# Wireless smart monitoring energy system: a proof of concept

Marco Costantino
*Department of Mathematics*
*University of Padova*
Padova, Italy
marco.costantino@studenti.unipd.it

Francesco Magarotto
*Department of Mathematics*
*University of Padova*
Padova, Italy
francesco.magarotto@studenti.unipd.it

***Abstract*—This document describes a prototype of domotic smart plugs. The system developed includes the smart plugs, a server, and a web interface that allow the user to monitor the power consumption of each individual appliance connected and power them on and off remotely. Furthermore, the system, provided that its settings are correct, keeps the user from turning on enough appliances to trigger the circuit breaker. The prototype has been developed keeping in mind issues of power consumption and reliability of communication between components of the system while keeping the traffic light.**

***Index Terms*—Domotic plugs; Reliable UDP;**

## I. INTRODUCTION

Nowadays, keeping track of domestic energy usage is a common scenario in domotic applications that aim to monitor the real-time watt consumption of home appliances. In most cases, the solution is a sensor connected to the main safety switch yielding the total energy consumption. Our approach, to get more detailed information and to have control over the use of power by the appliances, is to have an energy sensor for each plug connected to the 220V line. In the following paragraphs, we present our *proof of concept* (POC) which, functionally, consists of three main components: the smart plugs, the server, and the web interface. In our POC, we have modified the architecture described in [1] obtaining a specific low-energy consumption-driven infrastructure. This consists of the smart plugs, a computer (for the server), and a device (for the web interface). In a home scenario, these devices are connected to the home wireless router. The communication between smart plugs and server occurs over UDP with a simple form of reliability implemented at the application level and tested using Wireshark [2]. The architecture implemented differs from the most common ones in two major aspects: first, most homes aren't equipped with this kind of system, instead, a circuit breaker triggers for safety reasons when too much power is being consumed. Second, when a similar system is used it usually relies on cloud computing or external services. In contrast, our system offers the functionalities already described and both computing and data storage are performed locally. This can be an advantage in terms of both privacy and security. Furthermore, this is an advantage in terms of reliability: our system doesn't need an internet connection to work properly, instead, it just needs a working WLAN. However, while this aspect is significant, the development of the prototype hasn't focused on the security aspects and some issues will be discussed in the appropriate section. The architecture of the system is such that if need be it can be easily modified to use third-party services. The smart plugs have been prototyped using Arduino and the server has been written in Java. Let's now consider a use case example: the user sets the maximum power consumption possible in the home and then turns on two appliances. When this happens for the first time the system decides, based on the type of appliance connected, a default value for the maximum power usage of the appliance. This value is then updated to reflect the real maximum power usage of the appliance. Now that the system is running and some appliances are turned on, the user will be able to switch on only the appliances that consume less than the available power. Our POC covers this use case and shows that such a system can be practical and useful and that the traffic generated isn't enough to sensibly worsen the performances of the typical home WLAN. To summarize, the system keeps the power usage from reaching the maximum limit set by the user by forbidding the power-on of appliances that consume more than the available power. It also gives granular information about the energy consumption of the device and allows the user to turn on or off appliances remotely (via web interface). Aside from the main use case, in the age of climate awareness, such a system could be interesting to an environmentally conscious user that can monitor the power usage of appliances but also could set the maximum power consumption to be lower than the real one to use less power or save money.

## II. GENERAL ARCHITECTURE

The general architecture has three main components: the smart plugs, the server, and the web interface. The smart plugs have been prototyped using Arduino. The server has been developed in Java and the web interface is written in React. The rundown of what each part does is as follows: Arduino drives a sensor that gathers data on the power consumption of the plug and communicates with the UDP server that keeps track of the plugs and their energy consumption. The web interface uses an API exposed by the server to let the user see and control the status of each plug.


Fig. 1. General architecture of the system

### A. Smart plugs

The smart plug is a small power plug which consists of a WeMos D1 R2 board, a 220v relay that can be controlled using 5 volts digital signals, and a coil connected to a PZEM-004T sensor which provides the readings to WeMos through its digital pins. The connection scheme is represented in Figure 2, and has a cost about €15, so it is not so expensive. The WeMos D1 R2 board is equipped with a EPS 8266 wireless module operating at 2.4GHz. The connection settings are hardcoded in the software because storing it would require an SD module and an SD card. The board is programmed in C/C++ using Arduino IDE and uses the JSON format to serialize the data that will be sent, using a UDP library, to the server. In every Arduino-like board, there is a function which allows our program to wait for UDP package arrivals from the server, affecting the relay status. Another thread, once the board has recognized and registered the server, sends updates periodically if there is a variation greater than the 10% of the energy consumption. First, smart plugs wait for an *INIT message*, sent by the server via broadcast. Once the packet has been received, each plug saves the information about the server and is not replying to this type of messages for a certain amount of type, or until a hard reset is performed. Then, the plug starts to send *UPDATE messages* to inform the server about energy consumption variation. To do so, a thread keeps watch over the readings provided by the PZEM sensor. In order to switch correctly the relay, the board waits for *ON/OFF messages* from the server. After those requests have been received correctly, the correct action is performed and then an *ACK message* is sent to inform the server about the successful receipt of the command. We also used a different board called NodeMcu V3 in order to simulate a plug to get a real traffic scenario with at least two plugs. Since, NodeMcu V3 is not connected to a PZEM module, the power consumption readings are random values.

### B. Server

The server has been developed in Java 8 and managed using Maven. Tomcat is implemented to expose the servlet API. The server has the following responsibilities: to periodically broadcasts *INIT messages* to let the plugs know its address; It listens for connections and updates from the plugs; it waits for acknowledgments from the plugs and, in case they are not received, it retransmits the lost packets; it exposes servlets to let the web interface get information about the plugs and send *ON/OFF messages*. The periodic actions, namely, sending the *INIT message* and retransmitting those that are lost, are implemented as extensions of *Guava's AbstractScheduledService* which allows performing of these at constant intervals. Another thread waits and manages the

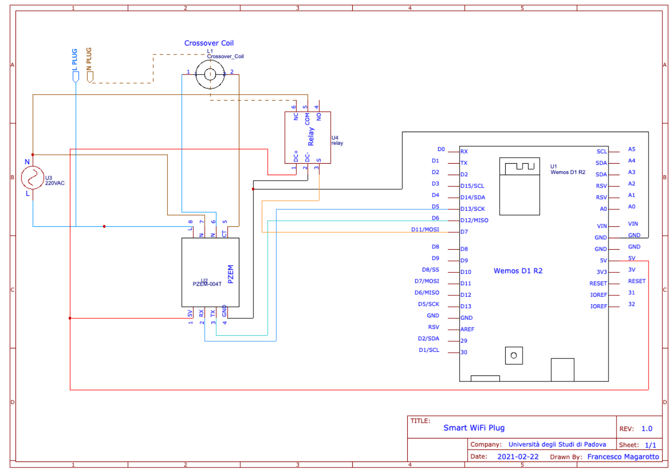


Fig. 2. Electric schema of the wireless smart plug

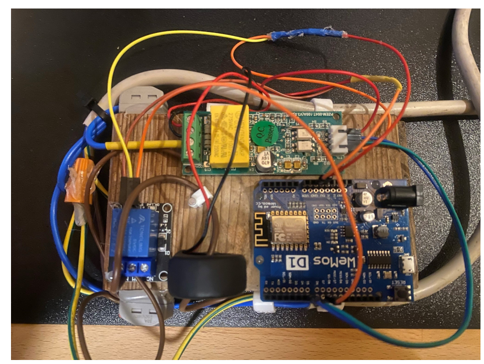

Fig. 3. Proof of concept of our smart plug

plugs' responses. When a plug responds to the *INIT message* the server will register its information and estimate the max power consumption for the appliance connected to the plug. This value is simply based on average values for the given appliance type as found on Google. When a plug sends an *UPDATE message* the server will update its registered information, most important: the current energy consumption and the max energy consumption. Finally, the server waits for the acknowledgments of previously sent messages. These messages are kept with a timestamp until an acknowledgment with the same timestamp is received. If a configurable number of seconds elapses before this happens the packet will be sent again. The servlets simply allow the web interface to get the server's information about the plugs (the type of appliance connected, the current energy consumption, and more) and to turn ON or OFF any given plug. In this last case, a packet will be sent by the server and for it the server will wait for an acknowledgment. The information about the plugs saved by the server is kept as a *thread-safe hashmap*. Some of this information, such as the IP address of the plugs, the type of appliance connected and the max power consumption is also saved in a *XML file* to avoid loosing it in case of a server malfunction or restart.

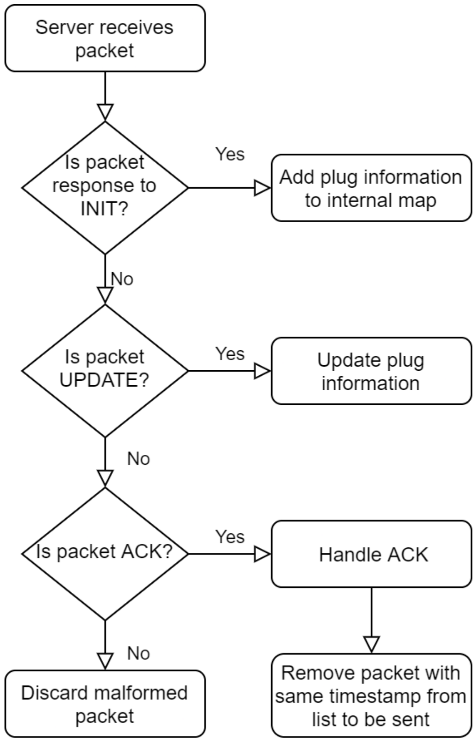


Fig. 4. Overview of server's logic for handling packets received by the plugs

### C. Web interface

The web interface is written in React using Bootstrap and exposes a way to interact with the server to get the real-time power consumption for each plug and managing the connected devices. It performs some to HTTP, so over TCP, to the API exposed by the server through the servlet, performing polling calls. The GUI shown in Figure 5 presents a switch where the user can turn on or off the plug. If there is not enough energy available to turn on a device, the switch will be displayed as disabled in order to avoid a blackout.

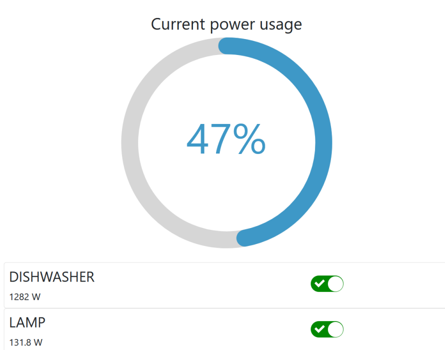


Fig. 5. Web interface used to control the whole system

## III. Methods

Throughout the development, the communication between server and the plug has been simulated and tested using PacketSender which is a tool that allow to send a configurable packet through the network. To check the system's traffic and the correct handling of acknowledgments, we used PacketSender to simulate the traffic generated by the plug but not the acknowledgments and Wireshark to filter the traffic by port and type and check the correct retransmition of packets by the server. Finally the entire system as been tested to check the correct transmission of acknowledgments by the plugs.

## IV. System results

## V. Consumption

## References

[1] Y. Tong and Z. Li, "Design of intelligent socket based on wifi," in *2017 4th International Conference on Information Science and Control Engineering (ICISCE)*, 2017, pp. 952–955.
[2] [Online]. Available: https://www.wireshark.org/docs/