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 or a Raspberry Pi running the server, and the control devices which interact with the system through the web interface. In a household 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 houses are not equipped with this kind of system, instead, a circuit breaker is triggered 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 does not 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 has not 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. For example, consider the following use case: 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 is not 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 avoiding blackouts. 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 and save money.

II. Related Work

Different designs have been proposed for this kind of application. In [3] a wireless sensor network is implemented using ZigBee, which is more indicated for IoT scenarios. In [4], [1] the communication protocol used is TCP over wireless. In our POC, we use UDP over a wireless connection. The theoretical advantage of using UDP is that we avoid the TCP overhead that is not needed in this kind of application, however, since the traffic generated is so little, empirical evidence of this advantage may only be manifest within a system with a large number of smart plugs connected. Differently from other proposed works, on top of the energy monitoring system, we implement a mechanism that keeps the user from exceeding a configurable maximum power usage.

III. State of the Art

Smart plugs with power consumption detection are commonly used in the same household scenarios, but they implement different workflows and protocols. Usually to realize these IoT devices an external platform is integrated with the main-board (Arduino, Raspberry Pi, generic boards with EPS 8266). In most cases, this platform is Blynk, which is a hardware-agnostic IoT platform and allows us to configure our board in many different ways. The user could use the cloud infrastructure provided by the Blynk company, or, since the source code is open-source, download and install their server. This latter scenario is the preferred one since is free and consists of the Blink server, the blink app, and the smart devices, which use Blynk API to perform the connections. A Blynk installation is generic-purpose, so you could install and control different types of devices and show the measurements using only the TCP protocols. Anyway, the app user interface needs to be customized and the server needs a database, so the configuration requires much time. Instead, our system - as described in the following paragraphs - has a different goal and provides essential features to manage the smart plugs. Our project aims to create a functional IoT system that avoids blackouts and in the meantime has a low impact on the wireless network using a modified version of the UDP protocol. Although, Blynk implements some security feature which makes it more secure respect to our system.

IV. General architecture

The general architecture, represented in Figure 1, has three main components: the smart plugs, the server, and the web interface. The smart plugs code has been written using Arduino IDE in C++. Instead, the server has been developed in Java and the web interface is written in React. The rundown of what each part does is made up of a WeMos D1 R2, an Arduino-like board, monitors a sensor that gathers data on the power consumption, and controls a relay that turns on or off the appliance. It communicates over wireless 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.

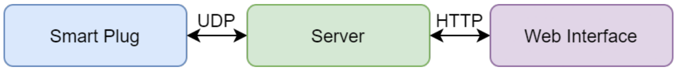


Fig. 1. General architecture of the system

A. Smart plugs

The smart plug is a small power plug that consists of a WeMos D1 R2 board, a 220v relay that can be controlled using digital signals, and a coil connected to a PZEM-004T sensor [5] [6] which provides the readings to WeMos through its digital pins. The hardware has been chosen to be low-cost and accurate [7]. The connection scheme is represented in Figure 2 and has a cost of about €15, so it is not that expensive. The smart plug can be seen in Figure 3. The WeMos D1 R2 board is equipped with an EPS 8266 wireless module operating at 2.4GHz. The connection settings are hard-coded 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 that 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 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 message for a certain amount of time, 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 watching over the readings provided by the PZEM sensor which integrates an ADC system in one integrated circuit [4]. A summary of the interactions between plug and server is given in Figure 4. The content of the packets is first defined as a JSON object and then serialized to the bytes that are the payload of the UDP packet. All of these packets have a field in common: `act` this field can have as values:

- `"INIT"` when responding to the server INIT packet;
- `"UPDATE"` when sending an update to the server;
- `"ACK"` when sending an acknowledgment to the server.

The `"INIT"` packet contains three more fields:

- `"type"` indicates the type of appliance plugged;
- `"max_power_usage"` indicates how many watts the appliance is expected to consume;
- `"sts"` indicates if the status of the relay, `0` for off, `1` for on.

The `UPDATE` packet has a `type` and `active_power` fields. This latter is the last watt usage read at time t. Finally, the `ACK` packet has the `timestamp` filed, which value is taken from the same field in the plug UDP packet received from the server as a unique identifier for that packet. 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 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 periodically broadcasts *INIT packets* to let

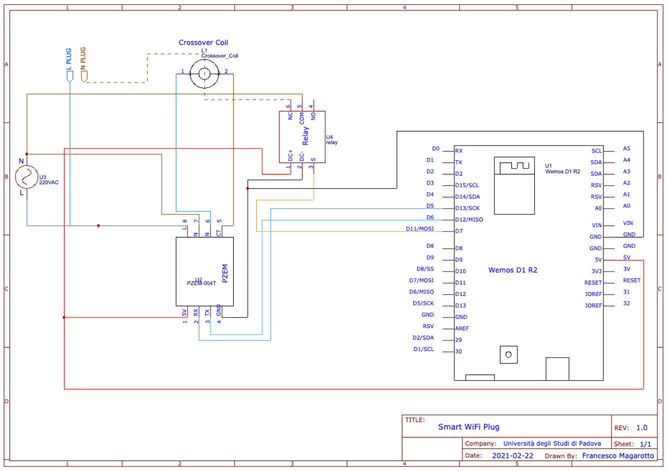


Fig. 2. Electric schema of the wireless smart plug

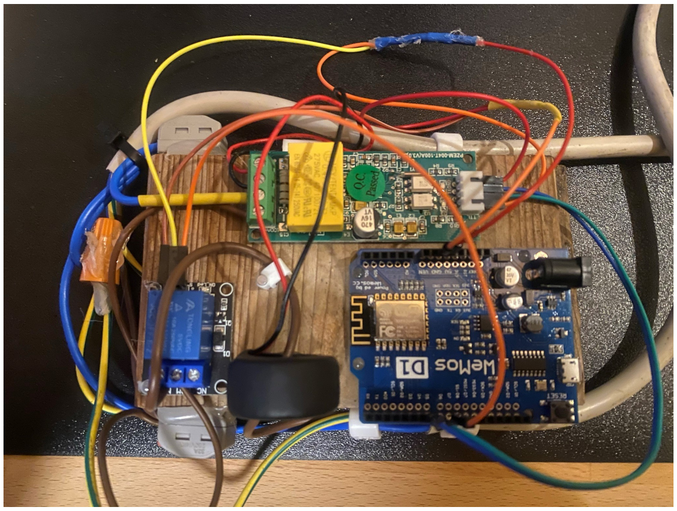


Fig. 3. Proof of concept of our smart plug

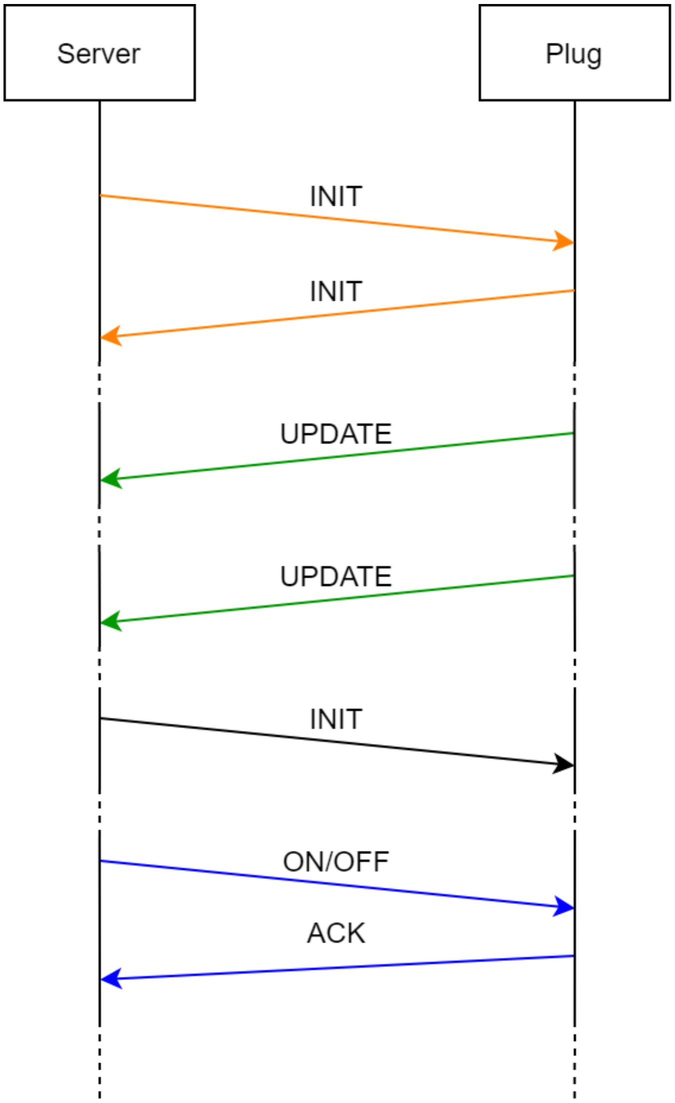


Fig. 4. Interaction between smart plugs and server

the plugs know its address and it listens for connections and updates from the plugs. Furthermore, it waits for acknowledgments and, in case they are not received re-transmits the lost packets. Finally, it exposes servlets to let the web interface get information, about the plugs and send *ON/OFF packets*. The periodic actions, namely, sending the *INIT packet* and re-transmitting 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 plugs replies. When a plug responds to the *INIT packet*, 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 packet* 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 ON or OFF packets. A summary of the logic used by the server is shown in Figure 5. ON/OFF packets contain a `timestamp` field which is sent back by the plugs in the acknowledgments. The server has a list of packets waiting for acknowledgment, in this way when the server receives an ACK it can discard the packet with the given timestamp. If a configurable number of seconds elapses before this happens the packet will be sent again. Like the plugs, the server first defines the content of the packets it sends as JSON objects. All the packets sent by the server have a field: `act` this field can have as values:

- `"INIT"` when broadcasting the INIT packet;
- `"ON"` when sending a plug the instruction to turn on;
- `"OFF"` when sending a plug the instruction to turn off.

The servlets simply allow the web interface to get the server 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, and this latter will wait for an acknowledgment. The information about the plugs saved

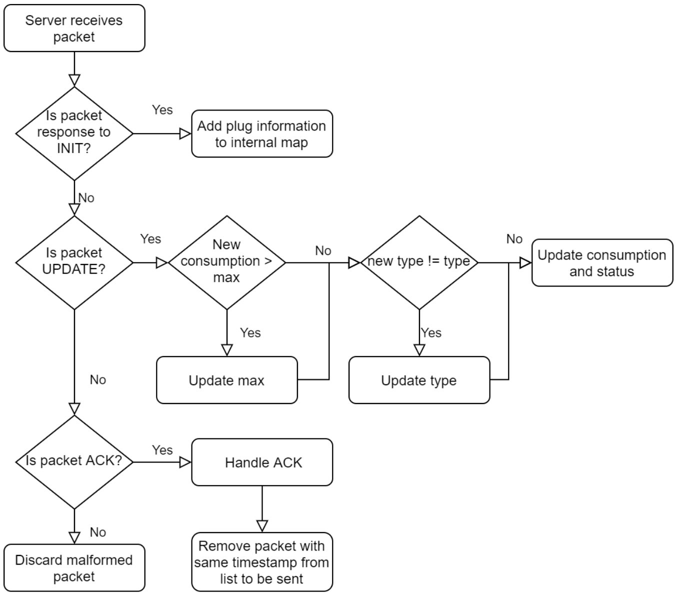
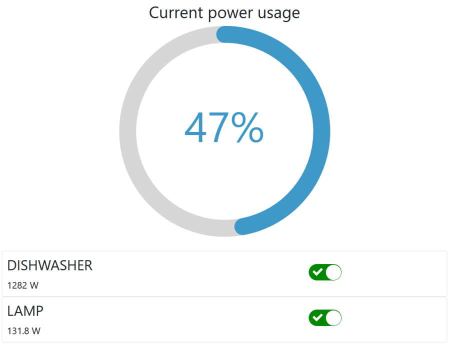


Fig. 5. Overview of server's logic for handling packets received by the plugs

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 an *XML file* to avoid losing it in case of a server malfunction or restart.

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. The polling calls to the API exposed by the server through the servlets are performed over HTTP and therefore using the TCP protocol. The GUI shown in Figure 6 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 to avoid a blackout.



Current power usage

47%

DISHWASHER
1282 W

LAMP
131.8 W

Fig. 6. Web interface used to control the whole system

V. Methods

Throughout the development, the communication between the server and the plug has been simulated and tested using PacketSender [8]. This tool allows the user to define the content of a packet in ASCII format, this is turned into bytes and sent to the specified address and port. For our tests, we specified the content of the packets by writing the JSON object in ASCII format, the type of packet (UDP), and the server address and port. In particular, we tested the system traffic and the correct handling of acknowledgments. To do this we used PacketSender to simulate the traffic generated by the plug but not the acknowledgments and Wireshark, a packet analyzer, to check the correct re-transmission of packets by the server by first filtering the traffic by port and type. The PZEM power monitor, according to [9], has an error rate based on the device characteristics, so we adopted different margin rates depending on the power consumption assigned to the device type. Finally, the entire system has been tested to check the correct transmission of acknowledgments by the plugs.

VI. Challenges

During the development, we found some issues outside the scope of this POC. Since the focus of the project was not on security we decided for ease of debugging to do not encrypt the packets exchanged between plugs and server. Another issue is that the configuration of the plug is hard-coded, for example, the Wi-Fi SSID and password are stored as a char array in our C++ code. Since the EPS 8266 could switch to access point mode which means generate a new wireless network, a configuration procedure could be implemented. A device - such as a smartphone - could connect to the plug wireless network and send the SSID and password required by our home wireless settings. That information should be stored in a persistent drive - like a micro SD card.

VII. Security issues

As we already mentioned, another challenge to face is the security issue, the information sent from a plug to the server is completely clear, so there is no encryption implemented. In fact, we aim to create a smart system and for debugging purpose is more convenient to use un-encrypted packets. The following paragraphs describe different attacks which we have implemented or could be performed quite easily.

A. Fake plugs

A fake plug could be easily added to our system sending fake readings. In our scenario, this could be a problem, because could lead to different behavior. If an attacker uses a fake plug could send a random value to the system avoiding the user to turn on the correct device, considering that the system sums the energy consumption to prevent blackouts. An authentication mechanism, for example using a token validation procedure, could be implemented to recognize and guarantee access to authorized devices. An example of a fake plug is the NodeMcu V3, which is used for debugging purpose, but from a different point of view could be seen as a malicious device and be a problem in a real scenario. We used the NodeMcu V3 board to perform this attack, and since it is a high-level threat an authentication procedure is required.

B. Man in the middle

A man in the middle attack could proxy the UDP traffic and change the content of packets - such as the measurements or ON/OFF commands. An attacker could even change the source IP address field of some INIT packets, and so the plugs would connect to the attacker server. Some tools, such as Scapy which is a Python library, allow to capture, modify and forward packets. Our system would not be able to detect that the packet has been modified, also because the UDP checksum is not enabled and so, the integrity of the header and payload is not guaranteed.

C. Creating interference using a magnet

This last threat consists of using a magnet and gets it closer to the coil. The coil generates an electromagnetic field to measure the power consumption and using the magnets the measurements may change due to interference. This kind of attack works if there is no housing to protect the socket, and therefore also the coil. By using it, the magnet will not be able to alter the measurements. This all depends on the size of the magnet used, since we tried with a very small one and, pushing it away a little, there were no effects on the measurement. In the end, this attack is an open-ended question and needs further investigation.

VIII. System results

Generally, TCP traffic worsens the performances of UDP in wireless applications. To check that this is not the case with our system, we used Wireshark to monitor our network while it was running to evaluate the impact of our traffic. During the test, we had both TCP and UDP third-party traffic, in particular, we had an Apex Legends game (UDP) and normal web navigation (TCP). The main objective of the test was to check the efficacy of our version of reliable UDP, so the focus was on the reception of acknowledgments by the server. The packets are sent to the correct client, even in a scenario with different traffic types. The traffic generated by the system is not much as can be seen in Figure 7. While the test was successful, it only had two plugs, one of them simulated. To achieve more realistic results the simulated plug generates more traffic than a real plug

IX. Conclusions

In the current day and age, we have the opportunity to make smart the most mundane objects. Smart plugs can be used to monitor energy consumption, give control to the user, and help the user reach a maximum consumption goal. These plugs can reliably operate using the UDP protocol provided there are some reliability checks at the application level and can do it in a busy home WLAN where traffic can be both TCP and UDP. Our POC provides an idea of wireless plugs which avoids the home blackouts, and in the meantime helps our planet making people more conscientious about wasting energy.

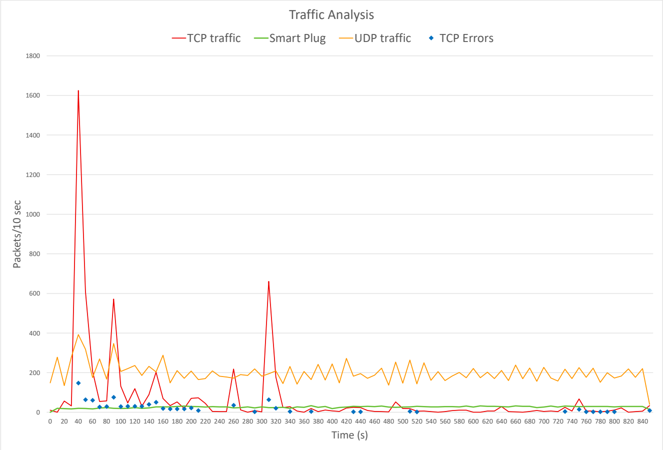


Fig. 7. Network traffic analysis chart. The green line is the UDP traffic generated by our system implementation, and it impacts on the whole network traffic

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/

[3] E. Irmak, A. Köse, and G. Göçmen, “Simulation and zigbee based wireless monitoring of the amount of consumed energy at smart homes,” in *2016 IEEE International Conference on Renewable Energy Research and Applications (ICRERA)*, 2016, pp. 1019–1023.

[4] K. Chooruang and K. Meekul, “Design of an iot energy monitoring system,” in *2018 16th International Conference on ICT and Knowledge Engineering (ICT KE)*, 2018, pp. 1–4.

[5] T. Tantidham, S. Ngamsuriyaros, N. Tungamnuayrith, T. Nildam, K. Banthao, and P. Intakot, “Energy consumption collection design for smart building,” in *2018 International Conference on Embedded Systems and Intelligent Technology International Conference on Information and Communication Technology for Embedded Systems (ICESIT-ICICTES)*, 2018, pp. 1–6.

[6] S. Wasoontarajaroen, K. Pawasan, and V. Chamnanphrai, “Development of an iot device for monitoring electrical energy consumption,” in *2017 9th International Conference on Information Technology and Electrical Engineering (ICITEE)*, 2017, pp. 1–4.

[7] R. Khwanrit, S. Kittipiyakul, J. Kudtonagngam, and H. Fujita, “Accuracy comparison of present low-cost current sensors for building energy monitoring,” 05 2018, pp. 1–6.

[8] [Online]. Available: https://packetsender.com/documentation

[9] Syafii, A. Luthfi, and Y. A. Rozzi, “Design of raspberry pi web-based energy monitoring system for residential electricity consumption,” in *2020 International Conference on Information Technology Systems and Innovation (ICITSI)*, 2020, pp. 192–196.