thus, it is good practice to follow some guidelines in the policies insertion procedure. In this work, the policies are provided directly by the manufacturer in order to create a communication pattern for each of its device. Nevertheless, these policies may be in

conflict with some administrator policies, which means having the performance and security issues described above. The future sections of this work analyse in more detail this problem.

2.3 Manufacturer Usage Description

The Manufacturer Usage Description (MUD) specification is used to allow a certain type of communications for IoT devices in a network. In fact, the goal is to provide a mechanism for edge devices to signal to the network what sort of access and network functionality they require to work properly. In order to do that, the manufacturer defines which is the behaviour that its devices must have. Hence, the MUD specification provides a standard way for the manufacturers to identify each device type and indicate the network communications that it requires to perform its intended function. When MUD is used, all the communication patterns not specified by the manufacturer are forbidden and so discarded.
According to IETF (RFC 8520) [6], a MUD deployment should consist of three architectural building blocks:

1) The **Manufacturer Usage Description file** (description), which is created by the device manufacturer to describe the device and its expected network behaviour.

2) A **Uniform Resource Locator** (URL), which is used to locate the *MUD file* from manufacturer's server.

3) A **mechanism** for local network management systems to retrieve the description.

The main components that result from these building blocks are: the **MUD File server** a web server that hosts MUD file (provided by the Manufacturer), and the **MUD Manager** the system that requests and receives the MUD file from the MUD file server.
The workflow between these two agents and other network components is shown in Fig. 2.

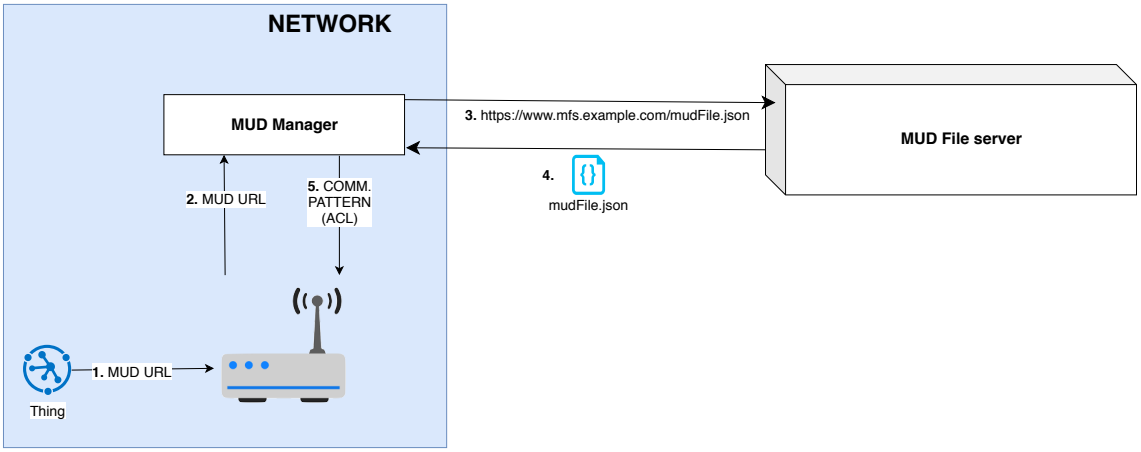


Figure 2: Manufacturer Usage Description workflow

In ①, when a thing adds in the network, sends the MUD-URL with a X.509 certificate or Dinamic Host Configuration Protocol (DHCP) or Link Layer Discovery Protocol (LLDP) (depending on the implementation). In ②, the local router (as Network Access Device (NAD)) sends the MUD-URL extrapolated to the MUD Manager. The latter, in case of X.509 authentication, validates the signature and requests (③)the MUD file to the manufacturer's file server, by using the MUD-URL to locate it. The MUD file retrieved (④), is a YANG-based JSON file (IETF RFC 7951 [11], e.g. Fig 3) signed with a public key signature from the device manufacturer. The MUD manager validates and processes the MUD file and stores the file locally for until its cerificate is valid and generates the Access Control List(ACL). In ⑤, the MUD manager sends the ACL to the NAD, which enforces the ACL received from the MUD Manager to all the traffic passing through it. The MUD manager may be a component of an Authentication, Authorization, and Accounting (AAA) system or a network management system. Introducing a AAA system means that between the step ② and ③ of the workflow just described, two additional steps must be added. First, the local router sends the MUD-URL to AAA server for the request authentication. Second, the AAA server sends the authenticated MUD-URL to the MUD manager. The AAA system [1] is not mandatory, but it provides a better level of authentication between the routers/switches and the MUD Manager in case of differnet subnets.

[1]There are different existing protocol for the AAA systems e.g. *Radius* and *Diameter*

```
"to-device-policy": {
  "access-lists": {
  "access-list": [
    {
      "name":"mud-96898-v4to"
    },
    {
      "name": "mud-96898-v6to
         "
    }
  ]}
}
```

```
"acl": [
{
 "name":"mud-96898-v4to",
  "type": "ipv4-acl-type",
  "aces": {
  "ace": [
  {
    "name": "cl0-todev",
    "matches": {
      "ipv4": {
        "ietf-acldns:src-
           dnsname": "cloud-
           service.example.
           com"
      }
    },
      "actions": {
        "forwarding": "accept
           "
      }
  }]}
}
```

Figure 3: Example of to-device-policy defined in a MUD file

According to the MUD specification, the MUD-URL, as described in the workflow, can be emitted in different ways:

- **DHCP option** that the DHCP client uses to inform the DHCP server. The DHCP server may take further actions, such as acting as the MUD manager or passing the MUD-URL along to the MUD manager.
- **X.509 constraint**. In this case, the IEEE has developed IEEE 802.1AR to communicate device characteristics. Various means may be used to communicate the certificate. By using this means, the devices can not be spoofed without detection, even if the device is compromised.
- **Link Layer Discovery Protocol** (LLDP).

It is possible that there may be other means for a MUD-URL to be learnt by a network. For instance, some devices may already be fielded or have very limited ability to communicate a MUD-URL, and yet they can be identified through some means, such as a serial number or a public key. In these cases, manufacturers may be able to map those identifiers to particular MUD-URLs (or even the files themselves).

From the previous analysis, can be concluded that MUD specification has provided enough flexibility and scope to choose different implementations to incorporate requirements of various deployment scenarios. The next sections focus on investigating three different implementations of MUD deployments: Cisco proof-the-concept (PoC) implementation [12], National Institute of Standards and Technology (NIST) [13] implementation based on Software Defined Network (SDN) and OpenSource MUD [14] developed by a consortium of companies (Cable Labs, Cisco, CTIA, Digicert, ForeScout, Global Cyber Alliance, Patton, and Symantec) in device manufacturing and network security, coordinated by a cyber security firm, MasterPeace Solutions.

2.3.1 Cisco Proof-of-Concept

The Cisco MUD deployment is an open-source proof-of-concept (PoC) implementation, which is intended to introduce advanced users and engineers to the MUD concept. The current version does not provide automated MUD manager implementation, and some protocol features are not supported currently [12]. The PoC deployment shown in Figure 4, is designed with a single device serving as MUD manager and *FreeRADIUS server* (as a AAA Server), which interfaces with the Catalyst 3850-S switch over TCP/IP. The Catalyst 3850-S switch contains a DHCP server that is configured to extract MUD-URLs from IPv4 DHCP transactions. In the implementation, the *things* are leveraged to use, in order to send the MUD-URL, either DHCP or LLDP. The Cisco MUD Manager is built as a callout from *FreeRADIUS* Server and uses MongoDB to store policy information. The MUD manager is configured from a JSON file that varies slightly based on the installation. This configuration file provides a number of static bindings and directives

as to whether both egress and ingress ACLs should be applied.

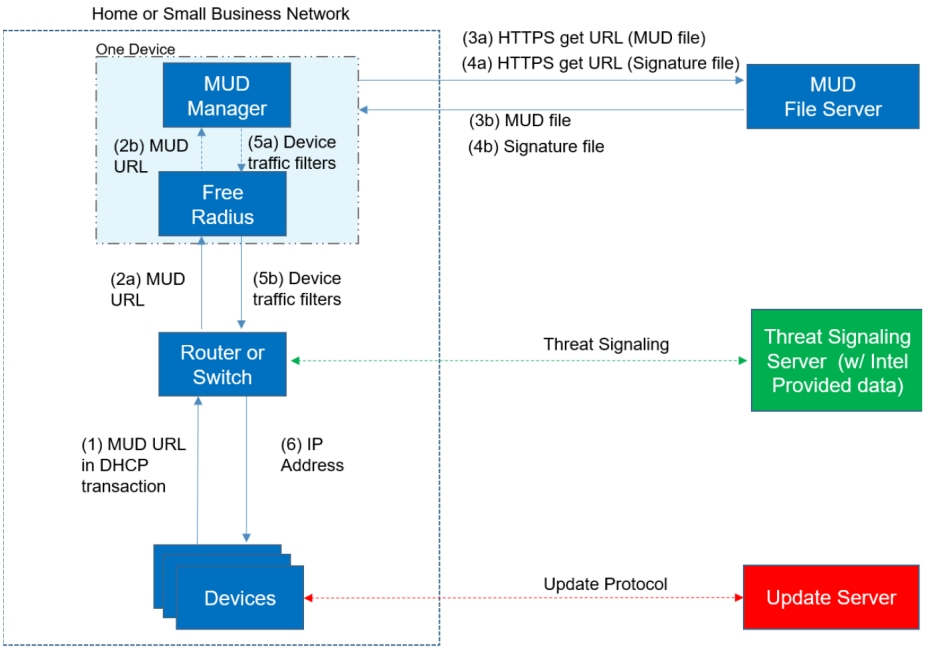


Figure 4: Logical Architecture of Cisco MUD implementation [12]

Along with the standard MUD implementation, Cisco initial architecture also proposed two additional components, Threat Signalling Server and Update Server. The former should refers, in case of DDoS Open Threat Signaling (DOTS), to the IETF RFC 8612 [15].

The limits of this implemetation are mainly related to the static configuration of the MUD Manager. In fact, it does not invoke a DNS resolution service to automatically resolve the fully qualified domain names (FQDNs), but the resolution is performed manually by human operator before beginning execution of the MUD manager service. Thus, this address resolution information remains static for as long as the MUD manager service continues to operate. This limitation makes deployment of this implementation in real environments impractical because of the following reasons:

- **Scalability**: manual configuration of DNS resolution.
- **Availability**: add new FQDN address resolutions or change existing ones man-

ually. The MUD manager process needs to be restarted to pick the new DNS changes from the JSON configuration file.

Furthermore, ingress Dynamic ACLs (DACLs), i.e., the traffic that is received from external sources to the network and directed to local IoT devices, are not supported by this version. Consequently, even if a MUD-capable IoT device's MUD file indicates that the IoT device is not authorised to receive traffic from a particular external domain, the DACL that is needed to prohibit that ingress traffic is not configured on the switch. Therefore, it is still vulnerable to attacks originating from that domain, even though the device's MUD file makes it clear that the device is not authorised to receive traffic from that domain. However, if an attacker is able to get packets to an IoT device from an outside domain, it is not be possible for the attacker to establish a TCP connection with the device from that outside domain, thereby limiting the range of attacks that can be launched against the IoT device, this thanks to the fact that the egress DCL are supported.

2.3.2 SDN based implementation

This implementation is based on the concept of Software Defined Networking (SDN), data plane and control plane separation, and uses OpenFlow SDN switches. An OpenFlow switch implements flow rules in the data plane, by arranging rules in more than one flow tables. The SDN controllers take the responsibility of inserting and updating the rules dynamically (when a packet arrives at the controller) or proactively when the switch connects to the controller. Ranganathan et al. ([13]) proposed a SDN based MUD architecture focused on achieving scalability of the class-based rules at SDN switch (Fig.5). It uses SDN flow rules in three flow tables: the first two, classify source and destination MAC address, whereas the third implements the MUD Access Control Entries (ACEs) with rules that stated in terms of packet classification metadata that is assigned in the first two tables. Once this flow pipeline is generated the packet being finally sent to a table that implements L2Switch flow rules, which is provided by another application.

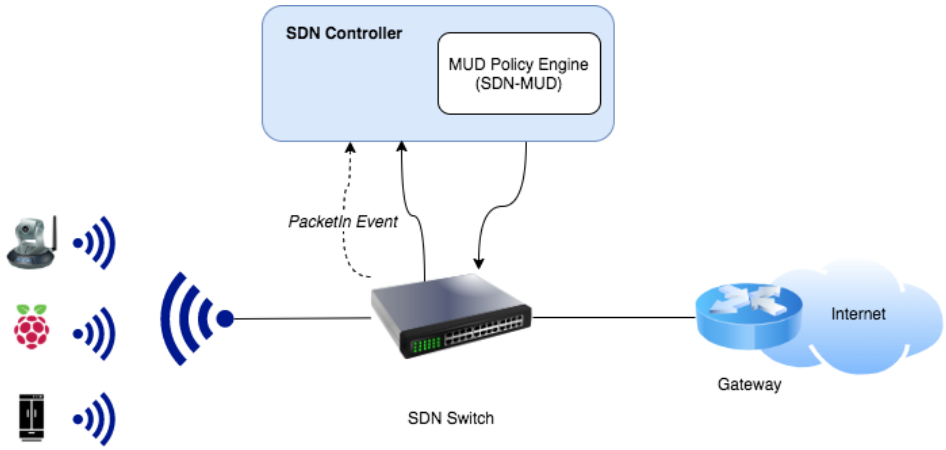


Figure 5: SDN based MUD implementation example

This scheme requires that a packet must be processed in the controller and a rule installed before packet processing may proceed. The first rule in the MAC address classification stage (first two flow tables) that is installed when the switch connects, sends the packet to the controller but not to the next table. Thus, a packet may not proceed in the pipeline before it can be classified. To address this problem, this implementation loosens up the interpretation of the ACE specification. It defines a "relaxed" mode of operation where packets can proceed in the pipeline while classification flow rules are being installed. This may result in a few packets being allowed to continue, in violation of the MUD ACEs with the condition that the system becomes eventually compliant to the MUD ACEs. Consequently, these packets could get through prior to the classification rule being installed at the switch. This could result in a temporary violation of the ACEs [13].

2.3.3 Open Source MUD

Open Source MUD (osMUD) implementation is developed by a consortium of device manufacturing and network security companies [14]. Comparing this architecture, illustrated in Fig. 6 , with those described in previous sections, it is noticeable that the MUD Manager directly run on the router. Thus, only the routers that have some characteristics are suitable for this architecture. In fact, osMUD is currently designed to

easily build, deploy, and run on Open Wireless RouTer (OpenWRT) platform[2], which limits the router choice to the only ones compliant with this platform. Furthermore, the architecture integrates *dnsmasq* in order to extrapolate from the DHCP packets the MUD-URL, which is designed to be lightweight, provide network infrastructure services and to have a small footprint, which means being suitable for resource constrained router and firewalls.

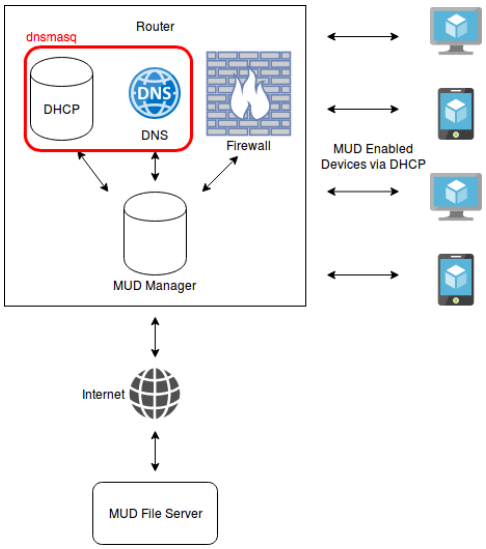


Figure 6: osMUD Architecture [14]

The limits that emerge from this MUD Manager implementation are listed below:

- it can run on OpenWRT platform, which means that routers must be OpenWRT compliant;
- it can be compiled outside of OpenWRT for most C compatible environments. Running osMUD outside of OpenWRT requires to use compatible firewall, and a DHCP server that can extract the MUD-URL from the DHCP header packet for MUD Enabled Devices;
- the current implementation does not have MUD file rules for lateral movement, thus attackers can progressively move through a network, searching for targeted key data and assets;

[2]OpenWrt is a Linux operating system targeting embedded devices that instead of trying to create a single, static firmware, it provides a fully writable filesystem with package management [16].

- it must use osmud-dnsmasq to read the DHCP header option for MUD enabled devices.

2.3.4 Security challenges

This section emphasises some MUD infrastructure and security concerns not addressed by the specification. Considering the description in 2.3, from an infrastructure perspective it is noticeable that:

- when MUD Server signing certificate gets expires, while MUD file is still deployed and in function locally, the MUD Manager should fetch a newly signed file with signature and updates rules without interrupting IoT devices availability and functionality;

- MUD enforced rules does not provide sufficient fine-grained control. Individual users may have their deployment setup, which may require additional rules (rules conflict problem see 2.2); thus, a mechanism to incorporate the user policy to filter traffic to provide more fine-grained control is needed;

- if a Threat Signalling Server is included, a mechanism to incorporate updates from it is needed. The further question and investigation in this direction is - should they go through MUD file or through a generalised common server updates (and filters) for all IoT devices.

All the elements listed above, represent degrees of freedom for the implementation. Furthermore, Users implementing MUD are advised to keep some security considerations in mind. For example, the specification does not provide any inherent security protections to IoT devices themselves. If a device's MUD file permits to receive communications from a malicious domain, traffic from that domain can be used to attack the device itself. Similarly, if the MUD file permits an IoT device to send communications to other domains, and if the IoT device is compromised, it can be used to attack those other domains. Further considerations to keep in mind can be found in section "*Security Considerations*" of [6].

In conclusion, there are some sophisticated attack traffic that can still pass undetected, by using spoofing techniques. This is due to lack of device authentication (if X.509 is not used as MUD-URL emission method), and to some badly defined MUD communication patterns. In fact, MUD is not intended to address network authorization of general purpose computers, as their manufacturers can not envision a specific communication pattern to describe. In addition, even those edge devices that have a single or small number of uses might have very broad communication patterns (such as personal assistant devices). Thus, in order to identify such threats an additional system that monitors the activities of all the MUD enabled devices flows is needed. The next section provides the backgrounds of a new distributed machine learning technique, which can be used to monitor the network traffic of MUD enabled IoT devices and which aims to preserve data privacy.

2.4 Federated Learning

Federated learning is a distributed machine learning approach, which enables the devices (IoT, smartphone etc.) to collaboratively learn a model while keeping all the training data on the devices. Thus, if in standard machine learning algorithms the approach is "*bring the data to the code*" in this case the approach becomes "*bring the code to the data*" by addressing fundamental problems of privacy, ownership and locality of data.
Federated Learning applies best in situations where on-device data is more relevant then the data that exists on servers, is privacy sensitive, or otherwise undesirable or infeasible to transmit to the server. The typical actions that a Federated Learning procol involves are: (1) send the global model pre-trained (if exists, otherwise might be a model with random weights) to the clients, (2) train the model with local data, (3) sending the model updated back to the server and finally (4) aggregate all the updates with the global model, by using an aggregation technique (e.g. Federated averaging). Note that these actions, in order to optimise the training procedure, could be repeated more than once, which means creating the concept of **round**.
Current application of Federated Learning regards only **supervised learning** tasks,

such as:

a. On-device item ranking

b. Content suggestions for on-device keyboards

c. Next word prediction

d. Anomaly detection

In the next two subsections, two examples of protocol that involve the Federated Learning in two of the learnig tasks listed above are exposed. Whereas, the next two subsections provide two work in progress frameworks, which can be used to simulate (and also create) a Federated Learning environment. Subsequently, two algorithms which are fundamental for the global model computation and its privacy are described. The last subsection defines which are the problems and the challenges that affect the Federated Learning infrastructure.

2.4.1 Basic protocol

The first Federated Learning protocol proposal given by its designers [17], provides a system able to train a deep neural network on data stored on smartphones.
As illustrated in Fig.7, the participants to this protocol are *devices* and a *FL server*, which is a cloud-based distributed service. The *devices* announce to the *server* that they are ready to run an *FL task* for a given *FL population*. The former, is a specific computation for an *FL Population*, such as **training** to be performed with given hyperparameters, or **evaluation** of trained models on local device data. The latter, is specified by a globally unique name which identifies the learning problem, or application, which is worked upon.

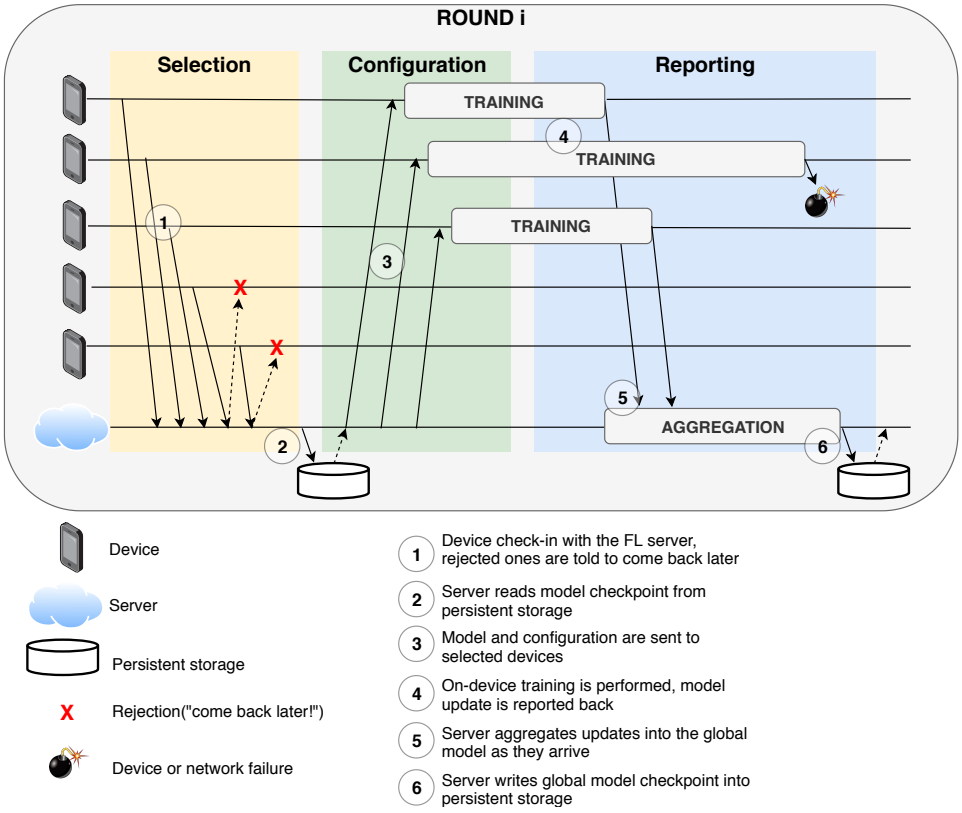


Figure 7: Federated Learning Setup for mobile phones [17]

This system involves three phases:

1) **Selection** Devices that meet the eligibility criteria (e.g. charging and connected to an unmetered network) check in to the server by opening a bidirectional stream. The server selects a subset of connected devices based on certain goals like the optimal number of participating devices (typically a few hundred devices participate in each round).

2) **Configuration** The server is configured based on the aggregation mechanism selected for the selected devices.

3) **Reporting** The server waits for the participating devices to report updates. As updates are received, the server aggregates them using the aggregation technique selected and instructs the devices when to reconnect.

The implementation is based on Tensor Flow [18], and uses as model aggregator the

Federated Averaging algorithm. This system has been tested in some large scale applications, such as the realm of a phone keyboard [19].

2.4.2 Anomaly detection protocol: DÏoT

The system provided by Nguyen et al. [20] implements another application of Federated Learning. In fact, the Federated protocol is applied as anomaly detection system and, if in the previous case the protocol participants were smartphones, the devices involved are either IoT devices or any kind of limited resources devices.

As shown in Fig. 8, the system consists of different *Security Gateways*, one for each devices, and a *Security Service*. The former, acts as the local access gateway to the Internet to which IoT devices connect over WiFi or Ethernet, and hosts the Anomaly Detection component. The latter, supports *Security Gateway* by maintaining a repository of device-type-specific anomaly detection models. When a new device is added to the local network, Security Gateway identifies its device type and retrieves the corresponding anomaly detection model for this type from IoT Security Service.

In more detail, the phases of the Federated Learning process are six. In the first two phases, the Security Gateway asks for an initial model, which could be either with random weights or pre-trained. This model represents the global model of the network managed by the *IoT Security Service*. In the next step, the global model is trained locally by each *Security Gateway* using data collected by monitoring the communication of the IoT device related to it. Each model updated is then sent to the *IoT Security Service*, which then aggregates all the model received to obtain a new global model. The re-training of the model is performed on a regular basis to improve the accuracy.

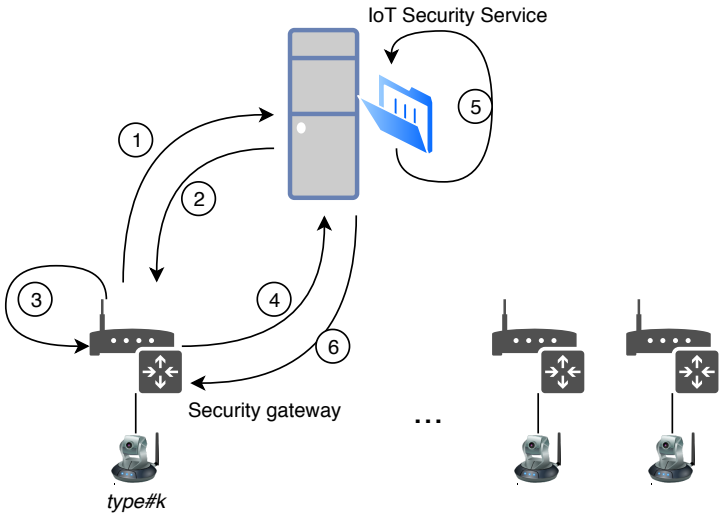


Figure 8: Federated Learning setup for IoT devices [20]

The implementation of this system, is based on *flask* [21] and *flask_socketio* in order to distribute the model, and Keras [22] for the model creation and training.

2.4.3 Tensor Flow Federated

This framework provided by the Tensor Flow team relies on graphs creation [23]. In fact, the server sends *TensorFlow graphs* to the edge nodes where they are executed, which means running machine learning operations (training, evaluation or inference) on local data. Thus, the server becomes a sort of device *orchestrator* that coordinates all the operations executed edge side.
The framework provides all the baselines needed for a first Federated Learning environment simulation. However, the framework has two important requirements:

- the edge nodes have installed the last version of Tensor Flow, in order to avoid version conflicts;
- the model must be serializable as *TensorFlow graph*.

Furthermore, it supports, at the time of writing this thesis, only **local execution simulation** [24], which makes this framework unsuitable for real case scenario.

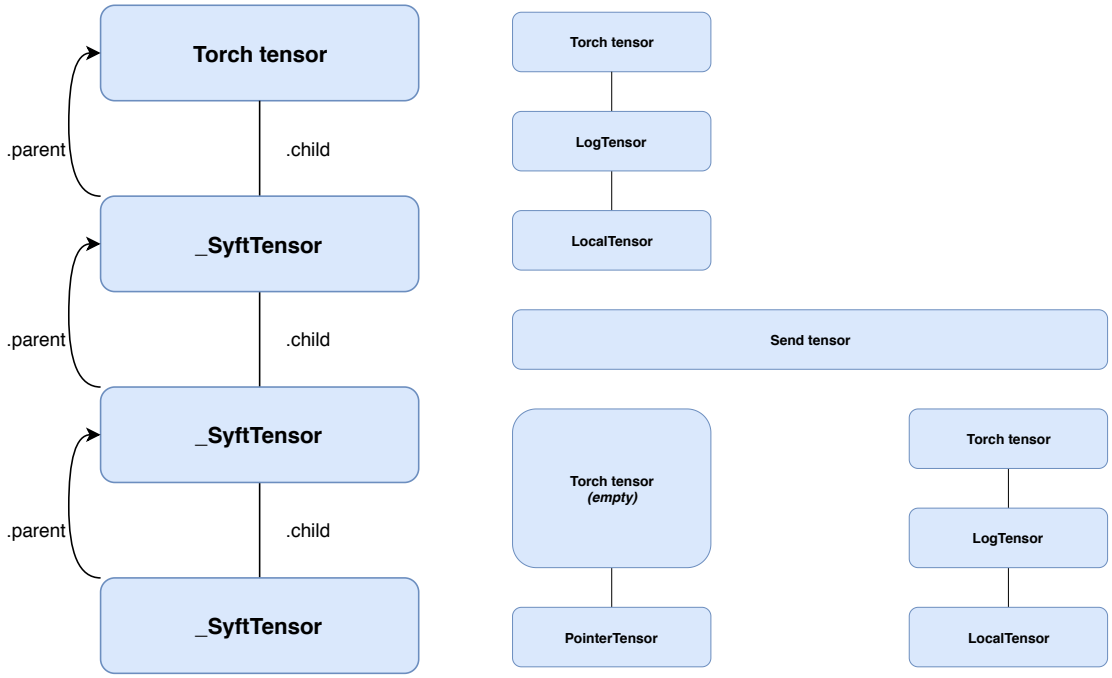
2.4.4 PySyft

The PySyft framework is designed for privacy preserving deep learning [25]. It introduces a representation based on chains of commands and tensors, by allowing to implement different privacy preserving constructs, such as Federated Learning and Secure Multi-party Computation. Furthermore, it is build on top of **PyTorch** (machine learning framework [26]), which means that is able to provide transparent APIs for privacy preserving deep larning to PyTorch users. Recently, the support to Keras [22] and so Tensor Flow has been provided.
Compared with the previous framework, PySyft provides a support for **remote operations** through Web Sockets [27], which makes it suitable for real case scenario. Thus, this framework provides:

- a standard protocol to communicate between worker nodes, which made the federated learning possible;
- a chain abstraction model on tensors to efficiently ovverride operation, such as sending/sharing a tensor between worker nodes;
- the elements to implement privacy methods such as multiparty computation (locally) and training on worker nodes' data.

Performing transformations or sending tensor to other workers can be represented as a chain of operations, and each operation is embodied by a special class. To achieve this, the framework provided an abstraction called *SyftTensor*. The SyftTensors are meant to represent a state or transformation of data and can be chained together. The chain has at its head a PyTorch tensor, and all the transformations and states embodied by the SyftTensor, can be accessed in both the direction (downward and upward). This structure, illustrated in Fig. 9a, suggests that all the operations are first applied to the PyTorch tensor, and then propagated to the rest of the chain. There are two important subclasses of SyftTensor: *LocalTensor*, which is created automatically when the Torch tensor is instantiated, and the *PointerTensor* which is created when a tensor is sent to a remote worker. The former, is used to perform on Torch tensors native operations

corresponding to the overloaded operation. The latter, is used to manage sending and getting back of a tensor. When this happens, the whole chain is sent to the worker and replaced by a two-node chain: the tensor, now empty, and the PointerTensor which specifies who owns the data and the remote storage location (Fig. 9b).
Thus, the chain structure described above allows the framework to provide all the baselines needed for the development of a Federated Learning environment with real devices.



(a) General structure of a tensor chain

(b) Impact of sending a tensor

Figure 9: PySyft framework: tensors structure (source [25])

2.4.5 Federated Averaging

Most deep learning trainings rely on variants of Stochastic Gradient Descent (SGD), where the gradients are computed on small sample of huge dataset and then used to make one step of gradient descent. In the federated settings this approach is applied on a random C-fraction of clients on each round, and compute the loss using all the data held by these clients. Therefore, C controls the global batch size, with C=1 corresponding to the gradient descend (not Stochastic) on a signle client. This algorithm is called **Federated SGD** (FedSGD).

McMahan et al. [28] provide a generalization of FedSGD, which allows the local clients to perform more than one batch update on local data and exchanges the updated weights rather than gradients. In particular, each client takes locally a step of gradient descent on current model using its local data, and the server then takes a weighted average of resulting models. Now that this interaction has been defined, it is possible to add more than one computation to each client by iterating the local update multiple times before the averaging step. This algorithm is called **FederatedAveraging**.

2.4.6 Secure Multi-Party Computation

The Secure Multi-Party Computation (SMPC) aims to compute a function on encrypted data without using keys. The data is divided among two or more different parties; the aim becomes to run a function (e.g. data mining algorithm) on the union of the parties without allowing any party to view another individual's private data, so that only the information learnt by the individuals is the function output. The classical example used to make more clear the SMPC scenario is the hospitals case, where they wish to jointly mine their patient data for medical research purpose. For patient privacy reasons, these hospitals can not reveal the patient data to each other, by consequence the classical data mining algorithm can not fit in this scenario. Thus, it is necessary to find a solution that enables the hospital to compute the desired data mining algorithm on the union of their data, without ever revealing them. This example becomes in the Federated Learning scenario as a number of distinct, yet connected, edge-devices that wish to carry out a joint computation of the same function (model learning). In these conditions, the aims of SMPC is to enable the parties to carry out such distributing computing tasks in a secure manner. However, the secure multiparty computation is concerned with the possibility of deliberately malicious behaviour by some adversarial entity. For example, the aim of this attack may be to learn private information or cause the result of the computation to be incorrect. Consequently, two important requirements to be satisfied by any secure computation protocol are **privacy** and **correctness** [29]. The former means that nothing should be learnt beyond what is absolutely necessary;

parties should learn their output and nothing else. The latter implies that each party should receive its **correct output**. Thus, a malicious actor must not be able to cause the result of the computation to deviate from the function that the parties had set to compute.

Only in recent years interest has arisen in pratical side. In this direction one protocol proposed is **SPDZ** (nickname of the MPC protocol defined by Damgård et al. [30]). This protocol is secure against active static malicious actors, and tolerates corruption of n-1 of the n parties. It is characterised by two phases: the *offline* phase where some shared randomness is generated, but neither the function to be computed nor the inputs need to be known; *online* phase in which the actual secure computation is performed. The main advantages of this protocol, is that the performance of the online phase scales linearly with the number of participants, and the basic operations are almost as cheap as those used in the passively secure protocol (in [31] an improvement of this protocol can be found).

2.4.7 Problems and challenges

In this section some of the problems and challenges that affect the Federated Learning concept from different perspective are summarised.

The Federated Learning structure, described in previous paragraphs, may suggest novel challenges due to the heterogeneity of the devices involved. First of all, in terms of device computational performance, which can influence the model training time especially in synchronous cases. Secondly, the moment in which a device is available could be different from the others, which can affect the overall infrastructure training start time. Additionally, the devices may also be unreliable, by meaning of dropping out the communication at a given iteration due to connectivity or energy constraints. Even the communication represents a sore subject for the Federated networks. In fact, the network communication can be much slower than local computation. Thus, a communication-efficient method that reduces as much as possible the bandwidth used and that guarantees that the model converges in a reasonable time is needed. For example, by applying a random mask to

the model weights in order to reduce the its dimension [32]. Furthermore, it is possible that one participant device, which produces rare and unique data, causes the data memorization in the model. In order to avoid this problem, a mechanism that limits how much a device can contribute on the model, by adding noise to its data, is needed (e.g. *differential privacy*).

From a security perspective, as confirmed by the designers [17], the training with malicious participants is a realist threat. In fact, the Federated Learning by design has no visibility into how the model updates are generated. Thus, an attacker can exploit this property to compromise the training data and so change the model's behaviour (**data poisoning**). Furthermore, a malicious participant can use model replacement to introduce a backdoor functionality in the global model (**model poisoning**) [33]. For example, a backdoored word-prediction model could predict attacker-chosen words for a certain sentences. Defensens against backdoors use techniques such as fine-pruning, filtering or various types of clustering. Typically, an additional machine learning algorithm, which analyse the weights of the model updates received, is adopted to detect this attack. Obviously, if the Secure multi-party computation is used, it is impossible to detect anomalies in models submitted by participants in the Federated network, due to the weights encryption.

In summary, the Federated Learning has some gaps due to its distributed nature, which influence its infrastructure creation and deployment. Even in terms of security, it still requires a lot of research. In fact, as highlighted above, there are some vulnerabilities, such as *data poisoning* and *model poisoning*, still not solved. Thus, in this work, the design and deployment of a Federated Learning architecture consider some strategies that fill, as much as possible, the gaps presented in this section.

Chapter 3

Related works

The MUD specification [6] has been approved as a method to define IoT devices communication pattern in order to reduce vulnerabilities in home networks. The standard allows and encourages the IoT manufacturers to provide a MUD file consisting of access control rules that describe the device's proper communication behaviour [11]. There are several recent works that focus on implementing and extending MUD. A scalable implementation of the MUD specification in a Software Defined Network was presented by Ranganathan et al. [13]. From an extension point of view, Afek et al. [34] extends MUD in order to enforce the MUD file whitelist rules at ISP (Internet Service Provider) layer, by combining an NFV (Network Function Virtualization), which monitors many home networks simultaneously, with router/switching filterning capabilities. However, MUD is typically supported by a machine learning algorithm, because its employment simply reduces the attack surface. In this direction, Hamza et al. [35] proposed an anomaly detection system based on deploying classical machine learning algorithms and MUD together in a SDN network, in order to detect volumetric attacks. Always the Hamza group presented a tool to automatically generate the MUD file, by simply monitoring network traffic [7].

Conversely, the system provided by this work is deployed together with a different approach of machine learning called Federated Learning [17], in which the model is learnt collaboratively by edge devices. At the moment of writing this work, this is the only

MUD solution that is deployed together with a Federated Learning approach, with the common aim of reducing possible attacks that involves IoT devices. The Federated Learning design fundamental are defined by Bonawitz et al. [17], where a basic protocol for the model propagation is proposed. Nguyen et al. [20] suggest another communication protocol designed with the goal of enabling IoT devices to be Federated Learning participants. Furthermore, they propose an approach to perform device-type-specific anomaly detection. Another problems that afflict the Federated Learning environment regards the optimization of model distribution among the devices. Konečnỳ et al. [32] presented two ways to reduce the uplink communications cost based on random mask applied to the model weights. In the same direction Wang et al. [36] provide an algorithm to optimise the frequency of global aggregation from a theoretical point of view, so that the available resource is most efficiently used. Some guidelines to adopt in order to optimise the model distribution for static case (not dynamic adaptation) are presented by this work, after evaluating the Federated architecture 5.2.2.4.

Recently Bagdasaryan et al. [33], in addition to the classic data poisoning, demonstrate that a malicious participant can use model replacement to introduce functionality into the joint model (model poisoning). In order to reduce this vulnerability an approach able to distinguish between normal and IoT devices should be used. In this direction, the Bremler-Barr group [37] introduced some classifiers able to confirm if a device is IoT or not in a short time scale. In order to deal with this problem, this work proposes a distributed architecture that exploits the MUD concept to make the Federated Learning environment able to distinguish IoT devices (MUD compliant) from the other (4.3 and 5.3).

Chapter 4

System design

In this chapter the design choices that lead this work to the final architecture are motivated. The chapter follows a bottom-up approach, in order to produce the pillars of different sub-systems that in the final phases are unified. In particular, the sub-systems design embraces two new approaches with the common aim of acting on devices with limited resources. The first approach chosen for system design is the **Manufacturer Usage Description** (MUD) specification, which is focused on lock down and verify rigorously IoT devices communication patterns. It has been adopted for several reasons. First of all, it is easy to deploy from a user perspective and reflects perfectly the objectives of this work, i.e. reduce as much as possible the vulnerabilities generated by IoT devices in a home network. In fact, it reduces the communication allowed for an IoT device to those defined by the manufacturer, which means that the devices can perform their intended functions without having unrestricted network priviliges. Secondly, it receives a lot of attentions in both industry and academia, and it addresses vulnerabilities even in systems that are no longer suppported or where patching the system is infeasible.
Subsequently, considering that the informations produced by IoT devices most of the times include sensitive data, the work focuses on creating an architecture able to learn a model without looking at the data. The reason why the work focuses on introducing a machine learning algorithm in support of MUD deployment, is related to filling up some of the security gaps that characterise the MUD standard. Hence, the second approach

involved in this work is the **Federated Leraning**, which enables a way to collaboratively learn a shared model while keeping all the data on the edge devices. Considering the work focus described above, the approach has been chosen as result of its properties of preserving privacy, ownership and locality of data, which are obtained by distributing the model learning computation across the devices.
Thus, previously described MUD implementations are analysed first(2.3.1, 2.3.2 and 2.3.3) to lay down the foundations of a MUD enabled environment. In order to give more authority to the network administrator in a MUD deployment and make all the IoT devices within a network MUD compliant, the User Policy Server is presented and the directions that arise from it are emphasised. It extends the classical MUD environment to give more authority to the network administrator and fills up some of the MUD manager gaps. Additionally, a distributed architecture able to support the Federated Learning concept applied on real devices is explored. In order to support a real device deployment, the architecture extrapolates the features from the two solutions already described (2.4.1 and 2.4.2) suitable for real case scenario. Finally, the sub-systems previously obtained are pieced together in order to produce the design of a more complex system based on MUD and able to execute learning of a model on the edge devices, by exploiting the benefits of both approaches.

During the design analysis outlined in the following sections, the actions adopted to solve some of the challenges described in the chapter 2 are underlined.

4.1 Manufacturer Usage Descriptions environment

4.1.1 Identification of an easy-to-use MUD manager

Considering the characteristics of the three existing MUD manager implementations 2.3.1, 2.3.2 and 2.3.3, the aim in this work is to find the most suitable and **easy-to-use implementation** for either a typical user network or a small business network. The term *user network* refers to a canonical local network composed by either *router* or *switch*,

Implementation	Support	URL via DHCP	URL via LLDP	URL via x509	URL via another ways	Component of an AAA system	Sig Verification
Cisco-MUD	Cisco Catalyst 3850	Yes (no DHCPv6)	Yes	No	No	Yes	OpenSSL
NIST-MUD	OpenVSwitch	Yes	No	No	with MAC + MUD	No	No
osMUD	Run on openwrt	Yes	No	No	No	No	OpenSSL

Table 1: Comparison of MUD implementations

which provides the Internet access, and user devices. Thus, this identification procedure represents the first existing analysis of different MUD implementations and even the first challenge that has to be addressed during the design of this work, which means understanding which of these implementations is suitable for a typical *user network*.
It has been shown that the Cisco's work (2.3.1) makes use of static files in order to apply the DNS resolution service, which means that the address resolution informations remain static for as long as the MUD Manager continues to operate. Furthermore, without considering other limits related to the ACLs dynamicity, it generates rules adoptable only by the Catalyst 3850-s switch. This implementation, as also underlined by Dodson et al. [12], is so intended to introduce advanced users and engineers to the MUD standard, without following the easy-to-use or plug-and-play paradigm. Thus, the Cisco's work does not respect the constraint imposed by this work.
In order to understand why the MUD manager solution based on SDN (2.3.2) has not been used, a brief definition of Software Defined Networking is needed. The SDN concept allows separating the data plane from the control plane, which becomes software defined. The partition allows the network operators to make their networks easy to manage and customizable, as well as free from the *vendor lock-in*. Thus, the setting up of a Software Defined Network is not intended for the typical user and in addition requires new components. Furthermore, this solution, at the moment of writing this work, does not support the MUD file validation (Table 1), which is considered a "*must*" by the standard [6]. Even if is easy to make it completely MUD compliant (by adding the OpenSSl validation), for the reason previously described, it is not used as MUD manager solution in this work.

After examining and excluding two implementations, the only one left is Open Source MUD manager (2.3.3). As already described, osMUD is currently designed to easily

build, deploy and run on Open Wirless RouTer (OpenWRT) platform. Hence, the design has to face up with the first big constraint, which led it to choose routers OpenWRT compliant[3] (they represent the most of general purpose routers). This type of choice makes the deployment of osMUD not so obvious. However, beside the drawback just mentioned, the setting up of osMUD on a router running OpenWRT is accessible to any users. Thus, the choice falls on this implementation.
By examining the osMUD design, illustrated in Fig. 10, the MUD manager cooperates with two central components:

- **dnsmasq**, which extracts the MUD-URL, by using the DHCP option (161 for DHCPv4 and 112 for DHCPv6), and manages the address resolution informations;
- **firewall**, based on Packet-Filtering (2.2), where rules extrapolated from MUD files by the MUD manager are enforced.

It is worth noting that the dnsmasq used is a custom implementation, which is provided by the osMUD designers. It records each DHCP request in a file, that is processed by osMUD to detect which are the MUD compliant devices. In this file the MUD-URL is also inserted, which represents the **distinction key** between general devices and MUD compliant devices. The multiple DHCP requests that come from the same device, can cause multiple MUD file requests for that device. In order to manage this problem, the file produced by dnsmasq includes different states, which makes the MUD manager able to avoid multiple requests and able to delete rules of retiring devices (DHCP release).
The MUD manager after receiving and validating a MUD file from the MUD File Server (MFS), starts the parsing procedure. In this phase, the YANG-based MUD file is processed and, considering the specfication [6], all the rules are generated. Hence, osMUD enforces the rules extracted from the file in the router firewall, by calling a script that injects Packet-Filtering rules. As matter of fact, being based on very simple concepts, this implementation is the most suitable for the purposes of this work, and, considering its open source nature, it is the easiest to adapt in any network deployment.

[3]supported devices: `https://openwrt.org/toh/start`

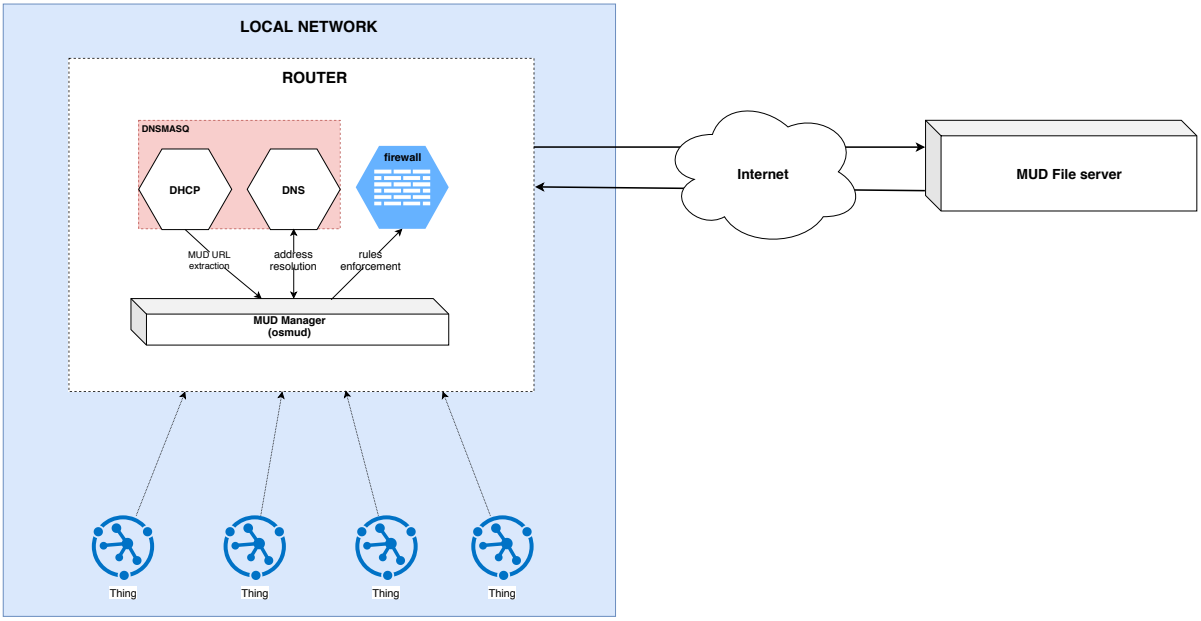


Figure 10: First sub-system: enable MUD by using osMUD

4.1.2 MUD File Server for non-MUD compliant devices

After that choice and analysis of a MUD manager implementation has been finalized, the idea is to provide a MUD enabled local network where all the **IoT devices are MUD compliant**, which means enabling the MUD standard even for non-MUD compliant devices. In order to do that, the sub-system needed is an internal MFS for all the IoT devices not consistent with the MUD specification. The only challenge to be addressed in order to allow the internal MFS to work properly and generate valid MUD files, is to make it **trusted**. The "*trusting*" is reached by using a Certification Authority, which is an entity that issues digital certificates. The latters certify the ownership of a public key that allows others to rely upon signatures. After making the internal MFS trusted, the interaction with the MUD manager can start. The Fig. 11 illustrates how the internal MFS performs the queries and its communication with the MUD manager. In particular:

- when the osMUD manager requests a **MUD file** (*e.g. https://mfs/mudFile.json*), the internal MFS generates a query to a mongo database in order to obtain it;
- when the osMUD manager requests for the **MUD file signature** (*e.g. https://mfs/*

mudFile.p7s), the internal MFS, after the file has been obtained by making another query towards the database, signs the file and returns it.

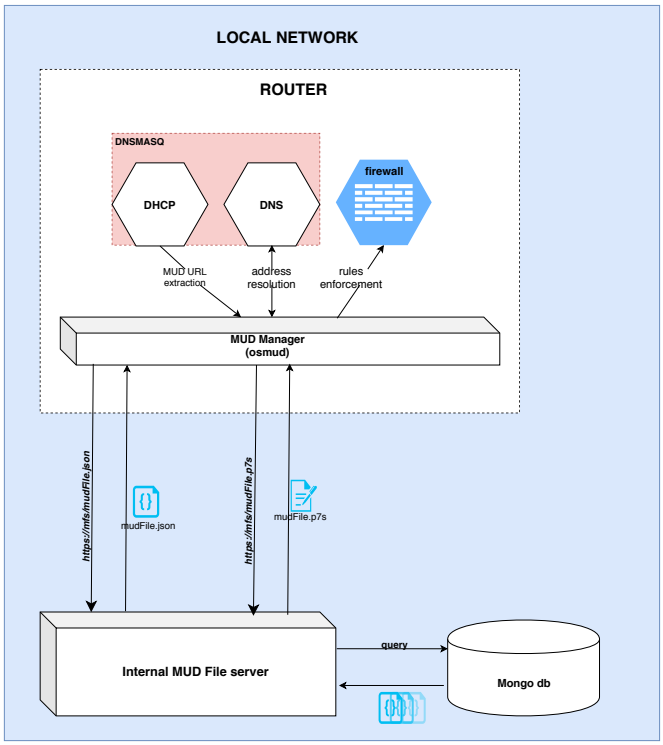


Figure 11: Sub-system: internal MUD file server

It is noticeable that this model is not thought to be efficient. In fact, for each request (MUD file or signature) an additional query is perfomed. Furthermore, the file is signed each time that is requested, which means to have a vast usage of resources in this time interval.

After the internal MFS realization, the IoT devices non-MUD compliant can look for a MUD file, by configuring the DHCP request (DHCP MUD option enabled and MUD-URL equal to the internal MFS address). Note that the MUD files must be defined by the network administrator, who manually inserts them in the database. The last observation, changed the initial aims of the internal MFS design. In fact, thanks to the interaction with a network administrator and the fact that osMUD is by definintion open source, the internal MFS can be exploited to create an access point through which

the network administrator is able to insert new MUD files even for the MUD compliant devices. From this investigation emerges a new architecture called User Policy Server (UPS), which adds new purposes to those already defined for the internal MFS.

4.1.3 User Policy Server: an access point for the network administrator

The osMUD manager is by definition open source, which means giving more flexibility when new functions are needed. Thus, the introduction of a new conception of internal MFS can be adopted easily.

The User Policy Server (UPS) arises from the needing of a network administrator/end-user to enforce new rules that can be different from those defined by the manufacturer. In fact, the manufacturer does not have an overview of internal network behaviours, i.e. the rules provided out of the box are not sufficient. Furthermore, could be a tough process specify a complete communication pattern for more general purporse devices (e.g. voice assistants). Thus, the UPS, in addition to the opportunity of making all the devices MUD compliant, allows to define categories of rules which are hard to be delineated by the manufacturer and more suitable for the network in which MUD is deployed, by preserving the YANG MUD file structure. It should be noted that the UPS introduction causes important implementation challenges to be addressed. First of all, the osMUD manager requires to be changed in order to request new MUD files (UPS files) for each MUD compliant device. Secondly, the UPS must identify network administrators and keep a separate session for each of them. Being implementation related, the challenges solution is discussed in 5.1.2, whereas in this section is delineated the resulting interaction between UPS and osMUD manager.

In order to obtain the behaviour described and identify correctly the IoT devices for which the rules are defined, the MUD file naming must contain the device ***mac-address***, while the UPS location is known a priori by the osMUD manager. In Fig. 12 is shown the workflow of the new concept of internal MFS, where the MUD manager, after the MUD file has been obtained (classical MUD specification plan Fig. 2), requests for a

further MUD file (which can be called *UPS file*) for the device with that *mac-address* (**7**). The *UPS file* contains the rules defined by the network administrator, and, as in the specification [6], after their validation (**10**) are enforced in the firewall (**11**). In order to guarantee a higher security level, the MUD files insertion procedure requires the user authentication.

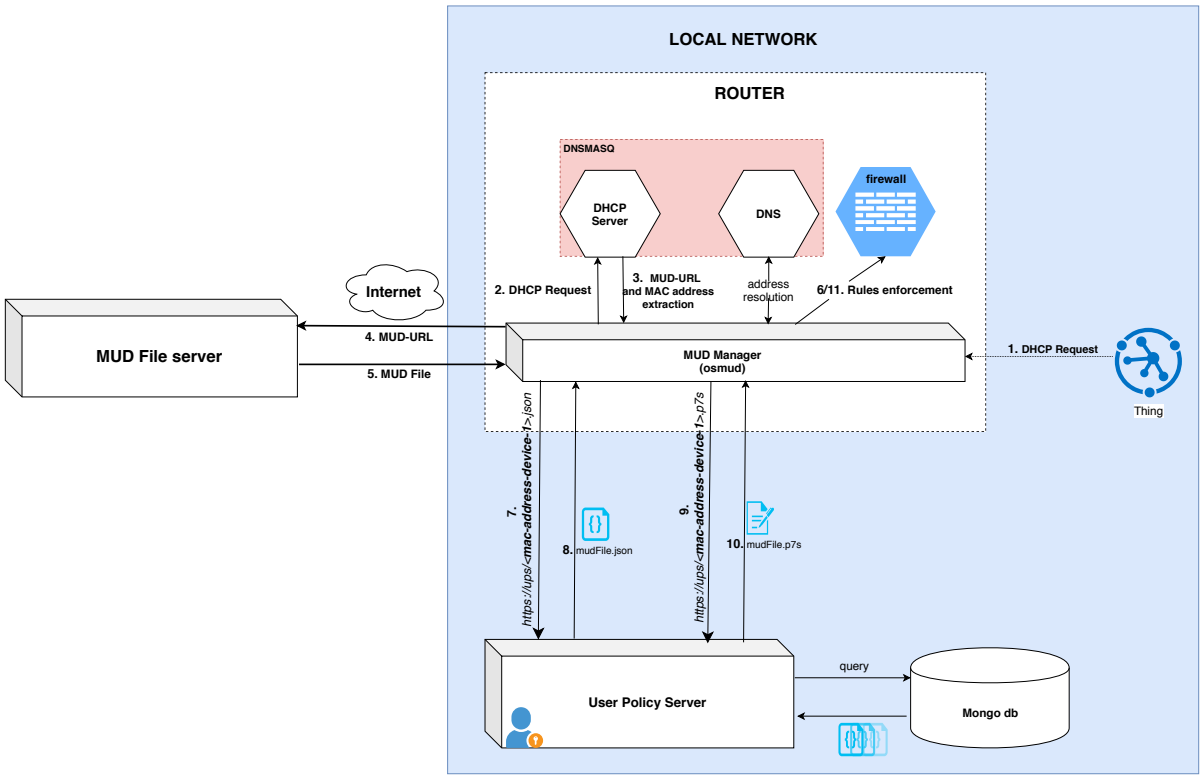


Figure 12: User Policy Server workflow

The results of this procedure could be either the **union** or the **intersection** of the manufacturer rules (**MR**) and administrator rules (**AR**):

$$NewRules = MR \cup AR$$
$$NewRules = MR \cap AR$$

The former is appropriate to define either internal communication pattern or more relaxed rules. For example, in personal assistant's case it is hard for the manufacturer to define a communication pattern that allows all the communication that it needs. Thus,

the administrator can easily build new communication patterns for the necessary services. The latter, considering that the MUD file contains whitelist rules, is helpful to restrict the communication pattern defined by the manufacturer. In fact, by default all the communication are rejected and only the whitelist rules included in both the MUD files requested are enforced in the firewall.
Thus, the UPS concept pave the way to three possible scenarios:

1) **Automatic UPS File generation**: Use a tool that automatically generates the YANG compliant JSON files by monitoring the internal traffic [7, 38] (helpful to generate the MUD/UPS file for non-MUD compliant IoT devices).

2) **New MUD File structure for the UPS Files**: Extension of current file structure to obtain more restrictive rules such as packet rate, maximum packets, time restrictions, maximum ingress and egress points.

3) **Publish/subscribe architecture**: it is possible to extend the UPS infrastructure with a publish/subscribe model, where server becomes the publisher and the router becomes subscriber. The server can publish, by understanding the traffic behaviour, new rules that must be inserted in the IP tables of the router. The architecture produced, needs the authentication between the router and server, otherwise an attacker could impersonate the server and insert less restrictive rules.

In conclusion, all MUD whitelist rules are inserted by default, so adding new whitelist rules using UPS is easy. However, for removing/reversing MUD rules using UPS thoughtful consideration are needed.

4.2 Federated Learning: design of a distributed architecture

As already outlined in 2.3.4, MUD is not intended to address some security scenarios. Consequently, the MUD compliant network, designed in the previous section, needs of an additional system that detects possible anomalies in network traffic. In order to achieve this behaviour, the initial idea was to design an anomlay detection system based on a classical machine learning algorithm, able to identify discrepancies in the traffic generated by the IoT devices. Nevertheless, the huge amount of traffic to be analysed, can reduce the performance of the classical *user network*. Therefore, instead of performing the computation of both training and inference phase on a single device, it could be better to distribute it across the devices within the network. Hence, if in standard machine learning algorithm *the data is brought to the code* in this case *the code is brought to the data*, which means addressing fundamental problems of privacy, ownership and locality of data. This approach is called Federated Learning.
The Federated Learning differs from other distributed learning algorithm, because of the assumptions made. In fact, a common underlying assumption of different distributed learning algorithms is that the local datasets are identically distributed and roughly have the same size. None of these hypothesis are made for Federated Learning. Here the datasets are typically heterogeneus and their size may span several order magnitude. Furthermore, it does not have a full control of the computational resources and does not get back data from the participating clients (only the model updates), compared with the distributed learning where typically the data is distributed uniformly among the nodes by a central server.

The introduction of the Federated Learning concept in a home network brings out new challenges to deal with. In design terms, it is necessary to define a pattern to communicate the device intentions (e.g. ready to train or do inference) and another to distribute the model across the devices. Additionally, from an implementation perspective, the

devices are required to be able to receive and learn a model, which implies thoughtful considerations especially for establishing the number of interactions needed. It should be noted that the devices even necessitate of a logic in order to better manage their intentions on the basis of their available resources. However, the logic challenges are beyond the aim of this work. Thus, considering tha nature of this chapter, in the proceeding sections is explained how this work approaches the design challenges. Particularly, this section delineates the design cores of a distributed architecture based on Federated Learning, which enables the learning capability on the IoT devices. Furthermore, the section emphasises the approaches used to deal with the design challenges that emerge from the Federate Learning concept.

4.2.1 Federated Learning design

The design of an architecture Federated Learning enabled represents the core of this work, because of the following reasons:

- it enables the learning on the IoT devices, by solving data privacy concerns;
- it allows to reach a high level of accuracy, thanks to the data and devices heterogeneity;
- it creates an environment able to host an anomaly detection system, which can further reduce the attack surface in a MUD deployment;

The Federated Learning concept relies on the communication of devices eligibility, which is advertised by using a network protocol. In [17], the designers provide a basic protocol, described in 2.4.1, in which the participants are *Adroid phones* that represent a different category from those devices on which this work is focused on. The latters include devices with limited **resources** and **functions**. Thus, the first architecture concern that has to be addressed to make the Federated Learning available on real devices, regards the choice of a protocol to be adopted to communicate the devices eligibility. First of all, it is needed a central entity that coordinates all device learning activities and receives the events of devices availability; in this work, this entity is called **Coordinator**.

The Coordinator must be capable to recognise when the devices are **ready to start a training phase**, and when they are **ready to receive a model in order to make inference** on their data. At first glance, this description is analogous to the **pattern observer**, where the Coordinator is the observer of the IoT device states (observable): a change of state causes the execution of some associated **actions** Coordinator side. Nevertheless, this pattern typically requires homogeneity in the communication, which means requiring a sort of adapter able to enable the IoT devices for a heterogeneus communication. Conversely, a pattern designed to be suitable with devices heterogeneity is the **publish/subscribe**, where sender of messages, called **publishers**, do not program the messages to be sent directly to specific receivers, called **subscribers**. Thus, the model provided is ideal to takle the heterogeneity communication concerns. In fact, here is not required a direct connection between the devices involved, but all the communications are mediated by a third part, called **Broker**. One possible solution, which replicates this pattern, is the **Message Queue Telemetry Transport** [39] (MQTT), which targets IoT devices and is designed to make this pattern extremely lightweight.

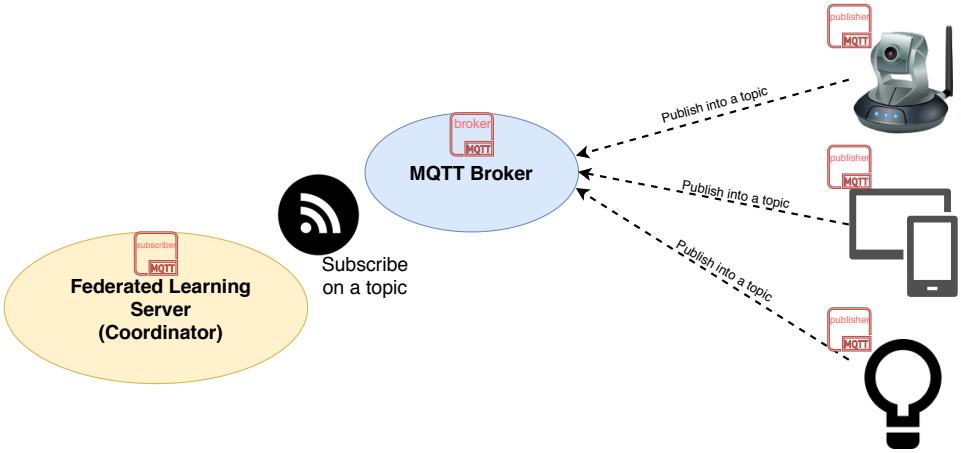


Figure 13: Federated Learning: device states communication through MQTT

The Fig. 13 shows how the integration between the pattern publish/subscribe and the Federated Learning can take place through the MQTT protocol: the Coordinator becomes a subscriber and the devices, which want to communicate their state, become the publishers.

The next step in the Federated Learning procedure is to **send the model** to the elegible

devices. The model sending requires some considerations: except for the protocol used, it strictly depends on either the framework used or the implementation itself. Thus, the next chapter conducts a detailed analysis and evaluation for this step(5).

The description above, provides the right elements to designate the possible states that a device can assume. First of all, two trivial states which are a consequence of the classical machine learning algorithm are defined: on one side the devices need a state that shows when they are ready to **train** a model, on the other side one that shows when they need to do the **inference** on their data. However, the training phase can be misleading, so it requires more attentions. It is possible that, even if a device declared its training intention, can not be available anymore when the training start. In fact, even according with [17], the Coordinator waits until the amount of devices is enough to obtain improvements in the model. Thus, the design of this work provides a further state, which makes the devices able to remove themselves from the Coordinator's devices list. The latter, requires a time in which the Coordinator is in a *wait state* to be built. The wait state in this work is designed as **temporal window**.
The temporal window, represents the time where the Coordinator waits and collects **training requests**. It is started after the first training request has been received. Whereas, the end of the window depends on the architecture design (e.g. training requests frequency), and represents the trigger for the training execution. Nevertheless, the temporal window introduces another not trivial challenge to be addressed. It is possible that some devices declare their training intentions when the training has already begun. On Coordinator side two are the behaviours that can occurs: **discard** these devices or **keep** them until a new temporal window starts. The former, might make sense when either it is not necessary a further model training, because the model has been trained enough, or in order to reduce the Coordinator overloading. The latter represents the **default case** especially in the early stages of training. However, this behaviour requires further consideration.

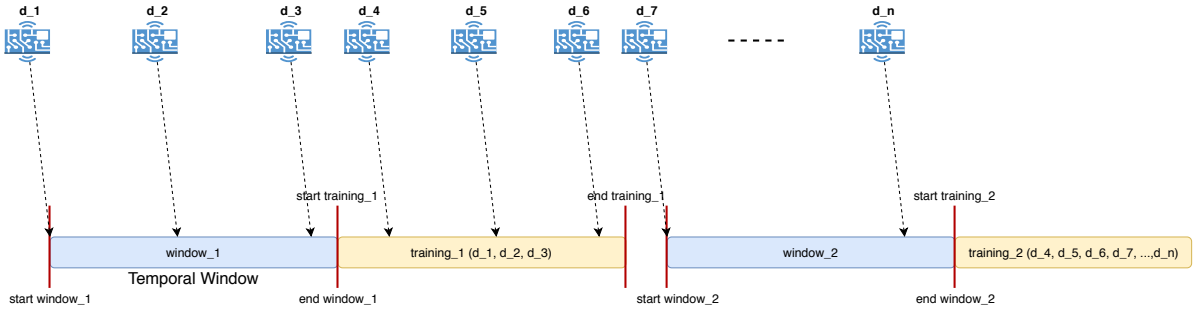


Figure 14: Coordinator's temporal window: devices lost problem

Referring to the Fig. 14, the devices *d_4*, *d_5* and *d_6* communicate their training intention after the window is expired (to identify them the keyword *d_expired* is used). These devices can be collected in the training devices queue, which is the same of the devices *d_1*, *d_2*, *d_3* (*d_window_1*). However, the *d_expired* devices can not be trained in the same training phase of the *d_window_1* devices. This is due to the fact that the training is **partially asynchronous**, which means that the Coordinator waits for the results coming from the *d_window_1* devices after the training has been started in parallel on all *d_window_1* devices . Furthermore, if the training involves more than one *round* (reiteration of the model training), the *d_expired* devices can participate to a lower number of *round* than those defined by the Coordinator. Hence, the *d_expired* training starts after the end of the *d_window_1* devices training and when a **new device triggers the creation of a new temporal window** (new training request received *d_7*). Thus, considering all these problems: *how can the temporal window be useful?*
It allows to define two important thresholds on the number of devices: **lower bound threshold** and **upper bound threshold**. The former, can be used when the improvements of the model produced by the devices involved are not enough to allocate all those resources (bandwidth reduction). The latter, more relevant, allows to define a **selection criteria of the devices**. For example, the devices selected could be the ones with higher computational performance. However, the design of the scenarios generated by these thresholds is not provided by this work. Furthermore, the temporal window allows to solve one of the challenges related to the different device times availability, described in section 2.4.7.

Now that the design pillars have been finalized, the work provides an overview of the a typical Federated Learning scenario, which relies on the sub-systems described above.

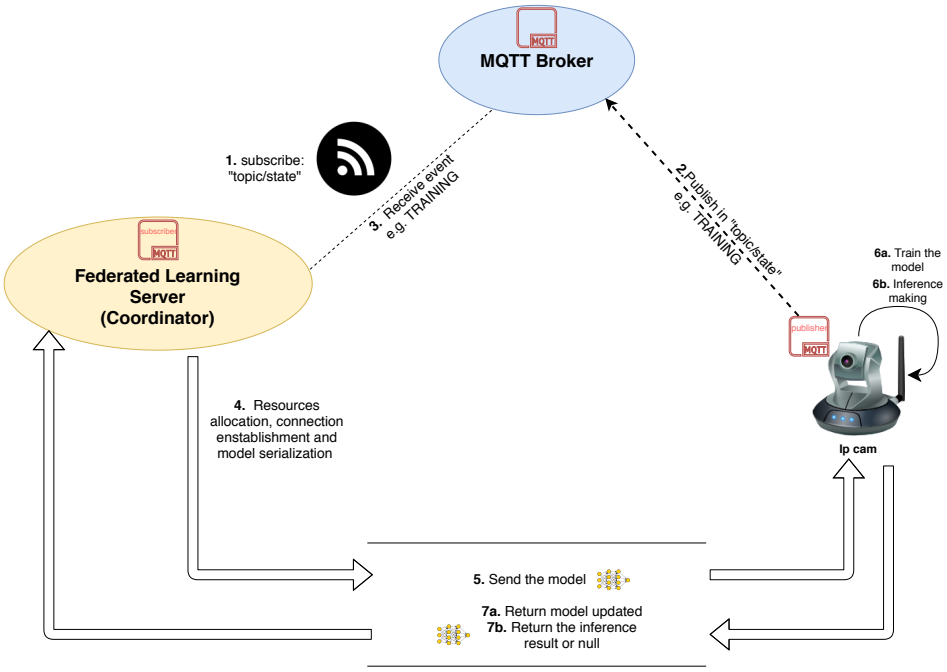


Figure 15: Distributed architecture Federated Learning enabled

The Fig. 15 shows the general steps that IoT devices and Coordinator follow in order to start the Federated Learning procedure. In particular:

- in the first steps, Coordinator and devices establish a connection with the Broker, which then mediates the communication. At the same time, the Coordinator subscribes itself to a particular *topic* (in figure "*topic/state*"), so that it can receive the device (publishers) status updates;
- the nexts steps involves the model sending. It is worth noting, that before that this phase takes place, the Coordinator allocates and prepares all the paramenters useful for the model communication (resource allocation, model serialization etc.);
- lastly, the participant devices start training or inference depending on their state.

It is noticeable that, in case of training, the figure also suggests the definition of **round**. In fact, the **round** includes: (1) selection of the clients that demonstrate their training

intention; (2) training of the model received and computation of the model updates; (3) sending of the model update; (4) aggregation of the models, Coordinator side, to construct an improved global model. Typically, considering the limited capabilities of the devices, the round is repeated more than once, in order to reduce the overloading and to reach a good level of accuracy. However, the rounds increase the bandwidth used, which implies the necessity of finding the best tradeoff between number of rounds and device performance (e.g. by using some automatic algorithms [36]), in order to reach higher accuracy and reduce communication efficiency problems [32].

In conclusion, this section has provided the logic Coordinator side and a logic for model and state communication. However, the work does not provide a design for the **logic client side**. The reason why no client logic has been provided lies on the strict dependency on the type of device used. In fact, the change of state results from particular devices condition, such as device in charging, battery powered, underload and overload condition (e.g. during the night) etc. Thus, being this information devices dependent, the best solution becomes having a **client logic pattern** either provided by the manufacturer or built by using other machine learning techniques.

4.3 MUD deployment and Federated Learning together

This section performs the last step of the bottom up approach. Now that all the pieces have been finalized, it is presented the strategy adopted by this work to put together the two sub-systems designed in previous sections, and it is shown the procedure used to take advantage of the benefits of both.
The first challenges to be addressed is to place the Federated Coordinator in an entity belonging to the MUD enabled network. By considering the additional trusted entity that turns out from the architecture provided at the end of section 4.1.3 (where the UPS is running), the Coordinator hosting can be easily handled.
After having placed all the components in the network, the design of the final architecture proceeds towards a strong integration between the two sub-systems, in order to reduce **vulnerability scenarios**. Referring to section 2.4.7, the Federated Learning suffers of two poisoning attacks that can cause the model misleading. Typically, these attacks turn out from the possibility of having malicious participants in Federated scenarios. The malicious devices, **without considering compromised IoT devices**, are mainly general purpose machines with high computational resources. Thus, the goals consists in removing from the Federated Learning participants all the non-IoT devices. According to Bremler-Barr et al. [37], by using some machine learning techniques, it is possible to distinguish between IoT and non-IoT devices in a matter of minutes. Nevertheless, introduce a new classifier in this network may not be the best solution to be adopted in terms of network performance. Hence, by exploiting the MUD definition, it is possible to achieve the same goal so that all the non-IoT devices do not participate in the Federated Learning procedure. The osMUD architecture, as already described in section 4.1.1, uses dnsmasq to record the DHCP requests in a file and, in case of MUD compliant devices, the MUD-URL is extracted and written in that file. The latter represents the element of distinction used, which means that only the **MUD compliant** IoT devices can participate in Federated procedures. Although not all IoT devices are MUD compliant, the

architecture provided gives the chance to make them consistent with the MUD standard (4.1.2). Thus, the design idea is to **exploit this file used by osMUD** to understand if a device is IoT or not, and then send the IoT device identifiers (e.g. *ip address*) towards the Federated Coordinator. The latter applies a filter on the events received from the devices, in order to consider only those that have a valid identifier (**device filtering**). It has been shown that the entity, which hosts the Coordinator, is **trusted** (section 4.1.2). Therefore, all the communications between osMUD manager and UPS, and between osMUD manager and Coordinator are trusted and encrypted. The final design of the distributed architecture is shown in Fig. 16.

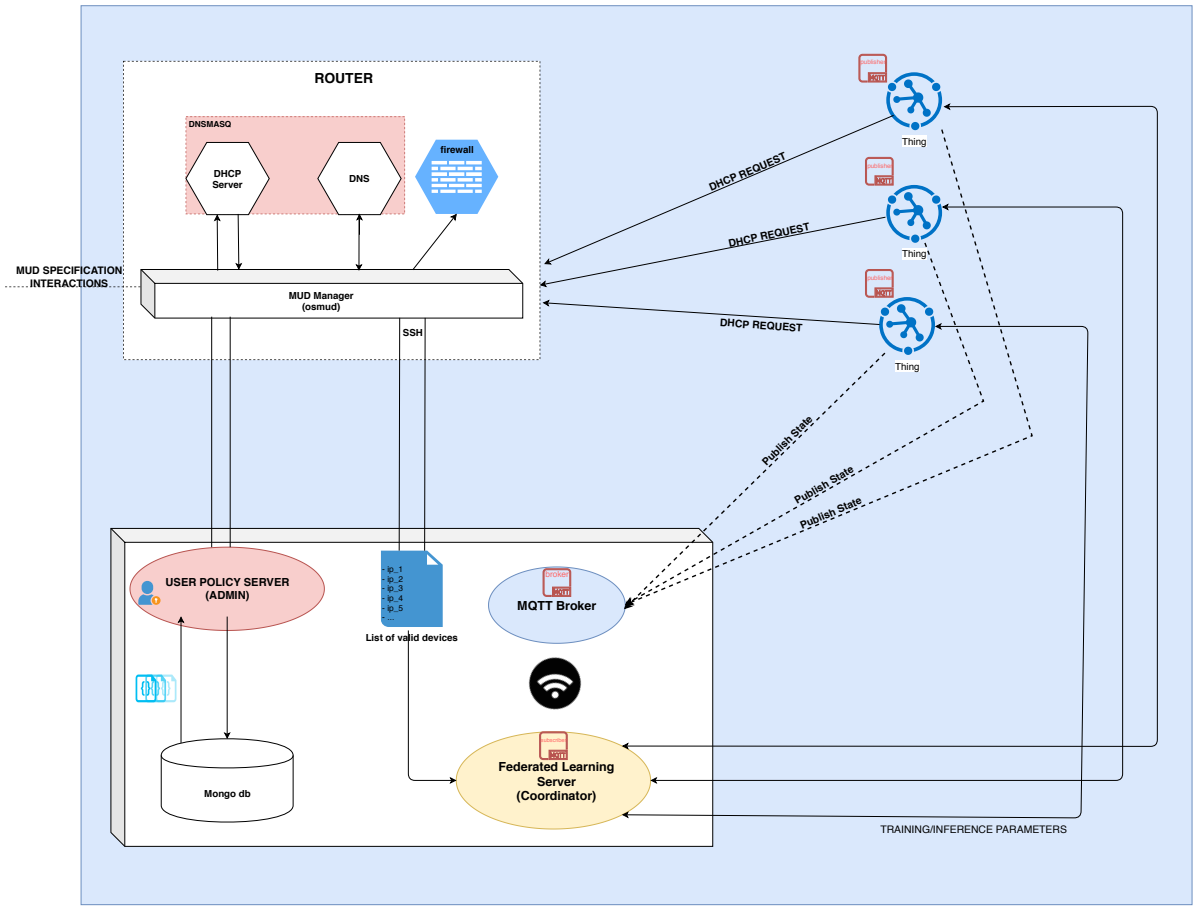


Figure 16: Final distributed architecture design

The final architecture illustrated in Fig. 16 guarantees the reduction of possible *poisoning* attacks. Nervetheless, it is **not always effective**. In fact, according also to the specification [6], an IoT device may be able to lie about what it is. Likewise, the

ip-spoofing attack can be used to steal the device identity. The former to be mitigated, requires an anomaly detection model to be adopted as Federated Learning global model. The latter is smoothed out by using authentication algorithm, such as *WiFi Protected Setup* and *X.509* as MUD-URL emitter.
The next section delineates a more detailed analysis about the unsolved challenges.

4.4 Design gaps

The final architecture obtained and outlined in the previous section, still demonstrate some security and efficiency deficiency.

Firstly, the UPS introduction could influence the osMUD manager performance. In fact, it generates further MUD file requests and requires new file processing in the osMUD manager and new rules injection in the firewall, if the UPS file is defined. However, does not generate great repercussion on the overall network performance, even because not all devices have an additional MUD file (section 5.1.3).
In case of rules union (4.1.3), the firewall could contain meaningless rules because redundant, but they do not cause relevant reduction of firewall performance. In addition, the rules generated by the network administrator may conflict with those defined by the manufacturer. Thus, the use of UPS rules requires thoughtful considerations.

Secondly, the pattern publish/subscribe used as baseline for the Federated Learning architecture lacks of authentication elements. In fact, an attacker can steal the IoT devices identity, which means reduce the device filtering effectiveness. Additionally, in terms of architecture there are no algorithm adopted for device selection and amount of rounds to adopt. The former, provokes the training time increasing, because of possible scarce devices performance. The latter, worsens the bandwidth management, which can be critical in case of huge amount of devices. There are already some solutions focused on the communication-efficiency, which propose adaptivity and model size reduction techniques [32, 36].

The analysis above is completed in section 5.4, where future directions for the architecture provided are proposed.

Chapter 5

Implementation Insights and Performance Results

The chapter provides the implementation description that results from the system design presented in past units. The description starts from the MUD environment, by focusing on problems encountered in the osMUD manager and on describing the UPS implementation. Thus, the mainly features of osMUD, useful to understand the rest of the architecture, are highlighted. After implementing the first sub-system designed in section 4.1.3, the chapter illustrates an essential evaluation to check the feasibility of running such sub-system in a home router or small business environment. Afterwards, the chapter focus moves on producing the sub-system resulting from section 4.2.1. Before building such sub-system, which relies on the Federated Learning, the chapter displays the motivations behind the framework choice, which further describes its main characteristics. Hence, the implementation of the MQTT based architecture, which makes devices within a network Federated Learning compliant, is described. The sub-system created represents a real case scenario where each device is able to do learning on its own data. In order to evaluate the performance of the implemented sub-system, a dataset and a model are required. The former is the Bot-IoT Dataset [3] that allows to create a model able to detect BotNet attacks. The latter is a very simple model used only for **evaluation** purposes. Thereby, all the elements necessary to have a complete evaluation

have been provided. In particular the evaluation involves:

- cumulative bandwidth tests;
- time for each round;
- total training time;
- temperature tests on real devices.

These provide a sufficient background that allows to suggest well-defined rules to maximise the communication-efficiency and the device performance.
In conclusion, the chapter describes a possible implementation of the final architecture described in section 4.3 and proposes new directions that can be taken to improve it.

5.1 MUD network

5.1.1 Open Source MUD Manager: implementations and problems

The osMUD manager implementation is designed to be integrated with the dnsmasq and OpenWRT services. As already described in section 4.1.1, each time that a DHCP request is received, dnsmasq calls a script (`detect_new_device.sh`) that **extrapolates the MUD-URL from that request**. In order to guarantee its execution, dnsmasq must be configured by editing the file `/etc/dnsmasq.conf` (adding the line dhcp-script = detect_new_device.sh).
The script called defines **three** different states that make the MUD manager able to distinguish the type of DHCP request received.

- The `NEW` state implies the execution of a MUD file request towards the Manufacturer File Server only in case of MUD compliant devices (DHCP request with MUD-URL);

```
45 if [ "$1" == "add" ]; then
46   msg="|NEW|‘uci get system.@system[0].hostname‘.‘uci get dhcp.
       @dnsmasq[0].domain‘|DHCP|${DNSMASQ_REQUESTED_OPTIONS}|MUD|${
       DNSMASQ_MUD_URL}|${DNSMASQ_VENDOR_CLASS}|$2|$3|$4|"
47   echo ‘date +%FT%T‘$msg >> /var/log/dhcpmasq.txt
48 fi
49
50 if [ "$1" == "old" ]; then
51   msg="|OLD|‘uci get system.@system[0].hostname‘.‘uci get dhcp.
       @dnsmasq[0].domain‘|DHCP|${DNSMASQ_REQUESTED_OPTIONS}|MUD|${
       DNSMASQ_MUD_URL}|${DNSMASQ_VENDOR_CLASS}|$2|$3|$4|"
52   echo ‘date +%FT%T‘$msg >> /var/log/dhcpmasq.txt
53 fi
54
55 if [ "$1" == "del" ]; then
56   msg="|DEL|‘uci get system.@system[0].hostname‘.‘uci get dhcp.
       @dnsmasq[0].domain‘|DHCP|${DNSMASQ_REQUESTED_OPTIONS}|MUD|${
       DNSMASQ_MUD_URL}|${DNSMASQ_VENDOR_CLASS}|$2|$3|$4|"
57   echo ‘date +%FT%T‘$msg >> /var/log/dhcpmasq.txt
58 fi
```

Listing 1: detect_new_device.sh: DHCP states

- The `OLD` state allows to avoid that further MUD file requests for the same MUD compliant device are made.
- The `DEL` state induces the MUD manager to delete the rules for a device that send the **DHCP Release** packet, which indicates that a device is about to leave the network.

It is worth noting that the script records a string (line 46, 51 and 56 in Listing 1) that reveals different informations about the device that made the request (manufacturer, mac address, MUD-URL etc.) in the file `/var/log/dhcpmasq.txt`. This file is monitored each 5 seconds (default value) by the osMUD manager, in order to generate the MUD file request in case of MUD enabled devices. If a valid MUD compliant string is detected, the osMUD manager call the method `executeOpenMudDhcpAction`, which takes as a parameter the structure `DhcpEvent`. The latter contains all the parameters useful in the MUD specification workflow (2.3), such as `mudFileURL`, `mudSigURL`, `mudFileStorageLocation` and `mudSigFileStorageLocation`. Thus, when the state `NEW` is detected, the osMUD

```
64 int buildPortRange(char *portBuf, int portBufSize, AceEntry *ace)
65 {
66         int retval = 0; /* Return > 0 if there is an error with port
               assignments */
67         if(ace->lowerPort == NULL || ace->upperPort == NULL){
68                 // Problem solution
69                 logOmsGeneralMessage(OMS_INFO,
                       OMS_SUBSYS_DEVICE_INTERFACE,"something null");
70                 snprintf(portBuf, portBufSize, "any");
71                 portBuf[portBufSize-1] = '\0';
72                 logOmsGeneralMessage(OMS_INFO,
                       OMS_SUBSYS_DEVICE_INTERFACE,portBuf);
73         }else{
74                 snprintf(portBuf, portBufSize, "%s:%s", ace->lowerPort
                       , ace->upperPort);
75                 portBuf[portBufSize-1] = '\0';
76                 logOmsGeneralMessage(OMS_INFO,
                       OMS_SUBSYS_DEVICE_INTERFACE,portBuf);
77         }
78
79         return retval;
80 }
```

Listing 2: MUD manager: port range problem

manger locates the MUD file by using the `mudFileURL` field of `DhcpEvent`. After the MUD file has been obtained, the osMUD manager creates a further request to retrieve the file signature (`mudSigURL`) so that the signature validation can take place ([6, Section 13.2]). Lastly, the file is processed in order to produce the rules to enforce in a router firewall.

It should be clear that the version of dnsmasq adopted is not the classical implementation. In fact, the dnsmasq used is a custom version provided by osMUD designers [40], able to extrapolate the MUD-URL (`DNSMASQ_MUD_URL`) from DHCP requests.

By trying to execute the osMUD code some errors occurred. First of all, the signature validation always failed due to some router environment problems, which bring this work to provide a script that runs the osMUD manager in admin environment. Thereby, the osMUD manager can access to MFS public key, which makes it able to validate the MUD

```
1     option mudurl code 161 = text;
2     send mudurl "https://ups/<mac_address_device>.json";
```

Listing 3: Example DHCPv4 configuration for linux-based system unsing dhclient

files. Secondly, the method `buildPortRange` does not handle the case of missing port in the MUD file, which could happen when the access to *any* port is granted by the manufacturer. This lack of port management lead the osMUD manager to fail at runtime. Thus, the implementation was modified in order to avoid runtime errors (Listing 2).

The brief description above is useful to have more clear the changes made by this work. These involve mainly the User Policy Server introduction in MUD deployments, which description is detailed in the next section.

5.1.2 User Policy Server implementation

The User Policy Server is a web server that relies on the *express* framework and works together with a mongoDB. The first UPS goal is to give the opportunity to all the non-MUD compliant devices to have their own MUD file (section 4.1.2). Thus, it respects the design priciples of a MUD file server, i.e. a trusted web server that hosts MUD file. In order to make the devices consistent with the MUD specification some changes in the DHCP requests are needed. In particular, it is necessary to change the DHCP client configuration file so that the MUD-URL is retrieved by the osMUD manager (e.g. Listing 3). In order to insert MUD files in the database, the UPS provides a javascript program able to execute insertion queries. Additionally, the UPS gives a Graphical User Interface with which it is possible to have a view of all MUD files that it is hosting.
By analysing the structure becomes straightforward to implement a new feature that provides the ability for an administrator to upload new MUD files. Thereby, an administrator can also define new MUD file for MUD compliant devices, which results in producing new rules different from those defined by the manufacturer. Thus, the UPS provided gives the possibility to **make MUD compliant non-MUD compliant devices** and **insert administrator defined rules**. The latter requires the implemen-

```
328                 case 'a':  adminMfs = copystring(optarg);
329                                                 break;
```

Listing 4: MUD manager: option for UPS file requests

```
1  if(adminMfs){
2      // Create the new addresses
3      strcpy(newFileURL, adminMfs);
4      // 1) File URL
5      strcat(newFileURL, dhcpEvent->macAddress);
6      strcat(newFileURL,".json");
7
8      dhcpEvent->mudFileURL = newFileURL;
9      // Change the name of the mudfile
10     dhcpEvent->mudFileStorageLocation = createStorageLocation(
           dhcpEvent->mudFileURL);
11
12     /*
13     * Signature URL setup
14     * Download UPS File
15     * Download Signature
16     * Validation signature
17     * Insertion administrator rules*/
18 }
```

Listing 5: MUD manager: further request to the UPS by using the MAC address

tation of new elements. First of all, the files inserted by the administrator represents further MUD files (UPS files) that must be retrieved by osMUD. As described in the system design (4.1.3), the osMUD manager must know in advance the UPS location and, considering that the devices do not provide the MUD-URL to locate the file, the file is located by using the `mac_address`. Thanks to the `DhcpEvent` structure, briefly introduced in the previous section, the `mac_address` is easy to obtain. In fact, the changes needed osMUD side are minimal:

- to know the UPS location another options can be added, so that it can be specified as an additional parameter at start time (Listing 4);
- to make the *UPS file* request osMUD uses the `mac_address` and the address previously obtained (Listing 5) (the workflow is illustrated in Fig. 12).

```
1    var re = new RegExp("^([0-9A-Fa-f]{2}[:-]){5}([0-9A-Fa-f]{2})$")
2    /* The file inserted must be a json file and must have a
        particular name format:
3    MACADDRESS.json */
4    if(fileName.endsWith('.json') && re.test(fileName.substring(0,
        fileName.length-5))){
5    }
```

Listing 6: UPS: check file name

Secondly, UPS side **administrator authentication** and **UPS file name validation** are required. The former is implemented as a **simple authentication form**, where the user data is stored in the mongo database. Furthermore, the form allows to have a separate environment for each end-user, in other words they can insert and remove only their files. The latter uses regular expressions in order to verify the name validity (Listing 6).

In conclusion, the UPS implements the union of rules, which means to take all the redudancy problems that come with it. Thus, using administrator rules that can either remove or reverse the manufacturer rules requires thoughtful considerations. However, as already described in section 4.1.3, the UPS introduction pave the way for different scenarios that can improve the security and reliability of a MUD deployment.

5.1.3 Evaluations

The aim of the system evaluations is to check the feasibility of running such a system in home routers or small business environments. The experiments are conducted in a deployment characterised by an OpenWRT compliant Netgear router (model WNDR 3700v2) and an entity that hosts the UPS (MacBook Pro, Intel Core i5 and 8 GB of RAM). In order to check the system feasibility, the tests are performed by considering the total time spent (**latency calculation**) by the osMUD Manager in:

1) retrieving the MUD file;

2) verifying and storing the file locally at the router;

3) processing the file and installing the rules.

These actions depict the **setting time of the rules specified in a MUD file**.
The tests consider the router in its **booting stage** that represents the highest overloading phase, because of the aggregation of DHCP requests received from all the connected devices (IoT and non-IoT). In case of MUD compliant IoT devices, the osMUD manager processes the MUD-URLs one by one in order to locate the MUD files. Hence, the file retrieving and processing is **sequential**. Additionally, in order to study the worst case scenario, the tests examine the situation where all the IoT devices are turned on before the router boot process, so that all the MUD files need to be retrieved at the same time. For testing purpose, the MUD file server provided by osMUD designers [4] is used.
The osMUD manager, in case of devices with administrator rules defined (UPS file), has to repeat the same operations listed above after installing the MUD file rules. It is worth noting that these UPS files are not defined for all the MUD compliant device, but only for a random number of them.

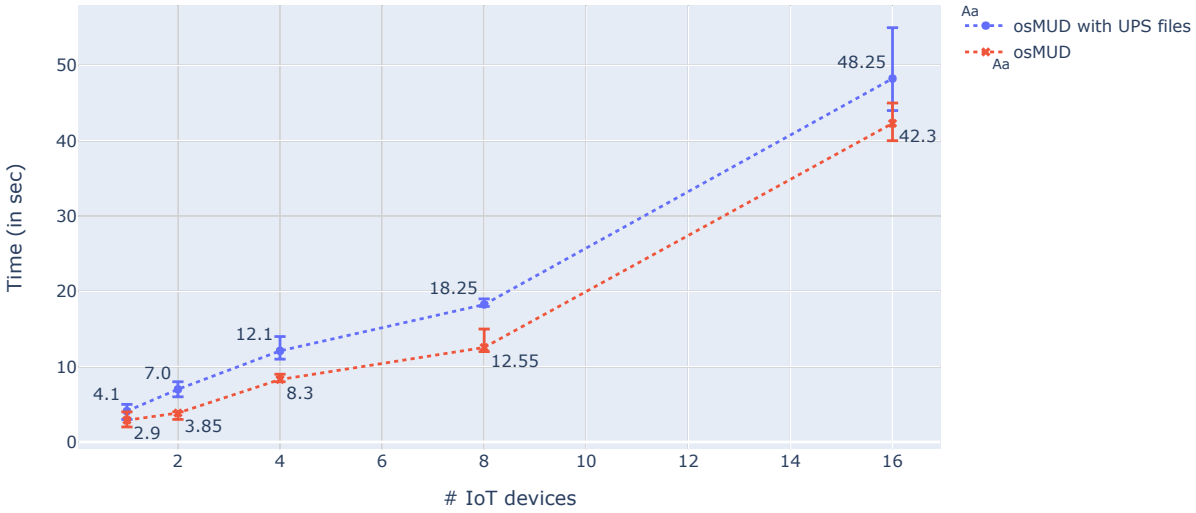


Figure 17: osMUD manager rules setting time

The Fig. 17 illustrates the performance of the osMUD manager in classical (without UPS files) and administrator (with UPS files) environment. Both the tests have been repeated 20 times with a file size range between 2-6K. In the test settings the UPS file requests and MUD file requests are setted as: (1,1), (2,2), (2,4), (3,8), (6,16). The

[4] https://mudfiles.nist.getyikes.com/

numbers in the brackets (a,b) indicate: a = number of MUD compliant devices (random), which request for UPS file; b = number of MUD compliant devices.

First of all, by considering only the one device scenario, the performance registered by the osMUD manager in presence of UPS file are a bit worse than those achieved by the classical MUD environment, as a result of an additional request done to the UPS. Nevertheless, the results obtained are better than expected; the latency average in osMUD with the UPS file is 4.1 seconds compared with 2.9 seconds of osMUD in classical cases, which means in average only 1 second worse. One of the reason that bring out this result comes from the evidence that the requests do not leave the local network, which implies a reduction in the file download time. These results are promising and motivate the usage of a User Policy Server in a MUD deployment.

Secondly, in the classical environment the time spent on doing these operations is not linear, as a result of not stable connectivity. However, the time is not exponential, so the number of devices do not excessive weight on the performance. This is not true for the administrator environment, in which the time depends on how many devices requires an additional requests. For example, the couple (6,16) reached a spike of 55 seconds, compared with 45 seconds of the classical environment.

In conclusion, in both cases with 16 devices, the time exceeds more than twice the time spent by 8 devices. The evaluation lead to think that when the number of devices is over a certain threshold, the router starts to slow its performance. For example, in the case illustrated in Fig. 17 the threshold value can be defined as 8 devices, as result of the huge growth reached after this number. It should be remembered, however, that the circumstance described represents the worst condition that could occur in case of router failure or unexpected reboot of it. In a **steady** condition in which at most 2 devices at a time are added, the performance registered are acceptable in both the environment even if both the devices requests an additional UPS file.

5.2 Federated Learning implementation principles

The architecture design (section 4.2.1) recognized the *model sending* concern as strictly related to either the implementation or the framework used. Thus, the first part of this section describes the implementation choices made in order to produce a standard approach for the model communication.
At first glance, the best decision might seem the implementation from scratch, which could result in the most optimal solution for the deployment used. Nervertheless, this method can lead to a huge correlation with the surrounding environment in which it is applied. Furthermore, the requirements to be satisfied necessitate thoughtful considerations on the characteristics to adopt, such as class of model serialization, machine learning framework, protocol for the model transfer etc. Therefore, in order to achieve a greater level of scalability and flexibility this problem is managed by using one of the framework already described in sections 2.4.3 and 2.4.4.
The framework choice problem requires some discussions on the environment in which the Federated architecture is applied. From section 4.2.1 emerges that all devices, which want to be participant of the protocol, are distributed. The devices distribution implies the needing of a **remote** model communication pattern. Thus, as result of the frameworks anlysis (2.4.3 and 2.4.4), the framework chosen as implementation basis is **PySyft**. It provides the *Network worker* structure that enables the remote communication of the model and lies on the Web Socket protocol. The latter is a network protocol designed to have a lower overhead and to facilitate real-time data transfer from and to the server.

Now that the building blocks has been finalized, the next steps involve the real architecture implementation. In particular, the next two subsections assembles the architecture showed in Fig.13 by providing a working local environment based on the Virtual Worker concept. The latter represents actual workers and allows to simulate a real communication without using the network, which makes it appropriate to simplify the first debugging operations. Subsequently, the resulting skeleton is employed to built the remote architecture designed in section 4.2.1. Both the solutions are evaluated by using the same

dataset and model. In particular, the dataset used, provided by Koronitiotis et al. [3], gives the baseline for allowing botnet identification across IoT based network. Whereas, the model is a simple feed forward neural network composed by 2 hidden layers, useful only for testing purposes. Furthermore, the evaluation in the local environment includes a comparison of model training without encyrption and with encryption, which is realised by using the Secure Multiparty Computation (SMPC 2.4.6) algorithm. The evaluation of the encrypted training is useful to underline the pros and cons of its adoption in a Federated Learning architecture.

5.2.1 Federated Learning implementation on the same machine

5.2.1.1 Local implementation: Plain environment

As previously described, in order to simplify the debugging operations and to obtain a skeleton for a remote architecture, the first part of the implementation involves Virtual Workers, which all live in the same machine and do not communicate across the network. The Virtual Worker abstraction provided by PySyft represents a virtual separation of the hosting machine, so that the limit of number of simulated edge devices depends on the machine capabilities.
The machine (subsequently refer to as Coordinator) must be a subscriber of a particular topic on which all the device states are published (Fig. 13). The state event follows the syntax `(device_name, state)`, where the `device_name` is ordinarily represented by the device's *ip address* that generates the event (not decisive in local environment), and the `state`, which commonly indicates the devices intention, selects the behaviour assumed by the Virtual Worker. As suggested in the architecture design (4.2.1), the states needed are three:

- `TRAINING`: the device wants to train a model;
- `INFERENCE`: the device needs to classify its own data;
- `NOT_READY`: the device is not available anymore.

```
1     def ip_address(self):
2         # self.__message represents the input event
3
4         # 1) remove the brackets
5         to_parse = re.sub(r'[\(\)]', '', self.__message)
6
7         # 2) obtain the ip address
8         ip_address = re.split(r',', to_parse)[0]
9
10        # 3) verify ip address
11        pattern = r'\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}'
12        if(re.match(pattern, ip_address)):
13            if not self.filtering:
14                return ip_address
15            else:
16                if valid_iot_ip_address(ip_address):
17                    return ip_address
18                else:
19                    return -1
20        else:
21            return -1
```

Listing 7: Coordinator: EventParser class ip_address method

Now that the states have been defined, the next step is to generate the event containing them. Typically, these events must be generated by the edge devices that want to become participants of the Federated Learning protocol. However, being a local simulation, the event for the Virtual Workers are simulated as well. For example, supposing that the Coordinator is subscribed to the topic "*topic/state*", the event that triggers the training start procedure is generated by using the command:

```
mosquitto_pub -t topic/state -m "(192.168.1.4, TRAINING)"
```

The Coordinator checks the event validity each time that an event is received. The validation involves a new component that parses the event and returns the part desired (name or state). For example, the method outlined in Listing 7, belonging to this component, returns the name of the device that generated the event. It is noticeable that the method checks the validity of an **ip address**, as result of building a skeleton for the remote case. Furthermore, it invokes the method `valid_iot_ip_address` which is

```
1 if state == "TRAINING":
2     settings.event_served += 1
3     settings.training_devices[worker.id] = worker # registration
```

Listing 8: Coordinator: training event collection

beyond the aim of this section (see section 5.3).

After that parameters useful for the communication have been obtained, the Virtual Workers are generated and their behaviours are defined. In the local case only the **training behaviour** has been defined and it implies the definition of a temporal window. Thus, if the event represents the first received, the temporal window is created. As already said, the temporal window defines the time where the Coordinator wait for other training state events. All devices that declare their training intentions before the temporal window exipiration are collected by the Coordinator in a dictorionary, where the keys are worker ids and the elements are workers (Listing 8). In order to trigger the training begin after the temporal window expiration, the implementation uses the **Timer** object. The latter creates a timer that runs the function specified as parameter, after a certain interval of seconds (parameter of the object) have passed. Thus, the Timer object represents a **singular temporal widow**, which brings the implementation to avoid its creation each time. As a result of this assertation, the variable `event_served` (Listing 8, line 2), which becomes zero at the end of the training of all the devices collected, has been introduced.

The function used as parameter of the Timer object (Listing 9) conducts the training initialization and starts the actual training. The first two function parameters represent the **lower bound** and **upper bound** that discern from the system design (4.2.1). The next two parameters contain training informations, whereas the last is used to remove the workers from the list `_known_workers` of the *local worker* (or owner) when the training ends. It is worth noting that, being the function executed by a separate thread (different from the one that collects the events), even if the training is started other events can be received. This behaviour can cause some problems in the training phase. In fact, if the rounds are used, the workers collected after the temporal window expiration (Fig.14) can

```
1  if settings.event_served == 1:
2      function_to_start = lambda :
3                          starting_training_local(
4                                                  lower_bound=self.
                                                      training_lower_bound,
5                                                  upper_bound=self.
                                                      training_upper_bound,
6                                                  path=self.path,
7                                                  args=self.args,
8                                                  server=self.server
9                                                  )
10
11     t = Timer(self.window, function_to_start)
12     t.start()
```

Listing 9: Coordinator: function parameter of the Timer object

```
1  to_train = settings.training_devices.copy()
```

Listing 10: Coordinator: devices to train copy

be trained for a less number of rounds, because the training has already started and the dictionary where the new workers are collected is the same used by the training function. In order to avoid that these workers (devices) are included in the running training, the actual workers to be trained are copied in a list before the training begin (Listing 10). Note that, the list of actual workers is different from the dictionary defined in Listing 8. The local case training procedure requires an additional step compared with the remote case, which is executed before the model sending. In fact, the Virtual Workers need some data on which the training is executed, which results in introducing an additional step where data are distributed by the Coordinator to devices. In order to realise this operation, PySyft provides the `FederatedDataLoader` which iterates on a `FederatedDataset` class (used as a PyTorch Dataset class 2.4.4) in a federated way. The `FederatedDataLoader` allows to split the dataset in random samples, which are then spread across all the Virtual Workers (tuple of list) passed as parameters (Listing 11).

```
1 federated_train_loader = sy.FederatedDataLoader( # <-- this is now a
      FederatedDataLoader
2             NetworkTrafficDataset(args.test_path, ToTensor())
3             .federate(tuple(to_train.values())), # <-- dataset
                 distribution across all the workers, it's now a
                 FederatedDataset
4             batch_size=args.batch_size, shuffle=True)
```

Listing 11: Coordinator: spreading the data among the Virtual workers

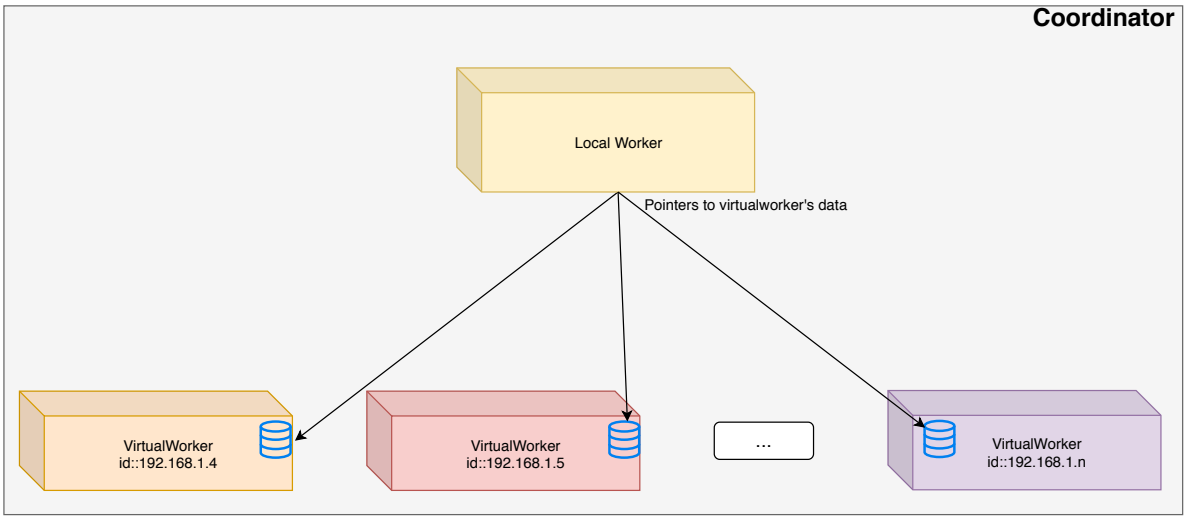


Figure 18: Coordinator: virtual workers and data pointers

The Fig. 18 illustrates the Coordinator machine structure after creating the Virtual Workers and spreading the data across them. Note that the Local Worker, also called owner, has one pointer for each virtualworker's data. These pointers are called **Pointer Tensors** and specify who owns the data and remote storage location (see section 2.4.4 for structure explanation). Thus, the Coordinator is able to coordinate the data operations, without having a direct access on it.

Now that all the components have been placed, the training can begin. The Listing 12 shows where the function invoked by the Timer object calls the method that executes the real training (`train_local`), which is a classical PyTorch training with two additional steps of **data location verification** and **model sending** (Listing 13). Furthermore, the Listing 12 shows that the training is executed sequentially, one virtual worker at a time. The sequential training can result in **bottle neck** states, because the devices

```
1  for worker in list(to_train.keys()):
2      # Calling training function
3      temp_model, loss = cf.train_local(worker=worker,
4      model=model, opt=optimizer, epochs=1, federated_train_loader=
           federated_train_loader, args=args)
5
6      # Collect model updates
7      models[worker] = temp_model
8
9  # Apply the federated averaging algorithm
10 model = utils.federated_avg(models)
```

Listing 12: Coordinator: Virtual workers training

```
1  for batch_idx, (data, target) in enumerate(federated_train_loader):
2      # verify that the data location is the same of the worker passed
           as parameter
3      if data.location.id == worker:
4          model.send(data.location) # sending model to the corret data
               location
```

Listing 13: Coordinator: verify data location

involved typically have different performance. However, this problem is not relevant to be managed in local case.

At the end of each virtual worker training the model's updates are gathered in a dictionary, which then is used to compute the global model by using **Federated Averaging** (2.4.5) algorithm(Listing 12, line 10).

In conclusion, even if there is no real separation between Coordinator code and "devices" code, the results obtained are of relevant importance for future remote architecture that involves real devices. In fact, this implementation provides:

1) **MQTT subscriber** able to receive and process state events from *unknown* devices;

2) **parser** able to process and verify the event syntax correctness;

3) **algorithm** able to collect workers and to manage the temporal window problem;

4) draft of **training** and **evaluation** method;

5) **simulation of Federated Learning** with data privacy preservation.

These parts obtained can be used as a skeleton for the implementation of a remote architecture with real devices.

5.2.1.2 Local implementation: Encrypted environment

This section describes the implementation of the Secure Multi-Party Computation (2.4.6) in a Federated Learning scenario. The implementation refers to the tutorial provided by the PySyft framework designers ([41]), where the SMPC algorithm is applied on model and data sharing. In particular, the Coordinator secret shares his model and send each share to a worker, and the workers secret share their data and exchange it between them. Once the model is trained, all the shares can be sent back to the server in order to decrypt it. In according with [41], the SMPC algorithm implementation relies on two crypto protocols SecureNN [42] and SPDZ (briefly described in 2.4.6). Thus, the tutorial is used as starting point for the local context of this work, which means introducing security principles in the architecture implemented in the previous section.

Before to start with the real implementation some backgrounds about the SMPC algorithm implementation design, which adopts the SPDZ protocol, is needed. In order to do that the framework designers provide another tutorial([43]) that simplify the math concepts that characterise the protocol used.
The SMPC protocol, as described in 2.4.6, removes the concept of private/public keys, by adding the **additive sharing** approach where the values are splitted in multiple **shares** each of which operates like a private key. Thus, the shares are distributed among multiple owners, so that all these owners must agree to allow the decryption. The assumption made by this protocol description is that the computation is performed by n workers over an **integer quotient ring** $\mathbb{Z}_Q = \mathbb{Z}/Q\mathbb{Z}$, in other words the integers between 0 and $Q-1$, where Q is a prime number "big enough" so that the space can contain all the numebers to encrypt. In more detail, each worker W has a uniform share $\alpha_i \in \mathbb{Z}_Q$ of a secret value $\alpha = \alpha_1 + \alpha_2 + ... + \alpha_n mod Q$ thought of as a fixed private

```
1     def encrypt(x):
2         # The number of shares generated varies according to how many
              of them are needed
3         share_a = random.randint(-Q,Q)
4         share_b = random.randint(-Q,Q)
5         share_c = (x - share_a - share_b) % Q
6         return (share_a, share_b, share_c)
```

Listing 14: Example of encryption function [43]

```
1 def decrypt(*shares):
2     return sum(shares) % Q
```

Listing 15: Example of decryption function [43]

key, which means that each worker that wants to generate the additive shares ($\alpha_i \in \mathbb{Z}_Q$) firstly calls an encryption function (e.g. Listing 14) and then spread the parties produced across other workers. Whereas, considering that the value encrypted is represented by $\alpha = \alpha_1 + \alpha_2 + ... + \alpha_n mod Q$, the decryption function can be defined as in Listing 15. The most relevant property of this algorithm, is that is possible to perform computation while the variable are still encrypted, which makes it perfect to compute a data mining function. The encryption and decryption functions are implemented by the framework PySyft, in order to avoid to hand-write all the primitive operations. The encryption function can be called on any PySyft tensor by calling `.share()`, while the decryption function is as simple as calling the method `.get()` on the shared variable (Listing 16).

After acquiring which are the operations done by the framework at low level to implement the SMPC algorithm, the real implementation can begin. As already said, the implementation refers to the tuorial [41] for the model and data sharing. The structure remains the same of the one introduced in the previous section so that the evaluation

```
1 x = torch.tensor([25])
2 encrypted_x = x.share(bob, alice, bill) # bob, alice, bill virtual
      workers
3 decrypted_x = encrypted_x.get()
```

Listing 16: Example of PySyft SMPC implementation [43]

```
1 if len(settings.training_devices) >= upper_bound:
2     logging.info("Applying selection criteria")
3     # Select only two devices
4     to_train = {k: settings.training_devices[k] for k in list(
          settings.training_devices)[:2]}
5 else:
6     loggin.info("No selection criteria applied")
7     to_train = settings.training_devices.copy()
```

Listing 17: Example of selection criteria (only the first two)

can be conducted on the same scenario. Hence, the Timer object calls the method `starting_training_enc` with an **upper bound** of two devices, which represents the max number of devices supported at the time of writing this work. Consequently, a selection criteria is applied (Listing 17) and as result of the definition (*shares distributed amongst 2 or more devices*) the **lower bound** can not be less than two. Subsequently, being a local case it requires the data distribution. Here, the method to call in order to distribute the data is `.share()` introduced previously, where, according to [41], each row of the dataset is shared among two workers. Considering that SMPC uses crypto protocols that require to work on integer, the method `.share()` necessitate of integer values as well (Listing 18). Thus, the framework provides the *Fixed Precision Tensors* which are used to store the PyTorch float tensors into *integer* (e.g. 0.123 with precision 2 becomes the integer 12). Furthermore, the function `.share()` requires another parameters called `crypto_provider`, which is a worker reponsibles for consistently generating random numbers and not colluding with any of the other parties (offline phase).

The next steps are sharing the model and starting the training. The former calls the method `.share()` as in the data case: `model = model.fix_precision().share(* to_train, crypto_provider = crypto_provider, requires_grad = True)`. The latter is a classical PyTorch training, where the only dissimilarity lies in the fact that all the parameters must be integer, e.g. the `.fix_precision()` function must be applied also to the optimizer (`optimizer.fix_precision()`). When the training ends, the model can be obtained by using the `.get()` method and in order to have float weights the method `.float_precision()` is called (Listing 19, line 5).

```
def secret_share(tensor):
        """
        Transform to fixed precision and secret share a tensor
        """
        return (
            tensor
            .fix_precision(precision_fractional=precision_fractional)
            .share(*workers, crypto_provider=crypto_provider,
                requires_grad=True)
        )
train_loader = torch.utils.data.DataLoader(NetworkTrafficDataset(
    args.test_path, transform=ToTensor()), shuffle=True)
result_train_loader = [
    (secret_share(data), secret_share(target))
    for i, (data, target) in enumerate(train_loader)
    if i < n_train_items / args.batch_size
]
```

Listing 18: Data encrypted distribution Network traffic scenario

```
    for i in range(args.epochs):
        cf.encrypted_training(model=encrypted_model, optimizer=
            optimizer_fixed, epoch=i, private_train_loader=
            private_train_loader, args=args)

    # Sending back the model
    model = model_encrypted.get().float_precision()
```

Listing 19: Training on encrypted data with encrypted model

```
1     for worker in to_train.items():
2         data_loader_testing[worker[0]] = list()
3         train_loader = torch.utils.data.DataLoader(
              NetworkTrafficDataset(args.test_path, transform=ToTensor()
              ), shuffle=True)
4         for i, (data, target) in enumerate(train_loader):
5             if i < n_train_items:
6                 data_loader_testing[worker[0]].append((data.send(
                      worker[1]), target.send(worker[1])))
7             else:
8                 break
```

Listing 20: Data distribution changes

5.2.1.3 Evaluations local scenario

This section provides a detailed comparison on the **training time performance** of the two implementations described in previous sections (5.2.1.1 and 5.2.1.2). The evaluations have been made by using the dataset provided by Koroniotis et al. [3], and a feed forward neural network with 2 hidden layers, one of 50 nodes and the other of 30 nodes, 10 nodes for the input layer and one output node. Despite the latter could be considered also to be used for real classification, in this work is used only to evaluate the architecture produced. Both the training tests have been repeated 5 times. Furthermore, they have been done by considering only one epoch to compute the training time and different amount of items (100, 200, 800, 1000).

In order to have the same scenario in both cases, one of them needs to be changed. First of all, the two data distributions must be allineated. Thus, the environment without encryption was adapted to be more congruent with the other environment. The data distribution that turns out in the plain scenario, by thinking to the production and distribution of additive shares among all the participant workers in the encrypted case, is showed in Listing 20. It is noticeable that the same number of items are sent to each worker, since the additive shares produced for each item are sent to all the workers.

Secondly, the number of devices must be the same, which implies to limit plain scenario to have an upper and lower bound of two devices.

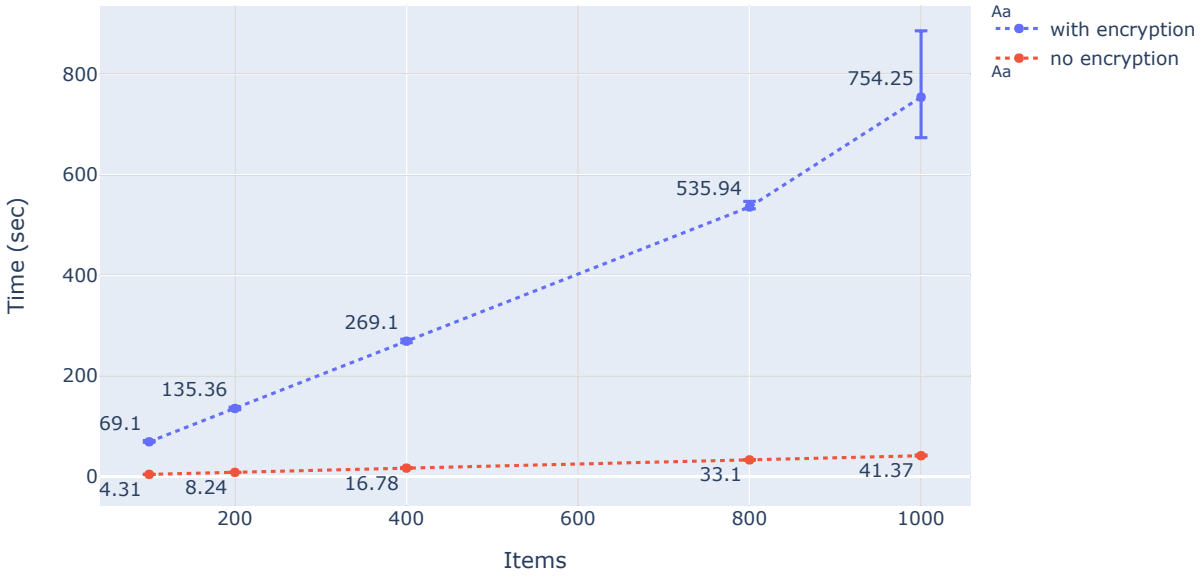


Figure 19: Comparison training time with and without encryption

The Fig. 19 emphasises the discrepancy in the time spent on training two workers in a cryptography and clear environment. The comparison is made on different amount of items from 100 to 1000. The latter is the only case which could represent a real case scenario (see graphs in section 5.2.2.4).

Overall, it is noticeable that the divergence between the two growth is huge. In fact, the training with 100 items in the cryptography scenario still requires more time (69.1 seconds) than the training time required in the plain scenario with 1000 items (41.37 seconds). However, the amount of discrepancy between the two training environments remains equivalent for almost all the amount of items considered (around 16 times). In fact, only in the last scenario with 1000 items the gap reaches a peak of more than 20 times (over 800 seconds). Furthermore, it is worth noting that the 1000 items case represents the scenario with the biggest variance achieved by the encrypted case.

In conclusion, the graph has emphasised the huge difference that characterise the two cases, which implies thoughtful considerations before introducing a cryptography algorithm. Thus, in order to reduce the training time, instead of encrypt both data and model a different strategy can be adopted. In fact, in the scenario of this work, where the data privacy is preserved thanks to the Federated Learning approach, can be embraced the

opportunity to encrypt only the model weights. Nevertheless, as described in section 2.4.7, adopting the SMPC in a Federated Learning scenario increases the probability of model and data poisoning.

5.2.2 A first real approach of Federated Learning

The skeleton resulting from the local approach creates the base infrastructure needed in a remote environment. It should be obvious that the components coming from the local case necessitate of some variations.

First of all, the events generated by edge devices must include the `port` where they await the model receiving. Thus, the event syntax becomes `(device_name, port, state)`, where, unlike the local case, the device's name **must** be an *ip address*. It is worth noting that the states are identical to those defined for local environment (5.2.1.1).

Secondly, the remote architecture implies to have a communication infrastructure between two different machines: Coordinator and remote device. Consequently, two abstractions representing the two ends point of a communication channel, through which the model is sent, are needed. The PySyft framework provides two new workers called *WebsocketClientWorker* and *WebsocketServerWorker*. The former, in the resulting architecture, acts as Coordinator end. The latter represents the Federated Learning participant side that processes all the messages received from the *WebsocketClientWorker*. The messages received typically contain a command (fit, async_fit etc.) that must be executed Server (device) side. It is worth noting that the communication protocol provided by the framework is similar to the concept of RPC (Remote Procedure Call) and RMI (Remote Method Invocation), where each command sent by the *Client* calls a function on the *Server*.

Considering the variantions produced on the skeleton, the next two subsections introduce new concepts necessary Coordinator and remote devices side so that the remote learning process can take place.

```
1  identifier = ip_address + ":" + str(port)
2  logging.info("Remote worker idetifier: " + identifier)
3  kwargs_websocket = {"host": ip_address, "hook": self.__hook, "
       verbose": True}
4  try:
5      worker = WebsocketClientWorker(id=identifier, port=port, **
           kwargs_websocket)
```

Listing 21: WebsocketClientWorker creation

5.2.2.1 Remote learning: Coordinator

In this subsection all implemented operations conducted by the Coordinator are explained. Considering the architecture obtained in the system design Fig. 15, after connecting to the Broker, the Coordinator starts to receive state events from edge-devices. After processing an event, the Coordinator creates a *WebsocketClientWorker* by using the parameters extrapolated from that event. In particular, as shown in Listing 21, the *WebsocketClientWorker*, in order to establish a connection with the *WebsocketServerWorker*, needs ip address and port where a device is waiting for a model. After that the network worker has been created, unlike in local case, the behaviour defined are **inference** and **training**. The inference is similar to the local environment. In fact, after obtaining a pointer to the inference data (Listing 22), the operations performed are: model sending to the remote worker, by simply calling the method `.send()`, and prediction making (Listing 22, line 9 and 14). After predictions have been made, they are picked up by using the method `.get()` (same method used to send back the model). It is worth noting that all predictions made edge side are sent back, which implies a high usage of bandwidth. Furthermore, the inference is executed on recorded data that is not ideal for an anomaly detection system (static data). However, the inference management and optimization is beyond the aim of this work. What instead is work core is the **remote training**.

It should be observed that the temporal window management, which is a mandatory operation in training cases, is equivalent to that defined for local environments (5.2.1.1). Hence, the Timer object calls another function designed for remote training.

```
# Obtain pointer to the inference data
data_pointer = worker.search("inference")# This return a list, we
    take only the first element (no logic defined)

# Only if we find the data we can conduct the inference
if data_pointer != []:
    data_pointer = data_pointer[0]

    # Send the model
    model = model.send(worker)

    # Apply the model to the data
    with torch.no_grad():
        # We are acting on pointer without looking at the data
        prediction_pointer = model(data_pointer)

        # The output needs to be 0 or 1, but it is sent back as
        # a probability which is a value between 0 and 1
        predictions = prediction_pointer.get()
```

Listing 22: Inference code

```
class FFNN(nn.Module):
    """
    Simple Binary FeedForward neural network
    """
    def get_traced_model(self):
        return torch.jit.trace(self, torch.zeros(10))
```

Listing 23: Method for model serialization

The remote case involves some steps that are completely new to the local case. First of all, to allow the model sending to edge devices, the model must be serialised. The serialization is performed by using an intermediate representation of the PyTorch model. PyTorch provides `torch.jit.trace()`, which is a set of tools used to capture the definition of the model and to create a serializable and optimizable version of it. Thus, in the class where is defined the feed forward neural network (example model of this work) is added an additional method that returns a serialised version of itself. In order to create this model version, the method `.trace()` requires mock data that initialises the model's input layer (Listing 23). The same procedure must be applied to the **loss function**.

```
1 @torch.jit.script
2 def loss_fn(target, pred):
3     # It depends on the model used (our model is binary)
4     return F.binary_cross_entropy(input=pred, target=target)
```

Listing 24: Definition loss function serialization

The method to be called in case of function is `torch.jit.script` (can be even used as decoration Listing 24), which is a script compiler that analyses the Python code and transform it into TorchScript (serialised version).

It can be observed that the inference case does not make a model serialization; this can lead to the following question: *why does the training require the model and loss function serialization?*

The framework PySyft provides another abstraction used in remote cases, which is of relevant importance for training environment. The **TrainConfig** abstraction adds a serializable wrapper, which allows the Coordinator to send all the training settings in one object. A comparison with the local case can make this concept more clear. The local case in Listing 13 shows that for each data in the `FederatedDataLoader`, which represents a set of data pointers, the model is sent to that data location. This action implies that when the training for that dataset row ends, the model is reclaimed by the Coordinator and at the next iteration it is sent back to that worker and so on. Thus, the code suggests that all the operations are coordinated remotely, which can indicate a high usage of bandwidth if a remote case is considered. The **TrainConfig** abstraction allows to send in a single object all training operators (epochs, optimizer, loss function etc.). Hence, the *WebsocketServerWorker*, when the command that starts the training (`fit`) is received, extracts from the TrainConfig object all the operators and starts a local training. The behaviour defined allows to reduce the bandwidth used and reflects more what the training should be in a Federated environment (an example of TrainConfig object can be found in Listing 25). However, the sending of this object **requires** model and loss function serialization.

Before to start with the training implementation, the device performance heterogeneity problem must be considered. The problem has already been mentioned in section

```
train_config = sy.TrainConfig(
    model=traced_model, # torch.jit.trace(model, mock_data)
    loss_fn=loss_fn, # torch.jit.script
    batch_size=batch_size, # Batch size of each training step
    shuffle=True,
    max_nr_batches=max_nr_batches, # If > 0, training on worker
        max_nr_batches
    epochs=epochs,
    optimizer=optimizer, # name of the optimizer to be used
    optimizer_args={"lr": lr}, # Learning rate of each training step.
)
```

Listing 25: TrainConfig Example

5.2.1.1 (sequential training), but it was not considered as critical for that environment. Conversely, in a remote environment the real devices could have different performance, which means that the training on some devices can be slower than the others. Thus, the implementation idea is to send the model to each participant without waiting for the model updates. The Coordinator waits for the model updates only after that each participant device has received the model. Therefore, the problem is partially managed, because slower devices can still influence the overall training performance. For now, the management insert a timeout after which the connection with the slowest client device is closed.

The Listing 26 shows the training implementation Coordinator side, in which the round concept is introduced(see section 4.2.1). The method `asyncio.gather()` runs the sequence of operations concurrently, which in this case is the method `train_remote` called for each worker. The keyword `await`, which precedes the `asyncio.gather()` calls, allows to define the behaviour defined above (partial asynchronism), i.e. wait only after that each participant has received the model. The method `train_remote()` creates an instance of TrainConfig (Listing 25), send it to the worker, start the training by calling the method `async_fit()` and reclaims the model back (Listing 27). After all the model updates have been gathered, the new global model is computed by using the **Federated Averaging** algorithm (as in the local case). All these operations are repeated for `round` times. When the training ends all the websockets are closed Coordinator side and the

```
for i in range(round):
    results = await asyncio.gather(
        *[
            cf.train_remote(
                worker=worker[1],
                traced_model=traced_model,
                batch_size=args.batch_size,
                optimizer="SGD",
                max_nr_batches=args.federate_after_n_batches,
                epochs=args.epochs,
                lr=learning_rate,
            )
            for worker in to_train.items()
    ]
    )
    models = {}
    loss_values = {}

    for worker_id, worker_model, worker_loss in results:
        if worker_model is not None:
            models[worker_id] = worker_model

    # Apply the federated averaging algorithm
    traced_model = utils.federated_avg(models)
```

Listing 26: Training with rounds

```
train_config.send(worker)
# We need to find the dataset remote side (dataset_key)
loss = await worker.async_fit(dataset_key="training", return_ids
    =[0])
model = train_config.model_ptr.get().obj
```

Listing 27: `train_remote` method

new global model is saved.

The Fig. 20 summarises all the Coordinator operations described in this section.

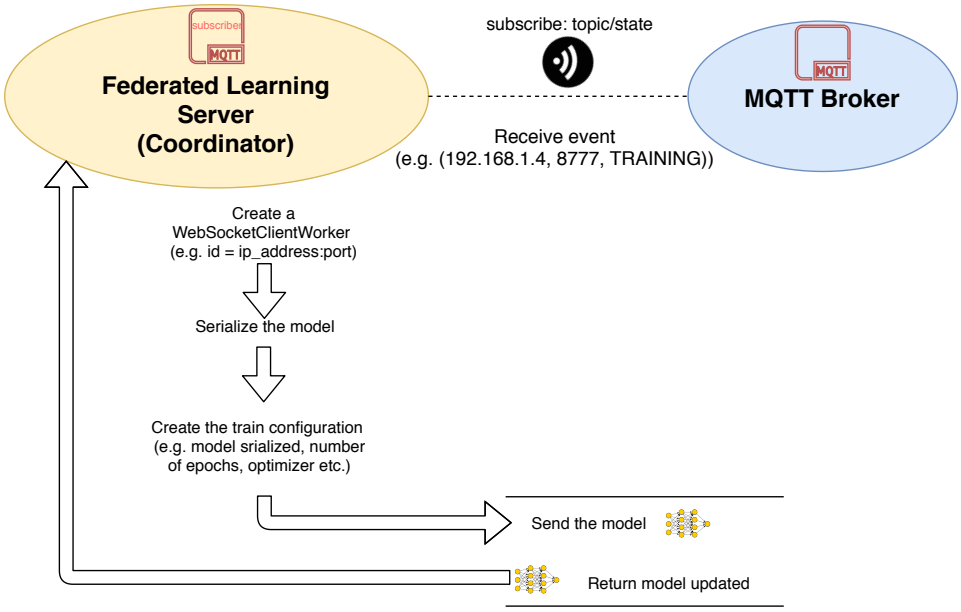


Figure 20: Coordinator operations

5.2.2.2 Remote learning: Edge-Device

This subsection summarises the middleware implemetation containing actions executed by the Federated Learning participants. Firstly, they are publishers in the MQTT infrastructure (section 4.2.1), as a result of state events generation. It is worth noting that the work does not provide a logic client side. In fact, it just concretises the implementation of a middleware that has to be executed when there is a state transaction. The latter can be determined by using a particular logic that depends on devices conditions.

Secondly, in order to identify the data location, the training and inference data must be registered. As result of the different implementation structure Coordinator side between training and inference, the data registration lies on the behaviour to carry out. In the training case it is as simple as call the method `add_dataset()` (Listing 28, line 6), which adds a dataset to a dictionary used in local training. This is possible because the training is executed locally, by extracting the training setting parameters from the single object (TrainConfig) received. Conversely, the inference phase is based on Pointer Tensors. In

```
1  # kwargs are all the parameter useful for a websocket creation
2  worker = WebsocketServerWorker(**kwargs)
3
4  # Training dataset registration
5  dataset = NetworkTrafficDataset(args.training, transform=ToTensor())
6  worker.add_dataset(dataset, key="training")
7
8  # Inference dataset registration
9  dataset_inf = NetworkTrafficDataset(args.inference, transform=
       ToTensor())
10
11 # Loading inference data
12 infrence_tensor = list()
13 for data in dataset_inf.data:
14     inference_tensors.append(th.tensor(data).float().tag("inference")
           )
15 worker.load_data(inference_tensors)
16
17 # WebsocketServerWorker start
18 worker.start()
```

Listing 28: Remote worker

fact, it requires that each row belonging to the dataset is made available as PointerTensor (Listing 28, line 12-15), so that the Coordinator can coordinate the inference operations without looking at the data. Nevertheless, considering that the core of the work is the remote training, this represents a temporary solution.

After registering the data, the *WebsocketServerWorker* can be started (Listing 28, line 18).

5.2.2.3 Deployment

The architecture deployment involves (Fig. 21): a laptop (Intel Core i5 2,4 GHz, RAM 8 Gb 1600MHz, SSD) as Coordinator, 2 Raspberry Pi 3 B+ as edge devices, and a router OpenWRT compliant (Netgear WNDR 3700 v2).

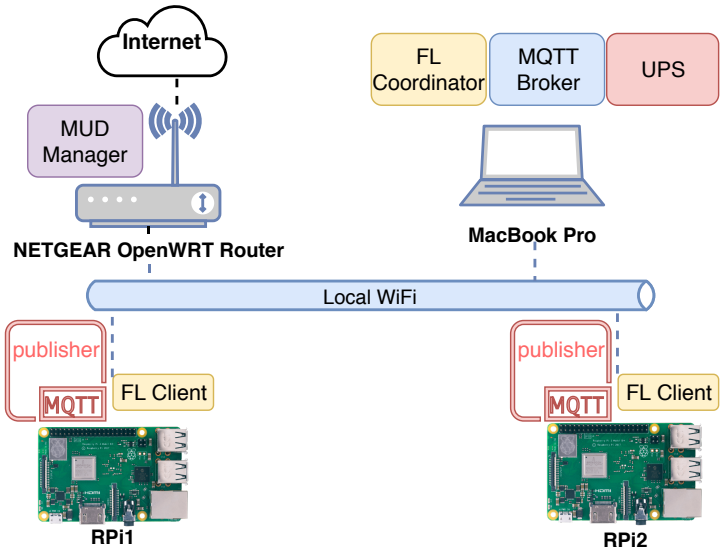


Figure 21: Lab deployment

The fact of using the Raspberry Pis on the edge makes IoT devices with computational limit and with no possibility of new deployment (vendor dependent) able to be easily integrated with the architecture provided. In fact, it is possible to introduce a two layers structure with a Raspberry top level and an IoT device bottom level (Fig. 22). Thus, an example of anomaly detection scenario could be: all the communications (internal and external to the network) pass through the Raspberry Pi, which collects the data in a static way and then performs the training or the inference on the basis of the device state. Furthermore, it should be noted that this deployment allows to make all the devices MUD compliant. In fact, as a consequence of interfacing IoT devices to the network through a Raspberry, it is possible to modify the DHCP client configurations to forge DHCP requests MUD compliant.

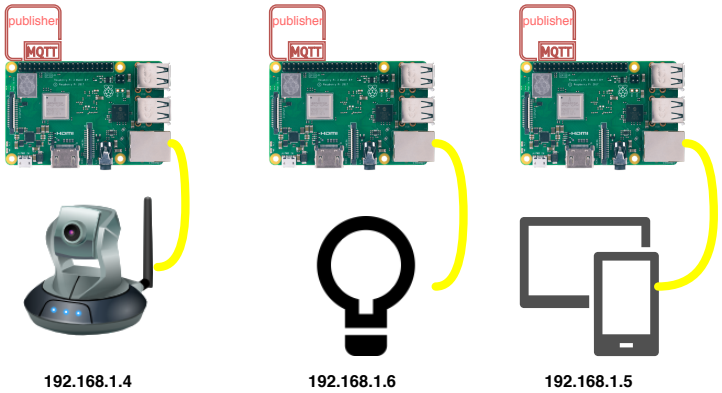


Figure 22: Example: Raspberrys as IoT device adapters

5.2.2.4 Evaluations

The evaluations of the implemented Federated Learning architecture describe how much the components are stressed in order to obtain certain performance. Thus, the experiments are performed on:

- bandwidth monitoring (stress on network interface);
- temperature monitoring (stress on IoT Devices);
- training time (performance of devices involved);
- model loss (performance of devices involved).

The design of the evaluations lies on **number of round** and **maximum number of batches**. The former regards the amount of interactions between the Coordinator and the edge devices. The latter regards how many iterations are executed on batches of data, e.g. 1000 max iterations means that random batches (i.e. `shuffle=True`) of data are selected to perform training for a maximum of 1000 times (if batch size is equal to 1, means 1000 random rows of the dataset are selected to perform training). In more detail, the experiments are carried out by considering: (3,6,12) number of rounds and (1000,2000,3000) max iterations. It is worth noting that the deployment (5.2.2.3) involves 2 RPi 3B+ (edge devices), a laptop (Coordinator) and a rounter (Netgear WNDR 3700 v2). The router involved in this scenario does not provide an Internet connection, so that

the bandwidth analysis can have a better level of accuracy. Additionally, each time that the outgoing data counter monitor (**cumulative** bandwidth measurement) is started it is set to 0. The model and the dataset used are the same used for the local test (see section 5.2.1.3 for further details).

It is of significant importance to analyse the temperature behaviour of a RPi in a idle case (Fig. 23), before to consider the training phase. It is noticeable that the temperature does not exceed the 50°C threshold for all the time in which it is monitored (around 1 hour). Thus, the value obtained can be used as comparison method to understand if the temperature reached during the training can cause slowdown problems, by also considering that the thermal throttling is achieved over 85°C.

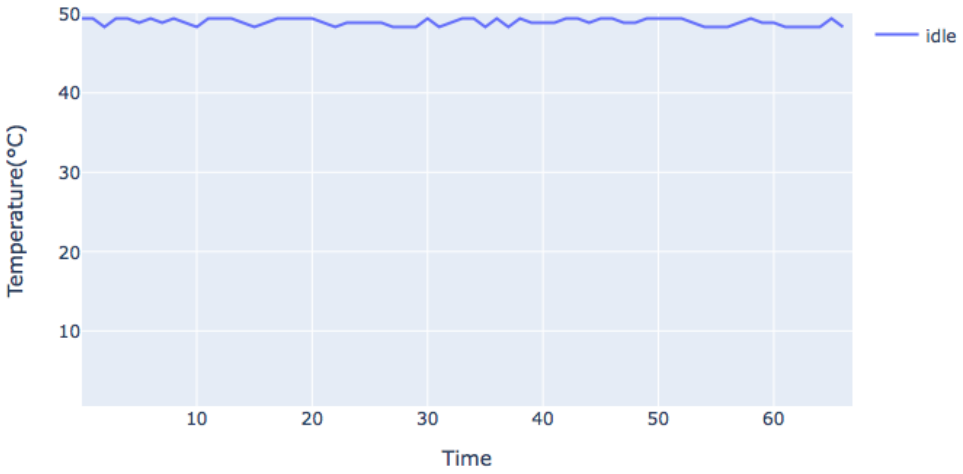
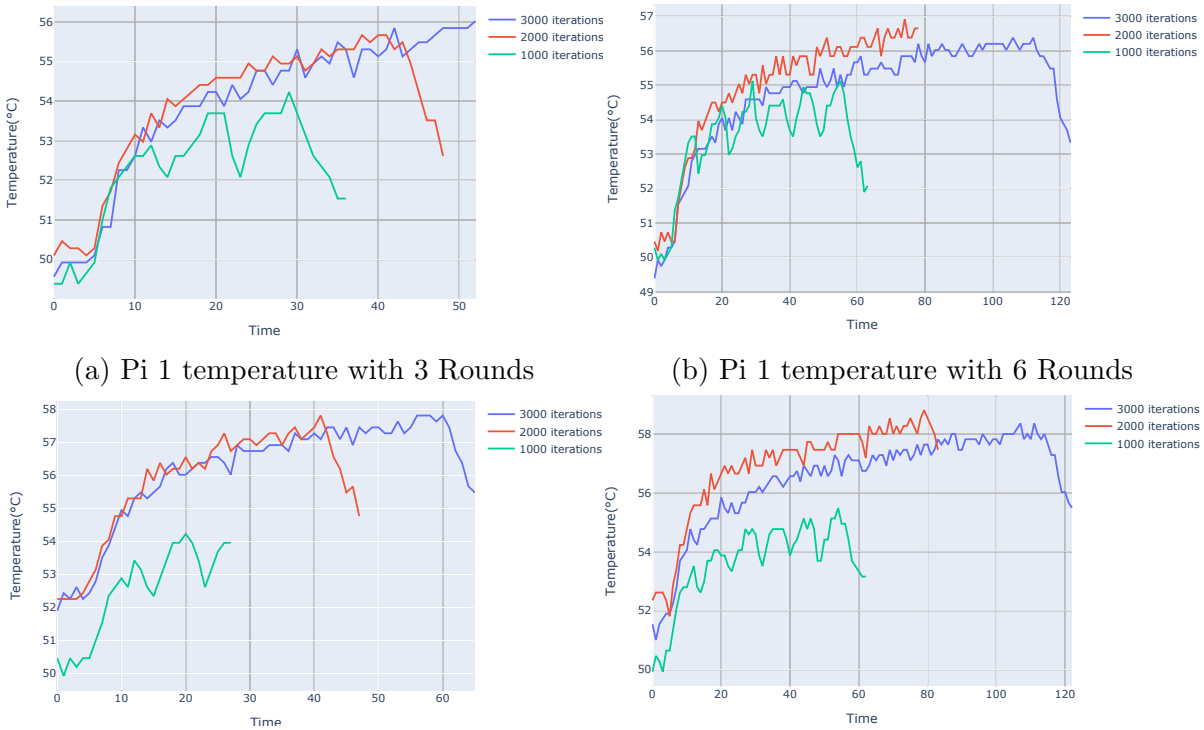


Figure 23: Raspberry Pi 3B+ idle CPU temperature

The temperature experiments are organised per round and repeated 3 times, in order to guarantee an average value. Furthermore, the temperature measurements are considered in a time window that starts 5 seconds before the training start time and ends 5 seconds after the training end time. The resulting graphs can give a first idea about the training time required by the devices involved. It should be observed that these experiments are based on a clock synchronization between devices and coordinator. In fact, the window considered does not have a complete accuracy. For example in the experiment on the first Raspberry with 3000 iterations showed in 24a the window ends before the training end time. However, it does not create relevant discrepancies for the final result.

Overall, the figure 24 demonstrates that the temperature does not exceed 60 °C. Rather, the max temperature reached is the 2000 iterations case illustrated in Fig. 24d with 58.4 °C, as result of the highest initial temperature. The Fig. 25 illustrates the temperature variation in a case with 12 rounds and 1000 max iterations, which represents the only one where 12 rounds have been tested.



(a) Pi 1 temperature with 3 Rounds

(b) Pi 1 temperature with 6 Rounds

(c) Pi 2 temperature with 3 Rounds

(d) Pi 2 temperature with 6 Rounds

Figure 24: Raspberrys temperature variations with 1000, 2000, and 3000 iterations

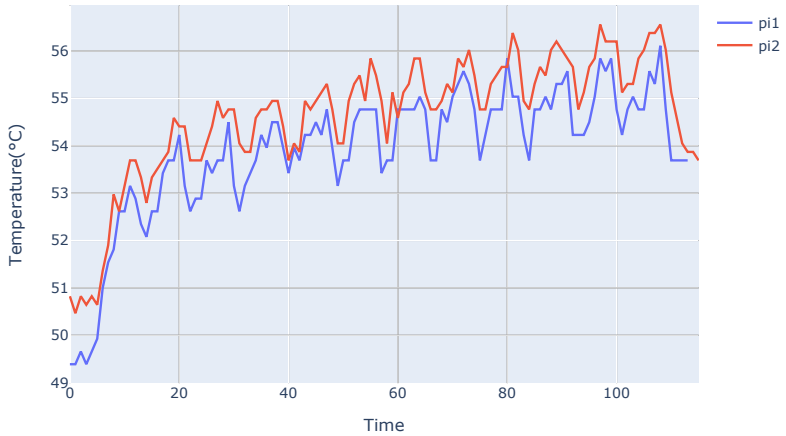
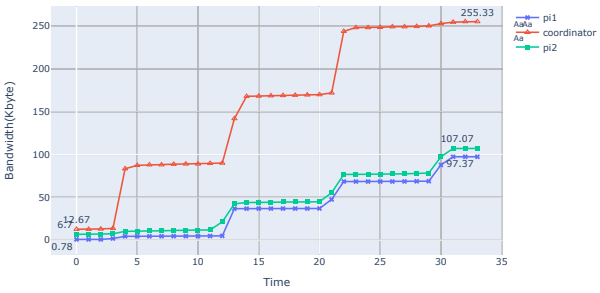


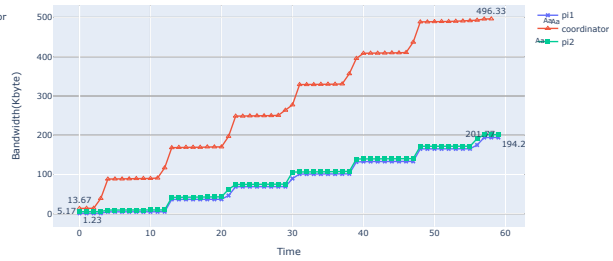
Figure 25: Temperature of both Raspberrys in 12 Rounds and 1000 iterations

It is noticeable that in all the graph the worst case is represented by the 3000 max iterations, where the temperature increase with an average of 6.75°C, followed by 2000 max iterations with around 6.18°C and lastly the 1000 max iterations with an average of 5.875°C. The 12 rounds case with 1000 max iterations, increases the average of 0.4°C which is considered negligible. From this analysis can be concluded that the number of max iterations influences the temperature of the components involved. Thus, when a paradigm, which defines a possible tradeoff value between the number of round and the max iterations used, has to be defined, it is preferred from a temperature perspective to increase the interactions rather than the data analysed. Obviously, as the next analysis confirm, the reduction of data analysed is made at the expense of other parameters.
The next stressed component that is considered is the devices' network interface. Considering that the model sent is **equivalent in all the scenarios**, the 1000 iterations case is the only one examinated. Furthermore, the experiments, for obvious reasons, look only at the **outgoing traffic** of each participant devices. As in the previous case, they have been repeated 3 times and they have been done by considering a smaller time window that starts 3 seconds before the training start time and ends 3 seconds after the traning end time.
At first glance, the three graphs (Fig. 26) are stairs-shaped where each step represents a round in which the model is sent. Thus, from the step dimension is possible to find an approximate size of the model. Note that the model size depends on different factors (e.g. number of weights). In order to compute it, only one chart can be used e.g. Fig. 26a. The bottom part of the first step in the coordinator stairs is around 13 Kbytes, while the top part around 85 Kbytes. By using this two values it is possible to define the approximate dimension of the TrainConfig object (around 36 Kbytes). Whereas, by looking the steps dimension of the outgoing raspberry bandwidth the approximate model dimension can be computed (around 30.61 kbytes). It should be observed that even the losses are sent by the Raspberrys, which means that the model dimension is a bit less then the one obtained. Furthermore, the model dimension is approximated because other parameters (e.g. WebSocket options) can influence the amount of traffic

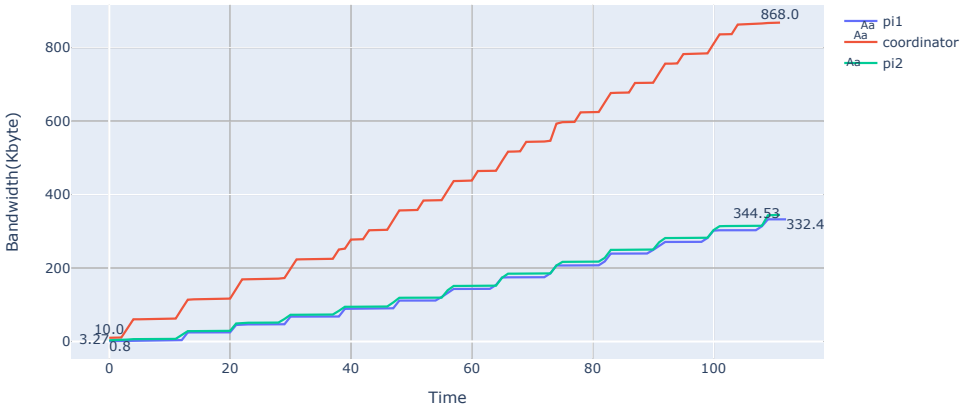
sent.



(a) Outgoing traffic 3 Rounds



(b) Outgoing traffic 6 Rounds



(c) Outgoing traffic 12 Rounds

Figure 26: Cumulative bandwidth 1000 iterations

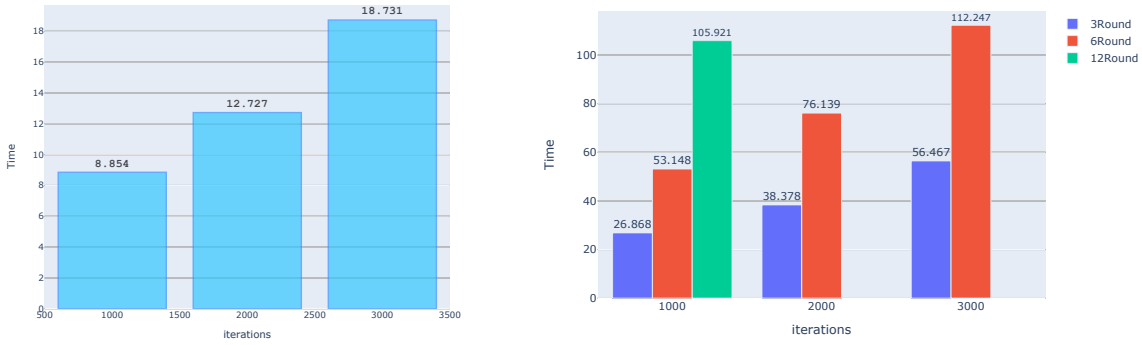
Thus, the graphs prove that the bandwidth, as expected, is strictly correlated to the model dimension, number of rounds and amount of devices involved (only Coordinator side). The two components analysed (network and temperature) have been tested also with a huge number of round (100, 200 and 400 rounds) by considering only one device and 1000 maximum iterations:

- **100** rounds: total bandwidth used for the entire training (round included) around 4.0 Mb and maximum temperature reached 60.148°C;
- **200** rounds: total bandwidth used for the entire training (round included) around 7.6 Mb and maximum temperature reached 60.148°C;
- **400** rounds: total bandwidth used for the entire training (round included) around 15.1 Mb and maximum temperature reached 60.148°C.

It worth noting that the bandwidth increase linearly as expected, but most importantly the temperature does not exceed 61°C, which represents a relevant factor in the paradigm definition.

Now that an analysis on the most stressed components involved in the federated protocol has been finalysed, the parameters helpful in generating a possible pattern to follow in order to optimise the resources usage are anlayzed.

First of all, the **training time** that typically depends on the device capabilities is useful to understand how long in average the Coordinator awaits for a response. The evaluation involves two training time scenario: the average of training times for each round (Fig. 27a) and total training time (Fig. 27b).



(a) Average training time for each round

(b) Average total training time

Figure 27: Training time analysis

Secondly, the model losses monitoring, which can be adopeted as evaluation method to discover the tradeoff between rounds and number of iterations used. In order to guarantee a correct comparison method, for each of the test carried out the model is set with random weights (created from scratch). Both the tests are performed on 1000, 2000, 3000 as max iterations and 3, 6 and 12 rounds (Fig. 28).

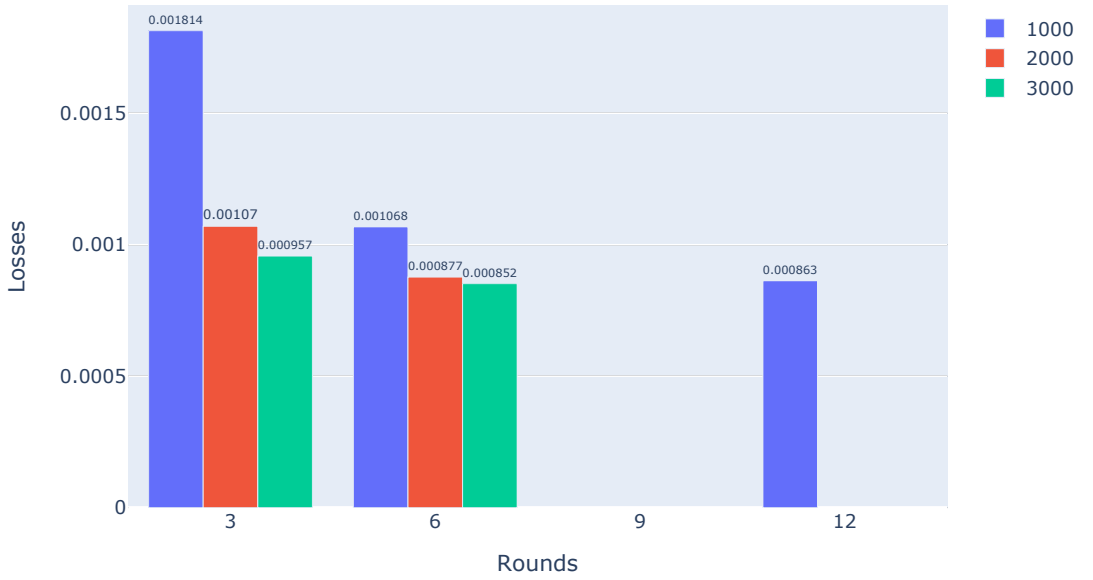


Figure 28: Model Losses

From the Fig. 28, it is possible to note that the training with 2000 and 3000 maximum number of iterations obtains approximately 0.0009 losses with only 6 rounds, which represents half of those needed with 1000 iterations. From a bandwidth perspective, can be asserted that increasing the number of iterations makes the protocol more efficient. In fact, considering the figures 26b and 26c, the bandwidth is reduced of the 43%. However, increasing the number of iterations can cause some problems on the device performance. Firstly, the training time for each round (Fig. 27a) is more than twice and around 1.5 times the case of 1000 max iterations, for respectively 3000 and 2000 max number iterations. The same situation is replicated for the total training time (Fig. 27b), where with 3000 max iterations and 6 rounds the time required is still more than that required by 1000 max iterations and 12 round, even including the delay time introduced by a classical network. This behaviour implies the existence of bottle neck problems Coordinator side. Although there are no huge difference, increasing the number of iterations influences the device's CPU temperature. In fact, comparing the 1000 and 3000 cases, it increases of almost 1°C. Furthermore, it has been shown that with a high number of rounds the temperature does not exceed a certain growth value (around 10°C). Thus, the right compromise in the tests carried by this work is the case with 6 rounds and 2000 max

iterations.

In conclusion, this analysis gives some of the right elements helpful to mitigate the ardous task of finding the right tradeoff between number of rounds and number of iterations (communication-efficiency and performance issues). The paradigm that arises from the analysis is to start to train a non-trained model with a huge number of round and low number of max iterations (1000 min value in these tests), when the devices involved have very limited resources. This helps to reach a high level of accuracy by preserving time and device's hardware wear. In case of devices with better performance (e.g. smartphones, cars etc.), it can be convenient to invert the paradigm, i.e. reduce the number of round and increase the max number of iterations, which reduces the bandwidth used in the network. Furthermore, another paradigm could be to train a non-trained model with high number of round and low number of iterations, and when a good level of accuracy is reached (model now trained) invert the paradigm, so that more data is analysed. At this paradigm a good logic client side to reduce the device overloading has to be added. Nevertheless, other solutions can be embraced: adopting the **transfer learning** technique (pre-training on public data Coordinator side, fine-tuning on sensitive data device side) or use an adaptive algorithm that chooses the right number of rounds and iterations to guarantee a good level of accuracy, by monitoring the surrounding environment.

5.3 MUD and Federated Learning in the same network

The next implementation step is to try to pick up the benefits of both the paradigms to reduce as much as possible the vulnerabilities in IoT devices environment. First of all, even if the work does not provide a machine learning model, the architeture is ready to embrace, especially in training terms, any kind of anomaly detection model for network traffic. Secondly, in this section is described the implementation of the final architecture obtained in 4.3 (Fig. 15), which further reduce the possible attack surface in a network by enabling the **device filtering**. As result of this implementation, the training and the

```
1  while true;do
2          input="/var/log/dhcpmasq.txt"
3          while read -r line
4          do
5                  COMMAND=$(echo $line | awk -F "|" '{ print $2}')
6                  MUD_URL=$(echo $line | awk -F "|" '{ print $7}')
7                  IP_TO_VALIDATE=$(echo $line | awk -F "|" '{ print $10}
                       ')
8
9                  if [ "$MUD_URL" = "-" -o "$COMMAND" = "OLD" ]; then
10                         echo "not valid"
11                 else
12                         ssh $REMOTE_USER@www.mfs.example.com "cd
                               $REMOTE_PATH/; python file_upgrader.py -c
                               $COMMAND -i $IP_TO_VALIDATE" < /dev/null
13                 fi
14                 sleep 1
15         done < "$input"
16         sleep $INTERVAL_OF_SCANNING
17 done
```

Listing 29: File monitor method

inference involve only MUD compliant devices, which means removing general purpose devices from protocol participants.

The implementation lies on a secure communication between the entity that hosts the Coordinator and the Router that hosts the osMUD Manager (Netgear WNDR 3700v2). As described in 5.1.1, the osMUD implementation of dnsmasq records all the DHCP request received in the file `/var/log/dhcpmasq.txt`(Listing 1). Thus, the file can be used to identify which are the MUD compliant devices, as a consequence of the MUD-URL extraction performed by the custom version of dnsmasq. The work provides a daemon process that each 10 seconds (default value) scans the file "`/var/log/dhcpmasq.txt`" and sends via SSH all the *ip addresses* that have a valid state and a MUD-URL. Indeed, the script executes remotely a python program that inserts the *ip address* received in a whitelist after checking the status (NEW or DEL), and checks if the *ip address* has already been entered.

This implementation gives the opportunity to the Coordinator to check if a device

with a particular *ip address* can take part in the Federated Learning protocol. The checking is carried out each time that an event is received, by executing the method `valid_iot_ip_address(ip_address)` (Listing 7). In future version, instead of continuously monitor the file, the remote method execution can be done directly in the script provided by osMUD (Listing 1) so that the number of remote invocation is reduced (bandwidth usage optimization).

Nevertheless, this implementation still exhibits some vulnerabilities that can be mitigated with a finest anomaly detection model. For example, as a result of the lack of device authentication in the network environment, an attacker can still steal the device identity by using an *ip spoofing attack*. Furthermore, a MUD compliant device can be compromised so that attacks like data and model poisoning still represent a threat. Thus, considering these observations, the next section tries to summarise which are the future works in security and scalability perspectives.

5.4 Future directions

A valuable improvement for the system can be to **extend the YANG-based MUD file** by adding a field containing structure and weights of a model. The idea is to give the opportunity to each manufacturer to define a model that describes normal behaviours of its devices in a classical network environment, which results in using the model to detect abnormal device actions. The Federated Learning architecture provided by this work can be exploited in order to employ the model directly on edge-device data. The application of this architecture can encourage further learning to improve the manufacturer model, by preserving the data privacy. Hence, a general workflow to adopt can be:

- the MUD compliant devices send a DHCP request, which generates a request of the new version of MUD file;
- the osMUD manager in processing this new MUD file creates a JSON file that contains structure and weights of the model;

- the receiving of state transaction events Coordinator side generates an interaction with the osMUD manager in order to find the model JSON file of that device category (e.g. located with either the ip addres or mac address);
- if the file is retrieved the model is built on the Coordinator, which then starts the training or inference phase;
- if the file is not retrieved the Coordinator uses a general anomaly detection model for network traffic.

This architecture solves the most massive problem in the Federated Learning infrustracture: **heterogeneity**. In fact, it is very tough to create a general model that identifies anomalies in a network considering the devices traffic heterogeneity. For example, the traffic generated by a camera has completely different properties if compared with that generated by a fridge. Thus, having a model provided by the manufacturer that knows the normal traffic generated by its own devices, creates that secure environment IoT devices need. This MUD extension requires also some changes in the Federated Learning architecture mainly in the training phase, when the global model has to be updated.

Another improvement involves the osMUD infrastructure. As already outlined, osMUD is thought to easily deploy in OpenWRT environment. However, if it is executed regardless of the router, beside reducing the impact on router performance, it paves the way for new interesting interactions. For example, instead of generating rules for a simple Packet Filtering firewall included in OpenWRT router, they can be generated for more complex system such as P4 programmable switch [44]. Furthermore, the osMUD manager can be used to manage the rules of distinct subnets, in order to improve the scalability of the architecture provided.

Finally, beside the extensions regarding the UPS infrastructure already outlined in 4.1.3, it is of relevant importance to apply an authenticated protocol for the MUD compliant devices. Although the MUD specification proposes a X.509 extension to support the MUD-URL, at the moment of writing this work there are no implemented solutions

that adopts this authenticated protocol for MUD compliant devices. The introduction of an authentication protocol allows to solve further security problems related to the publish/subscribe pattern used in the Federated Learning architecture, where there is no authentication among the devices involved.

Conclusions

Although the IoT devices are strong enough to host malicious code, due to resource and economic constraints they do not have the means to protect themselves from malicious actors. Thus, the accelerating employment of them in the home networks turns out in new threats. In this direction, the MUD specification has been approved as a method to define IoT device communication patterns. The standard allows and encourages the IoT manufacturers to provide MUD file consisting of access control rules that describe the user device's proper communication behaviour. Furthermore, even the devices no longer supported by the manufacturers can be helped by this standard. In fact, for MUD-capable devices that reach the end-of-life stage, the use of MUD provides additional protection that is not avilable for non MUD compliant devices. Nevertheless, even for MUD-enabled devices, there are still some behaviours that can be determined only by local policy. In fact, if the default policy provided by the manufacturer is not sufficient or too restrictive for the network standard, user actions are necessary to configure the device according to a different and desired policy. Additionally, MUD is not intended to address network authorization of general purpose devices, as their manufacturers can not predict a specific communication pattern. In order to identify abnormal behaviour in the network, the MUD deployment can be supported by a machine learning algorithm, which monitors the activities of all the MUD enabled devices.

It has been shown that this work, in order to create a MUD enabled network, employs the open source MUD (osMUD) implementation. The adoption of this solution allows to easy develop new functionalities, which could be of primary importance to increase

the network security. In fact, a new interaction between the osMUD manager and a new entity provided by this work has been introduced. The entity provides important services that aim to reduce vulnerabilities within a network.

First of all, the entity hosts a web server that provides a user-friendly interface needed to interact with the MUD components to modify their default settings when needed. The behaviour defined allows to build whitelist rules more suitable for the network in which MUD is deployed.

Secondly, the entity hosts one component of a distributed architecture that makes the network able to adopt the Federated Learning protocol. This architecture proposed by this work enables the opportunity to learn a model in the network, which aims to detect abnormal behaviours in outgoing and incoming packets generated by the IoT devices. Furthermore, thanks to the Federated approach adopted, the model is learnt collaboratively on the participant devices, by avoiding to send the data to a central server, which means preserve important properties of privacy, ownership and locality of data.

Subsequently, the work proposes an additional interaction between the osMUD manager and the distributed architecture component hosted by the entity. The aim of this is to reduce the Federated Learning participants to those devices that are MUD compliant, in order to reduce poisoning attacks (e.g. data and model poisoning) that characterise the classical Federated environment.

Finally, the work gives different evaluations that demonstrate the web server effectiveness in creating new local policies suitable for the network deployment, and that allow to give some suggestions which aim to optimise the communications involved in the Federated protocol. Furthermore, future directions have been proposed: these can improve and facilitate the employment of a model in the distributed architecture, and expand the MUD uses.

Bibliography

[1] S. Smith, "'internet of things' connected devices to almost triple to over 38 billion units by 2020." https://www.juniperresearch.com/press/press-releases/iot-connected-devices-to-triple-to-38-bn-by-2020, 2019.

[2] P. Newman, "Iot report: How internet of things technology growth is reaching mainstream companies and consumers." https://www.businessinsider.com/internet-of-things-report?r=US&IR=T, 2019.

[3] N. Koroniotis, N. Moustafa, E. Sitnikova, and B. Turnbull, "Towards the development of realistic botnet dataset in the internet of things for network forensic analytics: Bot-iot dataset," *Future Generation Computer Systems*, vol. 100, pp. 779–796, 2019.

[4] M. Miettinen, S. Marchal, I. Hafeez, N. Asokan, A.-R. Sadeghi, and S. Tarkoma, "Iot sentinel: Automated device-type identification for security enforcement in iot," in *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pp. 2177–2184, IEEE, 2017.

[5] Y. Meidan, M. Bohadana, Y. Mathov, Y. Mirsky, A. Shabtai, D. Breitenbacher, and Y. Elovici, "N-baiot—network-based detection of iot botnet attacks using deep autoencoders," *IEEE Pervasive Computing*, vol. 17, no. 3, pp. 12–22, 2018.

[6] E. Lear, R. Droms, and D. Romascanu, "Manufacturer Usage Description Specification." RFC 8520, Mar. 2019.

[7] A. Hamza, D. Ranathunga, H. H. Gharakheili, T. A. Benson, M. Roughan, and V. Sivaraman, "Verifying and monitoring iots network behavior using mud profiles," *arXiv preprint arXiv:1902.02484*, 2019.

[8] J. Ren, D. J. Dubois, D. Choffnes, A. M. Mandalari, R. Kolcun, and H. Haddadi, "Information exposure from consumer iot devices: A multidimensional, network-informed measurement approach," in *Proceedings of the Internet Measurement Conference*, pp. 267–279, 2019.

[9] "Collection of talks/links on manufacturer usage description." `http://www.sandelman.ca/SSW/ietf/mud-links/`.

[10] A. Wool, "A quantitative study of firewall configuration errors," *Computer*, vol. 37, no. 6, pp. 62–67, 2004.

[11] L. Lhotka, "JSON Encoding of Data Modeled with YANG." RFC 7951, Aug. 2016.

[12] D. Dodson, W. Polk, M. Souppaya, W. Barker, E. Lear, B. Weis, Y. Fashina, P. Grayeli, J. Klosterman, B. Mulugeta, *et al.*, "Securing small business and home internet of things (iot) devices: Mitigating network-based attacks using manufacturer usage description (mud)," tech. rep., National Institute of Standards and Technology, 2019.

[13] M. Ranganathan, "Soft mud: Implementing manufacturer usage descriptions on openflow sdn switches," in *International Conference on Networks (ICN)*, 2019.

[14] A. consortium of network security companies, "Open source manufacture usage specification." `https://osmud.org`, 2018.

[15] A. Mortensen, T. Reddy, and R. Moskowitz, "DDoS Open Threat Signaling (DOTS) Requirements." RFC 8612, May 2019.

[16] "Openwrt project." `https://openwrt.org`, 2004.

[17] K. Bonawitz, H. Eichner, W. Grieskamp, D. Huba, A. Ingerman, V. Ivanov, C. Kiddon, J. Konecny, S. Mazzocchi, H. B. McMahan, *et al.*, "Towards federated learning at scale: System design," *arXiv preprint arXiv:1902.01046*, 2019.

[18] Google, "Tensor flow." `https://www.tensorflow.org`.

[19] A. Hard, K. Rao, R. Mathews, S. Ramaswamy, F. Beaufays, S. Augenstein, H. Eichner, C. Kiddon, and D. Ramage, "Federated learning for mobile keyboard prediction," *arXiv preprint arXiv:1811.03604*, 2018.

[20] T. D. Nguyen, S. Marchal, M. Miettinen, H. Fereidooni, N. Asokan, and A.-R. Sadeghi, "Dïot: A federated self-learning anomaly detection system for iot," in *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pp. 756–767, IEEE, 2019.

[21] pallets, "A micro web framework written in python." `https://palletsprojects.com/p/flask/`.

[22] Google, "Keras." `https://www.tensorflow.org/guide/keras`.

[23] Google, "Tensor flow federated." `https://www.tensorflow.org/federated/get_started`.

[24] Google, "Federated learning for image classification." `https://www.tensorflow.org/federated/tutorials/federated_learning_for_image_classification`.

[25] T. Ryffel, A. Trask, M. Dahl, B. Wagner, J. Mancuso, D. Rueckert, and J. Passerat-Palmbach, "A generic framework for privacy preserving deep learning," *arXiv preprint arXiv:1811.04017*, 2018.

[26] C. driven project, "Pytorch." `https://pytorch.org`.

[27] A. Melnikov and I. Fette, "The WebSocket Protocol." RFC 6455, Dec. 2011.

[28] H. B. McMahan, E. Moore, D. Ramage, S. Hampson, *et al.*, "Communication-efficient learning of deep networks from decentralized data," *arXiv preprint arXiv:1602.05629*, 2016.

[29] Y. Lindell, "Secure multiparty computation for privacy preserving data mining," in *Encyclopedia of Data Warehousing and Mining*, pp. 1005–1009, IGI Global, 2005.

[30] I. Damgård, V. Pastro, N. Smart, and S. Zakarias, "Multiparty computation from somewhat homomorphic encryption," in *Annual Cryptology Conference*, pp. 643–662, Springer, 2012.

[31] I. Damgård, M. Keller, E. Larraia, V. Pastro, P. Scholl, and N. P. Smart, "Practical covertly secure mpc for dishonest majority–or: breaking the spdz limits," in *European Symposium on Research in Computer Security*, pp. 1–18, Springer, 2013.

[32] J. Konečnỳ, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," *arXiv preprint arXiv:1610.05492*, 2016.

[33] E. Bagdasaryan, A. Veit, Y. Hua, D. Estrin, and V. Shmatikov, "How to backdoor federated learning," *arXiv preprint arXiv:1807.00459*, 2018.

[34] Y. Afek, A. Bremler-Barr, D. Hay, R. Goldschmidt, L. Shafir, G. Abraham, and A. Shalev, "Nfv-based iot security for home networks using mud," *arXiv preprint arXiv:1911.00253*, 2019.

[35] A. Hamza, H. H. Gharakheili, T. A. Benson, and V. Sivaraman, "Detecting volumetric attacks on lot devices via sdn-based monitoring of mud activity," in *Proceedings of the 2019 ACM Symposium on SDN Research*, pp. 36–48, ACM, 2019.

[36] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan, "Adaptive federated learning in resource constrained edge computing systems," *IEEE Journal on Selected Areas in Communications*, vol. 37, no. 6, pp. 1205–1221, 2019.

[37] A. Bremler-Barr, H. Levy, and Z. Yakhini, "Iot or not: Identifying iot devices in a shorttime scale," *arXiv preprint arXiv:1910.05647*, 2019.

[38] "Open source manufacture usage specification." `https://github.com/ayyoob/mudgee`, 2018.

[39] "Message queue telemetry transport." `http://mqtt.org`.

[40] osMUD, "Dnsmasq able to extrapolate mud-url from the dhcp requests." `https://github.com/osmud/dnsmasq`, 2019.

[41] "Secure deep learning tutorial." `https://github.com/OpenMined/PySyft/blob/master/examples/tutorials/Part%2012%20bis%20-%20Encrypted%20Training%20on%20MNIST.ipynb`.

[42] S. Wagh, D. Gupta, and N. Chandran, "Securenn: 3-party secure computation for neural network training," *Proceedings on Privacy Enhancing Technologies*, vol. 2019, no. 3, pp. 26–49, 2019.

[43] "Secure multi-party computation tutorial." `https://github.com/OpenMined/PySyft/blob/master/examples/tutorials/Part%2012%20bis%20-%20Encrypted%20Training%20on%20MNIST.ipynb`.

[44] "P4 programming." `https://p4.org`, 2013.