Design and implementation of a low cost modular sensor

Augusto Ciuffoletti *Member, IEEE,*

***Abstract*—On the financial side, a key factor for the success of a smart-city initiative is the low cost of the sensors. On the technical side, the units need to be flexible enough to cover different roles, and to be reconfigurable for a distinct target. The capability to interact with an already deployed infrastructure, like a preexistent web-server, is a favorable feature from both the financial and technical point of view.**

In this paper we exploit an overlooked feature of the ESP8266 WiFi chip, i.e. the AT commands interpreter, to implement a sensor/actuator that meets the above specifications. To test our design, we implement a library that provides a transparent wrapper for AT commands.

Using a prototype consisting of a board hosting an ESP01 coupled with an Arduino Nano, of which the cost is around 10 Euros, we evaluate the static properties and the operational stability.

***Index Terms*—Internet of Things,** OCCI**,** REST**, WebSocket, Edge Computing.**

I. Introduction

The design and prototype deployment of complex IoT infrastructures, like in a smart-city project, require the availability of flexible, low cost devices that, while being adaptable to various scenarios, preserve a common architecture that reduces the cost, and ensures reuse of hardware and software components [10]. While it is a common sense statement that a modular design helps to go in that direction, we observe that IoT devices, especially those dedicated to prototyping, are often designed as monolythic as possible. For instance, the network device is often bundled with the Microcontroller Unit (MCU) in a way that makes difficult to exploit the computational capabilities of the network device, thus relieving the MCU from the management of complex networking tasks.

To go in the direction of a modular design, a fundamental issue is the presence of an interface, the API, between the modules: this element must give access to the capabilities of the involved modules, while being sufficiently abstract to simplify the interaction.

In this paper we focus on a serial WiFi transceiver board, the ESP01, that is sold with such an API already flashed in the on-board memory. This piece of software is open-source, and can be entirely replaced or extended. As a matter of fact, its presence is often overlooked, and designers usually prefer to re-flash the unit. We want to show how the ESP01, the cost of which amounts to a few Euros, is a building block for the implementation of a powerful prototyping device, that is obtained coupling it with a low cost MCU, like one of the Arduino family, using AT commands. The cost of the whole *thing*, excluding sensors and actuators, is estimated less than 10 euros, with prices on the retail market. This allows medium scale deployments at costs that are acceptable for a small community. Design modularity ensures that the device remains flexible, for instance allowing the replacement of the MCU or extending the network device with new functionalities,

The first step on this way is the production of the interface library. Using it, we show that the *thing* is sufficiently powerful to reliably feed a Thingspeak channel using POST requests, saving enough hardware resources for sensor/actuator control. Notably, the management of a POST entails the processing of both the header and the body of the HTTP response, which is a moderate challenge given the limited capabilites of the Arduino MCU.

The interest for a Restful *thing* is related to the features of a HTTP-based infrastructure, included the availability of software and hardware components that simplify its deployment. There are proposals to automate the deployment of IoT infrastructures that are based on Restful capabilities of IoT devices [1].

We evaluate the stability and reliability of our prototype with a run of nearly two days: after that lapse the device is stopped, and the number of lost feeds is evaluated .

II. Architecture of a IoT device

The design of an IoT device is usually split into two distinct components that are specialized to carry out two characteristic activities: one interacts with the environment using sensors and actuators, another provides the connection with the Internet. The two components exchange data using a low level protocol (e.g. SPI), and are often tightly coupled in the same board. For instance, to limit the scope to the Arduino family, the recent Arduino UNO Wifi (released in 2016) and the Arduino Fishino (on the market since 2015) share a similar architecture, based on a Atmel MCU and an ESP8266 WiFi controller. In that sense the architecture mimics that of a conventional PC, with the networking device embedded in the same chassis of the motherboard.

The network API offered to the programmer slightly depends on the physical carrier (be it cabled Ethernet or WiFi), and a representative is the Arduino Ethernet library. The functions offered by this library are a mixup of the three layers of interest — link, network and transport — and come at the price of a library that may have an heavy footprint, considering that the available program space is in the range of the tens of kilobytes. The library uses a low level protocol to exchange

Augusto Ciuffoletti is with the University of Pisa - Dept. of Computer Science

data and control between the MCU and the networking device, and is entirely opaque to the programmer.

However, network software is very sophisticated, and it is convenient to move it to the specialized device, so that it does not take space in the MCU RAM. In that way, it might be optimized for the specific hardware by the producer itself.

Among the boards that provide WiFi networking, the Wiznet WizFi family and the Expressif ESP8266 are two representatives of this approach. They both offer two distinguished interfaces, one through the SPI protocol, which is based on writing internal registers and bit patterns to run commands, another through a serial link, which uses human readable strings that mimic Hayes AT commands. Here we concentrate on an ESP8266 based board, the ESP01.

The ESP8266 AT commands interpreter is an open source software, and it is freely available from the Espressif site. It can be replaced with custom software using a simple procedure, similar to flashing a sketch on the Arduino. In fact the same Arduino IDE can be used to upload software on an ESP8266 board.

As we see, the designer is faced with a relevant alternative:

- keep the AT interface, and complete the project programming the MCU to interact with the ESP8266 using the serial device;
- re-flash the ESP8266, and implement the whole project on it.

The second option is the very popular, and libraries exist that use the internal network device in a sketch that is native for the ESP8266. This approach is appealing, but is open to some issues:

- the device is in charge of two roles, networking and environment, so that we lose the advantage of parallel execution
- the generic WiFi interface implemented by *one-fits-all* libraries does not give access to all the features of the ESP8266 device (e.g., power saving features)

In return, if the application uses a library that implements an interface similar to that of the legacy Ethernet shield, then the application becomes portable to a different architecture (for instance, MCU+Ethernet shield).

In contrast, if our project follows the first option and uses the AT commands, its portability to a system with a different networking device is complicated, since there is not an agreed convention about AT commands: for instance, WizFi and ESP8266 AT commands are not interchangeable. However, we account for the following favorable aspects:

- we de-couple networking and operation activities
- all the features of the ESP8266 are reacheable
- the AT interface is maintained by the producer itself, which gives high confidence on its quality

It turns out that the former approach, that trashes the AT interpreter in favor of a legacy API, is considered as more appealing: a quick tour in the Internet shows that the AT interpreter is used mostly for demo and introductory purposes, and seldom used in applications. The few libraries that make use of the AT commands are *opaque* wrappers, that hide the commands inside functions. For instance [8], but a more complete library, also used at MIT for educational purposes, is in [7].

In this paper we study a different approach, and implement a transparent *wrapper*, that provides a single function that is able to give access to most of the AT commands. The AT command itself is passed as a `char array`, and the response from the ESP board is returned in the same buffer. The library that implements the interface, named *atlib* is publicly available on a git repository [2]. Using it in a prototype device we check the reliability and effectiveness of the ESP8266 API based on AT commands.

III. Outline of the ESP01 board and of the AT commands

The ESP01 is a tiny board (0.5x1") that mounts the ESP8266 chip, an external Flash memory, a crystal and few minor components. An array of 8 pins gives access to the minimum required to use the AT interpreter and to reflash the firmware. Its cost on retail is around 3 Euros.

According with version 2.1.0 of the reference document [4], the AT commands are divided into three sections, depending on the controlled functionality. Following the same schema, we divide them into Basic, WiFi and TCP/IP.

- **Basic** commands are used to control the operation of the networking device, with no reference to network operation. For instance, among these commands we find a reset function, UART control, power saving, GPIO pins control, device inspection. This section counts 21 commands;
- **WiFi** commands are used to perform link layer operations. Besides the association to an AP, there are commands to control the DHCP function, to set and check IP and MAC addresses, as well as Zeroconf protocol features;
- **TCP/IP** commands are used to manage Client/Server TCP and UDP, and also to retrieve date and time information from SNTP servers, to control DNS servers, and to generate ICMP pings.

All commands follow a query/response pattern, and share some features. They all begin with the `AT+` string followed by the name of the function and a variable number of characters for switches and parameters. The commands return a variable number of text lines separated by a `\r\n` combination, with the last line closed by an OK in case of successful termination of the command. Otherwise, the line terminates with a `ERROR` or `FAIL` string.

To send a chunk of bytes across a TCP connection, the device is first prepared with an AT+CIPSEND command that informs about the length of the data, and next the data is directly sent through the serial link. When the ESP01 detects the termination of the transmission, it returns ready to receive AT commands.

IV. The *atlib* library

The *atlib* library takes approximately one Kbyte of program memory, and 200 bytes of dynamic memory; with reference to the Nano Arduino board, they are respectively 4% and 8%

```
class ESP
{
public:

  ESP(SoftwareSerial *mySerial, int baudrate);

  void reset();

  int state();

  int atflush();

  int atcmd(
     char cmd[],  // the two-way buffer
     int size,    // the size of the buffer
     int timeout);// operation timeout in seconds

 int session(
     char header[], // the two-way buffer for the header
     int headersize,// the size of the buffer
     char body[],   // the two-way buffer for the body
     int bodysize,  // the size of the body
     int timeout);  // operation timeout in seconds
private:
  SoftwareSerial * _mySerial;
};
```

TABLE I
DEFINITION OF THE ESP CLASS

of the corresponding resource. These figures do not take into account the buffers needed to store the contents.

The library depends on the SoftwareSerial library. This option, that limits the baudrate, is justified since we want to keep the unique hardware UART available for debugging and programming activities. This ensures that the board can be effectively used for prototyping activity, as in our initial statement. In addition, we want our software to be readily portable to simpler devices, like AtTiny MCUs, that do not have a builtin UART. We tested our design with a baudrate of 19200, corresponding to 2.3 Kbytes/sec, which is adequate to low bandwidth actuators/sensors. However, it is a soft design decision that can be easily modified.

The *atlib* library defines the ESP class, the wrapper for the AT commands (see table I). The constructor takes as arguments a reference to a SoftwareSerial object, and the desired baudrate. Five methods are defined for an ESP object: they are outlined in the next sections.

A. The `atcmd` transparent wrapper

The `atcmd` method is a transparent wrapper and takes as parameters, a `char` buffer containing an AT command, its size, and a timeout. The semantics of the method consist in forwarding the command found in the buffer to the ESP device, returning the device response in the same buffer. The method blocks until one of the termination patterns (`OK`, `ERROR` or `FAIL`) is received from the device, or when the timeout expires.

The `atcmd` returns an integer, with a meaning summarized in table II.

The buffer contains the full response, which is different for each command and follows the syntax described in the manual.

B. The `state` and `reset` opaque wrappers

We introduce two separate opaque wrappers to reset and to inspect the network device state.

1	successful termination (OK)
-1	unsuccessful termination (ERROR or FAIL)
-2	command timeout
-3	buffer overflow

TABLE II
RETURN CODE OF THE `state` METHOD

2	station associated with AP
3	TCP connection created
4	TCP not connected
5	station not associated to AP
6	command timeout after 2 seconds (builtin)

TABLE III
RETURN CODE OF THE `state` METHOD

The response to the `AT-CIPSTATUS` command does not end with one of the strings mentioned above, so we need a specific parser. In addition it returns a code that is incompatible with the `atcmd` return code. So we introduce a special `state` method for it.

The `state` method does not have input parameters, and returns an integer: see table III for its meaning.

The `reset` method wraps the `AT+RST` command, that returns a response which is always terminated by an *OK*, but is preceded by noise and a generic presentation. In this case we prefer to trash the response, since it might saturate buffer capacity. So the `reset` method has no parameters and does not return a result, but has the side effect of cleaning the state of the device.

C. The `session` request/response method

The `session` method manages a two way session with an HTTP server on the other side. It is designed to simplify a RESTful session, and its arguments are two buffers, one for the header of an HTTP message, the other for the body, their sizes, and a timeout.

The `session` method does not call AT commands on the network device, but simply delivers the *header* and the *body* of the HTTP request to the network device, that, in its turn, will deliver them to the destination, separated but an empty line. Next it waits for the response from the server: the response is split into a header and a body, and each of them is truncated to fit the dimension of the respective buffer.

The application is responsible for the compilation of the header: the startline and the attributes must comply with the HTTP protocol and with server expectation.

The return code of the `session` method has the meaning described in table IV.

D. The `flush` cleanup method

The `flush` method is used to recover from a timeout: when such an event occurs, part of the response may be received by the serial device at a later time as part of the response to

0	successful termination
-2	timeout

TABLE IV
RETURN CODE OF THE `session` METHOD

another command: this can easely disrupt a communication protocol. The `flush` reads all available bytes from the serial input, and terminates after a two seconds timeout. All contents are discarded.

E. Buffer management

The storage space is a scarce resource in a MCU, so we leave to the developer of the application the final word about where and how much memory to allocate.

Regarding how to allocate the memory, the user may decide to allocate it in the global dynamic memory as a fixed size characters array, or use space in the program memory and use a `malloc`. In the latter case there is no risk of fragmentation, since typically allocation occurs during the `Setup` operation and there is no `free`. In fact, we have tested both solutions with identical results.

The size of the buffers heavily depends on the application. We have found that 100 bytes are enough for the AT commands: as a design decision, the `atcmd` method terminates with a un-recoverable error in case of buffer overflow. The size of *header* and *body* buffers depends on the specific application. We noticed that, to POST a new feed to the Thingspeak server we need approximately 150 bytes in the *header*: besides the startline, we need Host, Content-Type and Content-Length fields to obtain a positive acknowledgment from the server. The *body* may be limited to the bytes needed to describe the feed, or a Thingspeak TalkBack operation.

Unlike the command buffer, the *header* and *body* buffers can overflow, and exceeding bytes are silently discarded.

V. The target application: a POST to feed a Thingspeak channel

The repository of the *atlib* library contains some examples: here we analyze the one that we consider as a significant milestone, which is the implementation of a POST request. Together with the GET, which is simpler since it does not entail the production of a *body*, they are the minimal toolset to interact with a HTTP RESTful API, as in the case of the Thingspeak service (`https://thingspeak.com`).

The challenge consists in reducing the size of the program, library and main, that are needed to manage the association to the AP, to open a new connection with the server, send the HTTP request and receive the HTTP response. Like program memory, also the buffers used for data management go through the limited memory of our target device, which amounts to 32Kbytes.

It turns out that the sketch that performs the POST using the *atlib* library uses the 1Kbyte (11%) less than a GET using the *WiFi* library (as reported in one of the examples of the *WiFi library*), with the advantage that the *atlib* library takes care of buffering the response and separating the *header* and *body* into two distinct buffers, which is a significantly complex operation.

Coming to the POST implementation, in figure V we see the core of the sketch that implements the request/response exchange. The first 9 lines in the code are for preparing the content of the request. The *body* is prepared first, since we need to include its length in the *Content-Length* field, and next the *header*.

```
// prepare query
  sprintf(body,"api_key=%s&field1=%d", CHANNEL_KEY,v++);
  sprintf(header, "POST /update HTTP/1.1\r\n");
  sprintf(header+strlen(header),
    "Host: %s\r\n",HOST);
  sprintf(header+strlen(header),
    "Content-Type: application/x-www-form-urlencoded\r\n");
  sprintf(header+strlen(header),
    "Content-Length: %d\r\n",strlen(body));
// connect to server
  do {
    sprintf(cmd,"AT+CIPSTART=\"TCP\",\"%s\",%d",HOST,PORT);
    esp1.atcmd(cmd,CMDSIZE,10);
  } while ( esp1.state() != 3 );
// prepare send operation
  do {
    sprintf(cmd,"AT+CIPSENDEX=%d",
      strlen(header)+2+strlen(body));
  } while ( esp1.atcmd(cmd,CMDSIZE,5)<0 );
// send query
  esp1.session(header,HEADERSIZE,body,BODYSIZE,10);
```

TABLE V
CODE FRAGMENT FROM THE THINGSPEAK_POST EXAMPLE

The lines that follow (11-14) are a loop that repeatedly tries to open a new connection with the server using an `AT+CIPSTART` command. When the connection is created, the ESP device is configured to receive and forward a number of bytes corresponding to the whole HTTP request: header, body, and a separating empty line. This is obtained with a `AT+CIPSENDEX` command, at lines 16-19. Finally the request is piped through the serial interface.

A typical HTTP packet that feeds one or more field values is 200 bytes long. At a baudrate of 19200 this corresponds to roughy 0.1 seconds: depending on the application this may be too much. In that case, the designer may use the hardware UART for communication — and the SoftwareSerial for debugging — and reduce the time to send the request of ten times (approx 10 msecs). However programming the MCU becomes more time consuming, since we need to disconnect the ESP01 from the RX/TX pins used for flashing.

Our test application envisions 200 Bytes for the *header* buffer, and 100 Bytes for the *body* of the message. When the call to the *session* method returns, the two buffers contain the *header* and the *body* of the HTTP response, truncated to the length of the buffers. In our case, in the *header* the response startline is available, as well as the *Content-Length* field. The *body* contains the cumulative number of values logged in the channel. In case of a GET it may contain a selection of past feeds, or the identifier of an action to perform (TalkBack service).

A. An experiment

To have an idea of the stability of the sketch, we run it for nearly two days, sending a new value every 30 seconds. The value is an integer incremented each time by one, so that we have been able to check software and network problems.

The hardware we used was the ESP01 coupled with an Arduino Nano, all mounted on a breadboard. The schematic is in figure 1. We also tried a similar schema using an Arduino

Mini Pro 3.3V, with similar results. In this latter case the soldered prototype board was approx. 2x1", including the power stabilizer.

We stopped the counting device after 5000 rounds, i.e. 41 hours of continuous operation, and we counted 323 (6.5%) "Bad Request" replies that result in a lost value. We considered this value excessive, so we investigated about the reason.

The first hypotheses was a SoftwareSerial synchronization problem. So we tried with different baudrates, but the loss rate did not change significantly. Then we investigated the time distribution of the events, and we found that they were concentrated during the lapse between 3pm and 2am UTC (see figure 2). Using an IP geolocation service we found that the Thingspeak server was located in the Amazon AS in Ashburn (VA-USA), in the EST timezone. Shifting the high-loss time period into the server's timezone (GMT-4 with DST) we found that it corresponds to working hours (11am-10pm), so we considered that the high failure rate was due to the server side infrastructure load, and not to a problem in our device. During a *low-loss* interval of fifteen hours the loss rate drops to 1.4%, and we consider this as a pessimistic failure rate for our device.

We conclude that the sketch is stable, since it did not crash during the test period, but it is exposed to the loss of some feeds, in the order of a few percent points. Notably, the loss rate can be easily lowered by introducing a retry strategy in case of *Bad Request* response, but this is outside our scope.

VI. Limits of the AT commands

The experiments we carried out, and that are summarized in the previous sections, allow to identify some weaknesses of the AT commands, as they are today. We list them in order of importance.

One relevant feature that is missing is security. It is clear that any aspect of an information management system must be designed with an eye to possible intrusions and threats, but the AT commands do not offer (or, at least, there is no document about) a way to open an SSL connection. The AT commands reference document mentions such possibility (namely, there is an option value dedicated to open an SSL connection, and a command to allocate buffer space for SSL), but we did not find an exhaustive documentation. A paper exists that describes the use of an on-board library in order to implement secure communication [3], but this is not in the scope of our design, since we want to use AT commands on the serial device.

Another serious drawback of the AT interface is its footprint in terms of the space used to store command strings. Consider that each string denoted with the "double quotes" syntax takes space as a data item, which raises the opportunity of an optimized design of command labels. Instead, command labels are clearly redundant, well beyond what is needed for their mnemonic function. All of them start with an `AT+` prefix, which is therefore completely redundant. A prefix of two or three letters identifies the group a command belongs to: e.g., the `CIP` prefix indicates an IP related command, and is common to nearly half of the available commands. Also in this case we have redundancy and verbosity.

The AT commands do not provide a native support to RESTful HTTP operations. With our experiments we have shown that, using the available commands, it is possible to implement a RESTful session, but at the cost of a significant fraction of MCU memory. Given the growing popularity of RESTful interfaces among providers (and the good reasons for this trend are found in the values of the REST paradigm [6]) it would be advisable that native AT commands were available to carry out REST verbs, as suggested by our sketch.

Even more valuable would be the provision of native WebSocket tools through AT commands. The WebSocket standard [5]is more and more popular since it allows the management of a stream of data (not a request/response session) in the familiar HTTP framework [9]. This eases the design of the infrastructure that gathers data from a multitude of sensors, and distributes control to actuators.

Therefore, while claiming that the AT serial interface has a relevant role in the design of modular IoT devices, we point out that it still misses some relevant features. Regarding SSL and native HTTP one may argue that they can be promptly implemented as extensions of the current AT interpreter: the tools exist to carry out this operation with limited effort.

As for command syntax, this issue hits the cornerstone of standardization. As of now, it wouldn't be a big problem to redesign the interface to have shorter labels. But, in case the AT commands were massively used in many projects, such a change might have disruptive effects.

If AT commands are considered a relevant tool, their design should be reconsidered in view of their utilization in MCU programs, and a common framework shoud be agreed to go in the direction of a common interface that allows the seamless replacement of a device with another, as mandated by a modular design.

VII. Conclusions

When I started out the design of a low cost Restful *thing*, I was prepared to find many projects on public repositories like Github and Fritzing to help me in the task. When I found that this was definitely not true, I decided to implement from scratch what I had in mind. During the implementation process, I noticed a number of relevant details in what I was making, and I decided to prepare a conference paper. The result is a DIY project with a research flavor, a hybrid document that shows that something is changing in computer science: the potential of a public repository needs to be combined with accuracy and method of a scientific research. I tried to put together all these aspects in this paper, and I am grateful to the reviewers that appreciated my effort.

Starting from methodological aspects, one detail that I have not found browsing projects in public repositories is the decoupling of computational and networking activities. This is usually considered commonsense, but I found that, among IoT projects, the opposite attitude is more popular indeed. So I decided to head forward a strictly modular architecture.

Communication between modules is consequential, but I found that AT commands are seldom used for purposes beyond simple educational TTY-driven exercises. I decided to focus on

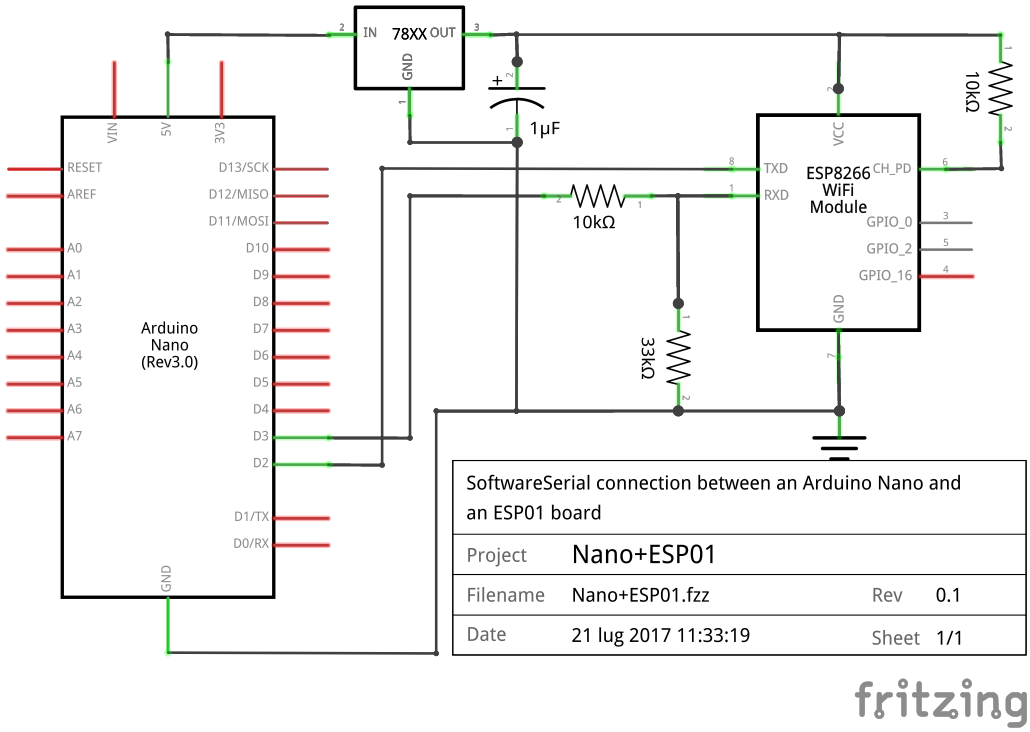

Fig. 1. Electronic schematics of the prototype used for the experiment. D2 and D3 on the Arduino are respectively RX and TX for SoftwareSerial

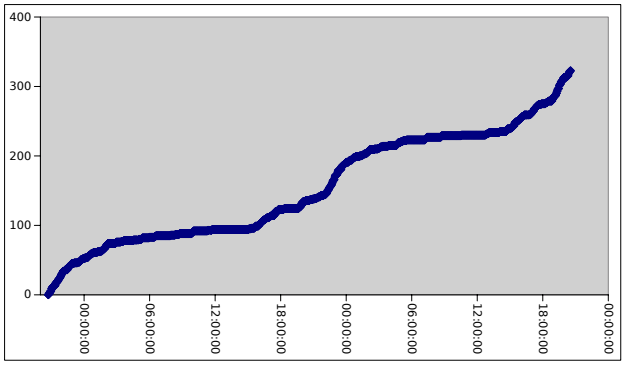

Fig. 2. Cumulated number of missing feeds during a two days period

this interface, and the results show that this allows the design of a thing with a modular architecture.

Many projects that I have found do not take into account that, once communication has been implemented, the residual resources must be sufficient to implement possibly sophisticated applications. I met two critical decision to take on this respect that are superficially documented and compared in the literature: the utilization of the UART, split between development and communication, and the allocation of memory buffers, either static or dynamic. I found many superficial discussions, and I finally opted for the less popular options. The hardware UART is dedicated to development, which is a relevant activity in a prototyping environment, and a software emulation of the serial interface is used for communication: since it is a controversial option, it is scientifically relevant to define its limits. On the storage side, I discovered that dynamic allocation is usually *disencouraged* for reliability reasons, justified by programming difficulties. But, while it is true that an `malloc()` is more difficult and error-prone than a variable declaration, a dynamic allocation saves precious memory that can be used for the application. So I opted for the latter alternative.

Also the co-existence of two electronic standards, CMOS for the ESP01, TTL for the Arduino, is often matter of confusion. For this reason I decided to make explicit the electronic scheme, to document my option.

It is part of the scientific method to perform significant experiments to verify the assumptions, to spot potential problems, and to identify directions for future investigation. After a number of test applications, I decided to implement a POST session, the cornerstone of a Restful thing. From this experiment I came to two basic results: we know that it is possible to manage a POST inside an Arduino leaving enough resources for a reasonable application, and we we have an idea of its performance and reliability.

Based on such results, I discuss the fundamental design decision, consisting in the adoption of the AT interface: there are a number of issues with it, discussed in Sect. VI, that appear to be approachable and solvable with a moderate effort.

Coming to the practical results of my investigation, we have the implementation of a library that provides a *transparent wrapper*, that gives access to the AT commands through a C++ object. This library provides generic networking functions, and

can be reused and improved in other projects. This happens thanks to the existence of software repositories that allow the publication of open source projects.

I claim that one of the scenarios where this solution is especially suitable is the design of devices and infrastructures for the improvement of urban environments. It is a field where a *trial and error* strategy is mandatory, implementing solutions that the target community will accept, refuse, amend. In these scenarios modularity, cost, and simplicity, the starting points of my investigation, have a fundamental role: the administrator wants to be able to change the solution without loosing investments of public funds, to keep costs low since the community is often small, and to use local limited competence to develop the solutions.

As a final remark, I observe that another scenario that exhibits similar features is *hands-on* education. Low cost, self assembled devices like that described in this paper allow a rich learning experience to a student in a related discipline. Just another reason to extend the utilization of AT commands beyond *hallo world* TTY exercises.

REFERENCES

[1] Augusto Ciuffoletti. OCCI-IOT: an API to deploy and operate an IoT infrastructure. *IEEE Internet of Things Journal*, PP(99):1–1, 2017.

[2] Augusto Ciuffoletti. A wrapper for the at interface of the esp8266. https://bitbucket.org/augusto_ciuffoletti/atlib.git, July 2017.

[3] Espressif Systems IOT Team. *ESP8266 SSL User Manual*, v 1.4 edition, 2016.

[4] Espressif Systems IOT Team. *ESP8266 AT Instruction Set*, v 2.1.0 edition, 2017.

[5] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011.

[6] Roy T. Fielding and Richard N. Taylor. Principled design of the modern web architecture. *ACM Trans. Internet Technol.*, 2(2):115–150, 2002.

[7] Shen Nong Min. Weeesp8266: Api documentation. https://docs.iteadstudio.com/ITEADLIB_Arduino_WeeESP8266/index.html, 2016.

[8] Rasmus Ljungmann Pedersen. Arduino esp8266 easyconfig. https://github.com/rasmuslp/ArduinoESP8266EasyConfig, 2014.

[9] M. Vujović, M. Savić, D. Stefanović, and I. Pap. Usage of NGINX and websocket in IoT. In *2015 23rd Telecommunications Forum Telfor (TELFOR)*, pages 289–292, Nov 2015.

[10] A. Zanella, N. Bui, A. Castellani, L. Vangelista, and M. Zorzi. Internet of things for smart cities. *IEEE Internet of Things Journal*, 1(1):22–32, Feb 2014.