D6.2

First version of Pilot Use Case Implementation

WP6 – Smart Home Pilot Use Case

SIFIS-Home

Secure Interoperable Full-Stack Internet of Things for Smart Home

Due date of deliverable: 31/01/2023
Actual submission date: 30/01/2023

Responsible partner: DOMO
Editor: Domenico De Guglielmo;
E-mail address: domenico.deguglielmo@domo-iot.com

30/01/2023
Version 1.0

Project co-funded by the European Commission within the Horizon 2020 Framework Programme		
Dissemination Level		
PU	Public	**X**
PP	Restricted to other programme participants (including the Commission Services)	
RE	Restricted to a group specified by the consortium (including the Commission Services)	
CO	Confidential, only for members of the consortium (including the Commission Services)	

The SIFIS-Home Project is supported by funding under the Horizon 2020 Framework Program of the European Commission SU-ICT-02-2020 GA 952652

Authors: Domenico De Guglielmo (DOMO), Luca Barbato (LUM), Ossi Saukko (CENTRIA), Olli Isohanni (CENTRIA)
Approved by: Luca Ardito (POL), Goran Selander (ERI)

Revision History

Version	Date	Name	Partners	Section Affected Comments
0.1	20/10/2022	Defined ToC	DOMO	All
0.2	28/11/2022	Inserted first description of testbed components	DOMO	All
0.3	5/12/2022	Completed DOMO testbed description	DOMO	All
0.4	16/01/2023	Inserted Centria Components	CENTRIA	All
0.5	27/01/2023	Received internal reviews	ERI, POL	All
1.0	29/01/2023	Ready to Submit	DOMO	All

Executive Summary

This deliverable reports the preliminary implementation of the SIFIS-HOME pilot architecture, which also represents the real-testbed for validation of SIFIS-Home mechanisms.
First, we describe in detail the involved devices and the various software components that are used in the SIFIS-HOME pilot implementation. We then detail the network and system architecture of our pilot and show how all the different devices involved in the pilot are interconnected. The interaction between the different devices and components is described by focusing on a specific smart home use case, i.e. allowing a user to control their lights and appliances using a web-based control panel. We then conclude the deliverable reporting the future actions that need to be performed in order to successfully demonstrate the different smart home use cases reported in deliverable D6.1.

Table of contents

Contents

1 Introduction

The smart home pilot is the real testbed used to show the possibility of integration of SIFIS-Home in existing architectures and devices, presenting an architecture, which fully matches with the SIFIS-Home paradigms. In particular, the architecture involves real devices classified in Smart Devices and NSSDs, fully distribution of functionalities among decentralized smart devices to improve reliability and resilience, secure communication and privacy aware data management.
This deliverable reports the details of the current SIFIS-HOME pilot implementation. It is structured as follows. First, we describe the different smart and not-so-smart devices that are used in the pilot. For each one of them we report its hardware components and describe its specific use in the pilot. We continue by describing the details of the network architecture of our pilot in order to show how the devices communicate and are interconnected. The various components that are installed and executed on the various devices are then described. Also, we show the specific actions that need to be performed to install the SIFIS-HOME software components on the pilot devices. We then describe how the devices, and the software components interact by focusing on a specific smart home use case, i.e., allowing the user of the smart home to turn on and off lights by using a web-based control panel. We conclude the deliverable reporting the future plans and actions that need to be performed in order to successfully demonstrate the smart home use cases reported in D6.1.

2 Devices used in the pilot

This section describes the different smart (SD) and not-so-smart (NSSD) devices that are used in the current pilot implementation. The main hardware characteristics of the various devices are reported and their specific use in the pilot is highlighted.

2.1 *Smart Devices*

Smart devices are powerful devices where it is possible to install a number of applications. They execute the set of SIFIS-HOME software components that compose the SIFIS-HOME Smart Device framework. In the following we describe the smart devices that are currently used in the pilot.

2.1.1 DoMO gateway

Figure 1 shows the DoMO gateway, i.e. the main smart device used in our pilot. The DoMO gateway is a quite powerful device, based on the Banana PI R3 board, that is provided with a Quad Core ARM A53 CPU and 2 GB of DDR RAM. Also, it has 8GB of EMMC flash available. Regarding network connectivity, the DoMO gateway is equipped with two 4x4 WiFi 6 network chips (2.4Ghz and 5Ghz bands), 5 Gb Ethernet ports and 2 2.5 Gb SFP ports. Also, it is provided with a user-accessible USB 3.0 compliant port that allows connecting external USB devices. Additional details of the device are reported in Figure 2.

The DoMO gateway runs an OpenWrt Linux distribution. In detail, we prepared a specific OpenWrt distribution for the DoMO gateway that includes the various SIFIS-HOME software components to be used. We describe the tools and the packages that we developed to generate the OpenWrt SIFIS-HOME distribution in section _DoMO_GW_Openwrt .

The DOMO gateway is the device where the most important SIFIS-HOME services, such as the SIFIS-HOME DHT Manager, are executed. It not only runs the vital SIFIS-HOME services but also provides WiFi connectivity to the NSSD that are installed in a SIFIS-HOME house.

Figure 1: Domo Gateway

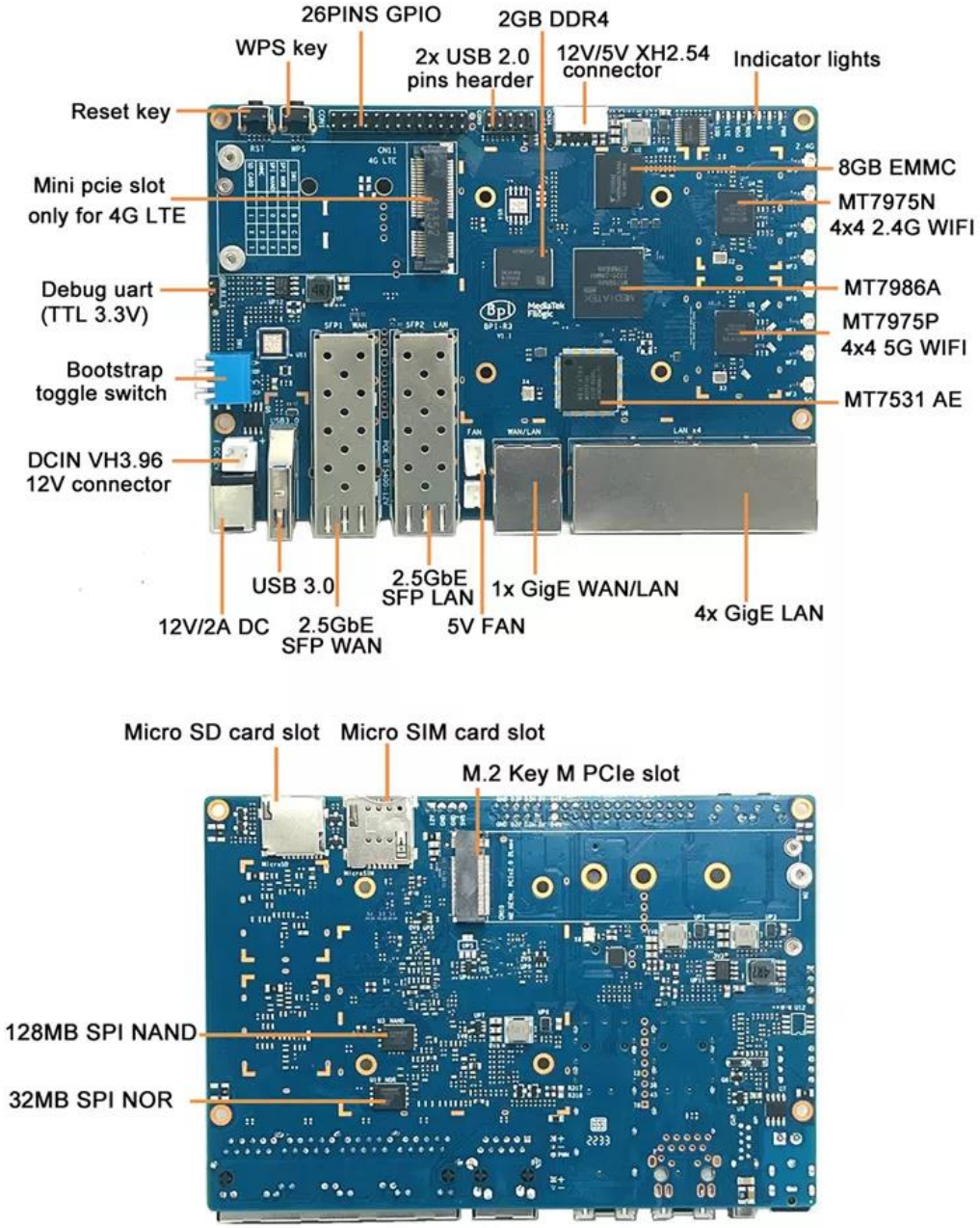


Figure 2: Domo Gateway details

2.1.2 Raspberry PI 4, Model B

For the development of the Smart Device Mobile API component we used a Raspberry Pi 4, model B device. It has 4 GB of RAM available and runs a 64-bit version of the Raspberry Pi OS. The device running the Smart Device Mobile API must be able to create a dedicated WiFi network to which the smartphone intended to run the SIFIS-HOME Mobile application will connect to. The Raspberry Pi 4 is equipped with a WiFi chip that can be configured to run in Access Point mode. Hence, it has all the needed features required to run the Smart Device Mobile API component. Once the Smart Device Mobile API component development is completed, we plan to prepare a specific OpenWrt package that includes it, so that it can also be executed on the DoMO gateways.

2.2 *Not So Smart Devices*

Not so smart devices are small, constrained devices that are mainly used to interact with the physical world. The set of SIFIS-HOME software components that they execute is named SIFIS-HOME NSSD framework. We report the details of the NSSDs used in the pilot in the following section.

2.2.1 DoMO WiFi actuators

The SIFIS-HOME pilot uses different types of WiFi actuators, provided by DoMO, to control and monitor the energy consumption of the lights, sockets, shutters and appliances installed inside the house. The WiFi actuators are simple devices that provide output and input channels and allow to turn on and off the appliances/light/sockets they are attached to as well as measure and report their energy consumption. Using the input channels of the actuators it is also possible to detect the state of attached buttons and bistable buttons as well as the state of attached window and door contact sensors. All the actuators are equipped with an Espressif ESP8266 WiFi chip that can be flashed with a custom firmware. The NSSD devices used in the SIFIS-HOME pilot should expose a WebOfThings (WoT) compliant API in order to be controlled and monitored. To this end, we developed a WoT compliant firmware whose details are described in section _DoMO_WiFi_actuators.

In the following we briefly describe the characteristics of the various types of WiFi actuators that are used in the pilot.

Shelly 1
Figure 3 shows the Shelly 1 WiFi actuator. It provides one input channel and a potential-free output channel. It is not provided with an energy monitoring chip. It can be used to turn on and off lights and appliances as well as heating systems. Also, it can detect state changes of buttons/contacts to which its input channel is connected to.

Figure 3: Shelly 1

Shelly 1PM
Figure 4 shows the Shelly 1PM WiFi actuator. It provides one input channel and one output channel. It can be used to turn on and off lights and appliances and monitor their energy consumption. Also, it can detect state changes of buttons/contacts to which its input channel is connected to.

Figure 4: Shelly 1PM

Shelly 2.5

Figure 5 shows the Shelly 2.5 WiFi actuator. It provides two input channels and two output channels. It can be used to turn on and off light and appliances and monitor their energy consumption. Also, it allows to open/close shutters and curtains. Finally, it can detect changes in the state of buttons/contacts to which its input channels are attached.

Figure 5: Shelly 2.5

Shelly Dimmer

Figure 6 shows the Shelly Dimmer WiFi actuator. It provides two input channels and one output channel. It can be used to control dimmable lights and monitor their energy consumption. Also, it can detect changes in the state of buttons/contacts to which its input channels are attached.

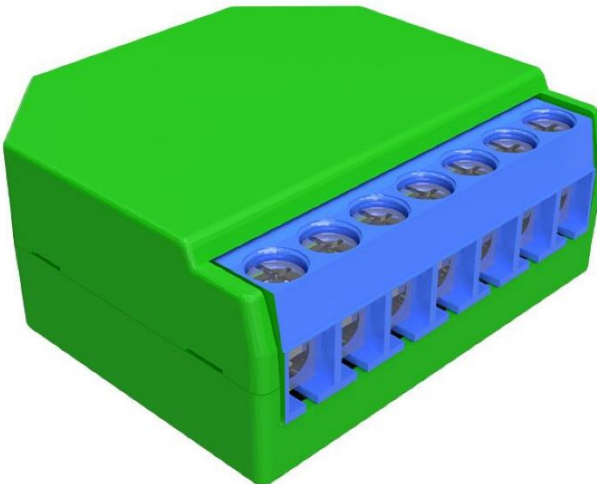


Figure 6: Shelly Dimmer

Shelly RGBW

Figure 7 shows the Shelly RGBW WiFi actuator. It provides one input channel and a number of output channels that can be used to control RGBW led lights. Also, it can detect changes in the state of buttons/contacts to which its input channel is attached to.

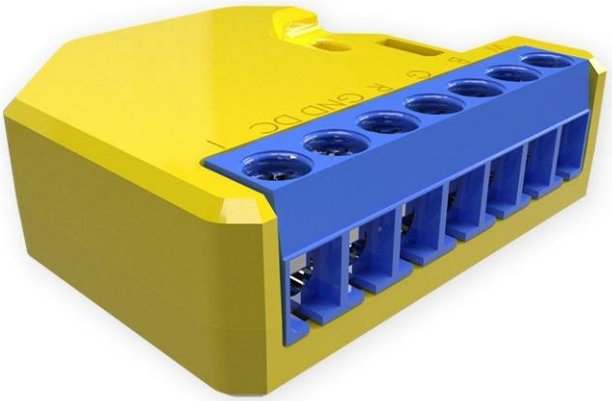


Figure 7: Shelly RGBW

Shelly EM

Figure 8 shows the Shelly EM WiFi actuator. It is a device that can provide the total power and energy consumption of the house where it is installed.

Figure 8: Shelly EM

3 Pilot network architecture

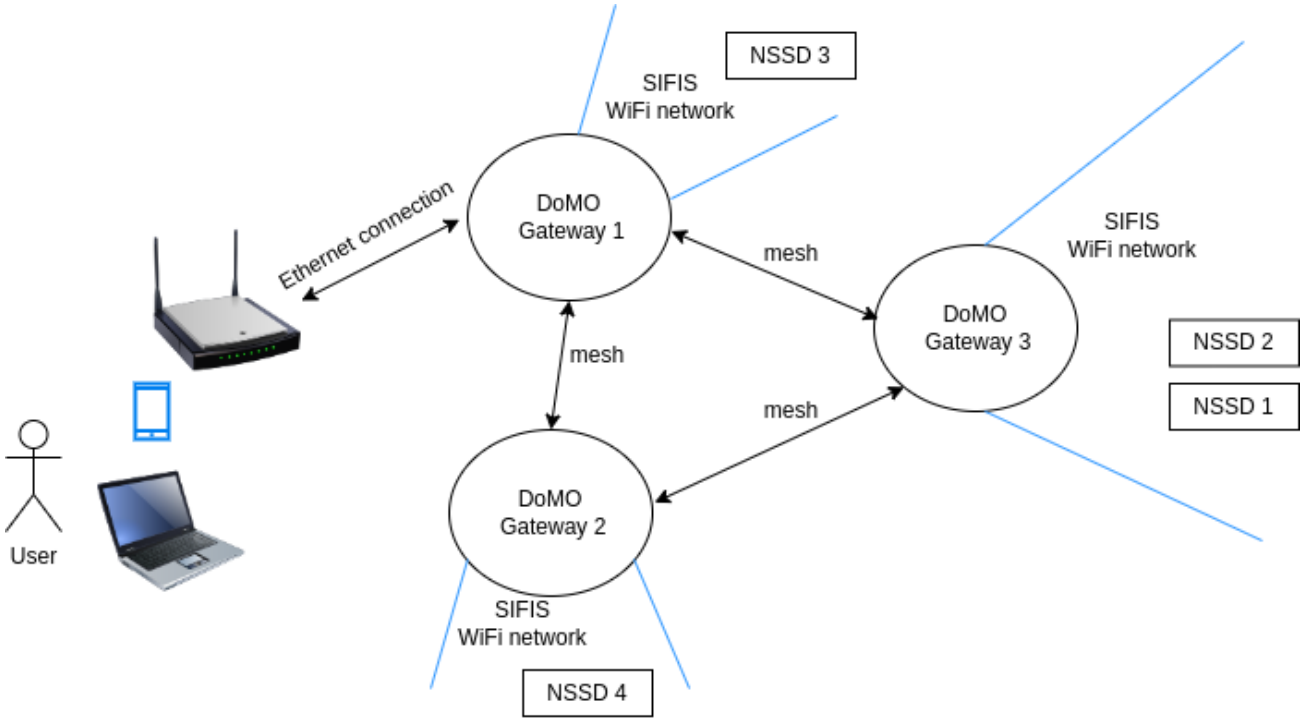


Figure 9: Pilot network architecture

Figure 9 shows the current network/system architecture of the SIFIS-HOME pilot. As it can be observed a number of devices are present in the pilot:

Smart Home Router: it is the device used to provide Internet Connectivity to the house. It is in general provided by the Internet Service Provider (ISP) and it does not execute SIFIS-HOME software components. We assume that it provides WiFi connectivity to the users of the smart home and

executes a DHCP server that assigns IP addresses to the network devices deployed in the house.

DoMO gateway: a number of DoMO gateways are present in the house. At least one of them is connected to the Smart Home Router using an Ethernet connection. The DoMO gateways are connected with each other by means of a dedicated WiFi mesh network. In addition, they advertise a specific WiFi network (SIFIS WiFi network in Figure 9) to which the NSSD devices used in the pilot connect to.

NSSD (DoMO WiFi actuators): there are a number of NSSD (WiFi actuators) inside the house. They are configured to connect to the network advertised by the different DoMO gateways present in the house. Once connected to the WiFi network of the DoMO gateway they expose a WoT compliant API.

User smartphone/PC: they are the devices that can be used by the users of the smart home to control the lights and appliances connected to the NSSD deployed in the house.

4 SIFIS-HOME Framework Integration

The DoMO gateways run a SIFIS-HOME OpenWrt distribution, i.e. an OpenWrt distribution that also contains SIFIS-HOME software components (the specific tools and actions needed to prepare the SIFIS-HOME OpenWrt distribution are described in the following). In detail, the current SIFIS-HOME OpenWrt distribution contains the SIFIS-HOME DHT Manager and the SIFIS-HOME NSSD Manager. We provide a detailed description of these components in the following sections. Also, in section _Smart_Device_Mobile we report the details of the Smart Device Mobile API component that has been currently developed and tested using a Raspberry PI 4 device. In the next months, we plan to prepare a specific OpenWrt package for the Smart Device Mobile API to include it in the SIFIS-HOME OpenWrt distribution.

4.1 *SIFIS-HOME DHT Manager*

The SIFIS-HOME DHT is a component that offers a completely distributed publish/subscribe mechanism through which SIFIS-HOME applications can exchange messages. The SIFIS-HOME DHT allows to publish both persistent and volatile messages. Persistent messages are messages that need to be stored in a persistent way, so that they are available even after a node reboot operation. In detail, persistent messages are stored on an Sqlite database. Volatile messages are instead messages that need to be delivered to all the available applications but that do not need to be persisted on disk.

The SIFIS-HOME DHT has a built-in mechanism to solve possible data conflicts that can arise when a network partition occurs. In detail, every time a message is published on the DHT, the DHT also stores its publication timestamp. Then, the publication timestamp is used to assure that only the most recently published messages will be stored and made available to the applications.

The SIFIS-HOME DHT has been developed using the Rust language. Rust applications can include the DHT by embedding it as a library. Non-Rust applications can access the DHT by means of a REST + WebSocket API provided by the DHT Manager. Please note that Rust applications can also use the Rest + WebSocket API provided by the DHT Manager to access the DHT. In the next section we report the details of the Rest and WebSocket API provided by the DHT Manager.

SIFIS-HOME DHT Manager REST API

The DHT Manager provides a REST API through which it is possible for an external application to access the DHT. Here we report the main REST API endpoints. In the following <DHT_ADDRESS> indicates the IP address of the node where the DHT manager executes while <DHT_HTTP_PORT> is the HTTP port used by the DHT Manager.

HTTP Method	Endpoint	Parameters	Description
GET	http://<DHT_ADDRESS>:<DHT_PORT>/get_all	-	Returns all the published persistent messages
GET	http://<DHT_ADDRESS>:<DHT_PORT>/topic_name/<topic_name>	<topic_name>	Returns all the persistent messages whose topic_name is <topic_name>
GET	http://<DHT_ADDRESS>:<DHT_PORT>/topic_name/<topic_name>/topic_uuid/<topic_uuid>	<topic_name> <topic_uuid>	Returns the message whose topic_name is <topic_name> and topic_uuid is <topic_uuid>
POST	http://<DHT_ADDRESS>:<DHT_PORT>/pub	The content of the message to be published is specified in the request body (type application/json)	Publishes a volatile message whose content is specified in the payload of the request
POST	http://<DHT_ADDRESS>:<DHT_PORT>/topic_name/<topic_name>/topic_uuid/<topic_uuid>	<topic_name>, <topic_uuid>, The content of the message to be published is specified in the request body (type application/json)	Publishes a persistent message whose topic_name is <topic_name> and whose topic_uuid is <topic_uuid>

SIFIS-HOME DHT Manager WebSocket API

The DHT Manager provides also a WebSocket API to access the DHT. In the following we report the WebSocket messages that can be sent to the DHT Manager to request operations on the DHT. The websocket API URL is ws://<DHT_ADDRESS>:<DHT_PORT>/ws.

Message	Parameters	Description
RequestGetAll	-	Returns all the published persistent messages
{"RequestGetTopicName": {"topic_name": "<topic_name>"}}	<topic_name>	Returns all the persistent messages whose topic_name is <topic_name>
{"RequestPubMessage": <payload>}	<payload>: payload of the message to be published	Publishes a volatile message
{“RequestPostTopicUUID”: { “topic_name”: <topic_name>, “topic_uuid:”: <topic_uuid>,	<topic_name>, <topic_uuid>, <payload>	Publishes a persistent message whose topic_name is

“value”: <payload> }}		<topic_name> and whose topic_uuid is <topic_uuid>

We want to highlight that, currently, access to the DHT HTTP API is unprotected. In the next months we plan to develop TLS + HTTP authentication to protect access to the DHT HTTP API.

SIFIS-HOME DHT Manager code and deployment

The SIFIS-HOME DHT Manager code is available on GitHub (https://github.com/sifis-home/libp2p-rust-dht). It can be easily built by running the command “cargo build”.

The command line parameters that are available are:

***SQLITE_FILE*:** absolute path of the sqlite file where persistent messages published on the DHT are stored.

***PRIVATE_KEY_FILE*:** path to the file containing the private key of the node in PEM format. A 2048 bytes long private RSA key file in PEM format can be generated using command "openssl genrsa -out private.pem 2048". If *private_key_file* does not exist, the key pair is automatically generated by sifis-dht and stored inside file *private_key_file* .

***IS_PERSISTENT_CACHE*:** if set to true indicates that sifis-dht is authorized to write messages to the provided sqlite file. If set to false, the *SQLITE_FILE* content will only be used to initialize the cache.

***SHARED_KEY*:** 32 bytes long shared symmetric key in hex format (command "openssl rand -hex 32" can be used to generate a random key)

***HTTP_PORT*:** port to be used for the HTTP interface

***LOOPBACK_ONLY*:** if set to true, only the loopback interface will be used, meaning that only other local instances of sifis-dht are discovered. If set to false, all the available network interfaces of the device will be used. Hence, two sifis-dht instances running on the same local network should discover each other.

The DHT Manager is built and added to the SIFIS-HOME DHT OpenWrt distribution by creating a dedicated OpenWrt package (see below for the details). Please note that an instance of the DHT Manager is present on every DoMO Gateway.

4.2 *SIFIS-HOME NSSD Manager*

The SIFIS-HOME NSSD Manager is the SIFIS-HOME component responsible for interacting with the NSSD devices present in the house. It has been developed using the Rust language and is composed of three main modules: the DHT module, the M-DNS Module and the Web of Things (WoT) Module.

- **DHT Module:** the DHT Module is the responsible for communicating with the DHT Manager. It uses the WebSocket API provided by the DHT Manager to access the DHT. In detail, it establishes a persistent WebSocket connection with the DHT Manager for being able to

receive commands from the user (e.g. “turn on a certain light”) and for updating the status of the managed devices (e.g. to signal that an actuator is connected to the system).

- **M-DNS Module:** the M-DNS Module uses the m-DNS protocol to detect the presence of WiFi actuators in the network advertised by the DoMO gateway where it is in execution. In detail, the m-DNS module periodically performs an m-DNS discovery operation that produces as a result the list of WiFi actuators that are connected to the DoMO gateway advertised network.
- **WoT Module:** the Web of Things module manages the communication of the NSSD Manager with the NSSD. It uses a WoT API to interact with the NSSD.

The interaction between the NSSD Manager and both the DHT Manager and NSSDs is shown in Figure 10 while the operations continuously performed by the NSSD Manager are reported in Figure 11.

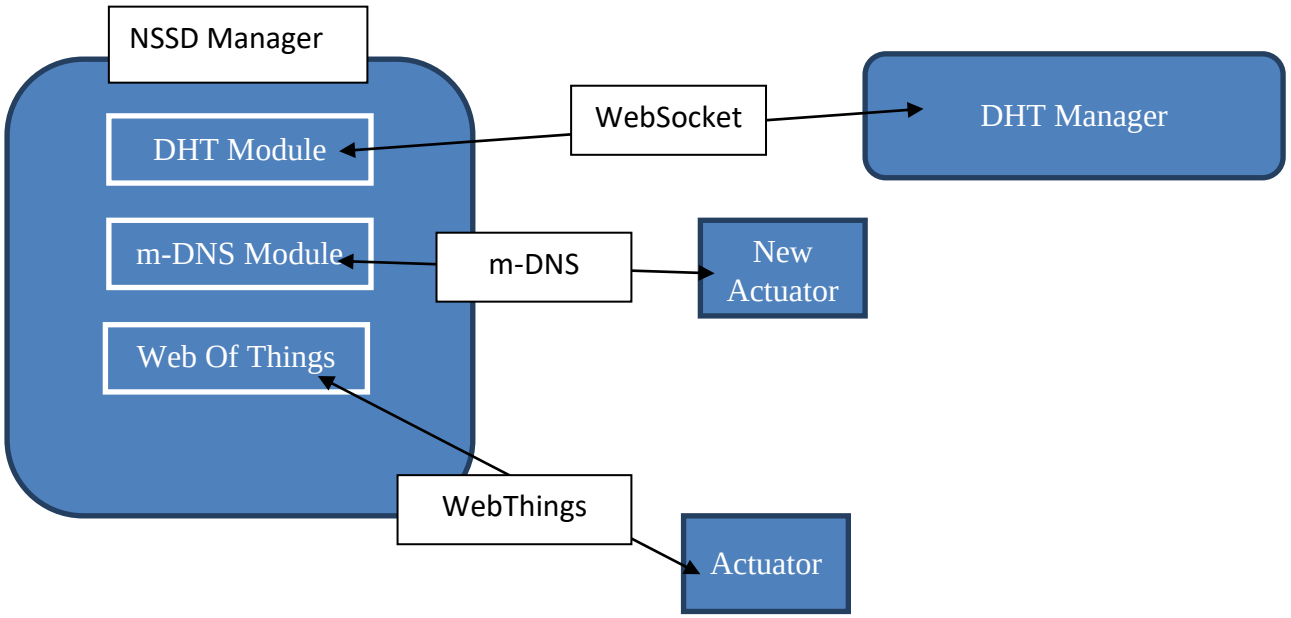


Figure 10: NSSD Manager interaction with the DHT Manager and WiFi actuators (NSSD)

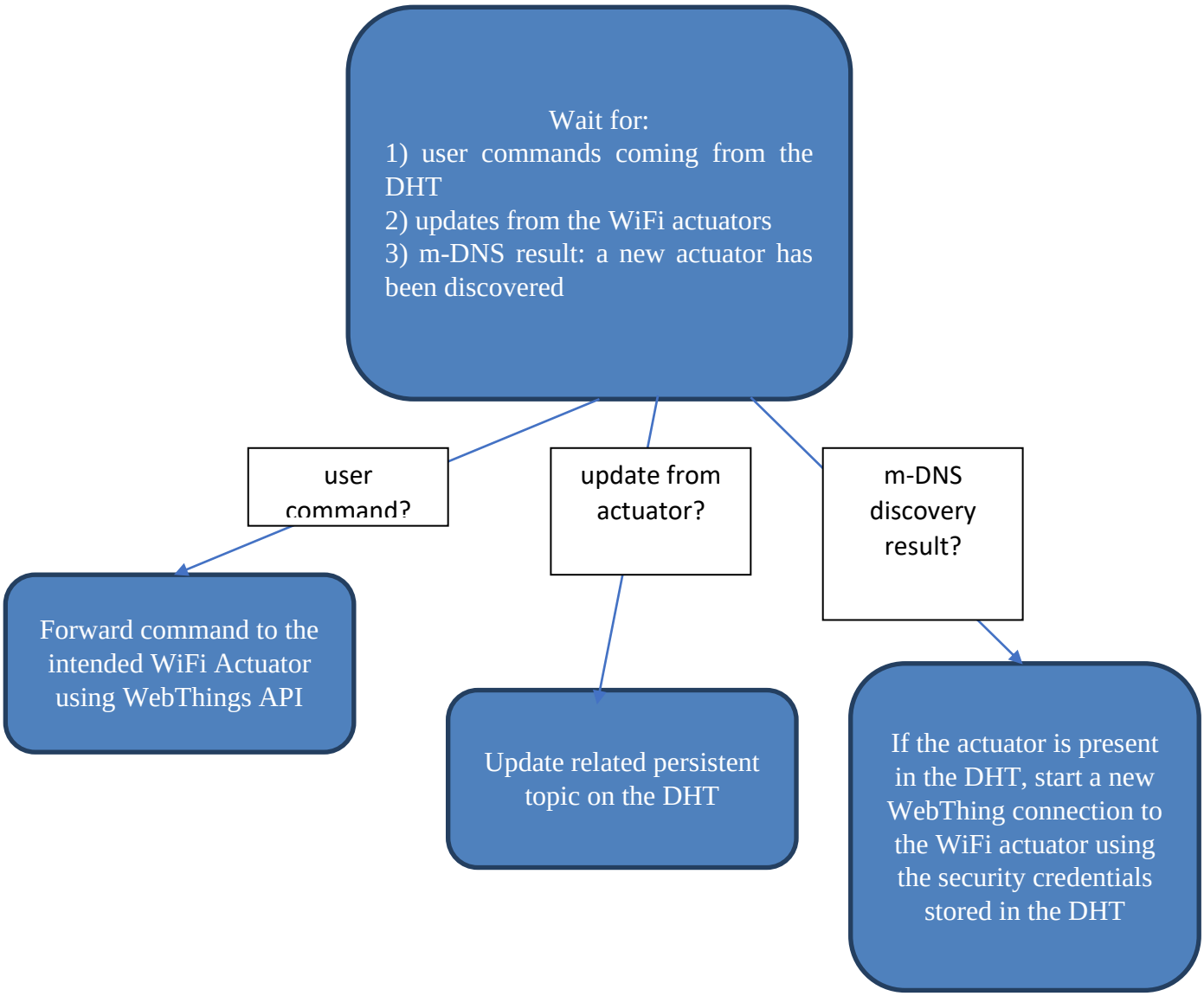


Figure 11: NSSD Manager operations

As it can be observed, the NSSD Manager continuously waits for an event to occur. The events can be of three different types: i) a user command is received from the DHT, ii) an update from one of the WiFi actuators is received, iii) a new WiFi actuator connected to the network advertised by the DoMO gateway. In case a user command is received from the DHT, it is forwarded to the intended WiFi actuator using the WoT API offered by the WoT firmware installed on the WiFi actuators. Conversely, if a state update is received from one of the actuators, the related persistent topic on the DHT is updated. Finally, if a new WiFi actuator has been discovered by the m-DNS module and the actuator has been registered on the DHT, a new WoT connection towards it is started using the security credentials stored in the DHT.

The NSSD Manager uses both persistent and volatile topics. In detail, we use persistent topics to store the status of the various managed actuators inside the DHT, so that it is accessible by all the SIFIS-HOME applications. Conversely, we make use of volatile messages to send/receive user commands.

Persistent topics used by the NSSD Manager

topic_name	topic_uuid	Description
shelly_1	MAC address of the actuator	Topic used to store a Shelly 1 Device.
shelly_1pm	MAC address of the actuator	Topic used to store a Shelly 1pm Device.
shelly_25	MAC address of the actuator	Topic used to store a Shelly 2.5 Device
shelly_dimmer	MAC address of the actuator	Topic used to store a Shelly Dimmer Device
shelly_em	MAC address of the actuator	Topic used to store a Shelly EM Device
shelly_rgbw	MAC address of the actuator	Topic used to store a Shelly RGBW Device

Volatile topic used by the NSSD Manager to receive user commands:

```
{
"command": {
        "command_type": "shelly_actuator_command",
            "value": {
                "mac_address": <actuator_mac_address>,
                "shelly_action": {
                        "input": {
                            "action": {
                                    "action_name": "set_output",
                                    "action_payload": <action_payload>
                            }
                        }
                }
            }
}
}
```

As it can be observed, the intended receiver of the command is specified by parameter "actuator_mac_address". Instead, the specific action to be executed is identified by means of the "action_name" "action_payload" parameter pair.

SIFIS-HOME NSSD Manager code and deployment

The NSSD Device Manager source code is available on Github (https://github.com/sifis-home/domo-wot-bridge) and can be easily built by running command "cargo build". The NSSD Manager is built and added to the SIFIS-HOME DHT OpenWrt distribution by creating a dedicated OpenWrt package (see below for the details). Please note that an instance of the NSSD Manager is present on every DoMO Gateway.

4.3 *SIFIS-HOME Smart Device Mobile API*

The Smart Device Mobile API is a component that allows the initialization of new Smart Devices by part of the SIFIS-HOME Mobile application. In addition, the component allows checking of the device status and running commands for restarting or shutting down the device and performing a factory reset. The component is executed on all the Smart Devices of a SIFIS-HOME enabled house.

Smart Device Initialization

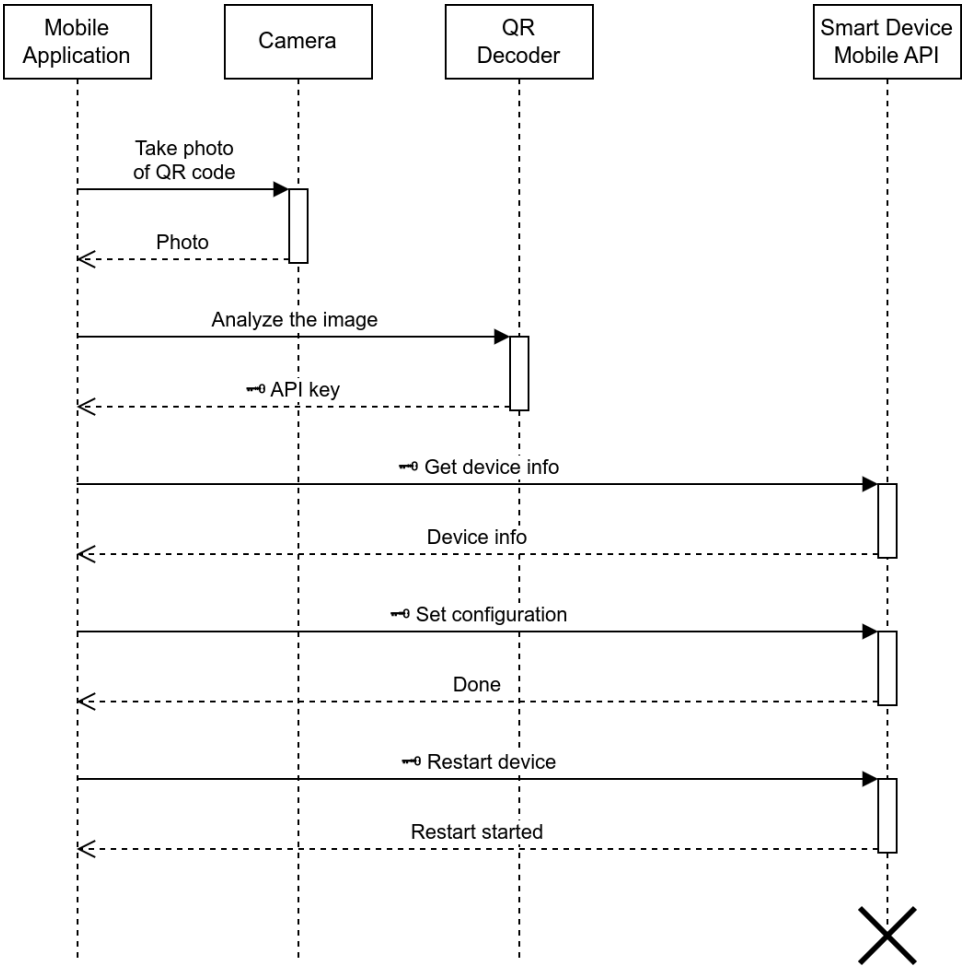


Figure 12: Smart Device Mobile API usage

The Smart Device Mobile API is executed on every Smart Device. The Mobile Application connects to the Smart Device Mobile API component using a dedicated WiFi network created by the Smart Device at boot time. An API key is needed to access and use the Mobile API. We assume that a QR code containing the API key needed to access the Mobile API is printed on the case/box of every Smart Device. The mobile application can scan the API key QR code with the smartphone camera (see also Figure 12, where full arrows represent API invocation and dotted arrows represent the response message).

The mobile application retrieves device information using the API key. The information contains both the device name and a unique device identifier. The mobile application shows this information to the user and helps him choose the correct device if several devices are available.

The mobile application allows to send and change the configuration of the device. After the configuration has been sent, the mobile application asks the device to restart. After the restart, the device can start joining the SIFIS-Home network.

Device Information File

We assume that a specific file, named *Device Information File,* is created on every smart device when

it is flashed. The file contains the unique data of the device and the API key needed to access the Smart Device Mobile API. The current file path is */opt/sifis-home/device.json*. The file *c*ontains the following information:

- Product name (product-name field)
- Unique identifier (uuid field)
- API key (authorization-key field)
- Private key path (private-key-file)

Device info file example:

```
{
   "authorization-key": "256-bits in hex format",
   "private-key-file": "/opt/sifis-home/private.pem",
   "product-name": "Name of the product (not unique)",
   "uuid": "128-bit UUID in standard hex format"
}
```

The file is accessed and used by the Mobile API component to provide device information to the mobile application.

Device Configuration

A specific file, named *Device Configuration File*, is used to store the device configuration. Its current file path is */opt/sifis-home/config.json* file. If the Device Configuration file is present on the device file system it means that the device has already been configured and, hence, the Smart Device has all the needed information to join a SIFIS-Home network. If the Device Configuration file is not present, the device has not been configured yet. In this case, the device goes in initialization mode. In detail, the WiFi chip of the Smart Device is configured to operate in access point mode and a dedicated WiFi network is created. Then, the SIFIS-HOME Mobile Application can be used to configure the device.
All the SIFIS-Home services running on the device can read the Device Configuration file, but only the Smart Device Mobile API is allowed to create and change it. The SIFIS-Home services that are not expected to run in initialization mode are not executed if the Device Configuration file is missing.

The configuration file contains the following fields:

- Device name (name field)
- Shared key for DHT (dht-shared-key field)

Example of configuration file:

```
{
   "dht-shared-key": "32 bytes in hex format",
   "name": "User-defined name for the device"
}
```

The operations executed by every smart device after boot are summarized in Figure 13.

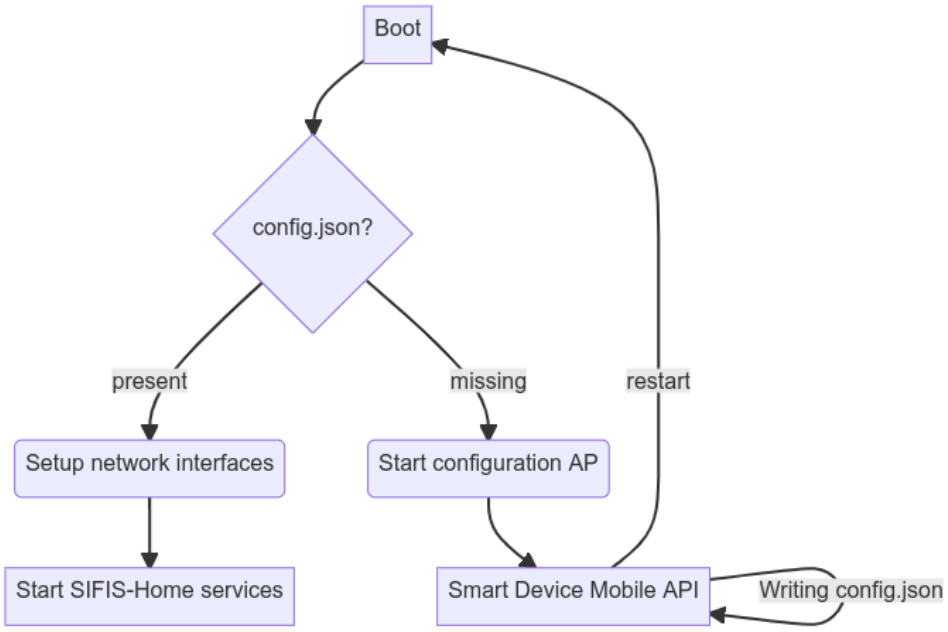


Figure 13: Smart device boot

SIFIS-Home Targets

Systemd is the most common solution for managing services on Linux systems. Figure 14 reports a simplified graph of default boot targets and services of the *multi-user* systemd target. The multi-user target has everything running except the graphical user interface. This is the target we used for testing the Mobile Application API component on the Raspberry PI 4 device.

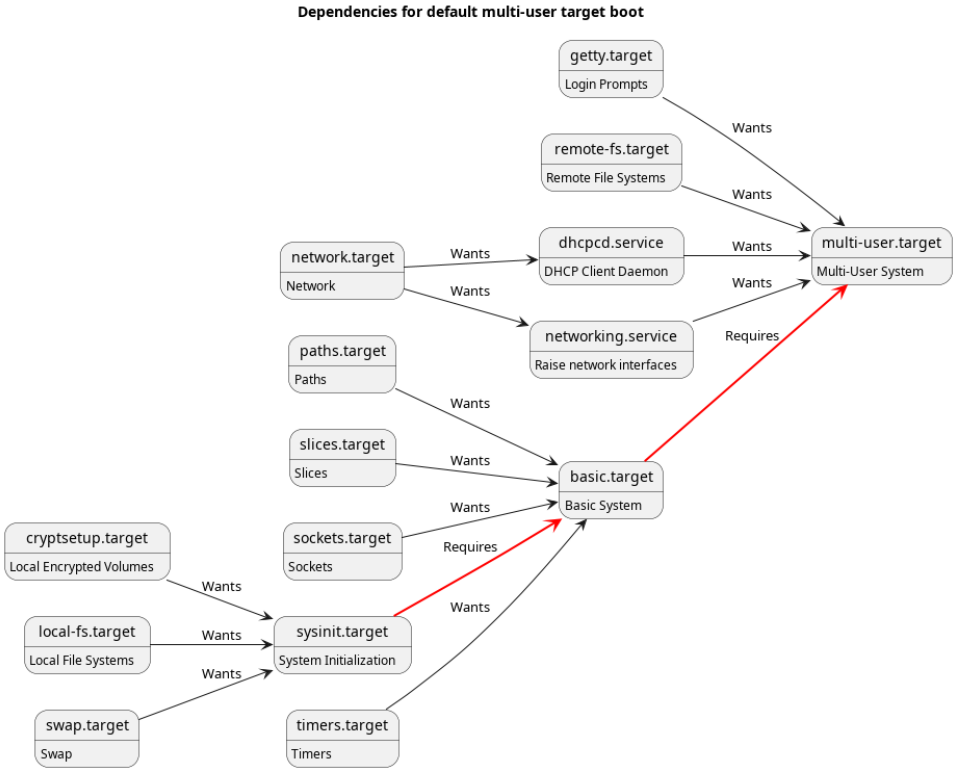


Figure 14: multi-user systemd target

For testing the SIFIS-Home Mobile API component we created two new systemd targets. The

selection of which target is active is based on whether the config.json file is or not present on the device file system. In detail, systemd services can be set up to be wanted by one of the targets to decide whether they are run at boot. Services that are only needed for configuration are installed under sifis-config.target, and services for the fully configured system are installed under sifis-home.target. Figure 15 below shows added targets with their conditions.

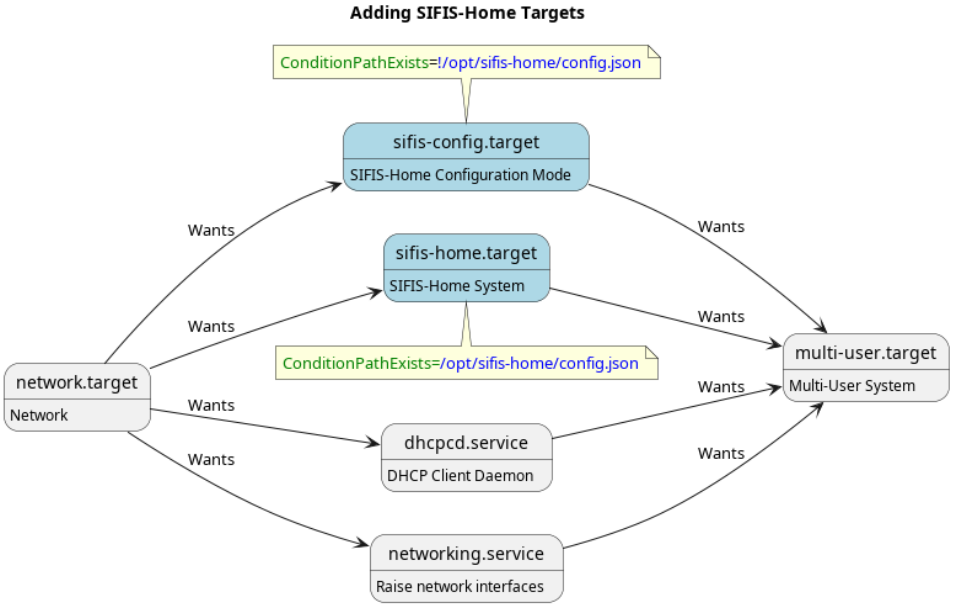


Figure 15: SIFIS-HOME systemd targets

Please note that some additional targets and services are left out of the picture for the sake of clarity.

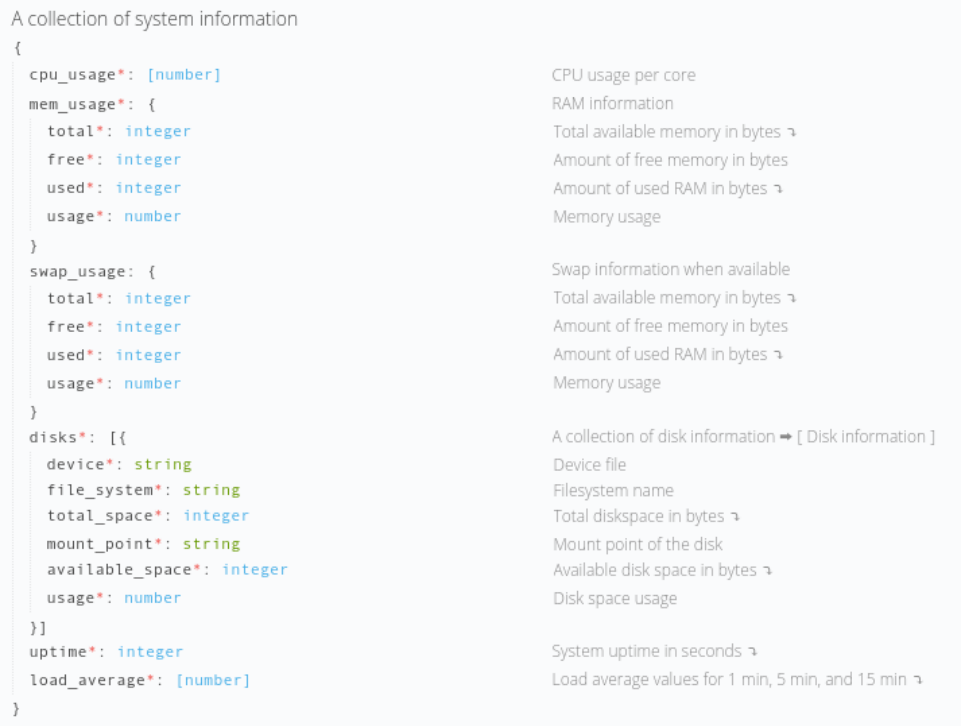
Smart Device Mobile HTTP API

All the endpoints offered by the Smart Device Mobile API component require an API key to function. The API key is provided to the Smart Device Mobile API component using the x-api-key header in an HTTP query.

HTTP Method	Endpoint	Parameters	Description
GET	http://<ADDRESS>:<PORT>/v1/device/status	-	Returns device status in JSON format
GET	http://<ADDRESS>:<PORT>/v1/device/configuration	-	Return device configuration in JSON format or 404 if the device has not been configured yes
PUT	http://<ADDRESS>:<PORT>/v1/device/configuration	Device configuration in JSON	Sets device configuration and returns

		format	200 OK or error status
GET	http://<ADDRESS>:<PORT>/v1/command/factory_reset	The confirm message "I really want to perform a factory reset"	Removes device configuration file and runs the *factory_reset* script on the target system
GET	http://<ADDRESS>:<PORT>/v1/command/restart	-	Runs *restart* script on the target system
GET	http://<ADDRESS>:<PORT>/v1/command/shutdown	-	Runs *shutdown* script on the target system

The structure of the JSON messages that are provided and used by the Mobile API component is reported below.



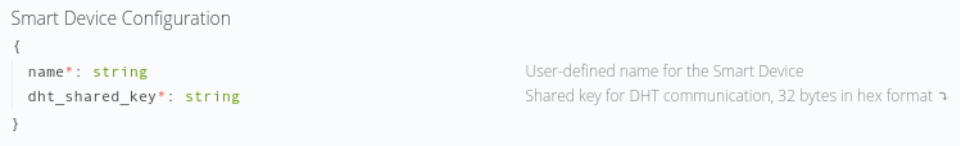


Figure 16: JSON messages stucture

SIFIS-HOME Mobile Application API code

The Mobile Application API component has been developed using the Rust language. The current source code can be found at *https://github.com/sifis-home/wp6_mobile_application_api*.

4.4 *DoMO GW OpenWRT distribution*

OpenWRT is a Linux distribution geared towards building firmware for routers. It is inherently focused on cross-compilation and is based on a GNUMake-based toolchain closely resembling the Linux Kernel build system.

Its source distribution is split in multiple git repositories, the `openwrt` one contains the core components and the toolchain machinery, additional `packages` repositories provide optional components.

Its main configuration files are `.config` and `feeds.conf`. The former matches the file with the same name used in the Linux Kernel, the latter resembles Debian's `/etc/apt/sources.list` in format and purpose.

The package layout is a directory containing a single Makefile implementing pre-named `defines` and variables that are then sourced by the main Makefile if its source tree is `installed` using the feeds management script. For each package present in the `installed` tree a metadata index is produced and used by the build system .config machinery.

The packages are built as installable packages (opkg) or built and installed in the base image depending on the `.config`.

In order to produce an openwrt image for the pilot the following are required:

- The OpenWRT core git tree
- A custom packages repository with packages for the custom software produced and the toolchain required to build them, in our case the Rust compiler is the only component missing.
- A package containing our custom configurations regarding the network and the boot process
- A `feeds.conf` pointing to our package repository.
- A `.config` targeting the hardware and adding to the base image the software

Additional care had been taken to not diverge from the distribution philosophy to reduce the odds of having clashing changes as the upstream distribution evolves:

- Every component providing a daemon has a procd-compliant initscript
- All the non-standard configuration is packaged as a uci-defaults script
- There is ongoing work on exposing all the daemon settings via uci.

4.5 *DoMO GW OpenWRT distribution setup scripts*

The whole SIFIS-HOME OpenWrt image creation is automated via a simple bash script. In detail, the bash script performs the following steps:

- It clones the openwrt github mirror
- It generates the feeds.conf to include the SIFIS-specific packages
- It populates the .config file using the settings extracted using the diffconfig script
- It builds the full image and the upgrade image

4.6 *DoMO GW upgrade procedure*

The standard OpenWRT system layout presents a production partition and a recovery partition, both read-only. The production mode mounts a read-write additional partition as overlay on top of the read-only production partition while the recovery mode simply boots its partition w/out mounting anything else.

The default OpenWrt upgrade process enables to save a tarball of the modified configuration files, kill all the services and pivot to a minimal ram-backed setup to overwrite the production partition, reset the overlay and optionally reload the configuration files.

Since there is a window of more than 10 seconds in which a power loss can cause the system to become unbootable, we developed a fallback process without that flaw. We briefly describe it below.

The process relies on the OpenWrt preinit system and diverges from the default by first storing the upgrade image and the modified configuration files tarball in a separate partition dedicated to store upgrade images. Then, the partition is unmounted and the system is forced to enter the recovery mode.

From the recovery mode the upgrade system detects if there is an upgrade image available in the dedicated upgrade partition and, in case, feeds it to the standard update system to update the production partition from the recovery mode.

Currently, the standard OpenWrt upgrade procedure cannot restore the configuration files if launched in recovery mode. Hence, our workaround is to have a preinit script that restores the configuration files from production and deletes the update files only once the boot process completes.

The event of a power outage would cause the process to restart from the last successful stage and complete once the power is back.

4.7 *DoMO GW flashing procedure*

The Banana-pi R3 SoC sports a 128MB NAND and a 32MB NOR memory in mutually exclusive access and a SD port sharing the I/O pins with the internal 8GB eMMC storage.

The OpenWRT build system generates images that can target all the possible I/O. The SD image can flash the NAND and the NOR with an image, directly from u-boot. The NAND image can flash the eMMC from u-boot; the NOR image cannot, due to the constrained memory available.

The manufacturer’s suggested procedure to flash the system is to boot from the SD with a modern image, flash the NAND storage, reboot from the NAND and flash the eMMC.

The alternative process leverages the u-boot tftp support to directly flash from network. Beside the

requirement of having a dedicated network configuration it has the potential to be completely unattended.

5 DoMO WiFi actuators firmware

We decided that the NSSD devices, being part of the pilot, need to expose a Web of Things compliant API. In detail, in a WoT-based architecture every NSSD is a server that exposes a set of functionalities to possible clients. Web of Things does not mandate the use of a specific protocol to make the functionalities of a WebThing accessible to an external application. In our implementation, we decided that our NSSDs are HTTPS servers exposing their functionalities through a WebSocket API. In a WoT server *properties* are used to expose settings and characteristics of a WebThing. For example, we can have a property *description* that is a textual description of a certain WebThing (e.g. “kitchen light”). In addition, *actions* are used to request the execution of a certain operation to a WebThing. A possible action to allow a user to turn on and off a certain light can be, for example, *turn*. Web of Things also provides *events* to allow a WebThing to signal anomalous conditions. Our implementation only uses WoT properties and actions.

5.1 *Firmware implementation and structure*

The WoT firmware for the NSSD devices has been developed using the C++ language and the Arduino ESP8266 Framework. Its code is available on GitHub (https://github.com/sifis-home/domo-wot-actuator). We also used PlatformIO (https://platformio.org/) to simplify the firmware development and building processes. All the different WiFi actuators that we use in the pilot share the same code base. This allows to reduce code repetition and speeds up testing operations.

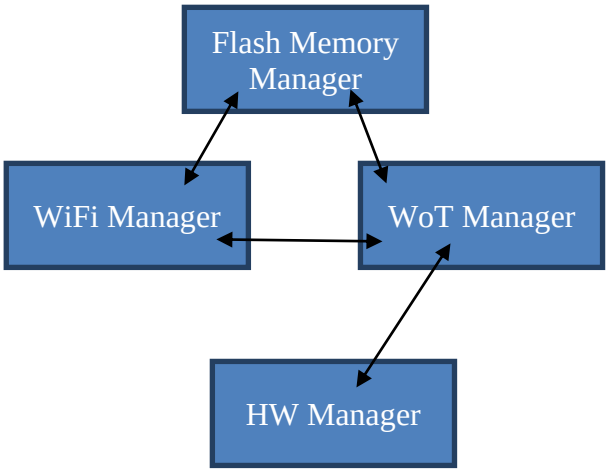


Figure 17: WoT firmware modules

In detail, our WoT firmware is composed of 4 different modules (Figure 17):

- **WiFi Manager:** it is the module responsible for managing the WiFi connection of the actuator. It communicates with the Flash Memory Manager to get the WiFi SSID and Password of the network to which the WiFi actuators should connect to. Also, it signals to the WoT Manager events of connection/disconnection from the WiFi network. The default WiFi network to be used is specified in the firmware code.
- **Flash Memory Manager:** it is the module that is responsible for reading/writing data from/to the persistent memory of the actuator. It provides WiFi credentials to the WiFi Manager. Also, it provides the WoT Manager with the server certificate and credentials to be used by the

HTTPS WebSocket server needed to expose the WoT API (see below for additional details).

- **HW Manager:** it is the module responsible for managing the physical peripherals/devices of the actuators. It uses the ESP8266 GPIO pins to activate/deactivate the actuator relays and to get the current status of attached input devices such as bistable buttons. Also, it communicates with the energy monitoring chips to provide power/energy readings to the user.
- **WoT Manager:** it is the module responsible for creating a WoT compliant API for the WiFi actuators. In particular, the WoT Manager main task is to start and monitor an HTTPS WebServer with WebSocket support. Also, the WoT Manager is responsible for starting up an m-DNS resolver that allows the discovery of the actuator by part of the NSSD Manager. In our implementation every actuator is identified by the m-DNS name <*shelly_model*>-<*mac_address*>.*local*, where shelly_model identifies the particular actuator type (i.e. shelly1, shelly_1pm, etc) and mac_address is the MAC address of the actuator. The WoT manager is informed about network connection/disconnection events from the WiFi Manager. Also, it receives the security material needed to correctly start up the HTTPS Web Server from the Flash Memory Manager (see section Security for additional details). Finally, it communicates with the HW Manager module to activate/deactivate the physical relays, get energy/power readings and updates on the input channels states.

5.2 *WoT API: properties and actions*

As mentioned before, our WoT implementation uses both WoT *properties* and *actions.* They are detailed below.

Properties

We use a single property named *status*, of type String, to represent the current state of the actuator. In detail, the property *status* is the serialization of a JSON Object that contains a number of different fields. We report in Figure 18 a possible value for the *status* property for a shelly1 actuator and a description of the various fields.


```
{
"ap_mac_address":"9483c413a0d4",
"fw_version":"v1",
"gateway":"192.168.1.1",
"input1":false,
"ip_address":"192.168.1.26",
"mac_address":"98:cd:ac:2d:4c:35",
"mcu_temperature":94.01399994,
"mode":0,
"output1":false,
"rssi":-61,
"topic_name":"shelly_1",
"wifi_ssid":"****"
}
```


Figure 18: example of WoT status property

Field name	Description
ap_mac_address	MAC address of the WiFi Access Point to which the actuator is currently connected

fw_version	Firmware version
Gateway	IP address of the WiFi actuator gateway
input1	State of input channel 1
ip_address	IP address of the WiFi actuator
mac_address	MAC address of the actuator
mcu_temperature	temperature of the MCU
Mode	Current operation mode
output1	State of output channel 1
Rssi	RSSI signal level
topic_name	topic_name of the persistent message used to store the status of the actuator inside the DHT
wifi_ssid	SSID of the WiFi network to which the actuator is connected to

Please note that the various fields of the status property are updated over time. For example, in case the relay number 1 of the actuator is activated the output1 field value changes from false to true. A WebSocket client connected to the actuator receives a *PropertyStatusUpdate* message whenever the status property value changes.

Action

We use an action named *shelly_action*, of type Object, to allow an external application to request the execution of specific operations to the actuators. The shelly_action contains two mandatory fields: *action_name* and *action_payload*. The former is used to identify the specific type of action that must be executed by the actuator. The latter contains parameters for the action execution. Our implementation currently provides the actions reported in the table below.

action_name	action_payload	Description
set_output	output_number: number of the relay to be activated/deactivated desired_state: desired state of the relay	Action that allows to activate/deactivate output relays.
set_dimmer	dim_value: desired dimming level	Action that allows to request a dimming operation.
pulse_action	output_number: relay to use for the pulse operation duration: duration of the pulse signal in ms	Action that allows to request a pulse operation using an output relay.
set_shutter	desired_state: OPEN, CLOSED, STOPPED	Action that allows to open/close/stop a roller shutter.
set_rgbw	Rgbw_value: desired rgbw value	Action that allows to set RGBW values .
set_led_dimmer	output_number: output channel to be used, dim_value: desired dimming level	Action that allows changing the dimming values of LED lights.
change_wifi	wifi_ssid, wifi_password	Action that allows changing the

		WiFi network to which the actuator should connect to.
change_mode	mode	Action that allows changing the actuator operation mode (i.e. RELAY mode or SHUTTER mode).
update_action	fw_url	Action that allows updating the firmware of the actuator.

A WebSocket client can request the execution of a specific action by sending a specific *ActionRequest* message.

5.3 *Security*

As mentioned above, the WoT Manager module takes care to expose a Web of Things compliant API that can be used by external applications to access the functionalities of the WiFi actuators. Communication between WebSocket clients and the WoT-enabled actuator are encrypted and protected. In particular, only allowed users/applications are able to communicate with the actuator and request the execution of specific actions. To this end, the WoT manager uses an HTTPS server with WebSocket support. The server certificate and server key to be used are generated during the actuator flashing phase and stored on the flash memory of the actuator (see section below). In addition, every WebSocket client should provide a user/password pair in order to access the WebSocket server functionalities. In detail, every WebSocket client should use the HTTP basic access authentication to send its username and password when making a request to the actuator. In basic HTTP authentication, the request contains a header field in the form of Authorization: Basic <credentials>, where credentials is the Base64 encoding of the username and password joined by a single colon. Our implementation uses a dedicated user/password pair for every actuator. They are generated and stored on the flash memory of the actuator during the actuator flashing phase. In this way, we provide encrypted communication and can guarantee that only allowed applications have access to the WiFi actuators functionalities.

5.4 *Flashing procedure*

The flashing procedure is the operation through which we install the WoT firmware on our WiFi actuators and provide them with the needed security material. We developed a flashing tool to ease the actuator flashing operation. Before starting the flashing procedure, the actuators should be put in programming mode and connected to a PC where the flashing tool is executed.

The flashing tool follows a number of steps that are detailed in the following:

1) The user selects the model (*actuator_model*) of the actuator to be flashed (i.e shelly1, shelly1pm, etc). The corresponding firmware is selected.
2) The MAC address (*mac_address*) of the actuator to be flashed is retrieved.
3) The security material to be used by the actuator is generated. In detail, a random user/password pair is generated. Also, a server private/public key pair and a server certificate with CN field equal to "*actuator_model-mac_address.*local" is created. The server certificate is signed using the SIFIS-HOME Certification Authority key. Please note that the SIFIS-HOME Certification Authority certificate is contained in the SIFIS-HOME OpenWrt distribution used by the DoMO Gateways. It is stored in the trust store of the DoMO gateways by installing a dedicated OpenWrt package that we created. Please note that we are currently assuming that the

device/PC used to flash the WiFi actuators is trusted. In detail, we are assuming that that the CA private key used to sign the actuator certificates has been provided and saved on the flashing device/PC using a secure channel. In a production environment, the use of an intermediate CA that is only used to sign the actuator certificates is recommended.

4) The flash memory of the actuators is completely erased.
5) The security material (serverKey, serverCert, user/password) is stored on the flash memory of the actuator (SPIFFS partition).
6) The WoT firmware is installed.

The user/password pair and the MAC address of the flashed actuator are saved in a local text file. Please note that the NSSD Manager needs to know the user/password pair used by a specific actuator for being able to connect to it. Hence, they need to be inserted in the DHT of the SIFIS-HOME house where the actuator will be installed. Currently, the credentials are stored inside the DHT using a Web-based control panel that we created (see below).

5.5 *WoT firmware operations*

The following figure reports the operations that are performed by our WoT-enabled actuators. When the actuator is turned on, the Flash Memory and the HW peripherals are initialized. Then, the WiFi Manager requests the WiFi network SSID/Password to use to the Flash Memory Manager. Then, the WiFi connection is activated and the HTTPS WebSocket WoT server is started by getting the security material from the flash memory. Also, the m-DNS resolver is activated. At this point, Websocket clients, such as the NSSD Manager, can connect to the actuator to get property updates and send action requests.
The actuator continuously waits for i) updates from the physical peripherals that produce an update of the WoT *status* property, ii) requests to execute a specific *shelly_action* by part of a WebSocket client that, in general, cause the WoT Manager to communicate with the Hardware Manager to start operations involving the physical devices.

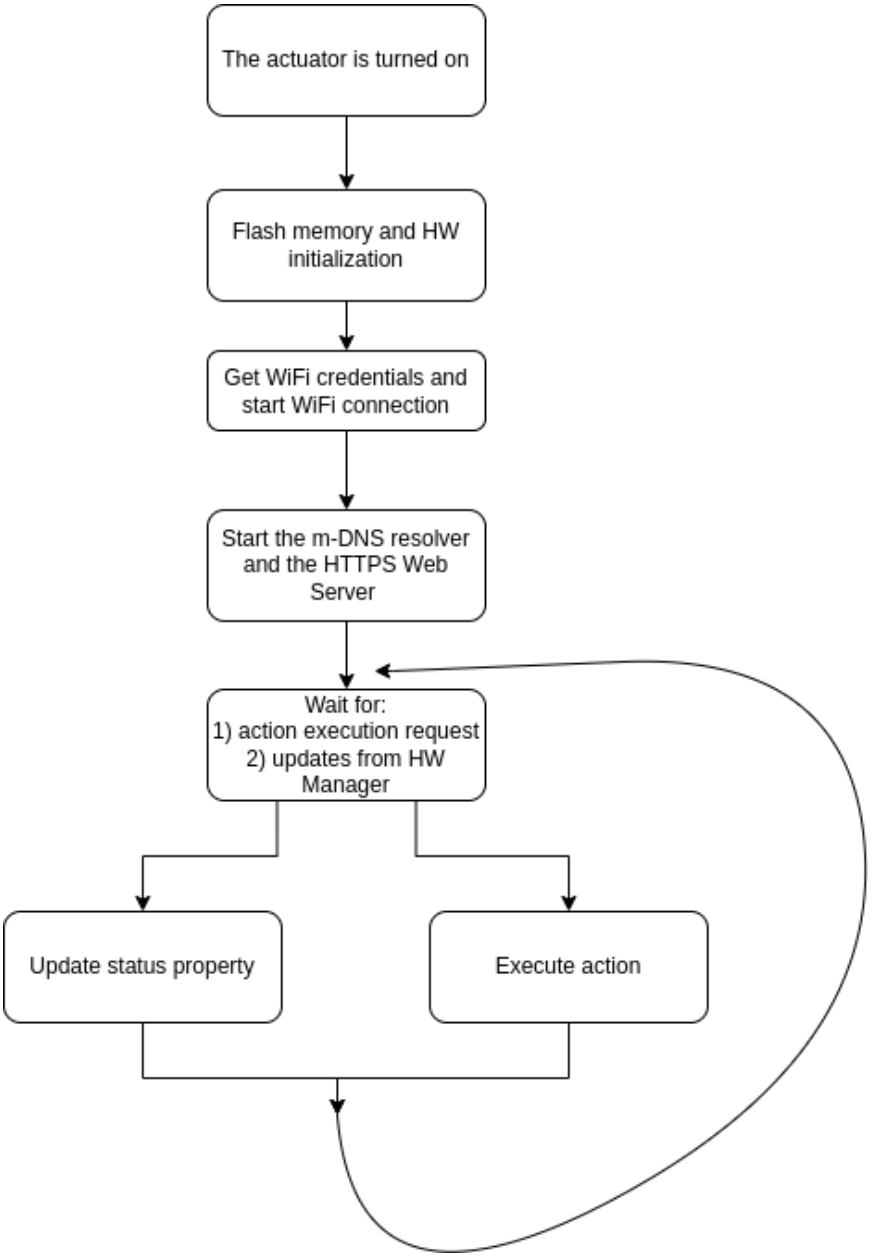


Figure 19: WoT firmware operations

6 Smart home use case workflow

In this section we focus on a specific smart home use case, i.e., allowing a user to control its lights by using a web-based control panel. Our goal is to show the interaction and the communication between the software components and the devices used in our pilot. We assume that a number of DoMO gateways and WiFi Actuators have been prepared and flashed according to the procedures described before and have been turned on. The actuators have also been connected to the lights they need to control. Hence, our environment is similar to the one reported in Figure 9. At the very beginning, an instance of the DHT Manager and NSSD Manager is present and operational on every DoMO gateway. Also, the WiFi actuators are connected to the network offered by the DoMO gateway and are waiting for WebSocket client connections. The DHT is also empty since no messages have been published yet. The user will use a Web-based control panel in execution on its PC to control its lights. In detail, we developed a Web application using Vue.js that allows the users of a SIFIS-HOME house

to control their devices. The Web application uses the WebSocket and the REST API of the SIFIS-HOME DHT Manager to interact with the system. The following picture shows all the involved components and devices.

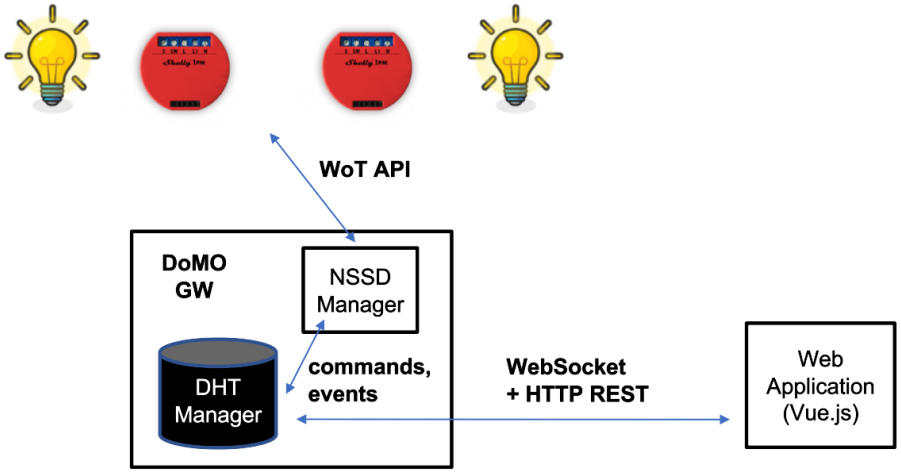


Figure 20: Smart home use case

As mentioned before, the NSSD Managers in execution on the DoMO gateways should be provided with the user/password pairs of the actuators they are intended to control. In detail, the security credentials to be used to connect to every WiFi actuator should be inserted into the DHT. We created a specific control panel in our Web application to allow a user to add a new WiFi actuator to its SIFIS-HOME house.

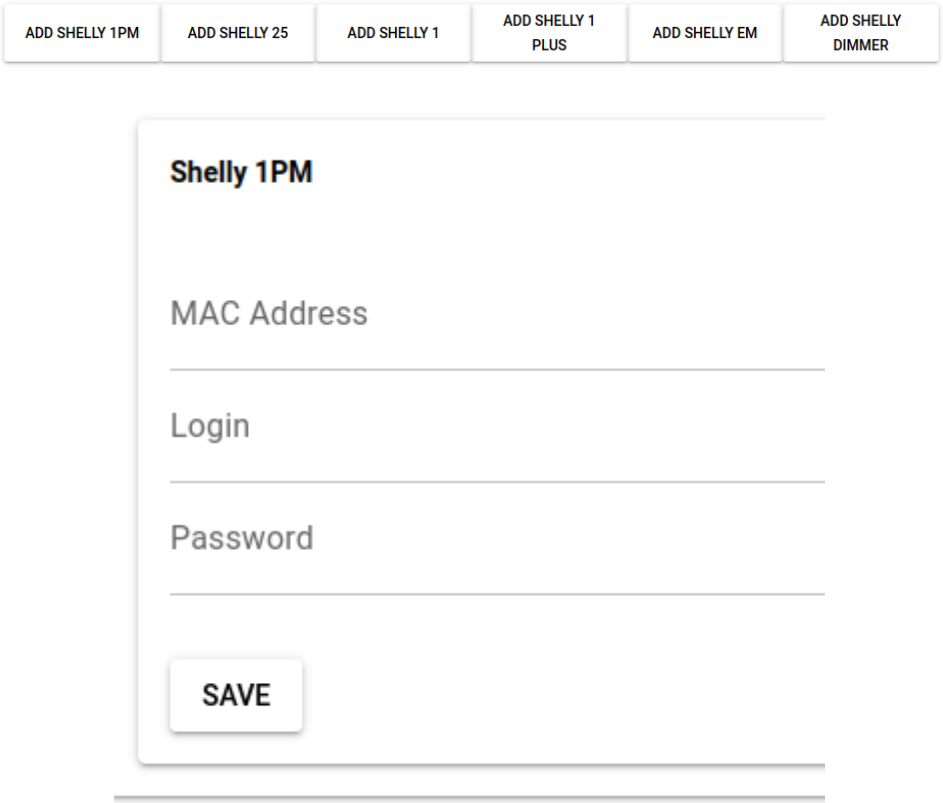


Figure 21: Add Shelly 1PM actuator to the DHT

Figure 21 shows the panel. As it can be observed the user should select the type of actuator to add and insert information such as its MAC address and user/password pair. The Vue application will use the DHT REST API to publish a persistent message with the provided information. Please note that the message is distributed by the DHT and available on every DoMO gateway.

Let us focus now on the actuator that has just been added to the DHT. It is connected to the network advertised by one of the DoMO gateways and it is operational, i.e. it executes a Web server providing a WoT API, whose access is protected by means of the user/password pair that has been provided to the actuator during flashing (see also Figure 22).

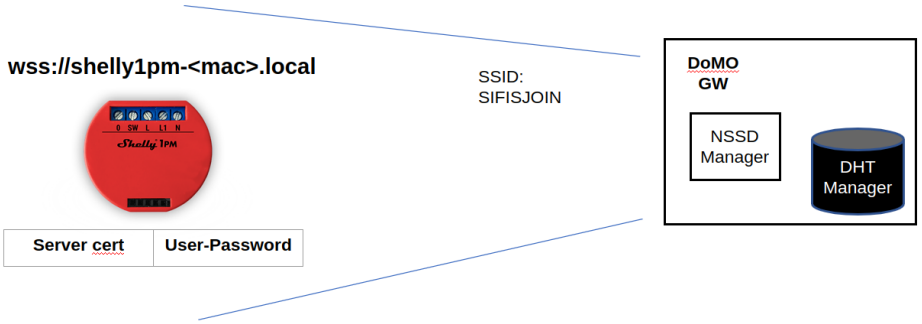


Figure 22: WiFi actuator connection to a DoMO gateway

The presence of the actuator on the network advertised by the DoMO gateway is discovered by means of the m-DNS protocol. Since the actuator is now registered on the DHT, a WebSocket connection to it is started by the NSSD Manager using the credentials stored in the DHT (see Figure 23).

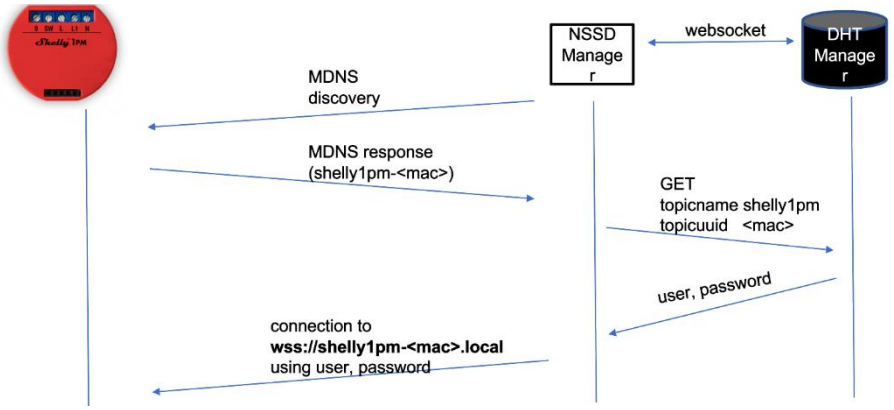


Figure 23: WiFi actuator discovery and connection

The user can now use the Web Application to send commands to the actuator, e.g. to turn on the light that is connected to it. The Web Application will use the DHT REST API to publish a volatile message that contains the command to be executed. The DHT Manager will forward the command message to all the NSSD Managers in execution on the various DoMO Gateways of the house. When the message is received by the DoMO gateway to which the destination actuator is connected to, the NSSD Manager in execution on the gateway, requests the execution of the WoT action corresponding to the requested command using the actuator WoT API. When the light is actually turned on, a Property update is received by the NSSD Manager. Then, the NSSD Manager updates the status of the actuator (the relay is now activated) on the DHT. The update is also received by the Vue application that can show the new updated status to the user.

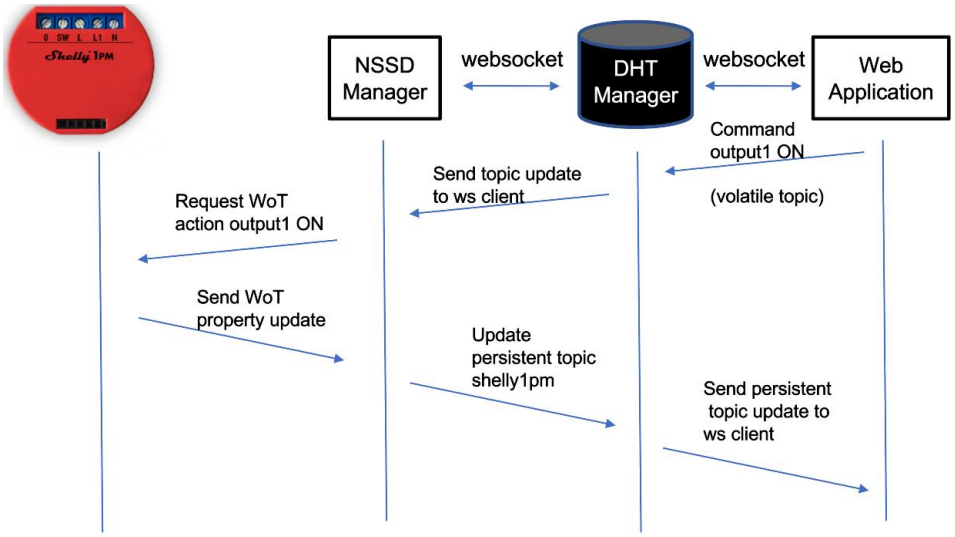


Figure 24: command execution workflow

7 Validation strategy

In deliverable D6.1 we defined the different smart home use cases to be demonstrated through our pilot implementation. For each one of them, we reported the functional, performance and security requirements. Also, an updated list of workflows has been reported in deliverable D1.4. Some of them (e.g. workflow 6.2 and 6.3) have a direct mapping with the use cases defined in D6.1. We not only need a validation strategy that verifies that our pilot implementation satisfies all the requirements of the different defined use cases but also a clear procedure to test the quality of the developed software components. In the following we detail our validation strategy reporting both the procedure to test the software component quality and the steps to verify that the use cases requirements are met by our implementation.

7.1 *Testing software component quality*

Every time the source code of a WP6 software component is uploaded on the SIFIS-HOME Github organization, a series of quality checks provided by the WP2 partners is applied to it. In detail, for every developed component it is required that both a README file and a LICENSE file are present in the component repository. Also, it is recommended that unit tests are present and that an automated procedure (or a Dockerfile) to easily build the component is provided. In addition, when possible, the *sifis-generate* tool is used to easily setup a Continuous Integration procedure. In detail, the *sifis-generate* tool automatically generates a series of GitHub actions that allow to calculate metrics such as the code coverage or signal the presence of known security issues in the project dependencies. Also, it is checked that code warnings are not produced by the build procedure. Finally, if possible, an additional manual code review is performed to improve code quality and signal possible problems.

7.2 *Validate the smart home use cases*

The validation strategy for the use cases will abide to the strategy defined by WP1 and WP5 activities. In fact, also for the use cases validation, we will exploit a General Quality Management (GQM) approach. The use cases validation will be fully addressed in D6.3.

7.3 *NSSD WoT API testing*

The WP6 smart home use cases require to interact and communicate with physical devices. As mentioned above, communication with the NSSD devices of the pilot is performed by means of a WebThing API. Given that it is of paramount importance to verify the correctness of the WoT API offered by the NSSD devices, the following procedure has been defined to test it. In detail, for every NSSD offering a WoT API, a wot-test program will be developed to automatically perform the below actions and verify that:

- Every WoT property that is read/write must be set to know values and its value retrieved to confirm

- Every WoT property that is read-only must fail an attempt to write

- Every physical sensor that can be influenced (e.g. contact sensors) should be manually manipulated and their status change recorded and then manually verified

- Every WoT action should be invoked and their effects evaluated (polled in WoT 1.1) until completion.

If the wot-test program reports successful tests, the following integration test is also performed for every WoT enabled NSSD. This is to verify that it is possible to successfully interact with the NSSD using the DHT layer. In detail, the test verifies that every NSSD controlled via the DHT layer has the same behaviour as when directly controlled via the WoT API, that is:

- Every WoT property mapped to a DHT topic must be confirmed to be written if its value changes

- Every action triggered via a DHT command should result in the expected outcome

The wot-test and wot-consume repositories contain the proof-of-concept of the testing programs mentioned above.

7.4 *Use cases testing*

The activities of WP6 are ongoing and implementation tasks have not finished yet. However, D6.1 use case requirements are periodically checked to verify that our final pilot implementation will be able to satisfy them. In detail, we plan to follow the below procedure to produce a final detailed testing report.

For every reported use case in D6.1 we are going to verify that:

1) The pilot implementation is able to demonstrate it (answer yes/no)
2) The performance requirements are met (e.g. the maximum latency to turn on a light is above the actual experimented latency to turn on a light in our testbed) (answer yes/no, deviation)
3) The security requirements of the use case are satisfied by the implementation (if not, specify what requirements are not met)
4) The use case implementation is compliant with the GDPR guidelines and personal data is handled correctly (if not, a clear explanation will be provided)

8 Next actions

In the next months a number of actions need to be performed to be able to successfully demonstrate the smart home use cases reported in D6.1. We report the main ones below:

- WP4 Analytics Integration: we need to integrate the analytics developed in WP4 in our pilot. To this end, we need to produce OpenWrt packages to include in our SIFIS-HOME OpenWrt distribution.
- Policy Manager Integration: the Policy Manager component needs to be integrated in our pilot. This will allow the definition and monitoring of smart home policies. We plan to include the Policy Manager by creating an OpenWrt package for our custom OpenWrt image.
- Remote Control: we are going to integrate the Fiware API component in our pilot so that it will be possible to control a SIFIS-HOME enabled house by using the Fiware APIs provided by Yggio.
- Third-party applications: we need to develop the Application Manager component that will allow the installation of third-party applications on the smart devices being part of our pilot.
- Testing: we need to test the quality of the developed applications and the performance of our system and report the obtained results.

Appendix A: List of Code Components

Code Component	**GitHub Public Link**
DHT Manager	*https://github.com/sifis-home/libp2p-rust-dht*
NSSD Manager	*https://github.com/sifis-home/domo-wot-bridge*
WoT firmware for DoMO WiFi actuators	*https://github.com/sifis-home/domo-wot-actuator*
Smart Device Mobile API	*https://github.com/sifis-home/wp6_mobile_application_api*

Appendix B: List of Acronyms

Acronym	Meaning
RGBW	Red Green Blue White Light
EMMC	Embedded MultiMediaCard
WoT	Web of Things
DHT	Distributed Hash Table
SD	Smart Device
NSSD	Not So Smart Device