TESIS CARRERA DE GRADO EN INGENIERIA EN TELECOMUNICACIONES

RED URBANA DE DATOS METEOROLÓGICOS

Gonzalo García Genta
Ingenieria

Dr. Rodolfo G. Pregliasco
Director

Ing. Horacio Fontanini
Co-director

Miembros del Jurado
Dr. Jorge Lugo (Instituto Balseiro - INVAP)
Dr. Felix Maciel (Centro Atómico Bariloche)

6 de Diciembre de 2019

Sección de Física Forense – Centro Atómico Bariloche

Instituto Balseiro
Universidad Nacional de Cuyo
Comisión Nacional de Energía Atómica
Argentina

Resumen

El contenido de este trabajo abarca la implementación de una red de estaciones meteorológicas, desde la generación de datos hasta la presentación en una página web. Se instalaron 3 estaciones meteorológicas y se recopilaron datos de la web con un programa. Se implementó un servidor de comunicaciones por protocolo MQTT. Los datos son recibidos y guardados en una base de datos de series temporales por un programa escrito en Python. Finalmente se brindan los datos con `Nginx` y `Grafana`.

Palabras clave: estación meteorológica, MQTT con TLS, túnel SSH y base de datos de series temporales.

Abstract

Implementation of a net of weather stations, from the generation of data to the serving of a web page. Three weather stations were installed and also a program for web scrapping meteorological data was made. A server for communications using MQTT protocol with TLS was implemented. Data is received and stored in a time series database by a program written in Python. Finally, data is served in a web page using `Nginx` and `Grafana`.

Keywords: weather station, MQTT with TLS, SSH tunneling and time series database.

Índice de contenidos

Glosario de términos y abreviaturas

MQTT	Message Queuing Telemetry Transport, protocolo de comunicaciónes sobre red TCP/IP.
SSH	Secure Shell Protocol, protocolo de comuniciónes sobre red TCP/IP.
GNU	GNU is Not Unix, proyecto para el desarrollo de un sistema operativo de código libre
GPL	GNU General Public License, licensia de uso original del proyecto GNU.
HTML	Hyper Text Markup Language, protocolo para trasnmisión de páginas web.
IOT	Internet of Things.
NIST	National Institute of Standars and Technology.
CAB	Centro Atómico Bariloche.
TLS	Transport Layer Security, protocolo se seguridad.
PN	Parques Nacionales.
AC	Autoridad Certificante, ente destinado a firmar certificados.

Capítulo 1

Introducción

1.1. Motivación y objetivos

La motivación del proyecto es contar con datos meteorológicos precisos. Al comienzo del trabajo los datos eran requeridos únicamente por proyectos de investigación de la sección de Física Forense del Centro Atómico, luego se detectó que muchas otras organizaciones requieren de los mismos. En la sección de Física Forense se estudian métodos para determinar el tiempo de muerte en función de la presencia de insectos en el cuerpo, para esto es necesario un registro preciso de la temperatura y humedad. Se tomó conocimiento de que el insumo datos meteorológicos es requerido por investigaciones que se realizan en el Centro Atómico, en las dependencias del INTI en Bariloche y por la prevención de incendios en el Parque Nacional Nahuel Huapi. Los únicos datos oficiales disponibles en Bariloche son los que brinda la estación del Servicio Meteorológico Nacional en el aeropuerto, pero estos no son representativos de lo que ocurre en otras zonas de la ciudad. Los datos del aeropuerto difieren mucho de lo que ocurre en otros puntos de la ciudad dado que la variación espacial del clima en Bariloche es significativa. La zona del aeropuerto en Bariloche es una zona de pocas lluvias y gran parte de la ciudad es una zona húmeda. Ante la necesidad de tener datos meteorológicos de mayor precisión se propuso armar un sistema para la generación y recolección de datos meteorológicos y luego servir estos datos en una página web pública.

Cabe aclarar que el alcance del trabajo se limitó a proveer datos confiables y de la mayor cantidad de fuentes posibles. Realizar pronósticos o análisis de datos queda fuera del alcance del proyecto.

1.2. Criterios

Se definieron criterios de diseño a modo de guía y objetivo, a continuación la lista y explicación de cada uno.

- **GNU**: Se desea que la red sea reutilizable por otras personas o que se reciban colaboraciones. Para esto se decidió seguir con la filosofía del proyecto GNU[1]. El proyecto GNU fue uno de los pioneros del movimiento para el desarrollo colaborativo de software libre y hoy en día lo sigue siendo.

- **Escalable**: el sistema diseñado debe ser capaz de expandirse para abarcar mayor área geográfica sin tener que realizar grandes modificaciones en el código.

- **Replicable**: el sistema diseñado debe ser capaz de replicarse en otros lugares de manera que sea de mayor utilidad.

- **Reutilizable**: el sistema debe ser reutilizable para cualquier tipo de datos, de manera de ampliar el uso del sistema.

- **Modularidad**: el sistema debe mantener un diseño modular, con partes separadas por funciones especificas, de modo que se mantenga mejor organizado.

- **Costo**: el sistema debe mantener un costo bajo en el diseño, realización y mantenimiento.

- **Servicio**: el sistema debe ser capaz de brindar un servicio útil a la comunidad.

- **Suma de Aficionados**: existen muchos aficionados con estaciones meteorológicas que mantiene sus datos para uso privado, se quiere sumar a estas personas a la red a cambio de brindarles algún beneficio. El sistema debe ser capaz de sumar a estos aficionados fácilmente.

1.3. Diseño del sistema

La red se dividió en 4 secciones principales que la separan por funciones. En la **Figura 1.1** se presenta un esquema del sistema implementado. Los datos se transportan desde la sección de la izquierda (donde se originan) hacia la derecha, pasando por la transmisión, el almacenamiento y llegando a la etapa final donde se sirven en una página web.

[1]El proyecto GNU fue fundado en 1978 por Richard Stallman en la universidad MIT con el objetivo de crear un sistema operativo de *software* libre basado en UNIX

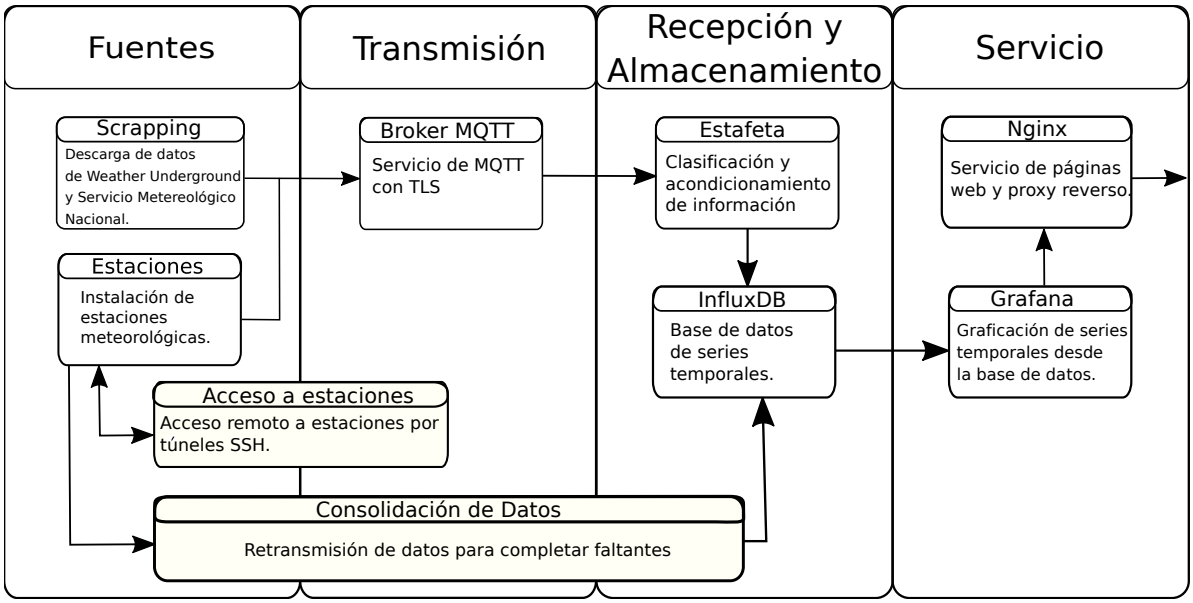

Figura 1.1: Esquema del sistema diseñado.

- **Fuentes de datos**: en esta etapa los datos son generados. Se pueden agregar cualquier tipo de sensor que pueda transmitir los datos por protocolo MQTT. Las fuentes se dividieron en dos grupos, las estaciones meteorológicas instaladas y la recolección de datos por medio de la técnica de *web scraping*. La técnica de *web scraping*(recolección de datos de páginas web públicas mediante programas).

- **Transmisión**: esta sección se encarga de implementar y mantener en funcionamiento el sistema de transmisión de datos. La transmisión se realiza por medio de protocolo MQTT[2] sobre la red de internet. Es necesario que las fuentes tengan acceso a internet para poder transmitir los datos. Este esquema requiere de un servidor MQTT llamado *broker* con dirección de IP pública.

- **Recepción y Almacenamiento**: los datos que se transmiten por MQTT son recibidos por un servidor que está preparado para organizar la información recibida y almacenarla en una base de datos. El programa diseñado para recibir los datos se llamó `Estafeta`, fue escrito el lenguaje Python. Por cada sensor con un formato diferente que se agrega a la red solo es necesario agregar un pequeño módulo al programa `Estafeta`. La base de datos utilizada fue una `Influxdb` la cual está específicamente diseñada para almacenar series temporales.

- **Servicio**: uno de los objetivos del proyecto es servir los datos en una página web pública y sumar estaciones a la red. Se utilizó el programa `Nginx` para servir una página HTML con un link a los datos públicos y ofrecer además un instructivo de

[2]MQTT es un protocolo de comunicación de capa 4 según el modelo OSI, es uno de los protocolos más utilizados en aplicaciones de IOT (*Internet of Things*)

cómo sumar estaciónes a la red. Los datos se públican con el programa `Grafana` que lee los datos directamente de la base de datos `InfluxDB`.

- **Acceso a Estaciones**: las estaciones meteorológicas instaladas transmiten los datos a través de una computadora *raspberry pi* que se conecta a internet. Estas computadoras estan en lugares remotos por lo que son administradas por internet utilizando el protocolo SSH[3].

- **Consolidación de datos**: por distintas razones puede ocurrir que no todos los datos transmitidos lleguen hasta la base de datos `InfluxDB`. Para mantener la base de datos completa se implementó un sistema que retransmite los datos diariamente y verifica si hay datos faltantes.

La tesis se dividirá en siete capitulos, uno para la introducción uno para la conclusión y uno por cada una de las secciones descriptas. La consolidación de datos se describe en el **Apéndice D**.

1.4. Implementación

Como fuentes de datos se instalaron 3 estaciones meteorológicas, una en la dependencia de PN en Puerto Pañuelo y una en la dependencia de PN en la costa del Lago Gutiérrez y la otra en el CAB. Estas son controladas por una computadora *raspberry pi* que extrae los datos y los envía por MQTT. Para la instalación de las estaciones en el Parque Nacional se realizó un convenio con las autoridades de PN.

Para la sección de Transmisión de datos se alquiló un servidor con IP pública fija a la empresa Digital Ocean. El servidor se llamó *central-comm* y su dirección IP es 159.89.80.129. El servidor se utilizó como *broker* MQTT y para crear túneles SSH que permiten acceder a las *raspberrys pi* de manera remota.

Tanto la sección Recepción y Almacenamiento como la sección Servicio se implementaron en el mismo servidor. Este servidor también se alquiló a la empresa Digita Ocean y se llamó *meteo-server* su IP pública es 138.197.228.125. En el servidor se corre el programa `Estafeta`, la base de datos `InfluxDB`, el servidor de páginas web `Nginx` y el graficador de bases de datos `Grafana`.

1.5. Métodos de trabajo

El trabajo generó un volumen considerable de programas, instructivos y registros. Para mantener la organización, resguardar la información y poder compartirla se utilizó la plata-

[3]Secure Shell Protocol, es un estándar para abrir una terminal de comandos de manera remota asegurando confidencialidad, integridad y autenticidad de la información trasmitida.

forma `Gitlab`. Este consiste en una página web que ofrece el servicio de *repositorios-git* en internet. Un repositorio es una estructura de directorios que se utiliza para guardar archivos y guardar distintas versiones de los mismos. La herramienta `git` se utiliza en proyectos de desarrollo de software para trabajar en conjunto, permite crear y manejar versiones de un programa. La plataforma también permite crear una *wiki* y un calendario de asuntos. Una *wiki* es una página web con formato tipo *wikipedia*.

Se utilizó el repositorio git para almacenar toda la información generada. Se utilizaron los programas de línea de comandos *git*, para manejar el repositorio. Además se creó una *wiki* para generar la documentación del proyecto. En **Figura 1.2** se puede observar la wiki del proyecto.

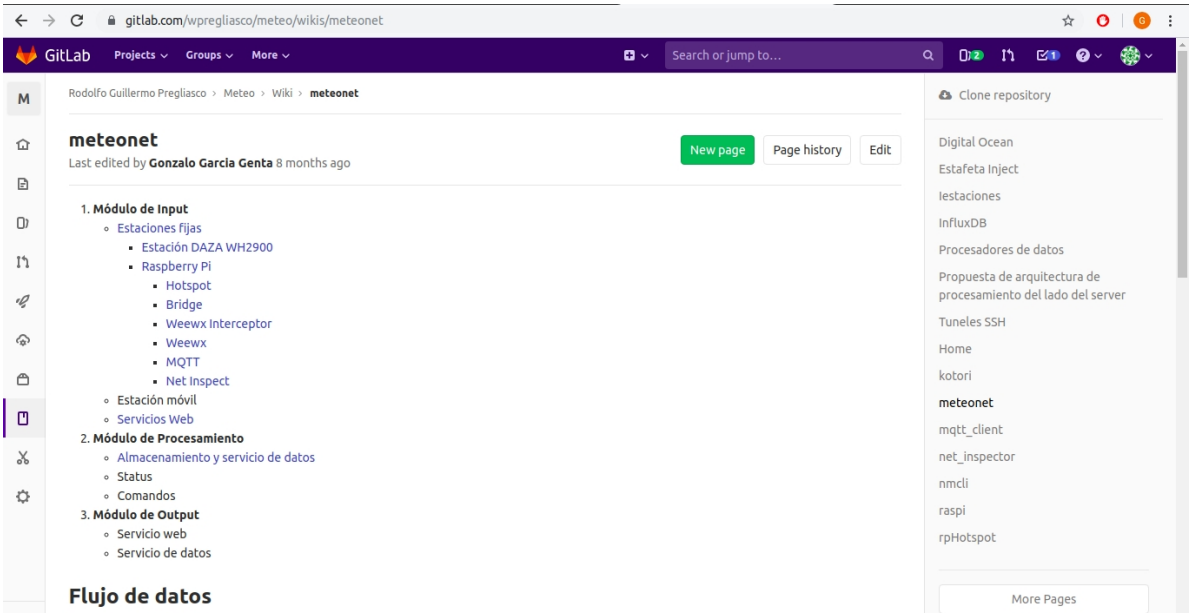

Figura 1.2: Wiki del proyecto generada con la plataforma gitlab.

El manejo de servidores, raspberrys y programas requiere una gran cantidad de contraseñas. Para mantener un nivel aceptable de seguridad se utilizó el programa `KeePassXC`, el cual está disponible directamente desde el repositorio de Ubuntu. Este programa permite generar y administrar contraseñas de manera segura. El programa se utilizó para generar contraseñas aleatorias y almacenarlas de manera segura.

Capítulo 2

Fuentes de datos

En esta sección describimos cómo se generan y transmiten los datos que ingresan a la red. La transmisión de datos se realizó por protocolo MQTT sobre la red de internet. Los generadores de datos pueden ser cualquier tipo de sensor capaz de enviar los datos medidos por MQTT, esto cumple con el Criterio de Reutilización ya que la red se puede aprovechar para utilizar sensores que no sean exclusivamente meteorológicos.

La red implementada está conformada por estaciones meteorológicas y por los datos recolectados mediante *web scraping*. Las estaciones meteorológicas son estaciones comerciales que tienen un costo de aproximadamente US$100. A cada estación se debe agregar una computadora *raspberry pi* para transmitir los datos medidos por protocolo MQTT, ya que las estaciones no tienen esta capacidad. Se instalaron tres estaciones, una en Puerto Pañuelo, una en el Centro Atómico y otra en la costa del Lago Gutiérrez. El `scrapper` es un programa que implementa la técnica de *web scraping*, esta técnica consiste en recolectar datos de páginas web mediante programas. Se implementó un programa en Python que recolecta datos de la página del Servicio Meteorológico Nacional y de la página *weather underground*. La página *weather underground* recolecta datos de estaciones meteorológicas de aficionados y publica los datos en la web, al día de hoy tiene 6 estaciones activas en Bariloche.

Para la transmisión de datos por MQTT cada sensor debe tener un usuario, contraseña, tópico y el certificado del Broker. Estos datos se asignan cuando el sensor se registra en la red. A continuación se describen las estaciones instaladas y el programa `scrapper`.

En la tabla 2.1 se listan las fuentes de datos implementadas.

Fuentes		Ubicación	Modelo
Estaciones Instaladas		CAB	WH-2900
		P. Pañuelo	WH-2900
		Los Coihues	WH-1080
scraping	Weather Underground	Colonia Suiza	Desconocido
		Las Marías	
		Villa los Coihues	
		Las Victorias	
		Perito Moreno	
		La Cumbre	
		Terminal de Omnibus	
		Perito Moreno	
		Las Margaritas x2	
	Servicio Meteorológico Nacional	Aeropuerto	

Tabla 2.1: Lista de las fuentes de datos.

2.1. Estaciones meteorológicas

Para el proyecto se instalaron tres estaciones meteorológicas, dos modelo DAZA WH-2900 y una Fine Offset WH-1080. Estas estaciones no están preparadas para enviar los datos por protocolo MQTT. Para extraer los datos de las estaciones y enviarlos por MQTT se instaló una computadora *raspberry* con cada estación. La computadora *raspberry* es una PC de bajo costo con capacidades reducidas, se utilizó para extraer los datos de las estaciones y enviarlos por protocolo MQTT. Para extraer los datos se utilizó el software WeeWX, este está diseñado para recolectar datos de estaciones meteorológicas y soporta el envío de los datos por MQTT. El software WeeWX es de gran utilidad dado que es capaz de comunicarse con la gran mayoría de las estaciones meteorológicas comerciales, a pesar que hay una gran variedad de protocolos e interfaces posibles.

Dos de las estaciones se instalaron en dependencias de la administración del PN Nahuel Huapi. Se generó un convenio con la Intendencia del PN Nahuel Huapi para instalarlas y utilizar las redes wifi existentes para enviar los datos por internet. La generación del convenio requirió reuniones con la Intendencia de PN y presentación de notas formales. Una estación se instaló en la dependencia de PN en el Puerto Pañuelo del Lago Nahuel Huapi y la otra estación se instaló en la dependencia en la costa del Lago Gutierrez.

La tercera estación se instaló en Centro Atómico, está tuvo inconvenientes para utilizar el servicio de MQTT a causa del *firewall*. Para solucionar este problema se utilizaron túneles SSH.

2.1.1. Recuperación y transmisión de datos

Las computadoras *raspberry* son computadoras de bajo costo, aproximadamente US$50 para el modelo 3 B+. Se utilizó una computadora por cada estación para extraer los datos y luego enviarlos por protocolo MQTT. Estas computadoras tienen capacidades reducidas, se utilizó el modelo 3 B+ que tiene un procesador ARM de 1.4 GHz de dos nucleos, 1GB de memoria RAM, una salida HDMI, 4 puertos USB 2.0, interfaz wifi IEEE 802.11.ac, Bluetooth LE e intefaz Gigabit ethernet. Las *raspberrys* no poseen disco duro, el almacenamiento se realiza en una tarjeta micro sd. El fabricante recomienda utilizar el sistema operativo *Raspbian* el cual es una version de Debian GNU/Linux diseñado especialmente para *raspberrys*. Raspbian se distribuye bajo una licencia GPL (GNU General Public License) la cual garantiza la libertad de usar, compartir, estudiar y compartir el software. En **Figura 2.1** se muestra una imágen de una raspberry pi 3 B+ como las utilizadas.

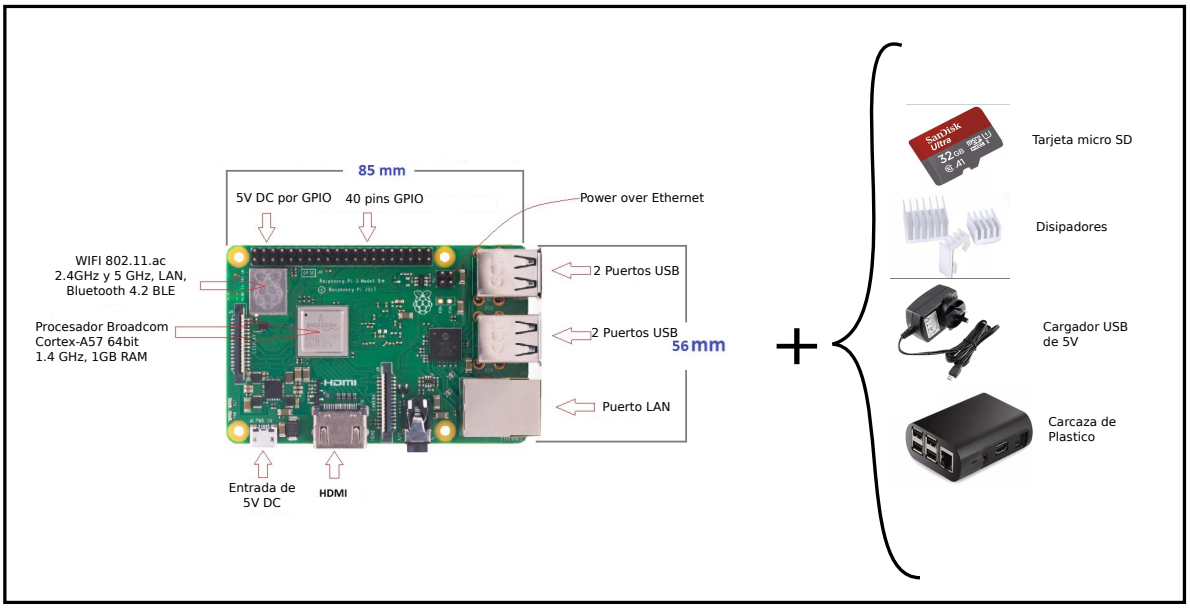

Figura 2.1: Raspberry pi 3 B+ con tarjeta de memoria, disipadores, cargador y carcaza.

La carcaza se utiliza para proteger el hardware que no tiene cobertura. Una de las roturas de hardware más común es las raspberrys es la tarjeta micro SD, si se realizan muchas escrituras y lecturas a la tarjeta de memoria la vida útil resulta corta.

El modelo pi 3 de las raspberrys tiene un procesador con mayor frecuencia de reloj que el modelo pi 2. El inconveniente con este procesador es que sobrecalienta. Para evitar que esto ocurra se baja la frecuencia del reloj automáticamente cuando éste alcanza altas temperaturas. Es necesario agregar un disipador a la raspberry para obtener un mayor rendimiento del procesador. Ocurrió que una raspberry sin disipador que funcionó durante un período prolongado de tiempo como *hotspot* de wifi se quemó.

Se utilizó el mismo sistema operativo para todas las raspberrys para evitar problemas ocasionados por utilizar distintas versiones. Se utilizó la version Stretch 9.6 de Raspbian, la

misma se puede descargar del repositorio con url *http://downloads.raspberrypi.org- /raspbian/images/*. Este sistema operativo es una versión de `Debian` modificada especialmente para raspberrys. Para instalar el sistema operativo en una raspberry se puede seguir el instructivo que se encuentra en el **Apéndice B**.

Las diferentes estaciones poseen distintos protocolos para el envío de datos o utilizan distintas interfaces. Para la extracción de los datos de las estaciones se utilizó el software *WeeWX*.Este es un software gratuito, de código abierto escrito en Python 2.7 que interactua con estaciones meteorológicas y brinda la posibilidad de producir graficos, paginas HTML y compartir los datos por MQTT, FTP y rsync. Esta diseñado para funcionar en Linux y en macOS. Es capaz de comunicarse con casi todas las estaciones meteorológicas comerciales. En este trabajo se utilizó para extraer los datos de las estaciones, guardar la información en una base de datos SQL y enviar los datos por protocolo MQTT. Para instalar y configurar WeeWX se puede seguir el instructivo que se encuentra en el **Apéndice A**.

A continuación se describen cada una de las estaciones instaladas.

2.1.2. Estación en Puerto Pañuelo

Se realizó un convenio con PN y se instaló una estación meteorológica DAZA WH-2900 en la dependencia de PN en el Puerto Pañuelo del Lago Nahuel Huapi. La estación se fijó en un árbol seco aproximadamente a cuatro metros de altura del suelo. Para la instalación se fabricó un soporte de metal en los talleres del Centro Atómico. Se solicitó ayuda a una persona con experiencia para colgar la estación del árbol.

La estación Daza WH-2900 es una estación meteorológica de uso hogareño. En **Figura 2.2** se puede observar una foto de la estación. El costo es de aproximadamente US$125, lo que satisface con el criterio de mantener un costo bajo. Como se observa en la imagen esta compuesta por la estación y por una base que muestra los datos en una pantalla. La base recibe los datos de la estación por RF, los muestra y los retransmite por wifi.

Figura 2.2: Estación DAZA WH-2900.

La estación es capaz de medir temperatura, humedad, radiación solar, viento, dirección

del viento y lluvia acumulada. La base es capaz de medir temperatura, humedad y presión. Al medir temperatura y humedad en la estación y en la base se obtienen valores de la temperatura en dos lugares distintos. Usualmente a la temperatura en la base se la llama temperatura interior, lo mismo sucede con la humedad. En la pantalla se muestran toda la información medida.

La base se conecta a una red wifi y envía los datos por protocolo HTTP a un dominio público en internet. Para conectarla a la red wifi y especificar el dominio se utiliza la aplicación para celulares Android WSTool. Los dominios disponibles en la aplicación son *weatherunderground.com* y *weathercloud.com*. Estos dominios están preparados para recibir los datos por protocolo HTTP y luego los brindan en su página web.

Para obtener los datos se utilizó una raspberry con el programa WeeWX. Es necesario instanciar una red wifi en la raspberry con un servidor DHCP y un servidor DNS. En el **Apéndice A** en la sección A.2 se describe el proceso para la recuperación de los datos.

La raspberry utilizada puede ser accedida de manera remota como se indica en el **Capítulo 6**.

2.1.3. Estación en Centro Atómico

Se instaló una estación en el Centro Atómico sobre una torre de hierro de aproximadamente 4 metros de altura. La estación es marca DAZA modelo WH-2900. Se fabricó un soporte de metal y se utilizaron elementos de seguridad para subir la estación a la torre.

La recuperación de datos se realizó de igual manera que para la estación en Puerto Pañuelo. Se utiliza una raspberry para servir una red wifi con servidor de DNS propio y hacer que la estación envíe los datos hacia la raspberry. Una vez que la estación envía los datos hacia la raspberry esta los recibe con el software WeeWX y los envía por MQTT.

El protocolo MQTT funciona en el puerto TCP 8883, pero ocurre que en la red del Centro Atómico existe un *firewall* que no permite las conexiones salientes hacia puertos TCP 8883. Este problema se solucionó pasando la comunicación por un túnel SSH que va desde la raspberry hasta el servidor donde se encuentra el *broker* MQTT. En la **Capítulo 6** (sección 6.3) se detalla la operación.

En un momento la interfaz wifi de la base dejó de funcionar, por lo que no se pudieron recuperar más los datos de la estación. Para solucionar este problema se utilizó una SDR con conexion USB. SDR es capaz de decodificar comunicaciones de RF en una variedad de frecuencias de portadora distintas y con una variedad de modulaciones diferentes. La SDR utilizada fue una comercial de las más comunes, su uso principal es para decodificar señales de televisión satelital con protocolo DVB[1]. En **Figura 2.3** se puede observar una imagen de la SDR utilizada.

[1]Digital Video Broadcasting, es uno de los protocolos más utilizados para la transmisión de televisión digital por satélite

Figura 2.3: SDR utilizada para recuperar datos.

Para manejar la SDR se utilizó la biblioteca `rtl_433`, de código abierto y diseñada para manejar este tipo de dispositivos. Así se pueden decodificar una variedad de dispositivos comerciales, como estaciones meteorológicas y controles remotos.

Una vez instalada la biblioteca se puede comenzar a recibir los datos trasmitidos por la estación con el comando `rtl_433 -F json -M utc -R 78`. La opción `-R 78` es el código para indicar que el dispositivo que se desea escuchar es una estación DAZA WH-2900. La opción `-F json` indica que la información recibida se imprime en formato *json*, la opción `-M utc` indica que se debe agregar la hora a la información recibida en formato *UTC*. Luego de correr el comando en una terminal se empiezan a recibir datos cada 5 minutos aproximadamente.

Para enviar los datos por MQTT se utilizó un cliente de línea de comandos. Corriendo el comando `rtl_433 -F json -M utc -R 78 | mosquitto_pub -t topic/ -u usuario -P contraseña -p puerto -h direccion --cafile path/al/certificado` se reciben los datos de la estación y se envían por MQTT. Es necesario especificar el usuario, la contraseña, el tópico, el certificado del broker y la dirección ip del broker.

Para dejar al comando funcionando se instaló un demonio en la raspberry que corre el comando. En caso de que el proceso se termine por algún error lo reinicia. En el **Apéndice C** se pueden ver en detalle las configuraciones utilizadas.

La raspberry utilizada puede ser accedida de manera remota como se indica en el **Capítulo 6**.

2.1.4. Estación en Lago Gutiérrez

La Fine Offset WH-1080 es una estación meteorológica de uso hogareño. La estación está compuesta por la estación misma que realiza las mediciones y por una base que recibe los datos. La base recibe los datos por RF, los muestra en una pantalla y los brinda por interfaz USB. La estación es capaz de medir temperatura, humedad, radiación solar, viento y dirección del viento. La base es capaz de medir temperatura, humedad y presión, estas mediciónes se diferencian llamandolas interiores. En la base se muestran los datos.

Se utilizó una raspberry como se describe en la sección 2.1.1 para recuperar los datos y transmitirlos por USB. La raspberry utilizada puede ser accedida de manera remota como se indica en el **Capítulo 6**.

2.2. Scraping

Se utilizaron dos programas diferentes: `Selenium` y `Scrapy`. `Selenium` es un controlador de navegadores web, este se descartó por consumir muchos recursos de la computadora. `Scrapy` es un navegador web y está diseñado para extraer información de páginas. Se utilizó este último por consumir menos recursos y ser más veloz.

`Scrapy` está escrito en Python y es de código abierto. Se puede instalar directamente utilizando el instalador de paquetes para Python *pip*. El programa funciona con *spiders*, estos son clases de Python estandarizadas que extraen información de una página web y la guardan en un archivo. Cada *spider* puede correr en un *thread* independiente, de manera que se pueden correr múltiples *spiders* simultáneamente.

Se creó un *spider* para la página *weather underground* y otro para la página del Servicio Meteorológico Nacional. Finalmente se creó un programa en Python que corre los *spiders* y envía la información por MQTT. El programa se corre como un *crontab* en una computadora dentro del CAB. Esto es un comando que se guarda en el archivo **/etc/crontab** y se especifica cuando se debe correr. Se especificó que se corra el programa cada 5 minutos. En el **Apéndice E** se describe el código utilizado.

Capítulo 3

Transmisión

Esta sección cumple la función de comunicar las fuentes de datos con el programa `Estafeta` que recibe toda la información, la acondiciona y almacena en una base de datos. La sección está compuesta por un servidor de MQTT que permite la comunicación a través de internet. MQTT es un protocolo de comunicaciónes útil para este tipo de aplicaciones.

Se alquiló un servidor con una dirección IP pública y fija para dar el servicio de broker de MQTT. El servidor se alquiló a la empresa Digital Ocean la cual brinda el servicio de servidores que se acceden por internet llamados *droplets*. Este se configuró el servidor para que funcione como servidor MQTT en su dirección de IP pública.

3.1. Protocolo MQTT

El protocolo MQTT está definido por el estandar ISO/IEC PRF 20922, es un protocolo de comunicaciones de capa 4 según el modelo OSI[1]. Usualmente se utiliza sobre una red TCP/IP como internet. Está diseñado para comunicaciones con poco ancho de banda, alta latencia y poca confiabilidad. Por su diseño es muy utilizado en aplicaciones M2M donde alguna de las partes puede estar en un lugar remoto o detrás de un link satelital. Los ejemplos usuales de utilización son aplicaciones de IOT o sensores en áreas remotas como pozos petroleros. Las características del protocolo cuadran muy bien con los requerimientos del proyecto ya que las estaciones meteorológicas están dispersas en un área geográfica grande y suelen estar conectadas a internet a través de redes de mala calidad.

El protocolo funciona con dos tipos de entidades, el *broker* y los clientes. Cada cliente puede publicar en tópicos y subscribirse a tópicos, esto sirve para crear canales de comunicación. Una públicación es el envío de un mensaje a tópico específico. Una subscripción a un tópico es una solicitud para recibir todos los mensajes que se publiquen en ese tópico. El broker recibe todos los mensajes de las publicaciones y los reenvía a los clientes que están

[1]El modelo OSI (Open Systems Interconnection model) está definido por el estándar ISO/IEC 7498-1. Es un modelo conceptual que organiza los sistemas de comunicaciones por capas con funciones especificas.

subscriptos a los tópicos correspondientes.

Todas las conexiones se inician desde los clientes hacia el broker, de modo que no es necesario que ningún cliente tenga una dirección IP pública. Además evita que el broker tenga que guardar la dirección de los clientes y mantenga un registro de su estado de conexión a la red.

El protocolo tiene como opción la implementación de usuarios con contraseñas y de acceso restringido a tópicos. El acceso restringido a tópicos permite que cada usuario tenga acceso a publicar y subscribirse en determinados tópicos. Al restringir el acceso a tópicos se generan canales de comunicación dedicados para cada cliente, de esta manera se mantiene la organización de la red y se controla el comportamiento de cada cliente. Si un cliente utiliza indebidamente la red o se compromete la seguridad de una fuente de datos su comportamiento no va a afectar la información que envían el resto de los clientes.

El protocolo permite la utilización de protocolo TLS para asegurar la autenticidad, confidencialidad e integridad de toda la información enviada. Por si solo el protocolo MQTT sin añadir TLS es inseguro, toda la información se envía en texto plano, incluidas las contraseñas de los usuarios. Es necesaria la implementación de TLS para mantener un mínimo de seguridad en la red.

3.2. Implementación

Para la implementación del sistema de transmisión se alquiló un servidor con dirección IP pública y fija. En el servidor se instanció un servidor de MQTT con protocolo TLS, acceso restringido a tópicos y usuarios con contraseñas. A continuación el detalle de la implementación del servidor.

3.2.1. Servidor de MQTT

Se contrató el servicio de servidores virtuales de la empresa *Digital Ocean* denominado *droplets*. Desde el lado del cliente un *droplet* es una computadora que se encuentra en un lugar remoto y se puede acceder por internet por medio de una dirección IP fija. Los droplets disponibles para alquilar poseen distintas capacidades y precios.

El droplet alquilado cuenta con 1GB de memoria RAM, 25GB de memoria para almacenamiento, 1 CPU, 1000GB disponibles para transferencias por internet y el costo es de 5 US$ mensuales. Al solicitar el servicio se elige el sistema operativo que se desea para la computadora, se puede elegir entre Ubuntu, Debian, FreeBSD, Fedora y Centos. El sistema operativo elegido fue Ubuntu versión 18.04 de 64 bits (última versión LTS), se eligió Ubuntu porque es el sistema operativo usual en el Laboratorio de Física Forense y tiene una licencia GPL. Las capacidades del servidor se consideraron suficientes para la red implementada. En caso de necesitar más recursos se pueden agregar sin problema.

Cuando se alquila el droplet se proveé una clave SSH para acceder al servidor. En la sección 6 se explica el funcionamiento del acceso remoto por SSH. Una vez especificada la clave SSH se procede con el pago. Luego del pago el servidor queda funcionando y listo para ser utilizado. Para acceder al servidor se utiliza un cliente SSH, solo es necesario poseer la clave SSH que se especificó y conocer la dirección IP del servidor la cual provee Digital Ocean. El nombre del servidor elegido fue *central-comm*, haciendo referencia a la función del servidor de comunicar los sensores con el programa `Estafeta`. En la sección 3.2.4 se detallan las configuraciones implementadas en el servidor.

3.2.2. Tópicos, usuarios y lista de control de acceso

La estructura de tópicos es similar a la de directorios, esta dada por palabras separadas por " / ". Cada públicación debe estar acompañada de un tópico y cada subscripción debe estar acompañada por un tópico o conjunto de tópicos. Para especificar un conjunto de tópicos se puede utilizar el caracter "#", por ejemplo la subscripción a "meteo/# " subscribe a todas las publicaciones que comiencen con "meteo/".

Para el proyecto se definió una estructura de tópicos particular para organizar la información de las estaciones. El tópico se definió según *proyecto/tipo/red/subred/estación/puerto*.

- **proyecto:** engloba a todo el trabajo realizado y se llama *meteo*.
- **tipo:** define qué tipo de datos se envían, es útil porque cada tipo define una base de datos distinta, manteniendo las bases de datos mejor organizadas. Al momento existe el tipo *station* para los datos de estaciones meteorológicas.
- **red:** clasifica la información según el administrador, al momento se utiliza la red *meteo* para datos de nuestras estaciones y la red *scrap* para datos recolectados de la web.
- **subred:** clasifica la información geográficamente dentro de cada red, nuestra red meteo recibe datos de barrio Los Coihues, Puerto Pañuelo y del CAB. A las locaciónes mencionadas se les asignó el nombre de subred *coihues*, *panuelo* y *CAB*.
- **estación:** define a cada sensor en particular para mantener un tópico por cada uno, usualmente se numeran.
- **puerto:** divide distintos canales para un mismo sensor, al momento se usa el puerto *data* para mandar información y el puerto *ping* para mandar señales de control.

Por ejemplo la estación meteorológica instalada en Puerto Pañuelo manda datos al tópico *meteo/station/meteo/panuelo/01/data*.

El protocolo MQTT permite la utilización de usuarios con contraseñas lo cual permite controlar quién se conecta a la red y además permite controlar quién se subscribe y pública

en cada tópico. Para controlar se utiliza una ACL (Access Control List), la cual permite definir qué usuarios pueden publicar y subscribirse a cada tópico.

En el broker MQTT se implementaron usuarios con contraseña, se creó un usuario llamado *admin* con capacidad para publicar y subscribirse a *meteo/#* y se crearon usuarios por cada red, de manera que cada administrador tenga su usuario para conectarse sólo a su red. Por ejemplo la estación en Puerto Pañuelo utiliza el usuario *meteo* el cual está autorizado a publicar en el tópico *meteo/station/meteo*.

La utilización de la lista de control de acceso permite que un usuario que tiene un comportamiento incorrecto no altere el funcionamiento de toda la red. Cada usuario tiene su propio canal para enviar mensajes de manera que no afecta al resto.

3.2.3. Encriptación con TLS

El protocolo MQTT permite la encriptación y autenticación de toda la información transmitida utilizando el protocolo TLS (Transport Layer Security). El protocolo TLS es el protocolo de encriptación más utilizado, se utiliza para la encriptación de tráfico web en HTTPS y es capaz de asegurar la autenticidad, integridad y confidencialidad de la información enviada. En su última versión el protocolo asegura *forward secrecy* en todas las condiciones.

- **Integridad**: la información recibida es idéntica a la enviada.
- **Confidencialidad**: la información recibida solo es legible por el emisor y el receptor.
- **Autenticación**: el emisor del mensaje es realmente quien dice ser.
- **Forward Secrecy**: si en el futuro la clave privada es descubierta, la confidencialidad de la información se sigue manteniendo.

En **Figura 3.1** se puede observar un esquema del funcionamiento del protocolo TLS.

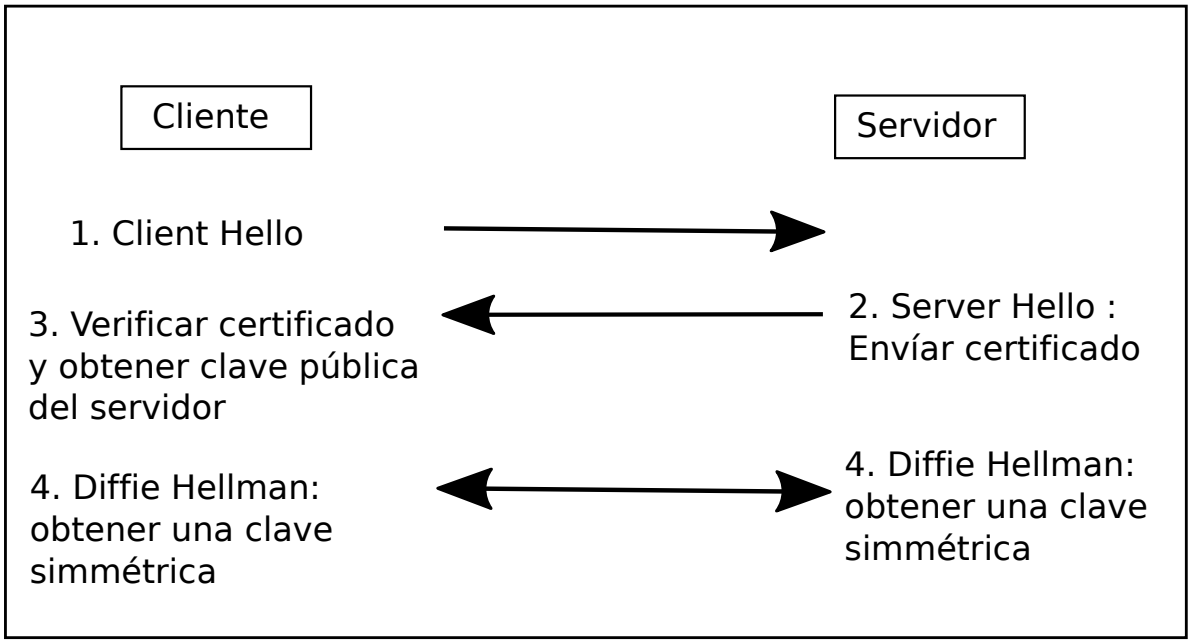

Figura 3.1: Esquema del protocolo TLS.

1 El cliente inicia una conexión hacia el servidor. En este primer mensaje incluye información de qué versión del protocolo soporta.

2 El servidor responde con la versión de TLS que desea utilizar. También incluye su certificado, este certificado incluye su clave pública y su dominio.

3 El cliente verifica el certificado que recibe con la clave pública de la Autoridad Certificante. El cliente debe haber conseguido la clave de la AC por algún medio seguro. Una vez que se verifica el certificado el cliente puede estar seguro que la clave pública que recibió es del ente correspondiente.

4 Cliente y servidor implementan el algoritmo de Diffie-Hellman, este algoritmo permite crear una clave simétrica que solo es conocida por el cliente y el servidor. La clave pública del servidor se utiliza para firmar la información transmitida para asegurar la autenticidad e integridad de esta.

5 Una vez que se obtuvo la clave simétrica se puede comenzar a transmitir información encriptandola con esta clave. Además de encriptar se añade una firma para asegurar autenticidad e integridad.

Resta explicar qué es una clave simétrica y asimétrica, qué es un certificado, qué implica firmar un mensaje, qué es una Autoridad Certificante y qué es el algoritmo Diffie-Hellman.

¿Qué es una clave simétrica y asimétrica? Una clave simétrica es una única clave que se utiliza de igual manera para encriptar y desencriptar. Las claves asimétricas funcionan

de a pares, una se utiliza para encriptar y la otra para desencriptar. Las claves asimétricas reciben el nombre de públicas y privadas, la clave pública puede ser poseída por cualquier persona y la clave privada debe ser poseída solo por el dueño. Las claves asimétricas requieren una cantidad de procesamiento muy grande para encriptar y desencriptar, por esta razón no son utilizadas para encriptar sino que son utilizadas para firmar. Las claves simétricas requieren una capacidad mucho menor de procesamiento para encriptar y desencriptar, por lo que son usadas para encriptar todo el trafico de información. Las claves asimétricas más comunes son las tipo RSA[2], para que sean seguras estas deben tener una longitud de al menos 2048 bits. Las claves simétricas más comunes son las tipo DES3[3] o las AES[4], estas suelen tener una longitud de 128 bits.

¿Qué es un certificado y qué implica firmar un mensaje? El certificado cumple la función de transportar una clave pública de un servidor y asegurar la autenticidad e integridad de la clave pública. Un certificado es un archivo que incluye: la clave pública de una Autoridad Certificante, la clave pública de un servidor, el dominio del servidor y la firma de la Autoridad Certificante. Los certificados que se usan en TLS están definido por la norma X509 de la ITU-T En **Figura 3.2** se puede observar el proceso de creación de un certificado.

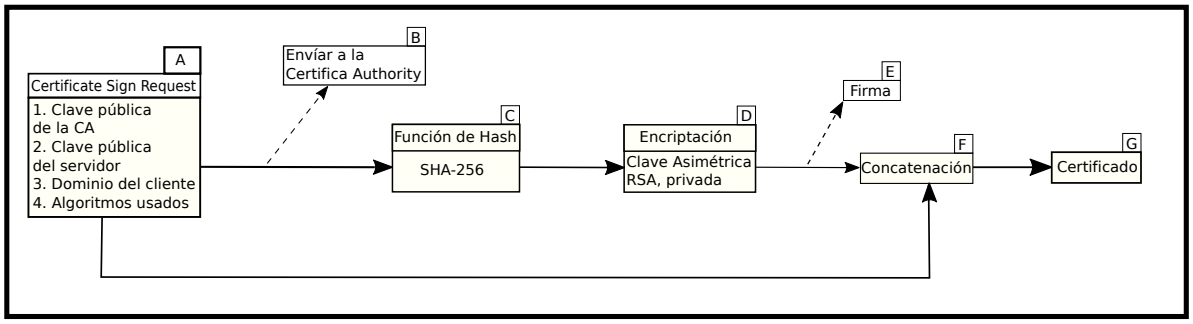

Figura 3.2: Esquema de fabricación de un certificado X509.

A El cliente arma un archivo denominado *Certificate sign Request*. El archivo incluye: la clave pública de la Autoridad Certificante, la clave pública del servidor, el dominio del servidor y los algoritmos utilizados en el certificado.

B El cliente envía el *Certificate sign Request* a la Autoridad Certificante.

C La autoridad calcula un hash del *Certificate sign Request*. Usualmente se utiliza la función de hash SHA-256.

[2]Rivest–Shamir–Adleman, algoritmo para la utilización y creación de claves asimétricas.

[3]Triple Data Encryption Standard, algoritmo de encriptación simétrica diseñado por IBM, definido por norma RFC 1851.

[4]Advanced Encryption Standard, es un algoritmo de acriptación simétrica desarrollado por NIST, está definido en la norma ISO/IEC 18033-3

D La autoridad certificante encripta el hash calculado utilizando su clave privada. Usualmente se utiliza una clave del tipo RSA.

E El archivo resultante del paso D se llama la firma del certificado.

F La Autoridad Certificante concatena el *Certificate sign Request* con la firma del mismo.

G El archivo resultante de la concatenación es el certificado del servidor. La Autoridad Certificante devuelve el certificado al cliente.

El ente que firma, en este caso la Autoridad Certificante, debe guardar su clave privada en resguardo y entregar su clave pública a cualquiera que la requiera. En **Figura 3.3** se puede observar el proceso de verificación de la firma de un certificado.

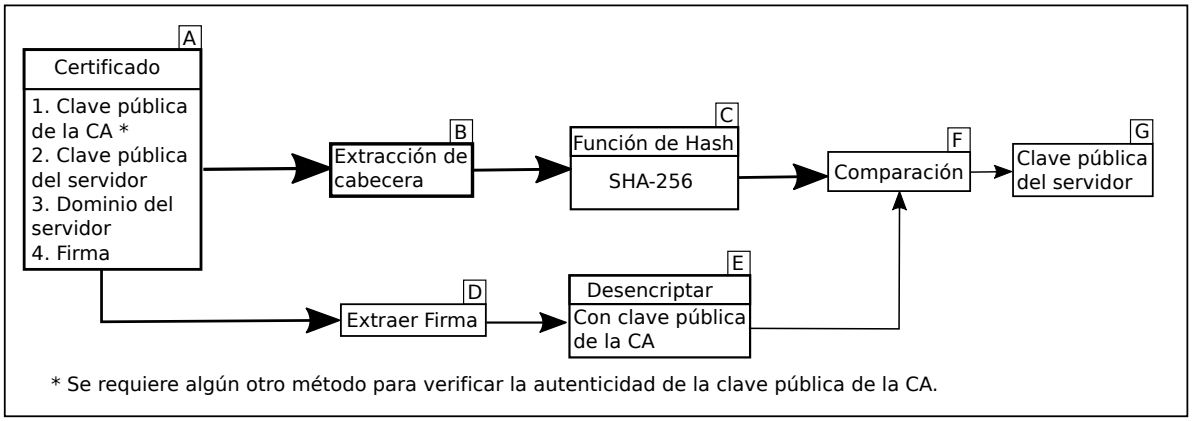

Figura 3.3: Verificación de un certificado X509.

A Se recibe el certificado del servidor.

B Se separa la cabecera del certificado. La cabecera está compuesta por todo el certificado menos la firma.

C Se calcula el hash de la cabecera, usualmente es la función SHA-256. La función esta definida en la cabecera.

D Se extrae la firma del certificado.

E Se desencripta la firma utilizando la clave pública de la Autoridad Certificante. Usualmente es una clave tipo RSA.

F Se compara el hash de la cabecera con la firma desencriptada.

G Si ambas firmas coinciden entonces el certificado es valido y la clave pública del servidor incluida en el certificado es valida. Se debe verificar que la clave pública de la Autoridad Certificante es valida.

En resúmen, una firma es encriptar un mensaje con la clave privada de uno y enviar el mensaje original más la firma. El receptor del mensaje desencripta la parte encriptada con la clave pública. Si se cumple que:

- La firma del mensaje y la calculada coinciden.
- El emisor cuidó bien su clave privada y solo él la conoce.
- El receptor recibió la clave pública del emisor por algún medio seguro y puede asegurar la autenticidad e integridad de la misma.

Entonces el receptor puede asegurar que el emisor del mensaje es quien cree que es y que el mensaje llego de manera integra.

¿Qué es una Autoridad Certificante? El protocolo TLS requiere adquirir de manera segura el certificado de la Autoridad Certificante. Una AC es un ente que se dedica a firmar certificados. La existencia de las AC reduce el número de certificados que uno necesita obtener de manera segura. Los certificados de las AC vienen preinstalados en los sistemas operativos.

¿Qué es el algoritmo de Diffie-Hellman? En las versiones actuales de TLS (TLS 1.3) se obtiene una clave simétrica a partir del algoritmo Diffie-Hellman. Este algoritmo es susceptible a un ataque del tipo MiT[5], para evitar este ataque se utiliza un par de claves público-privada. Las claves público-privada se utilizan para firmar los mensajes transmitidos de manera de asegurar autenticidad e integridad. Si uno asegura la integridad y autenticidad de la información entonces puede asegurar que no está bajo un ataque tipo MiT.

Para la implementación de TLS es necesario crear la AC y el certificado que va a utilizar el broker. Para esto se utilizó la biblioteca *Openssl* que viene preinstalada en Ubuntu. Se corren los siguientes comandos:

1 **Crear clave privada de la AC:**
`openssl genrsa -out ca.key -des3 2048`
Se genera una clave de RSA de 2048 bits de largo y luego se encripta con el algoritmo DES3 para mantenerla segura. El largo recomendado por NIST para una clave RSA es de 2048 bits. El comando solicita una clave para encriptar la clave con el algoritmo 3DES, es necesario recordar esta clave para poder volver a utilizar la clave privada.

[5]Man in the Middle, es un tipo de ataque criptográfico en el cual el atacante es capaz de alterar todos los mensajes enviados entre dos personas sin que estas lo noten.

2 **Crear certificado autofirmado de la AC:**
`openssl req -new -x509 -days 360 -key ca.key -out ca.crt`
El certificado tiene una validez de 360 días. Se solicitan campos para informar a quien pertenece la AC. En esta paso se genera la clave púlica de la AC.

3 **Crear clave privada del broker:**
`openssl genrsa -out server.key 2048`
Se genera una clave privada de 2048 bits.

4 **Crear el Cerficate sign Request:**
`openssl req -new -out server.csr -key server.key`
El comando solicita información de quien es el broker, el campo *Common Name* se debe completar con el dominio del broker. El dominio del broker es su dirección de IP. En este paso se genera la clave pública del broker.

5 **Crear el certificado del broker:**
`openssl x509 -req -in server.csr -CA ca.crt -CAkey ca.key -CAcreateserial -out server.crt -days 360`
Se firma el certificado del broker por la AC. La validez del certificado es por 360 días.

En **Figura 3.4** se muestra la decodificación del certificado creado para el broker. Para decodificar el certificado y pasarlo a un formato legible se utiliza el comando `openssl x509 -in <path_al_certificado.crt>-text -noout`. La información relevante del certificado está en la clave pública del broker, la firma de la autoridad certificante y en el *Common Name* del *Subject*. El *Subject* es el ente para el cual se firma el certificado y el *Common Name* debe completarse con el dominio del *Subject*. En este certificado el Common Name del Subject es la dirección IP del broker, que es 138.197.228.125.

```
gonchigg@gonchigg:~$ sudo openssl x509 -in gonchigg_cert_auth/server.138.197.228.125.crt -text -noout
Certificate:
    Data:
        Version: 1 (0x0)
        Serial Number:
            3f:6e:dc:3e:cf:79:12:86:d2:1b:98:5c:4f:3d:ca:b8:4d:bb:9e:e1
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = AR, ST = RN, L = BRC, O = FF, OU = FF, CN = gonchigg_ca, emailAddress = garciagentag@gmail.com
        Validity
            Not Before: Oct  4 20:45:26 2019 GMT
            Not After : Sep 28 20:45:26 2020 GMT
        Subject: C = AR, ST = RN, L = BRC, O = IB, OU = FF, CN = 138.197.228.125, emailAddress = garciagenta10@gmail.com
        Subject Public Key Info:
            Public Key Algorithm: rsaEncryption
                RSA Public-Key: (2048 bit)
                Modulus:
                    00:be:94:72:16:63:b0:5d:38:b4:bc:6f:9d:a3:32:
                    42:29:5b:5d:d7:4f:a9:87:90:d2:b6:5b:29:b5:de:
                    d9:a5:91:71:1a:52:20:37:aa:26:c9:d7:f4:71:69:
                    20:32:46:b0:42:d6:8a:04:fe:dc:24:e7:73:8a:d7:
                    e3:e6:00:8e:76:8a:5d:60:9a:95:f3:b5:7c:48:3c:
                    d4:2a:36:f8:85:61:90:97:1c:b0:73:d2:b7:40:78:
                    a6:99:40:01:73:82:ae:96:86:aa:0f:61:28:f0:f0:
                    93:88:df:d3:47:21:52:21:ec:ee:2c:92:ce:fb:84:
                    45:93:83:72:05:02:d6:e6:b8:a4:a4:89:7a:11:f7:
                    26:f0:00:28:78:07:4f:f3:e0:93:0f:a9:cd:42:2f:
                    10:3f:70:05:56:a8:50:ef:f2:a8:a8:33:1e:f4:af:
                    2f:c3:fb:17:a0:ea:74:2f:00:45:8d:e4:df:09:a7:
                    6f:52:6c:68:66:4e:fe:c0:fc:fd:5a:3c:9d:89:20:
                    1d:64:04:b6:76:d2:97:ac:97:a0:53:c7:59:bd:ad:
                    f2:b5:09:77:2b:98:97:1c:20:a4:45:0e:de:f7:b8:
                    b3:59:78:9b:bc:0f:9e:20:ca:c0:cc:da:1d:18:89:
                    1f:85:f0:b4:5f:9a:42:99:c3:63:ac:09:dd:8e:e8:
                    23:37
                Exponent: 65537 (0x10001)
    Signature Algorithm: sha256WithRSAEncryption
         0c:5d:f3:7d:08:c0:2f:5f:3e:97:18:4a:5b:76:20:76:c3:f0:
         4d:54:26:48:64:c9:76:ec:92:d2:7b:3b:29:49:23:93:29:8e:
         f4:20:26:c5:95:f8:0c:50:33:ff:37:01:ac:f1:a1:f4:14:bc:
         9f:e8:cf:3b:5e:5f:0b:f9:ab:85:d9:5c:5f:fe:e8:49:15:15:
```

Figura 3.4: Certificado del Broker de MQTT.

3.2.4. Configuración del broker MQTT

Para el sistema utilizado se utilizó le implementación del protocolo MQTT de la fundación Eclipse que se llama Mosquitto versión 3.1.1. Es de software libre y se puede instalar en Ubuntu corriendo el comando `apt install mosquitto`, luego de instalado se habilita para que comience a correr como servicio con el comando `systemctl enable mosquitto`. Una vez iniciado como servicio se puede detener, iniciar y reiniciar con el comando `systemctl stop|restart|start mosquitto`.

El archivo de configuración se encuentra en **/etc/mosquitto/mosquitto.conf** y se configuró de la siguiente manera:

Archivo de configuracion de mosquitto

```
# /etc/mosquitto/mosquitto.conf
# Instanciar un servicio en la direccion localhost puerto TCP 9993
port 9993

# Instanciar un servicio en la direccion publica puerto TCP 8883
listener 8883

# Se guarda el ultimo mensaje recibido en cada topico
```

```
persistence true
persistence_location /var/lib/mosquitto/

# Path donde se guardan los mensajes de error
log_dest file /var/log/mosquitto/mosquitto.log

# Inlcuir archivos de configuracion
include_dir /etc/mosquitto/conf.d

# Path al certificado de la Autoridad Certificante
cafile /etc/mosquitto/ca_certificates/ca.gonchigg.crt

# Path a la clave privada del broker
keyfile /etc/mosquitto/certs/test2.key

# Path al certificado del broker
certfile /etc/mosquitto/certs/test2.crt

# Solo usuarios con contrasena
allow_anonymous false

# Path al archivo donde estan los usuarios y contrasenas
password_file /etc/mosquitto/conf.d/userpass

# Path al archivo donde se indica el topico de cada usuario
acl_file /etc/mosquitto/conf.d/acl.txt

# Permitir solo la version 1.2 de TLS
tls_version tlsv1.2

# No se requieren certificados por parte de los usuarios
require_certificate false
```

El broker queda configurado para recibir conexiones en la dirección localhost puerto TCP 9993 sin protocolo de seguridad y para recibir conexiones en la dirección pública puerto TCP 8883 con protocolo de seguridad TLS. Es necesario que todos los clientes presenten su nombre de usuario y contraseña, además quedan únicamente habilitados los tópicos que están en el archivo ACL.

La razón para brindar el servicio de MQTT en la dirección local del broker es que algunos clientes no pueden realizar comunicaciones hacia puertos TCP 8883. Estó se descubrió cuando se quiso conectar la estación del CAB al broker MQTT y se notó que no era posible.

Como solución se utilizó un tunel SSH desde adentro del CAB hacia la dirección localhost puerto TCP 9993 del broker. De esta manera es posible utilizar el servicio MQTT sin que la comunicación sea interrumpida por un *firewall*. Esta técnica se detalla en el capítulo 6.

Se ponen los archivos en los directorios especificados, y se crean el archivo de usuarios y de restricciones de acceso a tópicos. Es necesario que el programa mosquitto tenga permisos para leer los archivos y que no cualquiera tenga acceso a leerlos. El archivo de usuarios debe tener un usuario por línea con la sintaxis *user:password*, donde *user* es el nombre del usuario y *password* es la contraseña. Una vez creado el archivo se utiliza el comando `mosquitto_passwd -U /etc/mosquitto/conf.d/userpass` para encriptar las contraseñas y que no sean legibles por cualquiera que pueda abrir el archivo.

El archivo de acceso restringido a tópicos creado es el siguiente:

Archivo de acceso restringido a topics de mosquitto.

```
# /etc/mosquitto/conf.d/acl
user admin
topic readwrite meteo/#

user meteo
topic readwrite meteo/station/meteo/

user scrap
topic readwrite meteo/station/scrap/#
```

El archivo permite al usuario admin subscribirse y publicar en todos los tópicos que comiencen con *meteo/* y al resto de lo usuarios les permite subscribirse y publicar en los tópicos correspondientes a su red.

Finalmente se utilizó el programa de línea de comandos `mosquitto_sub` y `mosquitto_pub` para verificar que el broker funciona en su dirección pública correctamente con TLS en el puerto 8883 y que funciona en el puerto 9993 solo en la dirección privada localhost. También se verificó los usuarios, contraseñas y el acceso restringido a tópicos funcione correctamente.

Capítulo 4

Recepción y almacenamiento

Esta sección es la encargada de recibir los datos por MQTT, organizar la información y almacenarla en una base de datos. Está compuesta por dos programas: `InfluxDB` y `Estafeta`.

`InfluxDB` es un demonio que administra una base de datos temporales, tiene un formato propio y utiliza una sintaxis similar a SQL. El demonio atiende pedidos de lectura y escritura a través de un socket. Tiene una licencia tipo MIT que es compatible con la licencia GPL.

Se diseñó el programa `Estafeta` en lenguaje Python, este se encarga de recibir los mensajes por MQTT, analizar cada mensaje, organizar la información y guardarla en la base de datos `InfluxDB`. Tiene un diseño modular de tal manera que por cada sensor que se agrega a la red solo hace falta agregar un pequeño módulo.

Para registrar a los sensores en la red se utiliza un archivo de texto plano con formato *json*[1]. En el archivo están registradas todos los sensores habilitados y sus características.

4.1. Base de datos

InfluxDB es una base de datos temporales desarrollada por InfluxData, es una de las bases de datos de series temporales más utilizadas. Esta optimizada para ser rápida y tener una alta disponibilidad para escritura y lectura. La base de datos es administrada por un demonio que atiende pedidos de escritura y lectura en la dirección localhost puerto TCP 8086, se puede configurar para que atienda en otras direcciones y puertos incluso puertos UDP. El demonio recibe mensajes de escritura y lectura que se llaman *queries* utilizando el protocolo HTTP. Se puede implementar encriptacion y autenticación con protocolo TLS.

La organización de las bases de datos tiene una estructura particular, cada base de datos está compuesta por *measurements*, *tag-keys*, *field-keys* y *time stamps*. A continuación se describe cada campo.

- **data-base**: es una base de datos, el demonio InfluxDB puede manejar múltiples bases

[1] JavaScript Object Notation, es un formato para guardar listas de diccionarios en texto plano. Un diccionario es un conjunto de valores referenciados por nombres.

de datos simultáneamente. Cada base de datos es independiente del resto, no se pueden realizar escrituras o lecturas a distintas bases de datos en un mismo pedido. Se pueden definir usuarios con distintos permisos de lectura y escritura sobre cada base de datos.

- **measurement**: las bases de datos están indexadas por *measurements*, cada *measurement* define un conjunto de *field-keys* posibles.
- **field-keys**: está identificado por un nombre y posee un valor. Cada *field-key* tiene un tipo en particular para su valor, los tipos pueden ser entero de 64 bits, número flotante de 64 bits, un string o un valor booleano.
- **tag-keys**: está compuesto por un nombre y un valor del tipo string. Los *tag-keys* sirven para indexar, un conjunto de datos con el mismo *measurement* y *tag-keys* forman una *serie*. Las series son funcionales para realizar lecturas a las bases de datos.
- **time stamp**: cada punto que se guarda en la base de datos debe tener un *time stamp*, este es un valor de tiempo que representa al punto.

Dentro de un *measurement* se forman *series*, una serie esta definida por puntos que tienen los mismos valores para sus *tag-keys*. La base de datos esta optimizada para realizar búsquedas por *measurements* y por *tag-keys*. Si se solicita una búsqueda por valores de los *fields-keys* o *time stamps* entonces el demonio debe leer toda la base de datos y filtrar la información correspondiente.

Cada sensor que se conecta a la red se le asigna un tópico específico dado por *projecto/-tipo/red/subred/estación/puerto*, la descripción del topico se puede ver en la sección 3.2.2. Por cada tipo de sensor se creó una base de datos distinta, el nombre de las bases de datos está dado por projecto-tipo. Entonces cada base de datos tiene información parecida, por ejemplo la base de datos meteo-station tiene información de estaciones meteorológicas. La base de datos tiene los *tag-keys* id, subred, estación y puerto. El id es la concatenación de subred-estación y sirve para acceder velozmente a la información de cada estación en específico ya que el id identifica unívocamente a cada sensor en la red. Dentro de cada base de datos existe un *measurement* para cada red identificado por el nombre mismo de la red. Las lecturas y escrituras de la base de datos se optimizan leyendo por *measurements* y *tag-keys*.

Para instalar el demonio InfluxDB y el cliente de línea de comandos se corren los siguientes comandos en una terminal.

Instalación de InfluxDB desde terminal de comandos

```
#agregar InfluxDB a lista de repositorios
wget -qO- https://repos.influxdata.com/influxdb.key | sudo apt-key add -
source /etc/lsb-release
echo "deb https://repos.influxdata.com/${DISTRIB\_ID,,} ${DISTRIB\_CODENAME}
    stable" | sudo tee /etc/apt/sources.list.d/influxdb.list
```

```
#actualizar la lista de repositorios apt
sudo apt update

#instalar grafana
sudo apt install influxdb

#habilitar servicio
sudo systemctl enable influxdb
```

Una vez instalado y funcionando InfluxDB se puede utilizar el programa de línea de comandos `influx` para verificar el funcionamiento. A continuación una demostración del funcionamiento del programa de línea de comandos para mostrar la estructura de la base de datos.

```
Terminal
gonchigg@meteo-server:~$ influx -precision rfc3339
Connected to http://localhost:8086 version 1.7.8
InfluxDB shell version: 1.7.8
> show databases
name: databases
name
----
_internal
meteo-station
> use meteo-station
Using database meteo-station
> show measurements
name: measurements
name
----
meteo
scrap
> show series from scrap
key
---
scrap,id=Botanico2,port=data,station=IRNBARIL5,subnet=wu
scrap,id=Botánico2,port=data,station=IRNBARIL5,subnet=wu
scrap,id=Coihues,port=data,station=IDEPARTA86,subnet=wu
scrap,id=Las_Cartas,port=data,station=IBARILOC6,subnet=wu
scrap,id=Las_Margaritas,port=data,station=IBARILOC2,subnet=wu
scrap,id=Onelli+,port=data,station=ISANCARL25,subnet=wu
scrap,id=Playa\ Serena,port=data,station=IRNBARIL2,subnet=wu
scrap,id=T.Omnibus_1,port=data,station=ISANCARL19,subnet=wu
```

Figura 4.1: Cliente `influx` en computadora meteo-server.

En **Figura 4.1** se puede observar el funcionamiento del programa de línea de comandos `influx`. Se corrió en la computadora *meteo-server* donde se guarda la base de datos del proyecto. El cliente se corre con el comando `influx -precision rfc3339`, la opción `precision rfc3339` indica que se deben imprimir los *time stamps* con el formato definido por la norma *RFC 3339*. El cliente inicia una comunicación al puerto TCP 8086 dirección localhost, esta es la dirección por default del demonio que atiende la base de datos. El mensaje *connected to http://localhost:8086 versión 1.7.8* que se muestra indica que el cliente logró comunicarse con el demonio InfluxDB.

Se corren los comandos:

- `show databases`: lista todas las bases de datos InfluxDB disponibles en la computadora.

- `use meteo-station`: indica que se van a realizar operaciones de escritura y lectura sobre la base de datos meteo-station.

- `show measurements`: lista las *measurements* de la base de datos, el resultado son las *measurements meteo* y *scrap*. Estas se corresponden con los datos que se recolectan en el tópico *meteo/station/meteo/#* y *meteo/station/scrap/#* respectivamente.

- `show series from scrap`: lista todas las series dentro del measurement scrap, se pueden observar todas las estaciones de las cuales se *scrapean* datos.

```
Terminal ▾
> show field keys from meteo
name: meteo
fieldKey     fieldType
--------     ---------
UV           float
alt          float
at           float
bar          float
chill        float
cloudbase    float
dew          float
dir          float
dir.gust     float
dir.gust.l   string
dir.l        string
geohash      string
heat         float
heigh        integer
hum          float
hum.in       float
humdx        float
lat          float
locationname string
long         float
pres         float
rain         float
rain.a       float
rain.x       float
rain.xx      float
rain.xxx     float
sol          float
temp         float
temp.in      float
vel          float
vel.gust     float
```

Figura 4.2: Cliente `influx` en computadora meteo-server.

En **Figura 4.2** se observan el resultado de correr el comando `show field keys from meteo` con el cliente `influx`. Al correr el comando se listan todos los *field-keys* correspondientes al measurement `meteo` con sus respectivos tipos de datos.

```
Terminal
> show tag keys from meteo
name: meteo
tagKey
------
id
port
station
subnet
> select temp from meteo where id='coihues_01' limit 4
name: meteo
time                  temp
----                  ----
2019-10-24T09:55:00Z 10.5
2019-10-24T10:05:00Z 10.983333333333336
2019-10-24T10:10:00Z 10.666666666666668
2019-10-24T10:15:00Z 10.833333333333334
>
```

Figura 4.3: Cliente `influx` en computadora meteo-server.

En **Figura 4.3** se observa el resultado de correr los comandos:

- `show tag keys from meteo`: lista los *tag-keys* del measurement meteo
- `select temp from meteo where id='coihues_01' limit 4`: lista 4 valores del *field-key temp* del measurement *meteo* donde el *tag-key id* sea *coihues_01*.

4.2. Recepción, acondicionamiento y almacenamiento

Para la recepción, acondicionamiento y almacenamiento de datos se diseñó el programa `Estafeta`. El programa se escribio en lenguaje Python. El programa debe ser capaz de recibir mensajes en cualquier tipo de formato, para esto se crea un módulo por cada sensor. En **Figura 4.4** se muestra en esquema del funcionamiento del programa `Estafeta`.

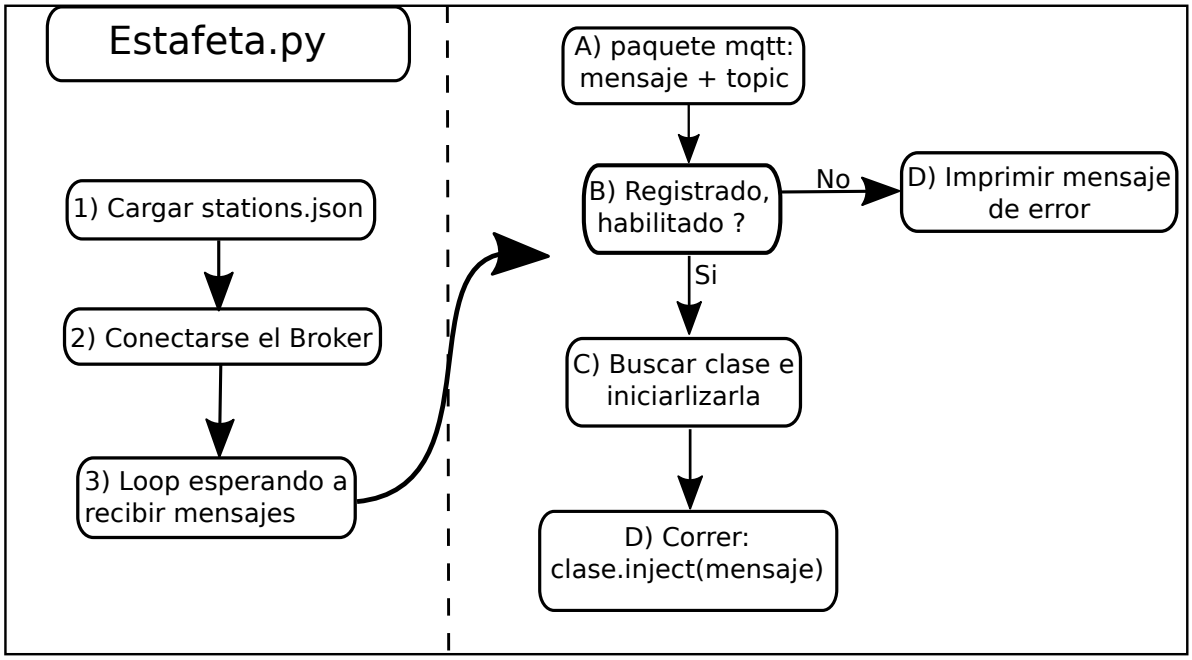

Figura 4.4: Esquema del programa Estafeta.

1 Se carga el archivo *stations.json*, este archivo posee información de todos los sensores registrados en la red.

2 Se subscribe al tópico *meteo/#* con el usuario *admin*.

3 El programa queda esperando a recibir mensajes. Con cada mensaje que se recibe se corre una rutina para acondicionar y guardar la información.

4.2.1. Registro de sensores

El registro de los sensores en la red se realiza con un archivo en formato *json* llamado *stations.json*, en el se guarda información de los sensores registrados en la red. Lo primero que realiza el programa es cargar el archivo *stations.json* y guardar la información en una lista de diccionarios de Python. Cada diccionario en la lista representa a un sensor. Un diccionario de Python es una lista con nombres, cada valor de la lista tiene un nombre asociado que lo identifica. El archivo tiene la información guardada de cada sensor registrado en la red, por cada sensor se guardan los siguientes campos.

- **id:** identificador del sensor, compuesto por la concatenación de los tópicos subred y estación.
- **desc:** una breve descripción del sensor.
- **model:** modelo sel sensor.
- **on:** estado del sensor, si está habilitado para guardar datos en la base de datos o no. Los valores posibles son *true* y *false*.
- **type:** el tipo se sensor, se utiliza para definir el tópico y el nombre de la base de datos.
- **net:** la red a la que pertenece el sensor, se utiliza para definir el tópico.
- **subnet:** la subred a la que pertenece el sensor, se utiliza para definir el tópico.
- **station:** el campo que identifica a cada estación dentro de cada red y subred, se utiliza para definir el tópico.
- **heigh:** la altura sobre el nivel del mar en metros a la cual se encuentra el sensor.
- **lat:** la latitud geográfica a la cual se encuentra el sensor.
- **long:** la longitud geográfica en la cual se encuentra el sensor.
- **parser:** el nombre de la clase que se utiliza para manejar la información que se recibe por MQTT.

4.2.2. Recepción de mensajes

El programa utiliza la biblioteca de Python *paho-mqtt* para conectarse al broker y recibir los mensajes. La biblioteca permite dejar al programa esperando a recibir mensajes y correr una rutina de comandos con cada mensaje que llega. Para conectarse se utiliza el usuario *admin* el cual recibe todos los mensajes que se manden al tópico *meteo/#*. Con cada mensaje que llega se corre una rutina.

4.2.3. Identificación del mensaje recibido

Con cada mensaje que se recibe es necesario identificar a qué sensor corresponde. Para identificar al sensor se utilizá el tópico, cada sensor tiene su tópico propio. Se verifica en el archivo *stations.json* si hay algún sensor registrado en el tópico en el cual se recibió el mensaje. Si el tópico está registrado se verifica en el diccionario del sensor si el campo 'on' esta habilitado. Si el sensor esta registrado y el campo 'on' esta habilitado entonces se procede con el acondicionamiento del mensaje, si no se descarta el mensaje y se imprime un mensaje de error.

4.2.4. Acondicionamiento y almacenamiento de la información

Cada sensor tiene un módulo propio para el acondicionamiento de la señal, este módulo es una clase de Python. El nombre de la clase está definido en el campo *parser* del archivo *stations.json*.

Se inicializa la clase definida en el campo *parser* con el tópico recibido. Luego se corre la función interna de la clase llamada *inject_mqtt*. Esta función recibe como argumento el mensaje enviado. La función se ocupa de realizar el acondicionamiento de la información y almacenarla en la base de datos.

Las clases definidas en el campo *parser* están estandarizadas para ser lo más simple posibles. Estás utilizan otra clase llamada *sensor* que se ocupa de la comunicación con la base de datos y el acondicionamiento de parte de la información.

Capítulo 5

Servicio

Esta sección cumple la función de servir una página web pública a modo de presentación del proyecto. La página contiene una descripción general, un link a un instructivo de como sumar estaciones a la red y un link los datos meteorológicos.

Para brindar la página web se utilizó un servidor `Nginx` que tiene una licencia de software libre. Este es el servidor de páginas web más utilizado, en este proyecto también fue utilizado como *proxy reverso*. Para ofrecer al usuario los datos metereológicos se utilizó el programa `Grafana`, este está diseñado para servir datos directamente desde una base de datos.

5.1. Página web

Se sirvió una página web desde la máquina de Digital Ocean con dirección IP 138.179.228.126. Este servidor es el mismo donde se corre la base de datos `InfluxDB` y el programa `Estafeta` La página web funciona como presentación del proyecto, tiene un link a un instructivo de como sumar estaciones a la red y otro link a los datos meteorológicos. Los datos meteorológicos se sirvieron utilizando el programa `Grafana`.

La página principal se sirve en la dirección *http://138.197.228.125/*, en **Figura 5.1** se puede observar la pagína.

Figura 5.1: Página web del proyecto, con la dirección *http://138.197.228.125/*

En **Figura 5.2** se puede la página que se utiliza como instructivo de cómo sumar esta-

ciones a la red.

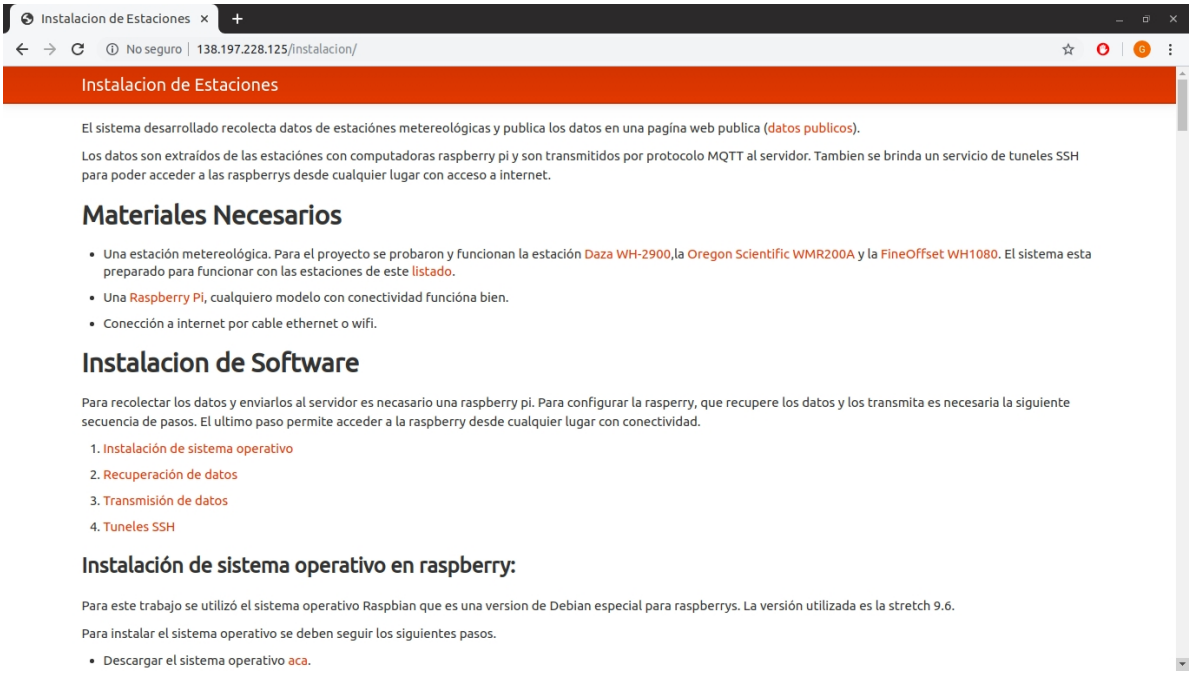

Figura 5.2: Página web con instructivo de como sumar estaciones a la red. Servida en *http://138.197.228.125/instalacion*

`Nginx` es un servidor de páginas web y *proxy* de alto rendimiento, es ampliamente utilizado y se puede encontrar en sitios web como *Netflix*, *Github* y *Facebook*. En Ubuntu se puede instalar directamente desde el repositorio del sistema corriendo el comando `sudo apt install nginx`. Una vez instalado el programa comienza a servir una página html estática por defecto que se encuentra en el directorio **/var/www/html/** con el nombre *index.html*. La página se sirve automáticamente en todas las direcciones IP de la computadora, de manera que es accesible desde cualquier computadora conectada a internet.

La página del proyecto se escribió en formato *markdown*, el formato *markdown* fue diseñado para escribir código html de una manera más clara. Para incluir código de markdown en una página html sin tener que utilizar compiladores en el servidor se utilizó un *javascript*. Un *javascript* es una porción de código escrito en lenguaje *Javascript* que se incluye dentro de un código de html, es utilizado para mejorar la interfaz con el usuario y servir páginas dinámicas. El código utilizado fue el siguiente.

Código de página web escrita en lenguaje Markdown.

```
<!DOCTYPE html>
<html>
<title>Titulo de la pagina</title>

<xmp theme="united" style="display:none;">
```

```
# Aqui se puede incluir codigo Markdown

</xmp>

<script src="http://strapdownjs.com/v/0.2/strapdown.js"></script>
</html>
```

Agregando la línea que comienza con *<script* se incluye el javascript y el código escrito en markdown se puede incluir dentro de la directiva *<xmp>*. En el campo *theme* se pueden definir distintos estilos de página, los estilos se pueden ver en *http://bootswatch.com/*.

El grafana se sirve en la dirección *http://128.197.228.125:3000* pero no es una practica adecuada servir páginas en puertos que no sean el estándar de HTTP. Si se quiere acceder a una página en el puerto TCP 3000 desde el CAB el *firewall* corta la comunicación. Por lo tanto se utiliza `nginx` como proxy reverso para redirigir las solicitudes que van a *http://138.197.228.125/grafana* hacia la dirección *http://138.197.228.125:3000*. De esta manera el usuario que visita la página ve como si grafana se sirviera en el puerto estándar de HTTP. Para configurar nginx como proxy reverso se edita el archivo de configuración que se encuentra en **/etc/nginx/sites-enabled/default**, el archivo se debe ver de la siguiente manera.

Archivo de configuración de Nginx

```
# /etc/nginx/sites-enabled/default
server {
  listen 80;
  root /var/www/html;
  index.html

  location /grafana/ {
    proxy_pass http://localhost:3000/;
}
```

El archivo *default* inicia un servidor http en el puerto indicado por la directiva *listen*. La directiva *root* indica el directorio donde se sirve la página web raíz. El archivo *index.html* que se encuentra en **/var/www/hmtl** es servido cuando se inicia una conección http a la dirección *http://138.197.228.125*. Cuando se realiza un pedido HTTP a la dirección *http://138.197.228.125/ejempplo* nginx intenta servir el archivo que se encuentra en **/var/www/html/ejemplo/html** si es que existe.

La directiva *location* indica que los pedidos http a la dirección *http://138.197.228.125/grafana* se deben reenviar hacia la dirección *http://localhost:3000* que es donde se encuentra el servidor de grafana.

5.2. Servicio de datos

Grafana es un programa de software libre con licencia Apache 2.0 diseñado para visualizar datos de series temporales en navegadores web. El programa realiza consultas a las bases de datos, grafica los resultados y los sirve en una página web. Es compatible con las principales bases de datos, algunas son *Cloudwatch, Graphite, Elasticsearch, OpenTSDB, Prometheus, Hosted Metrics e InfluxDB*. Es uno de los programas más utilizados para servir datos de series temporales, es utilizado por organizaciones como la Organización Europea para la Investigación Nuclear (CERN), Digital Ocean, el Laboratorio Nacional Fermi y muchas otras organizaciones

Se utilizó el servidor donde se sirve la página del proyecto, el programa `Estafeta` y la base de datos `InfluxDB` para servir también `Grafana` con los datos de las estaciónes meteorológicas. Se sirvió Grafana en el puerto por defecto TCP 3000. Se implementó el sistema de usuario y contraseñas que incluye grafana, este permite mantener un usuario administrador que puede modificar la página y un usuario de uso general que solo puede visualizar los datos.

Las páginas de grafana se llaman *dashboards* y están compuestos por paneles, variables y un rango temporal. El usuario de uso general puede definir el rango temporal de los datos que quiere visualizar y puede definir el valor de ciertas variables. En este proyecto las variables que se definieron fueron el nombre de cada estación, entonces el usuario que visualiza la página puede elegir los datos de que estaciones visualizar. Los paneles son los graficos que se muestran en la página, cada panel muestra un conjunto de datos en particular definidos por una consulta. En **Figura 5.3** se muestra la página de grafana servida.

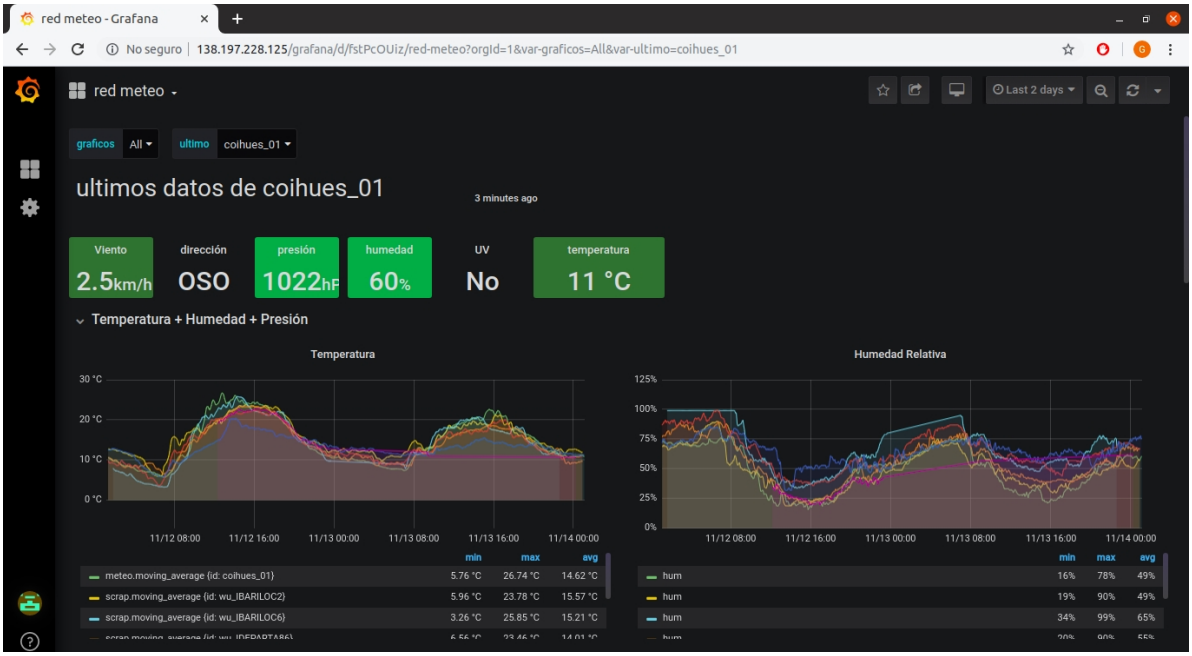

Figura 5.3: Página web generada por `Grafana`. Servida en *http://138.197.228.125/grafana/*

En **Figura 5.4** se muestran más partes de la página de grafana servida. Se puede observar que se implementó un mapa donde se visualizan las estaciones.

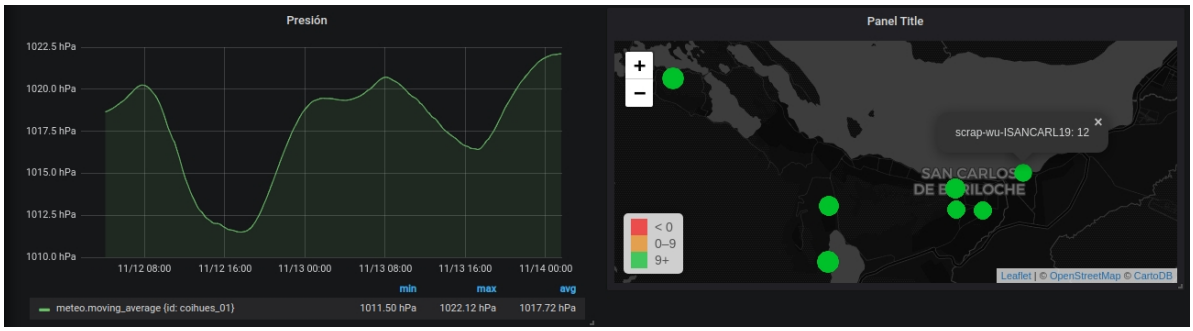

Figura 5.4: Página web generada por `Grafana`. Servida en *http://138.197.228.125/grafana/*

Capítulo 6

Acceso a estaciones

La función de esta sección es acceder de manera remota a las computadoras que extraen los datos de las estaciones. En muchas ocasiones puede ser necesario acceder a las computadoras y poder ejecutar comandos. Estas suelen estar en lugares remotos por lo que no es practico acceder a estas físicamente.

Para solucionar este problema se utiliza el protocolo SSH, que provee un canal seguro de comunicación en una red TCP-IP insegura. Funciona con una estructura cliente-servidor donde el servidor recibe conexiones en el puerto TCP 22. El cliente se conecta al servidor y ambos se autentican el uno al otro. Una vez autenticados el cliente puede abrir una terminal de manera remota en el servidor o redirigir tráfico utilizando túneles.

También se utilizaron túneles SSH para poder pasar el protocolo MQTT a través de un *firewall*.

6.1. Protocolo SSH

El protocolo SSH está definido por la norma RFC 4250, su función principal es acceder remotamente a servidores y poder ejecutar comandos en estos de manera segura. Otras funciones que incluye el protocolo son copiar datos, redirigir tráfico de red y la ejecución de programas con interfaz gráfica compartiendo la pantalla.

En este proyecto se utilizó la implementación del protocolo SSH llamada OpenSSH desarrollada por el proyecto OpenBSD[1]. Esta implementación viene instalada por defecto en la mayoría de las distribuciones de Linux.

El servidor recibe conexiones en el puerto TCP 22. Cada cliente que se conecta al servidor debe especificar un usuario, estos usuarios son los usuarios mismos del servidor. Cada usuario tiene su propio método de autenticación y permisos para ejecutar comandos.

El protocolo utiliza una combinación de claves asimétricas y simétricas para encriptar toda la información transmitida y garantizar autenticación de ambas partes, confidencialidad

[1]OpenBSD es un sistema operativo tipo Unix de código libre y abierto.

e integridad.

El servidor se autentica ante el cliente utilizando su clave pública, el cliente debe verificar si el *fingerprint* del servidor se corresponde con el que el cliente conoce de este. El cliente debe conocer el *fingerprint* con anticipación por algún método seguro. El *fingerprint* es un *hash* de la clave pública del servidor, un *hash* es una función que transforma la clave pública en un string más corto. Se compara el fingerprint en vez de la clave completa para que sea más facil para el cliente realizar la comparación. El cliente compara el fingreprint recibido con el que conoce del servidor y si ambos coinciden, entonces el cliente puede asegurar la autenticidad del servidor. En *Figura 6.1* se puede observar una conexión hacia un servidor en la cual se solicita la verificación del *fingerprint*.

```
gonchigg@gonchigg:~$ ssh pi@192.168.7.1
The authenticity of host '192.168.7.1 (192.168.7.1)' can't be established.
RSA key fingerprint is SHA256:AkQ6H1TiYmf2Cqmg+fk4iZRzv5OXonlYHEsMpuGrpKM.
Are you sure you want to continue connecting (yes/no)?
```

Figura 6.1: Conexión SSH hacia servidor con IP 192.168.7.1 y usuario *pi*, se solicita la verificación del *fingerprint*.

El cliente puede autenticarse ante el servidor de dos maneras distintas, con una contraseña o utilizando su clave pública. El cliente debe especificar un usuario al cual se quiere conectar, una vez que pasa la autenticación el cliente puede ejecutar comandos como si fuera ese usuario.

La autenticación con contraseña requiere que el cliente conozca la contraseña de alguno de los usuarios del servidor. La autenticación con clave pública requiere que la clave pública del cliente este registrada en alguno de los usuarios del servidor. La clave pública del cliente se debe guardar en el archivo **.ssh/authorized_keys** que se encuentra en el directorio raíz del usuario ante el cual se quiere autenticar.

El método de autenticación con clave pública se considera más seguro que con contraseña. Además, la autenticación con clave pública es más practica porque no hace falta escribir la contraseña cada vez que se quiere acceder a un servidor.

Para acceder a un servidor por SSH se corre el siguiente comando en una terminal `ssh user@ip`. Es necesario reemplazar *user* por el usuario en el servidor, y *ip* por la dirección de IP del servidor.

6.2. Acceso a estaciones

Se desea acceder a las *raspberrys* que manejan las estaciones, pero no es posible hacer esto directamente ya que las *raspberrys* no tienen una dirección de IP pública. Para solucionar este problema se utilizan túneles SSH, a continuación se describe el método utilizado.

6.2.1. Túnel SSH -R

Un túnel SSH permite el redirecciónamiento de tráfico de red, estos se utilizaron para acceder a raspberrys con dirección IP privada y detrás de *routers* no manejados. La problemática es la siguiente, las raspberrys pueden correr un servidor SSH pero como no poseen un IP público no es posible acceder a estas mediante internet. Entonces la raspberry abre un túnel en un servidor con IP publico y redirige alguno de sus puertos hacia la raspberry, esto se logra abriendo un túnel -R. Los túneles *-R* son túneles remotos, quiere decir que se redirige tráfico desde un puerto en un servidor remoto hacia un puerto en el cliente.

El método utilizado fue el siguiente, se tiene un servidor de SSH con IP público y una raspberry con otro servidor SSH pero con IP privado. Se corre el comando `ssh -R port:localhost:22 user_1@ip -N` en la raspberry. El comando redirige todo lo que vaya al puerto TCP *port* de la dirección *localhost* del servidor con dirección IP *ip* hacia el puerto TCP 22 del *localhost* de la raspberry. Es necesario que la raspberry logre autenticarse ante el usuario *user_1* del servidor y que el usuario este habilitado a redirigir tráfico en el puerto *port*. La dirección *localhost* del servidor es privada del servidor y solo se pueden acceder los programas que corren en este, lo mismo para la dirección *localhost* de la raspberry.

Ahora, si se pasa exitosamente la etapa de autenticación y el usuario *user_1* está habilitado para abrir el puerto *port* se puede correr el siguiente comando en el servidor para acceder a la raspberry `ssh -p port user_2@localhost`. El comando inicia una conexión hacia el puerto *port* y la dirección *localhost* del servidor, el túnel que se corrio previamente redirige esta conexión hacia el puerto 22 de la raspberry en la dirección *localhost*. El servidor de SSH de la raspberry recibe una conección por el puerto TCP 22 dirección localhost. La raspberry acepta la conexión y ahora resta autenticarse frente al usuario *user_2* de la raspberry.

Siguiendo la secuencia de dos comandos explicada es posible acceder a la raspberry desde el servidor siempre que esta tenga acceso a internet. En 6.2.4 se explica como hacer para mantener el túnel constantemente abierto con un demonio.

6.2.2. Túneles SSH -L

Con el túnel SSH -R y la conexión SSH se logra el acceso a raspberrys en redes privadas con *routers* no manejados, pero es necesario tener acceso a una terminal de comandos en el servidor. Se desea brindar un servicio para poder acceder a las raspberrys sin que los usuarios tengan acceso a correr comandos en el servidor. Para esto se utiliza un túnel SSH -L. Los túneles *-L* son túneles locales, quiere decir que se redirige tráfico desde un cliente hacia un servidor.

El usuario que desea acceder a la raspberry corre el siguiente comando en su computadora `ssh -L -N port:localhost:port user_1@ip`, el comando redirige todo lo que vaya al puerto *port* de la dirección *localhost* de la computadora que abre el túnel hacia el puerto *port* de la dirección *localhost* del servidor con IP *ip*. Es necesario que el cliente pueda autenticarse

ante el usuario *user_1* del servidor.

Una vez abierto el túnel -L se puede acceder a la raspberry corriendo el siguiente comando `ssh -p port user_2@localhost`. El comando inicia una conexión SSH contra el puerto *port* de la dirección *localhost* de la computadora donde se inicia la conexión. El túnel -L redirige la conexión hacia el puerto *port* dirección *localhost* del servidor, el túnel -R vuelve a redirigir la conexión hacia el puerto 22 de la dirección *localhost* de la raspberry. El servidor de SSH en la raspberry recibe la conección en el puerto TCP 22 dirección localhost. El usuario debe ser capaz de autenticarse ante el usuario *user_2* de la raspberry. Si se pasa la autenticación ahora es posible correr comandos en la raspberry sin tener acceso a correr comandos en el servidor.

Se debe cumplir que la computadora que abre el túnel -L pueda autenticarse ante el usuario *user_1* del servidor, que el usuario pueda abrir túneles en la direccion *localhost* puerto *port* y que la raspberry pueda autenticarse ante el *user_1* del servidor.

6.2.3. Configuración del servidor SSH

Para brindar el servicio de túneles SSH de manera segura y controlada se debe configurar correctamente el servidor. En el servidor existen distintos usuarios con distintas capacidades. Para configurar y administrar el servidor se tiene un usuario especial y para abrir túneles se usan otro tipo de usuarios.

Los usuarios que se crean para abrir túneles se le restringen los permisos unicamente para este fin, de manera que los clientes puedan abrir los túneles en sus puertos específicos pero no puedan correr comandos.

Los usuarios en Linux se encuentran en el archivo que se encuentra en **/etc/passwd** en cada linea se instancia un usuario con la siguiente sintaxis: *name:password:user ID:group ID:gecos:home directory:shell*. El campo *name* es el nombre del usuario, el campo *password* es la contraseña cifrada, el campo *home directory* es el path al directorio raíz del usuario y el campo *shell* es el path al ejecutable con la terminal del usuario. Para lograr que los usuarios que se utilizan para abrir los túneles no puedan correr comandos no se les asigna una terminal. En el campo *shell* se pone el path */bin/false* el cual genera que cada vez que alguien accede por SSH a este usuario nunca se abre una terminal.

En **Figura 6.2** se puede observar un recorte del archivo **/etc/passwd** en el servidor *central-comm*. Los usuarios *forwarderXXXX* se utilizaron para dar el servicio de tuneles. Estos no estan habilitados para correr comandos, solo pueden abrir tuneles en el puerto especificado.

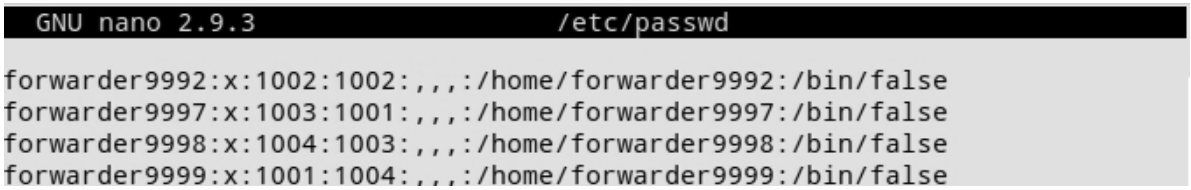

Figura 6.2: Recorte del archivo /etc/passwd en servidor *central-comm*.

El archivo de configuración del servidor SSH se encuentra en la dirección **/etc/ssh/sshd_config**. Para restringir las capacidades de un usuario para que solo pueda abrir túneles en un determinado puerto se agregan las siguientes líneas al archivo de configuración.

Archivo de configuración del servidor SSH

```
# /etc/ssh/sshd_config
# Se agregan las siguientes lineas por cada usuario
Match User forwarderXXXX
  # permitir autenticacion con clave publica
  PubkeyAuthentication yes

  # no permitir autenticacion con contrasena
  PasswordAuthentication no

  # permitir abrir tuneles solo en la direccion localhost puerto 9999
  AllowTcpForwarding yes
    PermitOpen localhost:port

  # no permitir compartir la pantalla
  X11Forwarding no
```

Está configuración establece que si algún cliente inicia una conexión dirigida hacia el usuario *forwarderXXXX* solo pueda autenticarse mediante el método de clave pública y solamente pueda abrir un túnel -L o -R en el la dirección localhost con puerto *port*. Para que un cliente que intenta conectarse pase la etapa de autenticación se clave publica debe concatenarse en el archivo que está en */home/forwarderXXX/.ssh/authorized_ keys*.

Por cada estación con una raspberry instalada se creo un usuario con un puerto asignado. De esta manera cada raspberry tiene su usuario para poder acceder remotamente.

6.2.4. Demonios autossh

La raspberry debe mantener un túnel -R abierto constantemente, de manera que se pueda acceder a esta en cualquier momento. Para esto se genera un demonio que ejecuta el

comando autossh. El comando autossh está diseñado especialmente para mantener una sesión SSH abierta constantemente. El demonio se instancia poniendo un archivo en el directorio */etc/systemd/systemd/*, el archivo debe tener la extensión *.service*, en nuestro servidor el arhivo se llamó *autosshR.service*.

El demonio instanciado es un archivo de texto plano que cumple con la siguiente sintaxis:

Demonio de autossh

```
# /etc/systemd/system/autosshR.service
[Unit]
Description=AutoSSH tunnel service
After=network.target

[Service]
ExecStart=/usr/bin/autossh -M 0 -o "ServerAliveInterval 30" -o
    "ServerAliveCountMax 3" -N -R port:localhost:22 user_1@ip
User=pi
Restart=always

[Install]
WantedBy=multi-user.target
```

La línea que comienza con *ExecStart* define el comando que corre el demonio cada vez que se inicia. El comando que se corre es `autossh -M 0 -o ‘‘ServerAliveInterval 30’’ -o ‘‘ServerAliveCountMax 3’’ -N -R port:localhost:22 user_1@ip`. El comando abre un túnel *-R* que redirige todo lo que vaya al puerto *port* del servidor con dirección IP *ip* hacia el puerto 22 de la dirección *localhost* de la raspberry. La raspberry debe ser capaz de autenticarse con su clave pública ante el usuario *user_1* del servidor. Las opción *-o “ServerAliveInterval 30”* definen que cada 30 segundos se manda un paquete de control para mantener el túnel abierto. La opción *-o “ServerAliveCountMax 3”* define que si no se recibe respuesta a más de tres paquetes se reinicia el túnel.

La línea que comienza con *User* define el usuario que corre el programa, esto va a definir que clave púbica que se utiliza para la autenticación. La línea que comienza con *Restart* es importante porque al completar con *always* asegura que el demonio se va a reiniciar cada vez que se caiga el túnel por cualquier razón.

Una vez que se crea el archivo se corre el comando `sudo systemctl enable autosshR.service` para habilitar que el demonio se corra cada vez que se inicia la raspberry. De esta manera el demonio va a intentar mantener el túnel abierto constantemente y ante cualquier falla se va a reiniciar.

6.3. MQTT a través de SSH

El protocolo de MQTT con TLS funciona en el puerto TCP 8883. Ocurre que en la red del CAB estas conexiones son interrumpidas por el firewall. Para solucionar este problema se utilizó un túnel SSH -L, el túnel se utiliza para encapsular el tráfico MQTT dentro de una conexión SSH. El tráfico logra pasar el firewall ya que las conexiones SHH salientes están permitidas.

Para abrir el túnel se utiliza un demonio que corre el comando autossh como se explica en la sección anterior. El demonio corre el comando `autossh -N -L 9993:localhost:9993 forwarder9993@138.197.228.125`. El comando redirige todo el tráfico que vaya al puerto 9993 dirección localhost de la raspberry hacia el puerto 9993 dirección localhost del servidor con IP 138.197.228.125. El servidor con dirección IP 138.197.228.125 es el servidor *central-comm* que se utilizó como broker MQTT.

El broker MQTT brinda su servicio en su dirección pública puerto 8883 y en su dirección privada localhost puerto 9993. De esta manera la raspberry que extrae los datos de la estación corre el túnel y luego puede pubicar en el broker MQTT iniciando una conexión hacia su localhost puerto 9993. El túnel redirige la conexión hacia la dirección localhost 9993 del broker y la comunicación de MQTT se da sin problemas.

Capítulo 7

Conclusiones

Se montó una red de datos meteorológicos con sensores distribuidos en la ciudad de Bariloche desde la generación de los datos hasta la presentación en una página web. A diferencia de las otras redes que existen, está es una red diseñada por los usuarios finales para estos mismos. Los usuarios finales pueden ser simples aficionados a la meteorológia, proyectos de investigación o emprendimientos comerciales. El resto de las redes son restrictivas desde el punto de vista de que se puede subir y como se debe subir. Son redes administradas por entes que buscan el beneficio propio y no el de los usuarios. Usualmente exigen un formato específico, requieren una validación de los sensores y hasta llegan a apropiarse legalmente de los datos.

En esta red cualquiera puede subir sus propios datos con mínimos requerimientos, la validación de los datos se generará posteriormente mediante el procesamiento de la información. De esta manera no se excluye a las personas que se quieran sumar a la red ni se les solicita procedimientos complejos.

Esta red brinda el servicio que los usuarios necesitan que es subir sus datos a una web donde se puedan ver y descargar. El objetivo de esta red es que se puedan subir información de todo tipo de sensores. Se tomó conocimiento que existen muchos proyectos de investigación que necesitan el servicio brindado por la red. Algunos son, control de fuego por PN, control de plagas por el INTAy control de contaminación en el Lago Nahuel Huapi por Investigadores de Universidad Nacional del Comahue.

Desde el punto de vista de lo implementado y aprendido. Hubo que instalar, configurar y mantener en funcionamiento una variedad de programas, en específico WeeWX, Mosquitto, OpenSSH, InfluxDB, Nginx y Grafana. También hubo que generar programas en Python, en Bash y ejecutarlos como Demonios. Hubo que manejar usuarios de Linux y administrar sus permisos, administrar una base de datos con contraseñas y multiples claves RSA.

La dificultad del proyecto fue la configuración de los programas, esta es una tarea difícil. Hubo que leer muchas páginas de documentación. Se aprendió que configurar programas es una tarea dura que solo resulta más simple cuando se entiende el programa completo y se

sabe claramente lo que se quiere hacer.

La implementación de MQTT con TLS requirió entender en profundidad los certificados X509. La implementación de los tuneles SSH requiere entender bien como administrar usuarios en Linux. Para que los demonios funcionen bien hay que leer detenidamente la documentación de estos. Leer los *man pages* de Linux lleva tiempo pero al final da resultados. Finalmente se aprendió que es de vital importancia la documentación de los métodos y la información aprendida, esta tarea es molesta y quita tiempo, pero al final es algo sumamente beneficioso.

Apéndice A

Recuperación de datos de estaciones

WeeWX es un software gratuito, de código abierto escrito en Python 2.7 que interactúa con estaciones meteorológicas y brinda la posibilidad de producir gráficos, páginas HTML y compartir los datos por MQTT, FTP y rsync. Está diseñado para funcionar en Linux y en macOS. En este trabajo se utilizó para extraer los datos de las estaciones, guardar la información en una base de datos SQL y enviar los datos por protocolo MQTT. Para instalar el software en distribuciones Debian de linux se puede utilizar el comando *apt* para instalar desde el repositorio del programa. Se corren los siguientes comandos en una terminal.

Instalación de WeeWX en terminal de comandos

```
# Agregar weewx a la lista de repositorios
wget -qO - HTTP://weewx.com/keys.html | sudo apt-key add -
wget -qO - HTTP://weewx.com/apt/weewx.list | sudo tee
    /etc/apt/sources.list.d/weewx.list

# actualizar la lista de repositorios
sudo apt update

# instalar weewx y sqlite3
sudo apt install weewx
sudo apt install sqlite3 #necesario para manejar la base de datos SQL
```

Para que el programa quede corriendo continuamente se lo instancia como demonio. Para instanciar un demonio en Linux se puede crear un archivo con la extensión *.service* en el directorio **/etc/systemd/system/**. Estos archivos deben cumplir con la sintaxis correspondiente, esta se puede leer corriendo el comando `man systemd`. En el archivo se especifica que comando se corre para iniciar el programa, como y cuando reiniciar, con que usuario correr y demás configuraciones del demonio. El demonio de WeeWX es creado automáticamente con la instalación del programa, se genera el archivo *weewx.service* en el directorio **/etc/systemd/system/**. El demonio no comienza a correr hasta que se lo habilita con el

comando `sudo systemctl enable weewx`. Los errores del programa se pueden leer en el archivo */etc/log/syslog*, este archivo es donde se escriben los mensajes de error de la raspberry. Otra manera de ver el estado del demonio es correr el comando `sudo systemctl status weewx`.

El programa necesita una configuración particular para cada estación dado que cada una tiene hardware y software diferente. En las secciones posteriores se explica como configurar el programa para las dos estaciones distintas que se utilizaron en el proyecto. También es necesario instalar y configurar una extensión para enviar los datos por protocolo MQTT. El archivo de configuración del programa se encuentra en **/etc/weewx/weewx.conf**. En la página oficial del programa en *http://www.weewx.com* se encuentra toda la documentación necesaria para configurar el programa.

A.1. Transmisión por MQTT

La extensión de MQTT para Weewx se puede encontrar en la página *https://github.com/weewx/weewx/wik* en esta se encuentra la información necesaria para configurar la extensión. Para instalar la extensión de mqtt se descarga y se utiliza el comando `wee_extension` para instalarla. Se corren los siguientes comandos en una terminal.

Instalación de extensión de MQTT para WeeWX en terminal de comandos

```
# descargar óextensin mqtt
wget HTTP://lancet.mit.edu/mwall/projects/weather/releases/weewx-mqtt-0.15.tgz

# instalar la óextensin
sudo wee_óextensin --install weewx-mqtt-0.15.tgz

# instalar pip (instalador de librerias)
sudo apt install Python-pip

# instalar dependencias
pip install paho-mqtt
pip install Python-cjson
```

Una vez instalado el plugin se debe editar el archivo de configuración que se encuentra en **/etc/weewx/weewx.conf**. En la sección *[StdRESTful]* se debe incluir la subsección *[[MQTT]]* y debe quedar como se muestra a continuación.

Archivo de configuración de WeeWX

```
#/etc/weewx/weewx.conf
```

```
[StdRESTful]
  [[MQTT]]
    # usuario, contrasena, IP, puerto
    server_url = mqtt://usuario:contrasena@IP:puerto/
    topic = meteo/statio/meteo/panuelo/01/data

    # enviar datos agrupados
    aggregation = aggregate

    # no retener el mensaje en el servidor
    retain = False

    [[[tls]]]
      # certificado de la Autoridad Certificante
      ca_certs = /etc/ssl/certs/ca.crt

      # no se requieren certificados del cliente
      cert_reqs = none

      # version del protocolo TLS
      tls_version = tlsv1.2
```

La directiva `server_url` indica el usuario, la contraseña, el IP del broker y el puerto TCP. La directiva `topic` debe incluir el topic en el cual se publicaran los datos, el usuario debe estar habilitado para publicar en ese topic. La directiva `aggregation` define si se manda cada dato meteorológico en particular o si se mandan agrupados, el modo *aggregate* manda los datos agrupados, este modo facilita el procesamiento de los datos. La directiva `retain` define si el broker mqtt va a retener cada mensaje hasta que llegue el proximo, el valor `false` desactiva esta opción. La subsección *[[[tls]]]* define las directivas del protocolo TLS, es necesario especificar el certificado del broker, la versión de TLS y si se requiere certificado del cliente. La directiva `ca_certs` especifica el *path* al certificado de la Autoridad Certificatente del broker. La directiva `cert_reqs` indica si el cliente requiere certificado propio, en este proyecto no se utilizó certificados para el cliente. La directiva `tls_version` define la versión del protocolo TLS que utiliza el broker, en este proyecto se utilizó la versión 1.2. Una vez instalada y configurada la extensión de MQTT se reinicia el demonio weewx con el comando `sudo systemctl restart weewx` y el programa debería comenzar a transmitir los datos por MQTT en el topic correspondiente. El registro de mensajes de error se puede leer en el archivo **/var/log/syslog** o corriendo el comando `sudo systemctl status weewx` para ver el estado del demonio.

A.2. Estación modelo WH-2900

Para el proyecto se utilizaron dos estaciones meteorológicas DAZA WH-2900. Esta estación se conecta a una red wifi y se configura para que envíe los datos por protocolo HTTP a algún dominio público dedicado para esta función. Para conectar la estación al wifi se utiliza la aplicación *WStool* disponible para sistemas operativos Android. La aplicación permite transferirle la contraseña del wifi a la estación e indicarle a que demonio enviar los datos. Para transferir la contraseña el celular debe estar conectado al wifi ya que la aplicación envía un paquete de broadcast con la contraseña de wifi. El broadcasting de la contraseña es peligroso para la seguridad de la red ya que cualquiera que esté escuchando al wifi puede capturar el paquete broadcast y obtener la contraseña del wifi, justamente de esta manera adquiere la contraseña la estación.

Para extraer los datos de esta estación es necesario instalar una extensión llamada *interceptor*. La extensión esta diseñada para recuperar los datos de cualquier estación que envié sus datos a la página *weather underground*. El funcionamiento de la extensión requiere que la raspberry tenga acceso a escuchar los paquetes de red que la estación envía por wifi. Esto se puede lograr de distintas maneras. En este proyecto se logra instanciando un hotspot en la raspberry de tal manera que puede ver todo el trafico que pasa por el wifi. En la sección A.2.1 se explica como instanciar un hotspot en la raspberry. Para descargar e instalar la extensión se corren los siguientes comandos.

Instalación de extensión para WeeWX en terminal de comandos

```
# descargar la óextensin
wget -O weewx-interceptor.zip
   HTTPs://github.com/matthewwall/weewx-interceptor/archive/master.zip

# instalar la estension
sudo wee_óextensin --install weewx-interceptor.zip

# reconfigurar el weewx
sudo wee_config --reconfigure --driver=user.interceptor --no-prompt
```

Al momento la extensión tiene un error en uno de sus archivos que debe ser modificado. El archivo que se encuentra en **/usr/share/weewx/user/interceptor.py** en la línea 1012 en la clase *Observer* la función *parser* dice:

Archivo a modificar

```
# /usr/share/weewx/user/interceptor.py
  def parse(self, s):
            pkt = dict()
```

```
            try:
                data = _cgi_to_dict(s)
                # FIXME: add option to use computer time instead of
                    station
                pkt['dateTime'] = self.decode_datetime(
                    data.pop('dateutc', int(time.time() + 0.5)))
```

y debe decir:

Archivo a modificar

```
# /usr/share/weewx/user/interceptor.py
  def parse(self, s):
            pkt = dict()
            try:
                data = _cgi_to_dict(s)
                # FIXME: add option to use computer time instead of
                    station
                #pkt['dateTime'] = self.decode_datetime(
                #    data.pop('dateutc', int(time.time() + 0.5)))
                pkt['dateTime'] = int(time.time() + 0.5)
```

Una vez corregido el error se debe editar el archivo de configuración para que utilice la extensión instalada. En la sección *[Station]* la línea que comienza con `station_type` debe quedar como se muestra a continuación.

Archivo de configuración de WeeWX

```
# /etc/weewx/weewx.conf
[Station]
  ...
  station_type = Interceptor
```

La sección *[Interceptor]* debe quedar de la siguiente manera.

Archivo de configuración de WeeWX

```
# /etc/weewx/weewx.conf
[Interceptor]
    driver = user.interceptor
    device_type = observer
    port = 8080
    address = 10.0.0.1
```

La extensión puede funcionar de dos maneras, en modo *listen* o en modo *sniff*. En modo *listen* los mensajes de la estación deben ser direccionados directamente hacia la dirección de

la computadora que corre el programa con la dirección y puerto especificados en las directivas *address* y *port*, en este modo el programa captura los paquetes y corta la comunicación hacia *weather underground*. En el modo *sniff* el programa solamente escucha el trafico de la red y guarda la información de los paquetes que se envían hacia *weather underground*, en este modo la información sigue llegando hasta el servidor de *weather underground*.

Configurando la línea que comienza con *device_type* en modo *observer* la extensión funciona en modo *listen*. En este modo la estación debe enviar los datos directamente hacia la dirección de la raspberry, para esto se instancia un servidor de DNS que le indica a la estación que la dirección de *weather underground* es en realidad la dirección de la raspberry. Es necesario indicar en el campo *address* cual es la dirección de IP de la raspberry. La estación envía los datos por protocolo HTTP al puerto TCP 80 de la raspberry, la extensión se configuro para escuchar en el puerto TCP 8080, por lo que es necesario redirijir todo lo que llegue al puerto TCP 80 de la raspberry hacia el puerto TCP 8080. En la sección A.2.1 se indica como instanciar un hotspot para que la estación se conecte a la red manejada por la raspberry y como instanciar el servidor DNS para que la estación envíe los datos hacia la raspberry. Además es necesario instanciar un servidor de DHCP para asignar las direcciones IP en la red, e indicarle a la estación que *gateway* y que servidor de DNS utilizar. Finalmente se indica como redirigir el tráfico de red del puerto TCP 80 al puerto TCP 8080.

A.2.1. Hotspot, DNS y DHCP

La estación meteorológica Daza WH2900 entrega los datos por internet a través de una red wifi. Se configura para que envíe los datos a algun *host* preparado para recibirlos por protocolo HTTP, en este trabajo se configuró para que envíe los datos al *host* con dominio *rtupdate.wunderground.com*.

El dispositivo que sirve la red wifi funciona como un switch de capa 2, al conectarse la estación esta no posee ni dirección IP propia ni conoce la dirección IP de ningún dispositivo en la red. Luego de conectarse a la red wifi la estación envía un paquete DHCP Discover (Dynamic Host Configuration Protocol), este es un paquete del tipo broadcast lo que permite que llegue a todos los dispositivos conectados a la red. El paquete funciona como una solicitud para que algún servidor de DHCP le indique a la estación cual sera su dirección IP, cual es la dirección IP del servidor DNS y cual es el IP del gateway. Un servidor DHCP es un dispositivo de red que administra las direcciones IP en una red, se encarga de asignar las direcciones IP e informar las direcciones IP del gateway y de los servidores de DNS. Si hay algún servidor de DHCP conectado en la red este recibirá el paquete DHCP Discover y responderá la solicitud con la información de red requerida.

Una vez que la estación adquiere el IP del gateway solo le hace falta adquirir la dirección MAC del gateway, para esto envía un paquete ARP del tipo broadcast y recibirá como respuesta la dirección MAC del gateway. El paquete ARP (Adress Resolution Protocol) es

un paquete de capa 2 que sirve para obtener la dirección MAC de algún dispositivo que solo se conoce la dirección IP. Si el dispositivo que tiene el IP de gateway está conectado a la red entonces responderá el paquete ARP con su dirección MAC. Ahora la estación se encuentra lista para enviar un paquete de DNS preguntando por la dirección IP del dominio *rtupdate.wunderground.com*. Una vez que recibe la respuesta de DNS con la dirección IP del dominio la estación comienza a enviar paquetes HTTP GET con la información de las mediciones con un período de aproximadamente 5 minutos. La información se envía como texto plano en el paquete HTTP GET.

Para obtener los datos de la estación meteorológica usando el software Weewx es necesario que la estación envíe los datos a la dirección de la computadora donde se está corriendo el Weewx. Para esto se configura un hotspot, un servidor de DNS y un servidor de DHCP en la raspberry. El hotspot es una red wifi, se utiliza la raspberry para brindar este servicio con el programa hostapd y se conecta la estación al hotspot. Una vez que se conecta la estación es necesario algún dispositivo conectado a la red que brinde el servicio de servidor DHCP y servidor de DNS. Estos dos servicios se brindan en la raspberry con el programa *dnsmasq*. En el servidor de DNS se indica que la dirección del dominio *rtupdate.wunderground.com* es la dirección de la raspberry misma, de esta manera la información de la estación es enviada directamente hacia la raspberry por protocolo HTTP. Para recibir los datos de la estación es necesario que la raspberry tenga algún servicio en el puerto TCP 80 de HTTP, de esto se encarga el programa Weewx.

El programa hostapd se puede instalar en la raspberry abriendo una terminal y corriendo el comando `sudo apt install hostapd`. Para que funcione el servicio es necesario configurar el archivo que se encuentra en **/etc/network/interfaces**, este archivo es el que indica como se deben configurar todas las interfaces de red de la raspberry. El demonio *networking* que corre en el `systemd` de la raspberry es el que se encarga de leer el archivo y configurar las interfaces como se describe en el archivo. La sintaxis del archivo de configuración **interfaces** se puede leer abriendo una terminal y corriendo el comando `man interfaces`. El sistema operativo le asigna un nombre a cada interfaz, es necesario identificar cual es el nombre de la interfaz que se va a utilizar, la interfaz wifi usualmente recibe el nombre *wlan0* y si hay mas de una la segunda suele recibir el nombre *wlan1*. Para ver el nombre de las interfaces se puede correr el comando `ifconfig -s` el cual lista las interfaces.

Para instanciar un hotspot en la interfaz wlan0 el arhivo en **/etc/network/interfaces** debe quedar de la siguiente manera, además se agregan las directivas para conectarse a una red wifi externa como cliente en la interfaz wlan1.

Archivo de configuración de interfaces

```
# /etc/network/interfaces

# interfaz ethernet en cliente dhcp
```

```
auto eth0
iface eth0 inet dhcp

#interfaz wlan0 en modo hotspot
auto wlan0
iface wlan0 inet static
    address 10.0.0.1
    netmask 255.255.255.0
    post-up iptables -t nat -A PREROUTING -i wlan0 -p tcp --dport 80 -j DNAT
        --to-destination 10.0.0.1:8080

#interfaz wlan1 en modo cliente
allow-hotplug wlan1
auto wlan1
iface wlan1 inet dhcp
  wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf
```

La línea que comienza con *auto* seguida del nombre de una interfaz indica que esa interfaz debe ser iniciada siempre que se prende la computadora. La línea con la directiva *allow-hotplug wlan1* indica que la intefaz debe ser inicializada siempre que sea detectada, se utiliza para interfaces que pueden ser removidas o conectadas despues de iniciar la computadora como un *dongle* wifi. La línea con la sintaxis *iface *nombre de interfaz* inet *opcion** indica como se va a manejar la dirección IPv4 de la interfaz indicada entre asteriscos, las opciones posibles son dhcp, manual y static.

- **dhcp** la administración de direcciones IP es realizada por el cliente dhcp por defecto del sistema. Agregando la línea *wpa-conf /etc/wpa_supplicant/wpa_supplicant.conf* debajo se habilita al demonio wpa_supplicant para que conecte la interfaz a una red wifi.

- **manual** no se asigna dirección IP por defecto y la interfaz se administra con los comandos *if up* e *if down*.

- **static** las direcciones IP se definen a continuación y se mantienen estaticas. Las directivas mas relevantes son address, netmask, gateway que se corresponden con la dirección IPv4 de la interfaz, la mascara de red y el *default gateway* de la interfaz. También se pueden agregar directivas de redirecciónamiento como *post-up iptables -t nat -A PREROUTING -i wlan0 -p tcp –dport 80 -j DNAT –to-destination 10.0.0.1:8080* que redirije lo que llegue al puerto TCP 80 de la interfaz wlan0 hacia el IP 10.0.0.1 puerto TCP 8080. Las directivas de redirecciónamiento deben cumplir la sintaxis de *iptables* que se puede ver corriendo el comando `man iptables`.

Para instanciar un servidor DHCP y DNS en la raspberry se utiliza el demonio dnsmasq que se puede instalar abriendo una terminal y corriendo el comando *sudo apt install dnsmasq*.

El archivo de configuración de dnsmasq se encuentra en */etc/dnsmasq.conf*, se configuró de la siguiente manera.

Archivo de configuración de dnsmasq

```
# /etc/dnsmasq.conf

interface=wlan0
# direcciones que el servidor dhcp va a otorgar.
dhcp-range=10.0.0.3,10.0.0.254,12h
# no leer /etc/resolve.conf
no-resolv
# no leer /etc/hosts
no-hosts
# leer de /etc/myhosts
addn-hosts=/etc/myhosts
```

En la primera línea se define en que interfaz va a funcionar el demonio, la interfaz es wlan0 que es la interfaz que sirve el hotspot. La línea que comienza con *dhcp-range* define cuales son las direcciones de IP que se van a servir, se completo con las direcciones desde 10.0.0.3 hasta 10.0.0.254 y al final se especifica que cada dirección IP que se asigna se reserva por al menos 12 horas. La línea con la directiva *no-resolv* indica que el servidor de DNS no debe resolver los pedidos que no conoce utilizando los servidores definidos en */etc/resolv.conf*. El archivo *resolv.conf* tiene registrado los servidores de DNS que utiliza el sistema para resolver pedidos propios. La línea con la directiva *no-hosts* indica que el servidor DNS no debe resolver los pedidos de DNS que no conoce con los *hosts* que estan guardados en el arhivo que esta en */etc/hosts*. El archivo *hosts* tiene registrados los IP's de algunos dominios, usualmente dominios locales como la asignación de la dirección 127.0.0.1 con el dominio *localhost*. La línea con la directiva *addn-hosts=/etc/myhosts* indica que el servidor de DNS puede responder pedidos con los *hosts* que estan guardados en el archivo que esta en */etc/myhosts*. La configuración del archivo en el *path /etc/myhosts* se muestra a continuación.

Archivo de configuración hosts

```
# /etc/myhosts

10.0.0.1 rtupdate.wunderground.com
```

En el archivo se declaro un unico *host,rtupdate.wunderground.com* con el IP 10.0.0.1 que es el de la raspberry. De esta manera el servidor de DNS va a responder pedidos unicamente con los *hosts* que estan guardados en el archivo que esta en */etc/myhosts*. El unico *host* que esta definido en el archivo es el de *rtupdate.wunderground.com*.

Para instanciar el hotspot se debe editar el archivo de configuración del demonio hostapd,

el arhivo se encuentra en la dirección */etc/hostapd/hotsapd.conf*. El archivo se configuró de la siguiente manera:

Archivo de configuración de hostapd

```
# /etc/hostapd/hostapd.conf

interface=wlan0
driver=nl80211
country_code=AR
ssid=meteo\_wifi
hw_mode=g
channel=4
wpa=2
wpa_passphrase=f64CTAm
ignore_broadcast_ssid=1
```

- **interface** define la interfaz sobre la cual se sirve el hotspot.
- **country_code** debe definir el país donde se encuentra el hotspot, esto tiene que ver con las frecuencias habilitadas para el wifi de cada país.
- **driver** define el driver que se utiliza para manejar la interfaz, el valor usado *nl80211* es correcto para la mayoría de los controladores.
- **ssid** define el SSID del wifi que es el nombre con el cual se identifica.
- **hw_mode** indica la version del protocolo 802.11 utilizada, la version g es la más reciente soportada por el dispositivo.
- **channel** especifica el canal de frecuencia utilizado por el dispositvo.
- **wpa** indica la version del protocolo de seguridad utilizado, la versión 2 es la más reciente.
- **wpa_passphrase** especifica la contraseña de la red wifi.
- **ignore_broadcast_ssid** especifica si se utiliza un ssid oculto o no, el valor 1 indica que se oculta el ssid.

Para conectarse a una red wifi se debe configurar el demonio *wpa_supplicant*, el archivo de configuración de este se encuentra en */etc/wpa_supplicant/wpa_suppicant.conf*. El archivo de configuración debe quedar de la siguiente manera:

Archivo de configuración de wpa_supplicant

```
# /etc/wpa_supplicant/wpa_supplicant.conf

ctrl_interface=DIR=/var/run/wpa_supplicant GROUP=netdev
update_config=1
country=AR

network={
         ssid="ssid"
         psk="ñcontrasea"
}
```

El campo entre comillas ssid debe ser completado con la el ssid al que se desea conectarse y el campo psk debe ser reemplazado por la contraseña del wifi.

Una vez implementadas las configuraciones mencionadas se deben habilitar e iniciar los servicios para que comienze a funcionar el hotspot como se desea. Se corren los comandos **sudo ifconfig wlan0 up**,**sudo systemctl umask hostapd**, *sudo systemctl enable hostapd*, **sudo systemctl start hostapd**, **sudo systemctl enable dnsmasq**, **sudo systemctl start dnsmasq** para levantar la interfaz wlan0, para desbloquear habilitar e iniciar el servicio hostapd y para habilitar e iniciar el servicio dnsmasq. Habilitar un servicio significa habilitar a que este se corra automaticamente cada vez que se prende la computadora. Para iniciar el demonio wpa_supplicant que conecta la raspberry a una red wifi se corren los comandos **sudo systemctl enable wpa_supplicant**, **sudo systemctl start wpa_suppicant** que habilitan e inician el demonio.

A.3. Estación modelo WH-1080

Para el proyecto se configuró una estación Fine Offset WH-1080 que se encuentra en Los Coihues. Esta estación se comunica por USB con un software del fabricante que solo corre en Windows y MacOS, por esta razón se utilizó el programa Weewx para poder recuperar los datos en la rasberry que utiliza sistema operativo Linux. Se instala el programa y se configura la extensión de MQTT como se indico anteriormente. El archivo de configuración para esta estación debe quedar de la siguiente manera. En la sección *[Station]* la línea que comienza con *station_type* debe quedar como se muestra a continuacción.

Archivo de configuración de WeeWX

```
# /etc/weewx/weewx.conf

[Station]
  station_type = FineOffsetUSB
```

La sección *[FineOffsetUSB]* debe quedar de la siguiente manera.

Archivo de configuración de WeeWX

```
# /etc/weewx/weewx.conf

[FineOffsetUSB]
    # Modelo de la estacion, opciones: WH1080, WS1090, WS2080, WH3081
    model = WH1080

    # Tiempo en segundos de pedido de datos a la estacion
    polling_interval = 60

    # Driver a utilizar
    driver = weewx.drivers.fousb
```

La línea que comienza con *model* debe ser completada con el modelo de la estación. La línea que comienza con *polling_interval* debe ser completada con el período en segundos con el cual se interrogara a la estación por USB para solicitar un dato nuevo. La línea que comienza con *driver* indica el driver que se utiliza para recuperar los datos, se debe completar con *weewx.drivers.fousb*.

A.4. Estación modelo WHM-200A

Otra de las estaciones que se utilizó en el proyecto fue una Oregon Scientific WMR-200A. Esta estación al igual que la WH-1080 entrega los datos por USB. Se utilizó el software Weewx para recuperar los datos, almacenarlos en una base de datos SQL y enviarlos por MQTT. Se instala el programa y la extensión para MQTT como se indicó anteriormente. El archivo de configuración para esta estación debe quedar de la siguiente manera. En la sección *[Station]* la línea que comienza con *station_type* debe quedar como se muestra a continuacción.

Archivo de configuración de WeeWX

```
# /etc/weewx/weewx.conf

[Station]
  station_type = WMR200
```

La sección *[WMR200]* debe quedar como se muestra a continuación.

Archivo de configuración de WeeWX

```
# /etc/weewx/weewx.conf
```

```
[WMR200]
  # modelo de la estacion
    model = WMR200A

    # usar tiempo de la computadora
    use_pc_time = True

  # Tiempo en segundos de pedido de datos a la óestacin
    polling_interval = 60

    driver = weewx.drivers.wmr200
```

La línea que comienza con *model* debe ser completada con el modelo de la estación. La línea que comienza con *polling_interval* debe ser completada con el período en segundos con el cual se interrogara a la estación por USB para solicitar un dato nuevo. La línea que comienza con *driver* indica el driver que se utiliza para recuperar los datos, se debe completar con *weewx.drivers.wmr200*.

Apéndice B

Instructivo de instalación de Raspbian

Se utilizó el sistema operativo Raspbian para correr en las *raspberrys pi*. Este sistema operativo esta especialmente diseñado para estas computadoras. Se decidió utilizar la misma versión de Raspbian en todas las *raspberrys* para evitar inconvenientes. La versión utilizada fue la *Stretch*, esta versión se puede descargar del repositorio que se encuentra en el url *http://downloads.raspberrypi.org/raspbian/images/*. Del repositorio se descarga la imagen comprimida en un archivo .zip. Se descomprime la imagen y se obtiene una archivo .img. Para instalar Raspbian se debe grabar la imagen en la tarjeta de memoria, esto significa copiar correctamente el archivo a la tarjeta de memoria. Para esto se utilizó el programa Balena-etcher, este es un programa de uso libre y codigo abierto disponible para Linux, Windows y macOS.

Una vez grabada la tarjeta de memoria se introduce en la raspberry y se enciende. La primera vez que se prende la raspberry es necesario configurarla utilizando un monitor y un teclado. Para poder acceder remotamente a la rasperry por protoclo SSH es necesario habilitar el servidor. Se corre el comando `sudo raspi-config`, en la sección *interface-options-¿SSH* se habilita el servicio.

Apéndice C

Instructivo de `Interceptor` con SDR

La estación meteorológica DAZA WH-2900 transmite la información sensada por RF en una frecuencia de 433 MHz. Se utilizó una SDR con interfaz USB conectada a una raspberry pi 3 B+ para recuperar los datos transmitidos. Para la raspberry se utilizó el sistema operativo Raspbian Stretch. Para interactuar con la SDR se utilizó la biblioteca `rtl_433`.

La biblioteca se puede instalar corriendo los siguientes comandos en una terminal.

Instalación de biblioteca rtl_433 en terminal de comandos

```
# instalar dependencias
sudo apt-get install git libtool libusb-1.0.0-dev librtlsdr-dev rtl-sdr
    build-essential autoconf cmake pkg-config

# descargar la libreria del repositorio en la carpeta rtl_433
git clone https://github.com/merbanan/rtl_433.git

cd rtl_433/
mkdir build
cd build/
cmake ../

# compilar la libreria
make
# instalar la libreria en la computadora
sudo make install

# instalar cliente MQTT
sudo apt install mosquitto-clients
```

Corriendo el comando `man rtl_433` se puede observar el manual de la biblioteca. Se pue-

de observar que la biblioteca soporta la decodificación de una gran cantidad de dispositivos comerciales. El código para la estación meteorológica WH-2900 es el 78. Corriendo el comando `rtl_433 -F json -M utc -R 78` se comienza a recibir la información transmitida por la estación. El comando detecta automáticamente la SDR, no es necesario realizar ninguna configración adicional.

Para envíar los datos recibidos por MQTT se puede utilizar el cliente de línea de comandos mosquitto. Este se puede instalar corriendo el comando `sudo apt install mosquitto-clients`. Una vez instalado se puede correr el comando `rtl_433 -F json -M utc -R 78 | mosquitto_-pub -t topic/ -u usuario -P contraseña -p puerto -h direccion --cafile path/al/certificado`, con este comando se escuchan los datos transmitidos por la estación y se retrasmiten los datos por MQTT. Es necesario especificar el topic, usuario, contraseña, puerto TCP, dirección IP y certificado del broker MQTT.

Para dejar a la raspberry recopilando datos se instanció un demonio en el systemd de la raspberry que corre el comando mencionado. Para instanciar un demonio se crea un archivo en el directorio **/etc/systemd/system** con el nombre *interceptor.service*. El archivo *interceptor.service* se ve de la siguiente manera.

Archivo de configuración del demonio interceptor

```
# /etc/systemd/system/interceptor.service

[Unit]
Description=Interceptor Service
# Iniciar despues de que estan iniciadas las interfaces de red
After=network.target

[Service]
# Reiniciar siempre en cualquier caso
Restart=always
# Comando que se corre al iniciar el demonio
ExecStart=/home/pi/interceptor/interceptor.sh

[Install]
WantedBy=multi-user.target
```

El demonio corre el *script* interceptor.sh, el script esta compuesto por el comando mostrado anteriormente. La línea *Restart=always* causa que el demonio sea reiniciado en cualquier ocasión en la cual el preceso se termine. De esta manera el demonio queda escuchando la información que transmite la estación y la retrasmite por MQTT.

Para habilitar el demonio y dejarlo corriendo se corren los siguientes comandos.

Habilitar demonio en terminal de comandos

```
# habilitar servicio
sudo systemctl enable interceptor

# iniciar servicio
sudo systemctl start interceptor

# ver estado del servicio
sudo systemctl status interceptor
```

Para recibir los datos con el programa `Estafeta` se crea la siguiente clase en el archivo *inject.py* y se registra al sensor en el archivo *stations.json*. La clase sigue la sintaxis necesaria para poder guardar la información transmitida por MQTT en la base de datos del proyecto.

Modulo para la recepción de mensajes del interceptor

```
# Inject.py

class Interceptor01(sensor):
    def __init__(self, typ, net, subnet, station):
        super(Interceptor01, self).__init__(typ='station', net=net, subnet=subne
t, station=station)
        self.datapath =
            "{}stations/{}_{}/".format(ROOT_PATH,self.subnet,self.station)
        self.angleN = 0
        #print 'usando INTERCEPTOR01'

    def inject_mqtt(self, mqttdata):
        dat = self.translate_mqtt(mqttdata)
        return self.inject([dat], 'data')

    def translate_mqtt(self,raw):
        def tconv(ts):
            time = dt.datetime.strptime(ts,'%Y-%m-%d
                %H:%M:%S')+dt.timedelta(hours=0)
            return time.isoformat("T")
        def dir_inv_conv(angle):
            return u.dir_inv(angle, self.angleN)

        dat = []
        lT = [['time' ,'time' ,tconv],
```

```
            ['outTemp_F' ,'temp' ,float],
            ['humidity' ,'hum' ,float],
            ['wind_speed_ms' ,'vel' ,float],
            ['wind_dir_deg' ,'dir' ,float],
            ['wind_dir_deg' ,'dir.l' ,dir_inv_conv],
            ['gust_speed_ms' ,'vel.gust' ,float],
            ['wind_dir_deg' ,'dir.gust' ,float],
            ['wind_dir_deg' ,'dir.gust.l' ,dir_inv_conv],
            ['rainfall_mm' ,'rain' ,float],
            ['light_lux' ,'sol' ,float],
            ['uvi' ,'UV' ,float],
            ['temperature_C' ,'temp' ,float],
           ]
        dat = self._t(raw,lT)
        return dat
```

Apéndice D

Consolidación de datos

Está anexo comprende la consolidación de los datos, esto significa asegurarse que los datos medidos lleguen a la base de datos. Se implementó las consolidación para las estaciónes meteorológicas instaladas.

En las estaciónes meteorológicas instaladas se extraen los datos utilizando una computadora raspberry que corre el software WeeWX. Este se comunica con la estación, guarda los datos en una base de datos SQLite[1] en el directorio **/var/lib/weewx/weewx.sdb** y luego los envía por MQTT. Los datos son recibidos por MQTT por el programa Estafeta que organiza la información y la guarda en una base de datos InfluxDB.

Por distintas razónes ocurre que los datos no siempre llegan hasta la base de datos InfluxDB, para esto es necesario la consolidación de datos. La solución implementada es la siguiente, se transmite toda la base de datos SQLite hacia la computadora donde corre el programa Estafeta y luego se corre el programa `inject.py` el cual guarda los datos en la base de datos InfluxDB.

La transmisión se realiza con el comando `scp` (Secure Copy) perteneciente al protoclo SHH. El comando que se corre es `scp /var/lib/weewx/weewx.sdb user@ip:meteo/datasources/`. El comando copia la base de datos que se encuentra en el directorio **/var/lib/weewx/-weewx.sdb** de la computadora que corre el comando en el directorio **/home/user/meteo/datasources** de la computadora con dirección IP *ip*. El usuario que corre el comando debe ser capaz de autenticarse ante el usuario *user* de la computadota con dirección IP *ip*.

El comando se corre en la raspberry donde se encuentra la base de datos y el IP de destino es el del servidor donde se encuentra la base de datos InfluxDB del proyecto. Para lograr pasar la etapa de autenticación se crea un usuario en el servidor y se habilita a la raspberry a autenticarse ante este usuario con su clave pública.

Una vez que se copia la base de datos se corre el programa `inject.py` en el servidor. El programa `inject.py` es capaz de leer la base de datos SQLite y guardar la información en la base de datos InfluxDB. La base de datos InfluxDB no guarda dos veces un punto con

[1]SQLite es una de las bases de datos más utilizadas, está escrita en lenguaje C y cumple con la sintaxis SQL.

el mismo valor de tiempo, de esta manera la información no queda repetida en la base de datos.

El programa `inject.py` esta diseñado en Python 2.7, utiliza el archivo *stations.json* para guardar de manera correcta la información en la base de datos. A continuación se puede ver una simplificación del código del programa.

Programa Inject.py

```
import datetime as dt
import json
import sqlite3
from influxdb import InfluxDBClient

class sensor(object):
    project = 'meteo'
    def __init__(self, typ, net, subnet, station):
        self.type    = typ
        self.net     = net
        self.subnet  = subnet
        self.station = station
        self.mqtt_btopic =
           "{}/{}/{}/{}/{}/{{port}}".format(self.project,typ,net,subnet,station)
        self.stdata = stationData(typ=typ, net=net, subnet=subnet,
           station=station)
        self.subnet  = subnet
        self.station = station
        if self.stdata=={}:
            print "ERROR: station not registered. 
               ({}/{}/{}/{})".format(typ,net,subnet,station)
            return
        self.influx = {'database': "{}-{}".format(self.project,typ),
                       'meas':net,
                       'tags': {'subnet' : subnet,
                                'station': station,
                                'id'     : self.stdata['id'],
                                'port'   :'data',
                            },
                                # hasta aca
                        'extra_fields':{
                                'lat'  : self.stdata['lat'],
                                'long' : self.stdata['long'],
```

```
                                'heigh': self.stdata['heigh'],
                                'geohash' :
                                   geohash2.encode(self.stdata['lat'],self
                                        }
                    }
        return

    def _t(self,din,listTrans):
        ou = {}
        for ki,ko,f in listTrans:
            try:
                val = din[ki]
                try:
                    val=val.strip()
                except:
                    pass
                if val not in ['--', None]:
                    try:
                        ou[ko] = f(din[ki])
                    except:
                        pass
            except:
                pass
        return ou

    def inject(self, tfields, port):
        db = self.influx['database']
        if db not in dbList:
            influx.create_database(db)
            dbList.append(db)
            print 'Database: {} created.'.format(db)
        influx.switch_database(db)

        indata = []
        for d in tfields:
      time = d['time']
            del d['time']
            xfields = self.influx['extra_fields']
            for key in xfields:
                d[key] = xfields[key]
            #tags
```

```
            tags = self.influx['tags']
            tags['port'] = port
            #out
            indata.append({
                        "time":         time,
                        "measurement": self.influx['meas'],
                        "tags":         tags,
                        "fields":       d
                    })

        influx.write_points(indata)
  return indata

class WeeWX(sensor):
    def __init__(self, net, subnet, station):
        super(WeeWX, self).__init__(typ='station', net=net,
           subnet=subnet, station=station)
        self.datapath =
           "{}stations/{}_{}/".format(ROOT_PATH,self.subnet,self.station)
        if self.stdata['id']=='CAB_01':
            self.angleN = 243.
        else:
            self.angleN = 0.

    def inject_date(self, date):
        raw = self.rawdata(date)
        dat = self.translate(raw, date)
        return self.inject(dat, 'data')

    def rawdata(self,date=dt.date(2001,12,20),query=''):

        #### read database
        filDB = self.datapath + 'weewx.sdb'
        # init
        dt0 = dt.datetime(date.year,
                          date.month,
                          date.day)
        dt1 = dt0 + dt.timedelta(days=1)
        dt0 = dt0.isoformat('␣')
        dt1 = dt1.isoformat('␣')
```

```
        Q  = "SELECT␣*␣from␣archive␣"
        Q += "WHERE␣datetime(datetime,␣'unixepoch','localtime')␣"
        Q += "BETWEEN␣datetime('{}')␣".format(dt0)
        Q += "AND␣datetime('{}')␣ORDER␣BY␣dateTime␣ASC␣".format(dt1)
        #Q += "limit 10" #(debug)

        if query != '':
            Q = query

        # connect
        with sqlite3.connect(filDB) as sqlite:
            query = sqlite.cursor()
            query.execute(Q)
            rows = query.fetchall()
            # keys
            keys = []
            for desc in query.description:
                keys.append(desc[0])

        ### json out
        ou = []
        for item in rows:
            d = {}
            for k,v in zip(keys,item):
                d[k]=v
            ou.append(d)
        return ou

    def show_fields(self):
        [raw] = self.rawdata(query="SELECT␣*␣FROM␣archive␣limit␣1")
        [dt0] = self.rawdata(query="SELECT␣datetime(dateTime,␣
           'unixepoch','localtime')␣FROM␣archive␣ORDER␣BY␣dateTime␣
           ASC␣LIMIT␣1")
        [dt1] = self.rawdata(query="SELECT␣datetime(dateTime,␣
           'unixepoch','localtime')␣FROM␣archive␣ORDER␣BY␣dateTime␣
           DESC␣LIMIT␣1")
        # out dates
        print 'dates␣range'
        for key in dt0:
            print '\t'+str(dt0[key])
            print '\t'+str(dt1[key])
```

```
        # out keys
        for i,key in enumerate(raw):
            print
               '{:3d}'.format(i)+str(key)+':\t'+str(raw[key])+str(type(raw[key]))

    def translate(self,raw,date=dt.date(2001,12,20)):
        def tconv(ts):
            t = dt.datetime.fromtimestamp(int(ts), tz=ARG)
      return t.isoformat("T")
        def dir_conv(angle):
            return u.dir0(angle, self.angleN)
        def dir_inv_conv(angle):
            return u.dir_inv(angle, self.angleN)

        dat = []
        for r in raw:
          if (r['usUnits']==1) :
            lT = [['dateTime'   ,'time'    ,tconv],
                  ['outTemp'    ,'temp'    ,u.F2C],
                  ['inTemp'     ,'temp.in' ,u.F2C],
                  ['dewpoint'   ,'dew'     ,u.F2C],
                  ['windchill'  ,'chill'   ,u.F2C],
                  ['heatindex'  ,'heat'    ,u.F2C],
                  ['inHumidity' ,'hum.in'  ,float],
                  ['outHumidity','hum'     ,float],
                  ['barometer'  ,'bar'     ,u.inHg2hPa],
                  ['pressure'   ,'pres'    ,u.inHg2hPa],
                  ['windSpeed'  ,'vel'     ,u.mh2kmh],
                  ['windDir'    ,'dir'     ,dir_conv],
                  ['windDir'    ,'dir.l'   ,dir_inv_conv],
                  ['windGust'   ,'vel.gust',u.mh2kmh],
                  ['windGustDir','dir.gust',dir_conv],
                  ['rainRate'   ,'rain'    ,u.in2mm],
                  ['rain'       ,'rain.a'  ,u.in2mm],
                  ['radiation'  ,'sol'     ,float],
                  ['UV'         ,'UV'      ,float],
                  ['extraTemp1' ,'temp.e01',u.F2C],
                  ['extraTemp2' ,'temp.e02',u.F2C],
                  ['extraTemp3' ,'temp.e03',u.F2C],
                  ['soilTemp1'  ,'temp.e04',u.F2C],
                  ['soilTemp2'  ,'temp.e05',u.F2C],
```

```
                    ['soilTemp3'  ,'temp.e06',u.F2C],
                    ['soilTemp4'  ,'temp.e07',u.F2C],
                    ['leafTemp1'  ,'temp.e08',u.F2C],
                    ['leafTemp2'  ,'temp.e09',u.F2C],
                    ['extraHumid1','hum.e01' ,float],
                    ['extraHumid2','hum.e02' ,float],
                    ['soilMoist1' ,'hum.e03' ,float],
                    ['soilMoist2' ,'hum.e04' ,float],
                    ['soilMoist3' ,'hum.e05' ,float],
                    ['soilMoist4' ,'hum.e06' ,float],
                    ['leafWet1'   ,'hum.e07' ,float],
                    ['leafWet2'   ,'hum.e08' ,float],
                  ]
            ou = self._t(r,lT)
            dat.append(ou)
        return dat


if __name__ == '__main__':

    fecha = dt.date(2018, 9, 27)
    date_end   = dt.date(2019, 10, 04)

  st = WeeWX(net='meteo', subnet='coihues', station='01')

    while fecha <= date_end + dt.timedelta(hour=1):
        r = st.inject_date(fecha)
        fecha = fecha + dt.timedelta(days=1)
```

Apéndice E

Recolección de datos con Scrapy

`Scrapy` es una plataforma escrita en Python de fuente libre y abierta para recolectar datos de páginas web. Trabaja con *spiders* que son clases de Python estándarizadas para realizar el trabajo de recolectar la información.

Para instalar `Scrapy` se utiliza la herramienta *Conda*. Esta es un instalador de bibliotecas que se encarga de resolver las dependencias entre programas. Se crea un *enviroment* para el proyecto, esto es una versión específica de Python con todas las dependencias necesarias para correr un programa en especial.

Crear enviroment para Scrapy e instalar dependencias

```
conda create --name scrap
conda install -n scrap -c conda-forge beautifulsoup4
conda install -n scrap -c conda-forge scrapy
conda activate scrap
```

Se crea un directorio para el proyecto de *Scrapy*. Se abre una terminal en el directorio y se corre el siguiente comando.

```
scrapy startproject myproyect
```

El comando crea la siguiente estructura de directorios:

- **myproyect/** directorio del proyecto. Aquí van los archivos de salida con los datos recolectados.
- **./scrapy.cfg** configuración general.
- **./myproyect** directorio para modulos de Python.

- __**init.py**__ archivo de inicialización del proyecto.
- **items.py**
- **spiders/** aquí se guardan los *spiders*.
- **items.py** items del proyecto.
- **middlewares.py** *middlewares* del proyecto.
- **pipelines.py** *pipeline* del proyecto.
- **settings.py** más configuraciónes del proyecto.

Todos los archivos de configuración se pueden dejar en su estado por defecto.

Para programar un *spider* primero hay que crearlo, para crearlo se corren los siguientes comandos

Crear un spider

```
cd myproject
scrapy genspider myspider https://myurl
```

Con el comando se crea el archivo myspider.py, que es el *spider* que recolecta datos de la web. Un *spider* hereda la clase *scrapy.Spider* y debe definir los siguientes items.

- **allowed_domains** dominio donde fuciona el *spider*.
- **customw_settings** define el formato y el nombre del archivo de salida.
- **parse** función que genera la salida del *spider*.

Para el proyecto se crearon 2 *spiders* uno para la página del Servicio Meteorológico Nacional y otro para *weather underground*. A continuación el código de cada uno.

Spider del Servicio Meteorológico Nacional "smn.py"

```
import scrapy
import datetime as dt
import json

class SmnSpider(scrapy.Spider):
    name = 'smn'
    custom_settings={ 'FEED_URI': "smnTest.json",
                      'FEED_FORMAT': 'json'}
```

```
allowed_domains = ['ws.smn.gob.ar']
today = dt.datetime.now()

def __init__(self, station= 'BARILOCHEAERO',
                 date = None , **kwargs):

    # Select station
    if station=='BARILOCHEAERO':
        self.station = 'BARILOCHE AERO'
        self.st_number = 87765
        self.lid = 137
    else:
        self.station = 'BARILOCHE AERO'
        self.st_number = 87765
        self.lid = 137

    # Now data / Date data
    if date is None or date==self.today.strftime('%Y-%m-%d'):
        self.datatype = 'now'
        self.start_urls = ['http://ws.smn.gob.ar/map_items/weather/']
    else:
        print ('DDDAAATTTEEE: ',date)
        self.datatype='date'
        d = date.replace('-','')
        turl =
            'https://ssl.smn.gob.ar/dpd/descarga_opendata.php?file=observaciones/datoho
        self.start_urls = [turl.format(s=station, d=d)]
    super().__init__(**kwargs)

def parse_now(self, response):
    data = json.loads(response.body_as_unicode())
    brc = list(filter(lambda x: x['int_number']==87765 and x['lid']==137,
        data))
    try:
        brc = brc[0]
    except:
        return None

    #print('-'*80)
    #print(brc)
    #print('-'*80)
```

```
        out = {}
        time = dt.datetime.fromtimestamp(brc['updated'])
        out['Time'] = time.strftime('%I:%M %p')
        wtr = brc['weather']
        out['humidity'] = wtr['humidity']
        out['pressure'] = wtr['pressure']
        out['st'] = wtr['st']
        out['visibility'] = wtr['visibility']
        out['wind_speed'] = wtr['wind_speed']
        out['temp'] = wtr['temp']
        out['wing_degL'] = wtr['wing_deg']
        out['tempDesc'] = wtr['tempDesc']
        out['description'] = wtr['description']

        return out

    def parse_day(self, response):
        data = str(response.body_as_unicode()).split('\r\n')
        for item in data[2:]:
            if self.station in item:
                fecha, hora, temp, hum, pnm, dd, ff = item.split()[:7]
                out = {}
                time = dt.datetime.strptime(fecha+'T'+hora, '%d%m%YT%H')
                out['Time']=time.strftime('%I:%M %p')
                out['temp']=temp
                out['humidity']=hum
                out['pressure']=pnm
                out['wing_degN']=dd
                out['wind_speed']=ff
                yield out

    def parse(self, response):
        if self.datatype == 'now':
            return self.parse_now(response)
        else:
            return self.parse_day(response)
```

Spider de la página weather underground, "wu.py"

```
import scrapy
from bs4 import BeautifulSoup
```

```
class WuSpider(scrapy.Spider):
    name = 'wu'
    custom_settings={ #'FEED_URI': "test_%(time)s.json",
                    'FEED_URI': "wuTest.json",
                    'FEED_FORMAT': 'json'}



    def __init__(self, station='IRNBARIL5',
                     date='2019-09-12' , **kwargs):

        turl =
            'https://www.wunderground.com/dashboard/pws/{s}/table/{d}/{d}/daily'
        self.allowed_domains = ['www.wunderground.com']
        self.start_urls = [turl.format(s=station, d=date)]
        super().__init__(**kwargs)

    def parse(self, response):
        ## Read Table
        soup = BeautifulSoup(response.body ,'lxml')
        table = soup.find_all('table','desktop-table')[0]
        ## Keys
        keys = []
        for item in table.thead.tr.find_all('th'):
            keys.append(item.string)
        ## Data
        rows = table.tbody.find_all('tr')
        for row in rows:
            out = {}
            for i,item in enumerate(row.find_all('td')):
                key = keys[i]
                for k,s in enumerate(item.stripped_strings):
                    if k==0:
                        out[key] = s
                    else:
                        out[key+'.u'] = s

            yield out
```

Para correr los spiders se activa el *enviroment* de `Anaconda`, y se corre el comando

`scrapy crawl -a station=IBARILOC2 -a date=12/11/2019` en el directorio del proyecto. El comando corre los spiders definidos en el directorio *spiders/* y les pasa los argumentos definidos con la opción `-a`.

Se creó un *script* en Python que corre los spiders y se ocupa organizar la información y enviarla por MQTT. A continuación el *script*:

"Script.py", corre spiders y envía la información por MQTT

```
import json
import datetime as dt
import os
import sys
import subprocess
from copy import copy
import paho.mqtt.client as mqtt

transmit = True
stationsFile = '../../stations.json'
tmpDataFile = 'Test.json'
dataPath = 'data/'
templateFile = '{s}_{d}.dat'
loadDataFile = 'LoadData.dat'

#runPath = '/home/willy/Programs/anaconda3/envs/scrap/bin/' # HP
runPath = '/home/willy/Programas/anaconda/envs/scrap/bin/' # Oficina

## mqtt comm
certData = '../../ca.gonchigg.crt'
mqtt_user = 'scrap'
mqtt_pass = 'f64CTAm'
broker_ip = '159.89.80.129'
broker_ip = '127.0.0.1'
broker_port = 8883
broker_port = 9993
## tunnel comand
# ssh -L 9993:localhost:9993 forwarder9992@159.89.80.129 -N -vvv

class Scrap():
    # .subnet
    # .today
    # .date0
```

```
# .date1
# .process
# .stations
# .scrapData(station, date) <- carga una óestacin a tmpDataFile

def __init__(self, subnet, date0, date1, station):

    self.subnet= subnet
    self.today = dt.datetime.now()
    self.date0 = date0
    self.date1 = date1
    self.tmpDataFile = subnet+tmpDataFile
    self.loadDataFile= subnet+loadDataFile

    # Load stations [net: scrap, subnet: wu, on:True]
    with open(stationsFile, 'r') as f:
        stations = json.load(f)

    self.stations=[]
    for st in stations:
        if st['net']=='scrap' and st['subnet']==subnet and st['on']:
            self.stations.append(st)
    if station is not None:
        self.stations = list(filter(lambda x: x['station']==station,
            self.stations))

def scrapData(self, station, date):
    ## Carga una óestacin a tmpDataFile
    if os.path.exists(self.tmpDataFile): os.remove(self.tmpDataFile)

    subprocess.run([ runPath+'scrapy',
                   'crawl', self.subnet,
                   '-a',f'station={station}',
                   '-a',f'date={date}'])
    ## Load data
    try:
        with open(self.tmpDataFile,'r') as f:
            data = json.load(f)
    except:
            data = None
```

```
        return data

    def lastData(self, station):
        '''Se fija la hora del último dato registrado.
           Si no hay archivo, se fija si falta completar el
           dia anterior.
        '''
        # Hay datos de hoy ?
        fname = templateFile.format ( s = station['station'],
                                      d = self.today.strftime('%Y%m%d')
                                    )
        print(fname) #
        if os.path.exists(dataPath+fname):
            # Leo el último dato
            with open(dataPath+fname) as f:
                   dataSaved = json.load(f)
            try:
                return dt.datetime.combine( self.today.date(),
                                            dt.datetime.strptime(
                                                dataSaved[-1]['Time'],
                                                            '%I:%M %p'
                                                           ).time()
                                          )
            except:
                return None

        # No hay datos de hoy
        # Hay datos de ayer ?
        yesterday = self.today - dt.timedelta(days=1)
        fname = templateFile.format ( s = station['station'],
                                      d = yesterday.strftime('%Y%m%d')
                                    )
        if os.path.exists(dataPath+fname):
            # Leo el último dato
            with open(dataPath+fname) as f:
                   dataSaved = json.load(f)
            try:
                return dt.datetime.combine( yesterday.date(),
                                            dt.datetime.strptime(
                                                dataSaved[-1]['Time'],
                                                            '%I:%M %p'
```

```
                                                    ).time()
                                    )
            except:
                return None

        # No hay datos de nada
        return None

    def timeToDatetime(self,day,row):
        dia = day.date()
        hora = dt.datetime.strptime( row['Time'], '%I:%M %p').time()
        out = copy(row)
        out['Time'] = dt.datetime.combine(dia, hora)
        return out

    def dateTimeToISO(self,row):
        out = copy(row)
        out['Time'] = row['Time'].isoformat()
        return out

    def loadData(self):
        if os.path.exists(self.loadDataFile): os.remove(self.loadDataFile)
        allData = []
        ## Actualiza los datos de todas las estaciones de la lista
        for st in self.stations:
            fname = templateFile.format ( s=st['station'],
                                    d=self.today.strftime('%Y%m%d')
                                )
            ## lastData
            ldata = self.lastData(st)
            newdata = []
            ## if yesterday
            if ldata is not None:
                if self.today.date() != ldata.date():
                    fnameY = templateFile.format ( s=st['station'],
                                            d=ldata.strftime('%Y%m%d')
                                        )
                    # Cargo datos de ayer
                    data = self.scrapData( station = st['station'],
                                    date = ldata.strftime('%Y-%m-%d')
                                )
```

```
                # Salvo el ída completo
                with open(dataPath+fnameY, 'w') as f:
                    if data is not None:
                        json.dump(data, f)
                    else:
                        json.dump([{}], f)

                # Datos nuevos
                if data is not None:
                    data = list(map(lambda x: self.timeToDatetime(ldata,x),
                        data))
                    newdata += list(filter(lambda x:
                        x['Time'].time()>ldata.time(),data))

        # today data
        # -- cargo datos
        data = self.scrapData( station = st['station'],
                               date = self.today.strftime('%Y-%m-%d')
                             )
        # -- salvo dia completo
        with open(dataPath+fname, 'w') as f:
            if data is not None:
                json.dump(data, f)
            else:
                json.dump([{}], f)
        # -- datos nuevos
        if data is not None:
            data = list(map(lambda x: self.timeToDatetime(self.today,x), data))
            if (ldata is None) or (self.today.date() != ldata.date()):
                # sin datos hoy
                newdata += data
            else:
                newdata += list(filter(lambda x:
                    x['Time'].time()>ldata.time(),data))

        newdata = list(map(self.dateTimeToISO, newdata))
        allData.append({'station':st,'data':newdata})

    ## Save !!
    with open(self.loadDataFile, 'w') as f:
            json.dump(allData, f)
```

```
        return allData

def publish(station, data, client):
    topic = 'meteo/{typ}/{net}/{subnet}/{station}/data'.format(**station)
    for row in data:
        ## mqttit !
        client.publish(topic, payload=json.dumps(row), qos=1, retain=False)

def errorChau(msg):
    print (f'\nscrap.py ERROR\n\t{msg}\n')
    print (''' Usage:
              scrap.py [subnet=wu [date0 [date1] [station]]]

              subnet = wu | smn
              date = 2019-12-25
              station= .station
           ''')
    exit(1)

def parseArgs(args):
    # parse args
    nargs = len(args)
    if nargs>5 : errorChau('Demasiados argumentos.')
    subnet = 'wu'
    date0 = None
    date1 = None
    station = None

    if nargs>1: # explicita subnet
        subnet = args[1]
        if subnet not in ['wu','smn']: errorChau('SUBNET desconocida.')
    if nargs>2: # explicita fecha0
        date0 = args[2]
        if len(date0.split('-')) != 3: errorChau('DATE0 con formato incorrecto')
    if nargs==4: # argumento 4
        arg4 = args[3]
        if len(arg4.split('-')) == 3:
            date1 = arg4
        else:
            station = arg4.strip()
```

```
    elif nargs==5: # expicita óestacin
        date1 = args[3]
        station = args[4].strip()
        if len(date1.split('-')) != 3: errorChau('DATE1 con formato incorrecto')

    return subnet, date0, date1, station

def main(args):

    subnet, date0, date1, station = parseArgs(args)
    print('------------------------')
    print(subnet, date0, date1, station)

    s = Scrap(subnet, date0, date1, station)
    newData = s.loadData()

    if transmit:
        ### mqtt init
        client = mqtt.Client()
        #client.tls_set(certData)
        client.username_pw_set(mqtt_user, mqtt_pass)
        client.connect(broker_ip,broker_port)

    print('\n\n-----------------------------------------------------------------')
    for item in newData:
        station = item['station']
        data = item['data']
        if len(data)!=0:
            if transmit:
                publish(station, data, client)
            print (f"{station['station']} ({station['id']})")
            for row in data:
                print('\t',row['Time'])
    print('-----------------------------------------------------------------')

    client.disconnect()


if __name__ == "__main__":
    # Los argumentos son
    # [subnet=wu [date0 [date1] [station]]]
```

```
main(sys.argv)
```

Agradecimientos

A quienes colaboraron para la concreción del proyecto. Especialmente, quiero agradecer a todos aquellos que luchan día a día **defendiendo la educación pública en Argentina**.