Putting together my PI weather station

This is a record of my experiences, issues faced, and solutions found for both the construction of a Raspberry Pi-based weather station and the remote uploading, archiving, and web-based presentation of its data. This involved also establishing a web-server with my domain name on my unix in-house server.

I started with a project found on-line, aptly named, Build your own weather station. I added most of the suggested pieces: a temperature/pressure/humidity sensor, an anemometer, and a rain gauge. I skipped the wind vane as that required addition HW since it provides an analogue signal, and for me it was of less importance knowing the wind direction.

The on-line project also provided some simple code to manage the external attached devices, and manage the data. The example project described was actually based on a project supported by Oracle, which was also available on the web, and served as a starting point for the project.

I found the needed pieces on Amazon. The digital sensor is known as a BME280, the rain gauge is a small "tipping bucket", and the anemometer is three-armed with a simple reed switch to count rotations. I also bought a GPIO header breakout HAT (GeeekPi - Modulo di breakout con blocco morsettiera a vite a 4 GPIO per Raspberry Pi 4, breakout per scheda di espansione, per Raspberry Pi 4B/3B +/3B). I used a Pi3 that I already had on-hand, but later switched to a Pi4, for reasons I will describe later.

The original project was written in Python, so this was my chance also to learn a bit of the language, which I had never used up to now. I followed pretty much verbatim the description for both setting up the HW and programming the Pi, including archiving the data in a local MySQL database on the Pi. The Oracle project also included code to upload the data to an Oracle machine, where apparently data was collected from various participants in that project. I leveraged and modified that code to upload the data to my web server in Milan.

Attaching the external devices and putting together the code to collect and archive the data was straightforward. The only missing bit from the on-line project was the need to

enable the I2C (and optionally SPI) interface via *raspi-config* command on the Pi, although this was mentioned on the Waveshare BME280 product page.

The anemometer works by closing the reed switch twice every rotation (clicks can be heard when spinning the device), which sends a type of "button pressed" message to a GPIO pin, programmed to record the signal. Counting the clicks/per time, dividing by two and multiplying by the circumference yields a wind speed. I followed the example project and also recorded the maximum wind speed during the interval, commonly known as wind gust.

The tipping bucket instead has a pyramid-shaped tray inside the bucket which collects water, and when it reaches "full" tips over, emptying the full side of the tray and allowing the other side to fill, and so on. Each tip sends a signal to a GPIO pin which once again is programmed to count the tips. Knowing the capacity of the tray, the amount of rain can be calculated. The sensor instead is "read" by the program on a regular basis to record the current temperature, pressure and humidity it registers. I followed the suggestion to collect readings every 5 minutes: the sensor is queried with this regularity and at the same time the counts from the anemometer and tipping bucket are registered, and an average value calculated for the wind speed.

I installed MySQL (*MariaDB*) and created the weather db and a table within it to hold the weather measurements. Most of the code for writing to the db was present in the example Oracle project. I then modified the Oracle code to send the data from the local db to my remote web server. Python is known for the useful libraries available, and here the *MySQLdb* and *requests* (includes HTTP support) libraries made things easy. I decided to use an HTTP POST to send the data from the local db to the remote one on my server after every 5-minute interval. This duplication of data in the local db instead of just sending immediately the data remotely, I did for robustness. The code is written to compensate for the situation where the remote server may not always be available – this could be because the local internet link is down or, alternatively, my remote server may not be reachable for some reason. Every time the Pi successfully Posts a 5-minute set of data it marks the row in the local db as sent, and every time it attempts to send data it sends all rows not yet sent. If it cannot reach the server it gives up and tries again when the next 5-minute interval is over.

At the moment I haven't implemented a mechanism to clean up the db regularly – it is keeping for the moment all records. But this may eventually be necessary to avoid filling up its disk. I also should implement a periodic backup on the Pi.

Other aspects included on the Pi installation include configuring a firewall (the program *ufw* was installed to make this easier), installing the *unattended-upgrades* (name self-explanatory) module, and installing the weather station program as a service so it is started by the system at boot.

As a service also allowed configuring a watchdog mechanism supported by systemd via *sdnotify*. Code is added to the program such that it sends a "ping" to systemd every *n* seconds, and systemd is configured such that if it doesn't receive a ping within a certain time > *n* seconds, the weather station program will be restarted. So in case the program hangs it will be restarted. Also systemd was configured so that if the weather program crashes for whatever reason it will also be restarted. In this latter case it also sends me an email.

The Pi also has a chip which supports a HW watchdog, which I also enabled. This causes the system to reset if the OS itself hangs.

All these last interventions are to make the Pi more robust against as many eventualities as possible, as it will be installed in a location not easily physically reachable.

One particular issue I resolved after some effort regards the sensor. At some point while testing over a long time period the weather program crashed and although the system properly restarted it, it crashed continually. I discovered that it was the sensor failing that caused these crashes, and even a reboot of the Pi would not help. However, when I power cycled the Pi the sensor recovered. Searching on Internet I found that someone else had experienced a similar problem. I learned that the Pi4 (but not earlier Pis) included the possibility to configure its EEPROM to cut 3V3 power to the GPIO during shutdown (halt). I thus moved my setup to a PI4 (also had a spare one) hoping this would solve the problem. I saw that in fact with this configuration power was cut to the sensor, but unfortunately not during a reboot, which is what I would need to recover the system automatically under such a failure. And once the system is made to halt (and goes into a low-power mode) the only way to restart it is by power-cycling the Pi itself! Which is what I was trying to avoid.

The solution I eventually found to this problem was to connect the 3V3 power lead from the sensor not to a 3V3 supply pin but to a GPIO pin (which also use 3V3), which then can be programmatically controlled to power the device. This solution works only for devices requiring very little power, like the sensor. When the program starts it turns off and then on the GPIO pin, thus power cycling the device, so that if the program crashes, when it is restarted, the chip should reset.

An additional precautionary measure would be to add an image of the SD card disk onto to a USB stick and attach that, so that if the SD card fails it could boot from the USB. Although the failure would need to be such that the system could not boot at all from the card, before it would try to boot from the USB.

The weather station of course needs to be physically installed. The Pi will be put into a waterproof electrical box, while the sensor is put in another nearby, as its box needs to have an opening (away from possible rain entering) to allow air flow getting to it. The anemometer is installed on a pole, with the rain gauge anchored at its base.

The most work for the project was actually on the server side. This work included:

- configuring a web server,
- dealing with having the server behind a NAT (actually two NATSs!),
- getting past my Fastweb router (which in the end meant configuring HTTPS and acquiring a TLS certificate),
- finding a way to capture the HTTP POSTs from the Pi and putting that data into a DB,
- and taking that data and creating a web page to display the information graphically.

I had already setup an Apache web server on my home server, at the time used just for monitoring traffic on my routers. Of course I wanted a separate web server to expose to the Internet, so I created a virtual server on another port, initially port 8080. I did not want to require people using the web page to also specify a port, so, since I needed to port map at the NAT anyway I thought to simply remap from port 80 externally to 8080 internally. I discovered, however, that this would not work since the Fastweb router does not pass port 80! Supposedly they use it internally for

configuring/monitoring the router itself. In the end, I decided instead to use HTTPS (port 443) as this port is not blocked by Fastweb, and as a plus adds a layer of security to my setup. As a side note when I was initially playing with using port 8080 also facing the Internet, I saw a lot of suspicious requests on my server, but when I switched to HTTPS, these mostly disappeared.

So my server is only visible via HTTPS.

In order to configure a server for serving pages via HTTPS, the server needs a TLS certificate. It is possible to create a so-called self-signed certificate; however, this requires users of most browsers to go through an annoying multi-step process to accept this "uncertified" certificate. A truly trustable certificate must be signed by a well-known Certificate Authority (CA.) Most CA require payment for this service, but there is one that performs the verification for free: it's called Let's Encrypt.

There were several methods for going through the required process to get a signed certificate, although most required that the CA access your web server on port 80 during a verification phase. This would not work for me since, as I mentioned above, I cannot use port 80. As an alternative, one can place a DNS record in your domain, which value the CA sends you during the registration process, and the CA then verifies that DNS has been properly updated with this record, and is thus assured you have control of that domain. I found SW on the web (a script), called *getssl*, that semi-automates this process. It worked well except for a small bug in the script that took me a while to find. Also the hosting server which manages my DNS domain records does not support a certain API for automatized updates (as some others apparently do), so I need to perform this process manually every 3 months (which is the expiration limit for the certificates.) I may look into migrating my domain to another hosting company which supports such an API.

The most difficult hurdle to overcome was getting my web server working correctly from behind the NATs present in my routers. The problem is that the server when serving a page with links to other local resources on the server, uses its local IP address in the links, which of course cannot be reached from the Internet. There were a couple of Apache modules / solutions which apparently would resolve this problem, but none seemed to work. After weeks of searching on the web I finally found a relatively simple solution which uses a so-called output filter (output, in the sense of outputting web

pages to the user) which could be configured to substitute the local IP address with my www domain name in the outgoing web pages.

For receiving the POST data from the Pi I found another open source program (php script) that reads data coming from a POST and then inserts it into a mySQL DB. Just what I wanted. I essentially duplicated the mySQL setup from the Pi on my server and used this script to feed the weather readings coming from the Pi into it. I put the script in a username/password protected directory, which credentials the Pi weather program then uses for access. The directory is also not visible from my home page so not directly visible to others.

Now that I have the data feeding into my server I need to present it in some manner. Once again I chose to write the program to do this in Python. Originally I planned to use one of the plotting libraries commonly available for Python, but then remembered an open source software that I used for plotting traffic usage, called MRTG. This SW also has a successor which is a more flexible and powerful tool for graphing time-based data, called RRDTool, which also supplies a Python module with bindings to the tool. This tool creates separate databases (yes another duplication) which are bounded (limited time frame) and maintain data aggregated at configurable time intervals on a round-robin basis, adding new data and discarding the oldest (hence, Round Robin Database tool). This keeps the databases at (relatively small) fixed sizes, according to the time frame and aggregation interval. Again, I could eventually truncate / clean up the data in the SQL db to save space if needed.

In any case, this tool allowed me to easily create different time length graphs: daily, weekly, monthly and yearly for each measurement type. So the Python script runs once every five minutes, reads any outstanding measurement data from the MySQL db (it uses a mechanism similar to that on the Pi to mark the records successfully read) adds this data to the RRDTool db and then recreates all the graphs.

I came up against two quirks of the RRDTool. First it is a tool designed for managing rate-based data, and there is no easy was to calculate total sums of data as I would like to do with rainfall – to indicate total rainfall during a given period. To do this I will need to do it outside the RRDTool itself – albeit a minor inconvenience. Secondly, it doesn't accept "old" data, meaning if you try to insert new data into its dbs that is older than a data point also present, it refuses, and until this data is removed the update will

continuously fail. This situation shouldn't normally occur, but once when the server was down for some time and the Pi needed to send many records, it sent some data out of order. For now I put a check in the server code to simply discard such data, but I probably could fix the issue on the Pi side.

Regarding the Web server, I could have created the pages manually, but decided to use a web design SW, since I may want to do other things on the site in the future. I installed Wordpress, one of the most common SW used on the web. It is mostly overkill for my purposes, but fairly user friendly. I also spent some time hardening the installation, since its commonality on the web also makes it a common target for security holes.

I created some simple pages for presenting the graphs. The Python program simply writes the RRDTool generated graphs in a directory under the web server, which the pages link to. So the pages are essentially static and get updated automatically every five minutes when the Python script runs and overwrites the old graphs.

That's it, in a nutshell!