POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Technical Report

The MPower Communication Protocol

Matteo Ferroni

ferroni@elet.polimi.it

Version 0.1 - Last update: N/A

NECST Laboratory

mpower.necst.it

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

API Application Programming Interface

FSA Finite State Automata

HAL Hardware Abstraction Layer

IMEI International Mobile Equipment Identity

IV Inizialitation Vector

JSON JavaScript Object Notation

TTL Time-To-Live

Chapter 1

Communication

Since MPower is a distributed system, a strong communication protocol has to be designed and implemented in order to allow communications between its components over the Internet.

The first authentication phase has been described in Section **??**, while describing the behavior of the AuthActivity component. Once the user and the device have been authenticated, the MPower app and the Meth server share a *secret token*, that has to be stored on both the sides. This token is used as a *symmetric key* for two main purposes:

- **encrypt** the log files, since these may contain sensitive data, i.e. hidden information about the user's habits. More details about this task will be given in Section 1.1.3;
- **authenticate** the device as soon as it starts communicating with the server, in order to confirm its identity.

We designed and implemented a custom protocol, thought to be secure with respect to malicious attacks (i.e., men-in-the-middle attacks) and to support new functionalities as soon as these will be developed. This is composed of two main parts: the first one, that is a preliminary and command-agnostic handshaking process, permits

the device identification and authentication. The second part is a command-specific phase.

Figure 1.1 shows the preliminary handshaking process. A comprehensive description is here given:

1. • **MPOWER:** sends a *#reset* message, to clear any pending sessions.

2. • **MPOWER:** sends a *#command* message, pointing out that it wants to begin an handshaking phase to execute a particular *command*.
   • **METH:** replies with another *#command* message, acknowledging the request and the specified command.

3. • **MPOWER:** sends the version of the protocol, *ver#2.0* in our case.
   • **METH:** replies with the same protocol version, if it's a know protocol. This has been designed to support future versions of the same command, guaranteeing support for older versions of the application.

4. • **MPOWER:** then sends a message containing its *device_id*, claiming to be the device identified by it.
   • **METH:** replies with the same *device_id*, adding a random 8 byte alphanumeric salt. This is done to prevent *reply attacks* from a malicious third party.

5. • **MPOWER:** encrypts its own device_id followed by the salt received from METH with AES-256 (CBC) using the secret token.
   • **METH:** extracts from the database the authentication token related to the *device_id* received before, decrypts the received message, parses the known *device_id* and isolates the salt, to check if its the same it just sent. If both checks are passed, it replies with an *auth#ok* message. All the messages are encrypted using the secret token from now on.

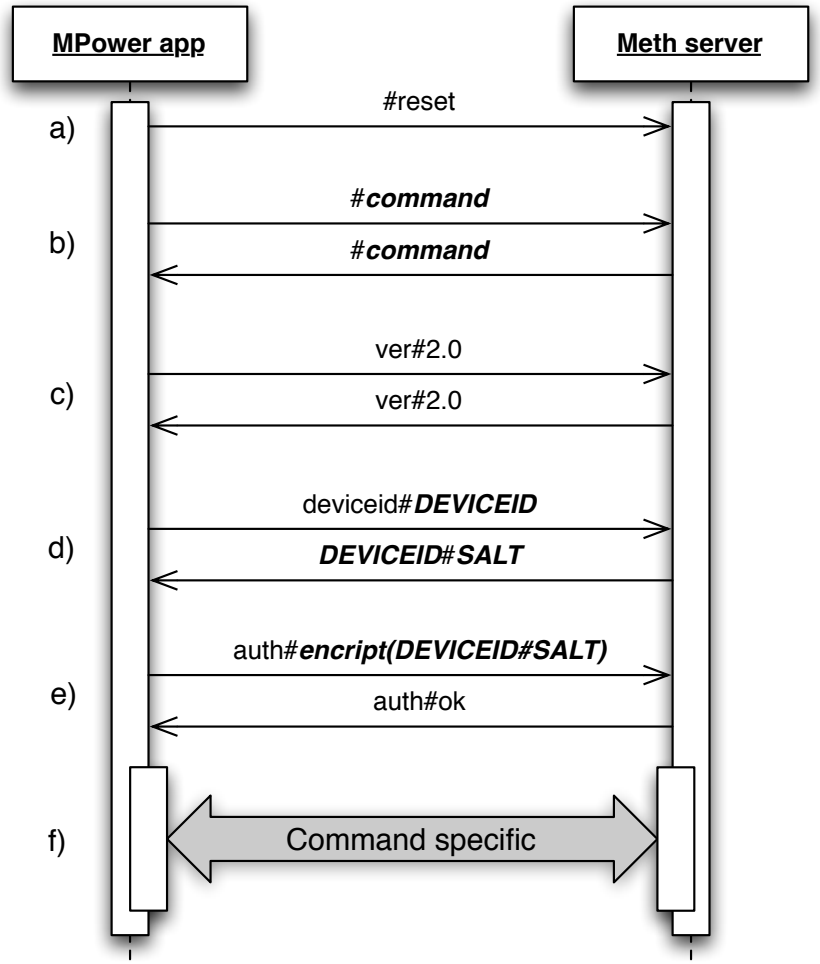

Figure 1.1: MPower communication protocol: preliminary and command-agnostic handshaking process

6. The command-specific phase of the protocol can now start.

At the time of writing, two commands are supported:

- A **send** command, that declares the intention of sending log files from the MPower app to the server. This is handled by the **DATASENDINGTASK** component.
- A **model** command, that declares the request of the MPower app for the Time-To-Lives (TTLs) computed for the device. This is handled by the **DATARETRIEVINGTASK** component.

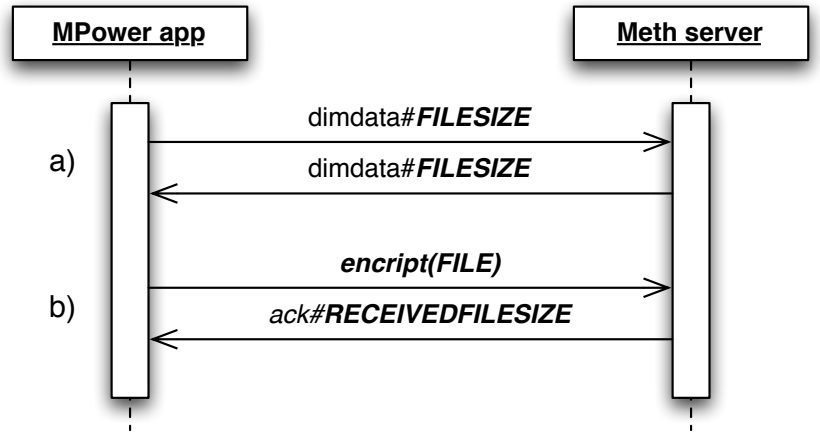

Figure 1.2: MPower communication protocol: file sending phase

DataSendingTask and DataRetrievingTask extends the `AsyncTask` Android class. This is meant to perform background operations, publishing results on the UI thread if needed: this is the case of the DataSendingTask, that can be launched by the user through the "Force data sending" button of the MainActivity.

DataSendingTask

A description of the command-specific phase of the protocol related to the *send* command is shown in Figure 1.2 and here detailed:

1. - **MPOWER:** MPOWER sends the size of the data it is about to send.
   - **METH:** stores the size to later compare it with the file size and replies with the same data size message. This is done in order to check the correctness of the file itself.
2. - **MPOWER:** finally sends the data and waits for the acknowledgement to complete the session.
   - **METH:** receives the data, check its size against the size declared before by the client, sanitize the file name, check the filesystem for duplicates and, at

last, store the data for further processing. An *#OK* message is finally sent to the MPower app.

DataRetrievingTask

A JavaScript Object Notation (JSON) object structure has been defined to be the representation used to send computed TTLs from the Meth server to the MPower app. An example is given in Listing 1.1: it is a set of *configuration* objects, each one containing hardware states information and a set of one hundred TTLs values, one for every battery level percentage. Again, to keep the file as lightweight as possibile, we decided to skim it of all the useless blank spaces and newlines.

```
[
        {"config_index": "000000",
        "content":
                {"airplane_mode": "0",
                "wifi_on": "0",
                "screen_brightness": "0",
                "gps_on": "0",
                "data_state": "0",
                "bluetooth_on": "0",
                "ttl":
                        ["1294",
                        "1282",
                        "1269",
                        ... ,
                        "38",
                        "26",
                        "13"]
                },
        },
        {"config_index": "000001",
        "content":
                {"airplane_mode": "0",
                ...
]
```

Listing 1.1: A small extract of a JSON TTL file

A detailed description of the command-specific phase of the protocol related to the *model* command is shown in Figure 1.3 and here detailed:

1. - **MPOWER:** sends a *#ttl* message, asking for the TTLs computed by the server.
   - **METH:** if a new model has been computed for the device, it encrypts the TTL JSON object with the secret token and sends it to the client. Otherwise, a *#nottl* message is sent back to the client and the communication stops.
2. - **MPOWER:** receives the data, checks its size and sends that value back to the server.
   - **METH:** compares the received value against the size of the data sent to the client, replying with an *#OK* message if these match.

The MPower app is then in charge of decrypting the string: the JSON object containing the TTLs is then passed to the DatabaseHelper component, that is in charge of storing them on the SQLite database. More details will be give in Section 1.1.3.

1.0.1 EventHandler

The EventHandler is a Broadcast Receiver component. It is created by the LogService, that registers it to listen for internal messages. Its main tasks are described in the following subsections.

Data events

The following messages are related to log files management tasks: a brief description for everyone of them is given in the following paragraphs.

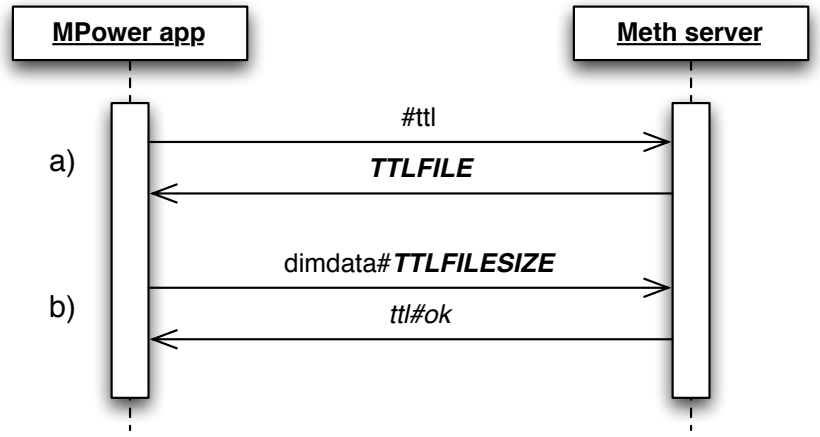

Figure 1.3: MPower communication protocol: TTLs retrieving phase

msg_check_file — We have to prevent that our log files become too big: otherwise, an excessive overhead may be experienced while accessing them. In order to do this, we decided to split the logged data in more than a file: a new log file is being created as soon as this message is received if *al least six hours* have passed from the previous split operation.

msg_plugged — File compression and encryption processes may be computationally expensive and thus power consuming. This is the reason why these operation are performed only when this message is received, pointing out that the device battery is charging: a wake lock is acquired and maintained for the whole length of these tasks. It is then released when all the log files have been processed. The actual implementation of the compression and the encryption operations is provided by the FileHelper component, as described in Section 1.1.

msg_wifi_connected — Mobile data traffic may have additional costs for the user: this is the reason why the standard policy of MPower is to send log data and ask for updated TTLs only when the device is connected to a Wi-Fi network. As soon as this

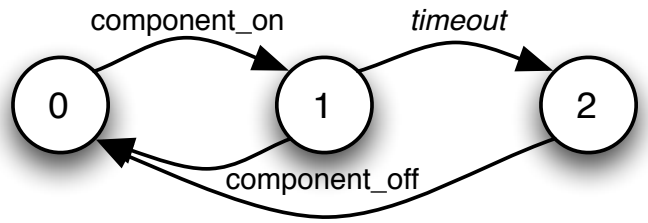

Figure 1.4: Notifications policy

happens, the DataSendingTask and the DataRetrievingTask are launched, in order to send the already compressed and encrypted files, and to retrieve an updated power model from the server (if this is available).

Notification events

MPower makes use of the Android notifications to warn the user as soon as an hardware component has been left on and it is not currently used.

The notification policy can be easily explained using a Finite State Automata (FSA) as the one in Figure 1.4. Here is a brief explanation:

1. In state 0, the hardware *component* is OFF or is ON and it is currently used by at least one Android application.

2. A transition from the state 0 to the state 1 is performed as soon as a "*component*_on" message is received: the component is then ON but it is not used. A timeout is here set.

3. A transition from the state 1 to the state 2 is performed as soon as the timeout has passed: a notification is then shown to the user, in order to warn him that he can save battery power by disabling the unused hardware component.

4. Transitions from states 1 or 2 back to the state 0 are performed as soon as a "*com-*

*ponent*_off" message is received: the component is then OFF.

This logic has been developed for every considered hardware module: the current implementations handle notifications for the GPS, the Bluetooth and the Wi-Fi modules.