TRABAJO FIN DE MÁSTER

Máster en Ingeniería Industrial

UNIVERSIDAD DE ALMERIA

ESCUELA SUPERIOR DE INGENIERÍA

Desarrollo de una arquitectura abierta y modular implementada en Python para la interacción de dispositivos y servicios industriales mediante estándar OPC

Curso 2020/2021

Alumno/a:

Juan Miguel Serrano Rodríguez

Director/es:

Francisco Rodríguez Díaz

Francisco García Mañas

UNIVERSIDAD DE ALMERÍA

ESCUELA SUPERIOR DE INGENIERÍA

MÁSTER EN INGENIERÍA INDUSTRIAL

TRABAJO FIN DE MÁSTER

Desarrollo de una arquitectura abierta y modular implementada en Python para la interacción de dispositivos y servicios industriales mediante estándar OPC

Alumno:	Juan Miguel Serrano Rodríguez
Director:	Francisco Rodríguez Díaz
Co-director:	Francisco García Mañas
Fecha:	Julio de 2021

Índice general

Agradecimientos

A mis directores ***Francisco Rodríguez Díaz*** *y* ***Francisco García Mañas****, Paco I y II respectivamente. A Paco II por apoyarme en todas las ideas para este TFM y a Paco I por ponernos los pies en la tierra y centrar la atención en lo prioritario.*

A ***Marina Martínez Molina****, gracias a ti no he llegado a junio sin saber ni lo que significa Modbus por ir dejándolo todo al final. Por haberme ofrecido la oportunidad y la confianza de usar un sistema real para probar el desarrollo y siempre estar disponible para pasarte cuatro horas en un meet.*

A ***Manuel Muñoz Rodríguez****, por ofrecer la API y la explicación que la acompaña. Gracias también por mostrarme tu trabajo con el servicio web que has desarrollado y darme cuenta así de que lo mío no es para tanto.*

Y finalmente a ***Corey Schafer*** *y la comunidad de código abierto. Al primero por sus vídeos con explicaciones de Python que han hecho del proceso de aprendizaje autodidacta algo que he verdaderamente disfrutado. A la comunidad porque sus interminables hilos de dudas y foros han sido la herramienta básica para el aprendizaje y resolución de dudas. Intentaré devolver el favor explicando las partes que siento no están suficientemente bien documentadas en la red.*

Acrónimos

TFM	Trabajo Fin de Máster Master Thesis
UAL	Universidad de Almería University of Almería
OPC	OLE para Control de Procesos OLE for Process Control
OLE	Enlace de objetos y empotramiento Object Linking and Embedding
PLC	Controlador Lógico Programable Programmable Logic Controller
TCP	Protocolo de Control de Transmisión Transmission Control Protocol
UDP	Protocolo de Control de Transmisión User Datagram Protocol
RTU	Unidad Terminal Remota Remote Terminal Unit
API	Interfaz de Programación de Aplicaciones Application Programming Interface
REST	Transferencia de Estado Representacional Representational State Transfer
SCADA	Supervisión, Control y Adquisición de Datos Supervisory Control and Data Acquisition
ARM	Grupo de investigación Automática, Robótica y Mecatrónica Automatics, Robotics and mechatronics research group

DBMS	Sistema de gestión de bases de datos Database Managment System
SQL	Lengua de Consulta Estructurada Structured Query Language
ACID	Atomicidad, Consistencia, Aislamiento y Durabilidad Atomicity, Consistency, Isolation, Durability
MVCC	Control de concurrencia multi-versión Multiversion Concurrency Control
HTTP	Protocolo de transferencia de hipertexto Hypertext Transfer Protocol
SSL	Capa de Conexiones Seguras Secure Sockets Layer
UA	Arquitectura Unificada Unified Architecture
AE	Alarmas y Eventos Alarms & Events
DA	Acceso a datos Data Access
HDA	Acceso a datos históricos Historical Data Access
TCP/IP	Protocolo de Control de Transmisión / Protocolo de internet Transmission Control Protocol / Internet Protocol
ASCII	American Standard Code for Information Interchange Código Estándar estadounidense para el Intercambio de Información
CRC	Comprobación de redundancia cíclica Cyclic Redundacy Check
CIM	Fabricación integrada por ordenador Computer-Integrated Manufacturing
ERP	Planificación empresarial de recursos Enterprise Resource Planning

MES	Sistemas ejecutivos para la fabricación Manufacturing Execution Systems
IoT	Internet de las cosas Internet of Things
IIoT	Internet de las cosas industrial Industrial Internet of Things
CIP	Protocolo industrial común Common Industrial Protocol
HMI	Interfaz hombre - máquina Human Machine Interface
AEMET	Agencia Estatal de Meteorología Meteorology State Agency
IFAPA	Instituto Andaluz de Investigación y Formación Agraria, Pesquera, Alimentaria y de la Producción Ecológica Andalusian Institute for Agricultural, Fishing, Food and Organic Production Research and Training
ST	Lenguaje Estructurado Structured Text

Índice de figuras

Índice de códigos

Índice de tablas

Resumen

La integración de servicios y dispositivos industriales es una necesidad básica de cualquier aplicación en un entorno industrial o de investigación desde la extensión del uso de señales para la monitorización y automatización de procesos. Existen diversos protocolos abiertos y probados para comunicación a bajo nivel con dispositivos concretos. También existen soluciones comerciales para la integración de capas superiores y la unificación de protocolos, pero estas suelen estar limitadas al uso de *hardware* del fabricante específico. Surge por tanto la idea de suplir esta necesidad de integración diseñando e implementado una arquitectura abierta y modular para la comunicación de una diversidad heterogénea de dispositivos y que permita la interoperabilidad de distintos agentes, tanto a nivel de campo como en un nivel de abstracción superior.

Con este objetivo se ha diseñado, implementado y posteriormente verificado el funcionamiento de una propuesta de arquitectura para comunicación de servicios y dispositivos. Para ello se ha aprovechado la existencia de protocolos abiertos tanto a nivel de campo (*Modbus*, p. ej.) como a mayor nivel de abstracción (*OPC*). Haciendo uso de librerías existentes para el lenguaje de programación *Python*, se han desarrollado dos programas que conforman la base de la arquitectura. Uno de ellos es el encargado de generar y gestionar el servidor OPC, así como automáticamente gestionar una base de datos que almacene los registros de todas las variables del sistema. Por otra parte, se ha programado un cliente genérico, el cual puede establecer una comunicación bidireccional entre dispositivo o servicio industrial y el servidor OPC. Además, se han desarrollado plantillas para la interacción con los diferentes elementos de la arquitectura con el *software* científico *MATLAB* así como la programación de un PLC para comunicación vía *Modbus* haciendo uso de un entorno de desarrollo basado en *CODESYS*.

Palabras clave: *Integración de sistemas, Comunicaciones Industriales, Informática Industrial, Software de utilidad.*

Abstract

The integration of industrial services and devices is a basic need for any application in an industrial or research environment since the widespread use of signals for process monitoring and automation. There are several open and proven protocols for low-level communication with specific devices. Tthere are also commercial solutions for the integration of higher layers and unification of protocols, but these are usually limited to the use of specific manufacturer's hardware, and these solutions are usually rigid and limited to the manufacturer's approach. Therefore, the idea arises to supply this need for integration by designing and implementing an open and modular architecture for the communication of a heterogeneous diversity of devices and to allow the interoperability of different agents, both at field level and at a higher level of abstraction.

With this objective in mind, the architecture has been designed, implemented and later on tested. To this end, the existence of open protocols at both field level (e.g. Modbus) and at a higher level of abstraction (e.g. OPC) has been taken advantage of, and using existing libraries for the programming language Python, two programs have been developed that make up the architecture. One of them is in charge of generating and managing the OPC server, and also responsible for automatically generating a database that stores the records of all the system variables. On the other hand, a generic client has been programmed which can establish a bidirectional communication between the device or industrial service and the OPC server. In addition, templates have been developed for the interaction with the different elements of the architecture with the *MATLAB* scientific software as well as the programming of a PLC for communication via *Modbus* using a development environment based on *CODESYS*.

Keywords: *System Integration, Industrial Communications, Industrial Informatics, Utility Software*

Capítulo 1

Introducción

El Trabajo Fin de Máster (TFM) supone el último requisito antes de la finalización de los estudios de máster oficiales en el ámbito de la Ingeniería Industrial. Este trabajo se acoge a la modalidad de Trabajo Técnico, el cual según la normativa corresponde a "un trabajo original que suponga la solución a un determinado problema práctico o un estudio de viabilidad en el campo de la Ingeniería Industrial, relacionado con las asignaturas contempladas en el plan de estudios cursado por el estudiante, y que da lugar a un producto o servicio determinado. El resultado de este trabajo será la memoria de producto o servicio final". En los siguientes apartados de este capítulo introductorio se justificará la motivación del tema seleccionado, su adecuación al contexto dentro del currículum impartido en el máster y se presentarán objetivos y resumen de la consecución o no de estos. Así como el modo en que se han alcanzado (*fases de desarrollo*) y la distribución temporal de los trabajos realizados. Finalmente se describirá la estructura de esta memoria mencionando los distintos capítulos y describiendo brevemente el objetivo y contenido de cada uno de ellos.

1.1 Motivación del Trabajo Fin de Máster

El primer paso en la historia de la humanidad hacia el aumento de la eficiencia en la producción de bienes fue el uso de herramientas en la prehistoria. El segundo gran paso fue comenzar a sustituir la fuerza de trabajo humana por máquinas que hicieran el esfuerzo, eso sí, estas máquinas dependían del manejo por parte de un humano o disponían de un nivel de automatización muy elemental. El tercer paso consistió en sustituir el trabajo intelectual humano por el de las propias máquinas, es decir, la configuración, control y

operación de la máquina realizándose de manera automática [1]. Con esta tercera ola en la evolución y la extensión de la automatización durante el siglo XX surge la necesidad de monitorizar y realizar operaciones de control automática sobre la maquinaria, lo que requiere de mecanismos de comunicación y trasferencia de señales.

En un inicio, esta necesidad quedaba satisfecha mediante conexiones punto a punto, que hacía que los sistemas rápidamente se volvieran muy complejos y difícilmente escalables. Posteriormente, apareció comunicación a nivel de campo con los llamados buses de campo (*fieldbus* en inglés). Uno de los mayores problemas de los protocolos que surgieron es que eran soluciones de nicho de fabricantes particulares, de manera que difícilmente se podían justificar los costes de desarrollo y mantenimiento [2]. Por ello, surgió la necesidad de implementar estándares de comunicación, apareciendo con el tiempo protocolos como *Modbus*, *RS485*, *I2C*, *EthernetIP*, etc. De esta manera, la comunicación a bajo nivel quedó mayormente solventada.

El siguiente paso fue la integración de los distintos subsistemas. Uno de los problemas de los buses de campo es que, en general, requieren su propia red de transmisión, incompatible con la usada típicamente en redes locales, *Ethernet* e IP. Para solventar esto han surgido protocolos o evoluciones de los mismos como *KNX*, *ModbusTCP* o *EthernetIP* que se integran en la estructura *Ethernet/IP*. Con este paso, se aspira a integrar la comunicación vía *Ethernet* a nivel de campo, siendo un paso muy importante hacia la integración real y, es por ello, que se ha visto promovido por los propios fabricantes [3]. Pero como menciona [2], si se observa con detenimiento la *Ethernet* industrial, tal y como es hoy, resulta que ni siquiera el uso de *Ethernet* estándar es realmente un denominador común, y por encima de la capa de enlace de datos, los enfoques son completamente diferentes. Algunos utilizan mecanismos estándar como Protocolo de Control de Transmisión (TCP) o Protocolo de Control de Transmisión (UDP) para transmitir datos, que pueden ser mejorados por capas de software adicionales para soportar la comunicación en tiempo real y no real, mientras que otros utilizan pilas de comunicación dedicadas que evitan todo el conjunto de IP. Algunos emplean protocolos de bus de campo conocidos para mantener cierta compatibilidad con las implementaciones ya existentes, y otros son desarrollos completamente nuevos (ver fig. 1.1).

Seleccionados algunos de los protocolos mencionados anteriormente, y sabiendo que es un tema todavía en evolución, sólo resta por integrar dispositivos con aplicaciones. Para ello, el estándar industrial más empleado es OPC UA. Una vez más, el problema radica en que mientras que el protocolo es abierto, y la conexión e interacción con cualquier servidor OPC se encuentra mayormente estandarizada independientemente del fabricante, la implementación de cada fabricante para la integración de los distintos dispositivos y servicios no lo es. Incluso dentro del propio fabricante puede existir incompatibilidad entre distintos modelos, y no sólo eso, dentro del mismo modelo usando distinto *software* del

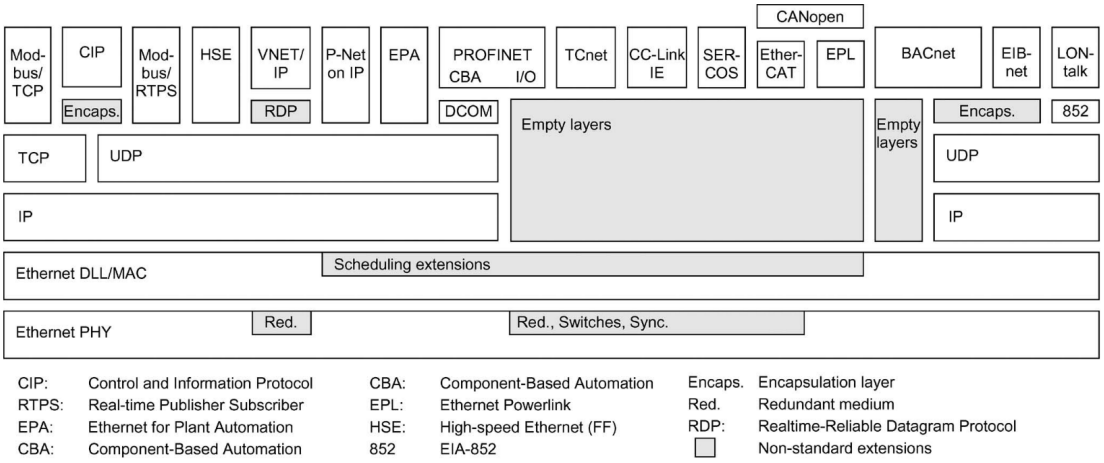


Figura 1.1. Variedad de protocolos basados en *Ethernet*. Imagen de [2].

fabricante se puede tener o dejar de tener acceso a distintas funcionalidades. Es por ello, que en este TFM la aproximación propuesta es la que tomaría un fabricante cualquiera: seleccionar los protocolos de bus de campo estándares a usar, con el requerimiento adicional de que dispongan de soporte en la comunidad de código abierto, y usar OPC UA como elemento integrador. Queda, de esta forma, únicamente por resolver la comunicación entre dispositivos y servicios con el servidor. Esta manera de proceder presenta las siguientes ventajas:

- Económicas. No se requieren de licencias del fabricante.
- Eliminación de la dependencia del soporte del fabricante.
- Expandible. Dado el carácter abierto de las librerías, éstas se pueden mejorar o ampliar sus capacidades en base a necesidad.
- Interoperabilidad. Usando distintas librerías se pueden integrar dispositivos de características y capacidad computacional muy diversa.
- Flexbilidad. Esta manera de proceder no limita el uso de soluciones comerciales Por ejemplo, hay ciertos controladores lógicos programables (PLC) que tienen soporte para generar su propio servidor OPC, esto también se puede aprovechar como herramienta de integración no debiendo limitarse a buses de campo.

1.2 Objetivos

El objetivo principal de este TFM es el desarrollo de una arquitectura integrada para dispositivos y servicios industriales, la cual permita el control y la monitorización de

los distintos elementos de un sistema real, inicialmente planteado para un invernadero experimental pero no limitado a este. Se pretende realizar la integración de sistemas diversos mediante herramientas de protocolo abierto, que otorguen interoperabilidad y sean de carácter heterogéneo. Para ello se plantean los siguientes objetivos:

- Diseño de la arquitectura a implementar.
- Generar un servidor OPC que haga de agente integrador de dispositivos, sea seguro y configurable.
- Dar soporte a *ModbusTCP* y *ModbusRTU*.
- Dar soporte a distintas Interfaz de Programación de Aplicaciones (API) de estaciones meteorológicas.
- Creación de programa cliente que simplifique la gestión e interacción con los dispostivos y servidor, dejando al usuario la única tarea de programar la aplicación.
- Definir casos de estudio a modo de demostración de concepto y comprobar su funcionamiento de manera que se pruebe el desempeño de la arquitectura implementada.

Además se plantean los siguientes objetivos auxiliares que completarían este trabajo:

- Generación y gestión de una base de datos para registro de datos históricos de todas las variables del sistema.
- Creación de plantilla para comunicación con PLC usando *Modbus*, paliación de limitaciones de *Modbus*.
- Creación de plantilla para conexión de *MATLAB* y servidor OPC.
- Creación de plantilla para conexión e interacción de *MATLAB* y base de datos.
- Dar soporte a EthernetIP.
- Dar soporte a *ModbusRTU* a través de TCP.

1.3 Contexto

Este TFM se enmarca en la temática de asignaturas relacionadas con las instalaciones industriales avanzadas y los sistemas de comunicaciones industriales que se imparten en la Universidad de Almería. Concretamente, en el marco de la asignatura *Instalaciones Industriales Avanzadas* del plan de estudios del Máster en Ingeniería Industrial, en la que se introduce al alumno en conceptos básicos relacionados con las comunicaciones industriales.

En el módulo II de esta asignatura, *Control y gestión Micro-redes energéticas*, se estudian aspectos como *Monitorización y supervisión en edificios. Domótica, Inmotica e IoT en edificación* y *Gestión de energía y ciberseguridad en micro-redes.*

Además, dentro del Grupo de investigación Automática, Robótica y Mecatrónica (ARM) y, en concreto, en proyectos de investigación que requieren de configuraciones experimentales para validación de resultados, se busca una manera genérica y estándar de unificar las medidas y variables de control de distintos sistemas de manera sencilla y fácilmente accesible para implementar lazos de control usando *software* científico como *MATLAB.*

En concreto uno de los casos de aplicación del trabajo desarrollado en este TFM ha sido implementar la arquitectura en un sistema real de uno de los proyectos actualmente en ejecución. El proyecto *AGROCONNECT* (EQC2019-006658-P: Sistema de Cultivo Intensivo Sostenible, Autónomo, Conectado y Abierto) [4] tiene como objetivo implantar una infraestructura de ensayo y demostración de investigación a escala y funcionalidad reales. Para ello, se dispondrá de un invernadero conectado en entornos de Big Data e IoT (Internet of Things), autónomo y totalmente controlado, autosuficiente y eficiente energéticamente gracias al empleo de energías renovables, y optimizado para la reutilización y uso del agua y energía, gestión y control de clima. En concreto, el proyecto se distribuye en cuatro subsistemas o lotes distintos, los cuales se enumeran a continuación:

1. Sistema de generación de energía y agua en un invernadero experimental.
2. Destilación por membranas.
3. Humidificación, enriquecimiento de CO_2 y fertirrigación.
4. Nube/*Big Data*/Simulación.

1.4 Resumen de resultados

Al término de este TFM se ha desarrollado una arquitectura flexible y modular que permite la integración de servicios y dispositivos industriales que soporten protocolos abiertos como *Modbus* y dos implementaciones de *API REST*. Para ello se han desarrollado dos programas, uno para generar el servidor OPC y otro que se debe configurar y ejecutar para cada dispositivo y servicio de manera que quede integrado en la arquitectura. Adicionalmente, se ha desarrollado una base de datos que registra automáticamente todas las variables del sistema. Un diagrama conceptual de la arquitectura desarrollada se puede apreciar en la Figura 1.2. Se han programado aplicaciones a modo de ejemplo en lenguaje estructurado (ST) y *MATLAB.* El trabajo realizado se ha validado con dos casos de estudio, uno con dos Controlador Lógico Programable (PLC)s ejecutando una simulación de

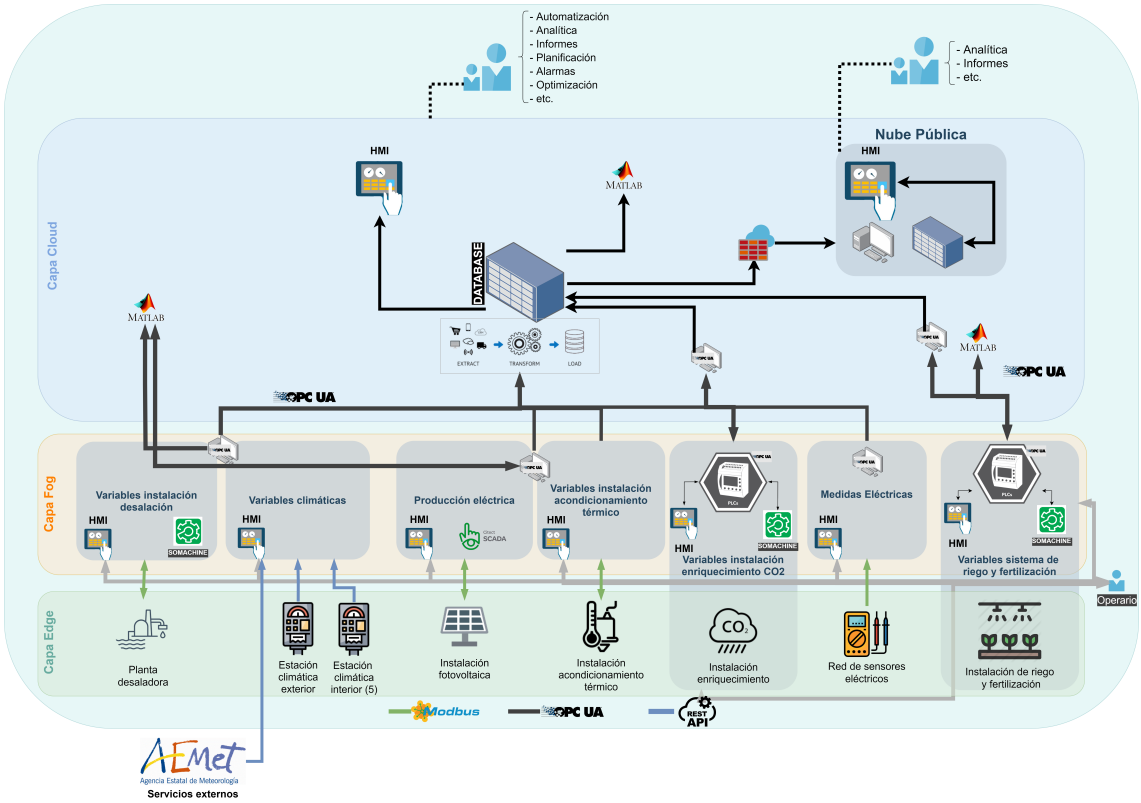


Figura 1.2. Diagrama conceptual de la arquitectura desarrollada

configuración experimental, y otro en una instalación física existente con dispositivos de distinta índole como medidores eléctricos y caudalímetros.

En el primer caso se ha logrado controlar el sistema de regado de dos invernaderos haciendo uso del *software MATLAB*, teniendo disponible en el servidor todas las variables de interés del sistema como nivel del tanque, estado de las válvulas, etc. Además, se ha generado una base de datos donde se han registrado las variables de interés del proceso durante un tiempo. Estos datos se han recuperado desde *MATLAB* y se han realizado representaciones gráficas.

En el segundo caso se ha aplicado el desarrollo en un sistema real, concretamente del proyecto *AGROCONNECT* [4]. Se han integrado los registros de medidas eléctricas así como de un caudalímetro por comunicación serie. En este caso no existía alternativa sencilla para el registro permanente de los datos y con la aplicación del trabajo desarrollado se ha obtenido una base de datos en la que se registran los datos históricos de los elementos mencionados, siendo estos de interés para evaluación del rendimiento obtenido de la aplicación de desarrollos tecnológicos.

Para la ejecución de los dos casos de estudio mencionados se ha realizado la progra-

mación de controladores lógicos haciendo uso del lenguaje Lenguaje Estructurado (ST). Habiéndose debido importar y trabajar con librerías para este lenguaje. Además, se ha configurado el adaptador *Modbus*, incluidos los registros para cada variable. Para acceder a la base de datos ha sido necesario también trabajar con el lenguaje *SQL*, desarrollando una plantilla para facilitar el trabajo a usuarios futuros.

1.5 Fases de desarollo y planificación temporal

El desarrollo de este trabajo se ha dividido en una serie de fases que una vez realizadas completan lo propuesto en el apartado de objetivos. Estos son los que se enumeran a continuación:

1. Búsqueda bibiográfica del estado del arte e implementaciones similares.
2. Creación de programa simple para generar servidor.
3. Creación de programa simple para comunicación vía *Modbus* (TCP o Unidad Terminal Remota (RTU)).
4. Creación de programa simple para comunicación via API con estación meteorológica.
5. Programación de aplicación ejemplo en PLCs.
6. Programación de *scripts* en *MATLAB* para aplicación de ejemplo y conexión tanto a servidor como base de datos.
7. Programación completa de cliente genérico con configuración mediante archivo.
8. Programación completa de servidor con generación de base de datos.
9. Verificación del funcionamiento de la arquitectura desarrollada en dispositivos reales.
10. Redacción de la memoria.

Estas fases establecidas se han organizado y distribuido en el tiempo según se muestra en la tabla 1.1. Se ordena por orden cronológico con el tiempo aproximado empleado en cada una de las fases de desarrollo a lo largo del año 2020-2021.

Según la tabla 1.1, el tiempo empleado en días suma un total de 85 días. La labor realizada ha supuesto una carga de trabajo de 3 horas de media al día, exceptuando junio y julio donde se ha promediado 8 horas por día, la duración total del proyecto en horas ha sido de 479.

1.6 Competencias del título usadas en el TFM

A la finalización de los estudios en la Universidad de Almería se deben haber alcanzado una serie de competencias que se acreditan con el título y tienen un carácter general, las aplicadas en la realización de este Trabajo de Fin de Máster se enumeran a continuación.

- CB6. Poseer y comprender conocimientos.
- CB10. Habilidad de aprendizaje.
- CB9. Capacidad de comunicar y aptitud social.
- CG11. Capacidad para aprender a trabajar de forma autónoma.
- UAL7. Conocimiento de una segunda lengua.
- UAL4. Comunicación oral y escrita en la propia lengua.
- UAL2. Habilidad en el uso de las TIC.

Asimismo, también se han aplicado las siguientes competencias específicas del Máster en Ingeniería Industrial:

- TI7. Capacidad para diseñar sistemas electrónicos y de instrumentación industrial.
- CT4. Capacidad de resolver problemas con iniciativa, toma de decisiones, creatividad, razonamiento crítico y de comunicar y transmitir conocimientos, habilidades y destrezas en el campo de la Ingeniería Industrial.
- CT10. Capacidad de trabajar en un entorno multilingüe y multidisciplinar.
- CB3. Conocimientos básicos sobre el uso y programación de los ordenadores, sistemas operativos, bases de datos y programas informáticos con aplicación en ingeniería.
- TI8. Capacidad para diseñar y proyectar sistemas de producción automatizados y control avanzado de procesos.
- TFM1. Ejercicio original a realizar individualmente y presentar y defender ante un tribunal universitario, consistente en un proyecto en el ámbito de las tecnologías específicas de la Ingeniería Industrial de naturaleza profesional en el que se sinteticen e integren las competencias adquiridas en las enseñanzas.
- G4. Conocimientos de contabilidad financiera y de costes.

1.7 Estructura de la memoria del TFM

La memoria se estructura en distintas secciones que aportan una idea detallada del trabajo realizado, de forma que un lector con noción básica de programación pueda seguir la metodología aplicada aunque no se llegue a entender detalladamente qué se implementa en cada línea de código.

En el capítulo 2 se describen la evolución de la tecnología en el ámbito de la comunicación industrial y la integración de sistemas, así como las soluciones actuales, identificando sus limitaciones. A continuación, en el capítulo 3 se presenta conceptualmente la arquitectura de integración propuesta, detallándose su jerarquía tanto a nivel de *hardware* como de *software*. Posteriormente, en el capítulo 4 se presentan los detalles de la implementación en *Python* y *SQLite* de todos los elementos constitutivos de la arquitectura mostrada en el capítulo precedente. En este se muestran las clases desarrolladas para dar soporte a los distintos protocolos y servicios, se explica el programa que hace de *driver* entre los dispositivos y el servidor, el propio servidor y finalmente la generación de la base de datos. Hasta aquí quedaría justificado el por qué se ha hecho y el cómo.

En el siguiente capítulo (5) se presentan resultados de la aplicación de la arquitectura desarrollada. En concreto se presentan dos casos de estudio y se muestra y explica el trabajo auxiliar desarrollado para su consecución, con la esperanza de que pueda servir como punto de partida para usos y aplicaciones futuras. En este capítulo también se presenta el presupuesto del trabajo técnico desarrollado, lo que precede a la sección final de presentación de conclusiones y mención de líneas de trabajo futuras.

Tabla 1.1. Planificación temporal del trabajo realizado a lo largo del curso escolar

Período de tiempo	Actividad realizada	Duración
Noviembre - Diciembre	· Estudio del estado del arte. · Redacción anteproyecto. · Programación y recordatorio de uso de *Unity Pro XL* de aplicación a modo de simulador. · Programación de aplicación en *Citect SCADA*. · Configuración de PLCs para conexión entre ellos creando servidor OPC en uno. · Interacción con *MATLAB*, gráfica de la evolución del sistema en tiempo real.	**15 días** (45 horas)
Enero	· Desarrollo de programa en *Python* para comunicación vía *Modbus TCP* de *loggers* eléctricos. · Desarrollo de servidor y cliente OPC.	**15 días** (45 horas)
Enero - Febrero	· Mejoras al programa. Registro de mensajes de información e incidencias, añadir gestión de fallos en ejecución sin cierre de programa, uso de clases.	**4 días** (12 horas)
Abril - Mayo	· Ampliación para aceptar *Modbus RTU* e integrar caudalímetro. · Estudio y mejora de programación en *Python* usando contenido de Corey Schafer. · Programación de servidor que actualiza su configuración dinámicamente en base a cambios en archivo de configuración. · Mejora de programa desarrollado.	**11 días** (33 horas)
Junio	· Descarte de aproximación anterior y aprovechando partes de ese programa para generación de Cliente Genérico. · Creación de servidor OPC. · Creación de generación automática de base de datos. · Implementación perfiles de usuario. · Implementación seguridad en los mensajes, generación de certificados con *OpenSSL*. · Definición de nuevo tipo de dato para estaciones meteorológicas. · Intento de implementar soporte para *EthernetIP*. · Soporte para API para estaciones meteorológicas. · Inicio redacción de memoria.	**25 días** (200 horas)
Junio - Julio	· Ampliación de base de datos para crear tabla unida por dispositivo. · Mejora sustancial del programa de *MATLAB* para representación de variables en tiempo real a partir de servidor OPC. · Creación de plantillas para conexión a servidor OPC y base de datos. · Creada plantilla para conexión a base de datos usando driver. · Prueba de versión actual en sistema con Marina. · Pruebas con configuración experimental de PLCs y estaciones meteorológicas. Obtención de resultados. · Creación de plantilla para programación de PLCs y comunicación vía *Modbus* con soporte para variables reales. · Redacción de memoria.	**18 días** (144 horas)

Capítulo 2

Antecedentes

En este capítulo se contextualiza el desarrollo realizado frente a las tendencias actuales en la automatización industrial, mencionando las distintas aproximaciones comerciales y comentando sus principales características y el porqué el desarrollo de una arquitectura integrada sigue siendo una cuestión abierta en el área de comunicaciones industriales e Industria 4.0.

El concepto de integración en automatización no es algo nuevo, fue formulado en los años setenta para la idea de Fabricación integrada por ordenador (CIM), aunque debido a las limitaciones tecnológicas de la época nunca se llegó a desarrollar. Su objetivo era integrar ciertas islas (aisladas) de procesos asistidos por computador en un sistema comprensivo y transparente de información a nivel de empresa. Con el tiempo se tendió hacia dos ideas diferentes, una enfocada en la gestión Planificación empresarial de recursos (ERP) y otra en los procesos Sistemas ejecutivos para la fabricación (MES) [5].

En la Figura 2.1 se puede observar un mapa cronológico con la aparición de distintas tecnologías en el ámbito de redes y comunicaciones industriales. Se puede apreciar que, a pesar de la diversidad de orígenes de los protocolos de campo, es de gran importancia la definición del modelo ISO/OSI. Tras este surge una ola de buses de campo. Conforme la mayoría de desarrolladores tomaron una aproximación pragmática, la mayoría de los buses tenían un carácter propietario, con una arquitectura de integración vertical en un Supervisión, Control y Adquisición de Datos (SCADA) directamente conectado a este bus en una sala de control [5]. Esto es lo que puede entender por una arquitectura de integración tradicional.

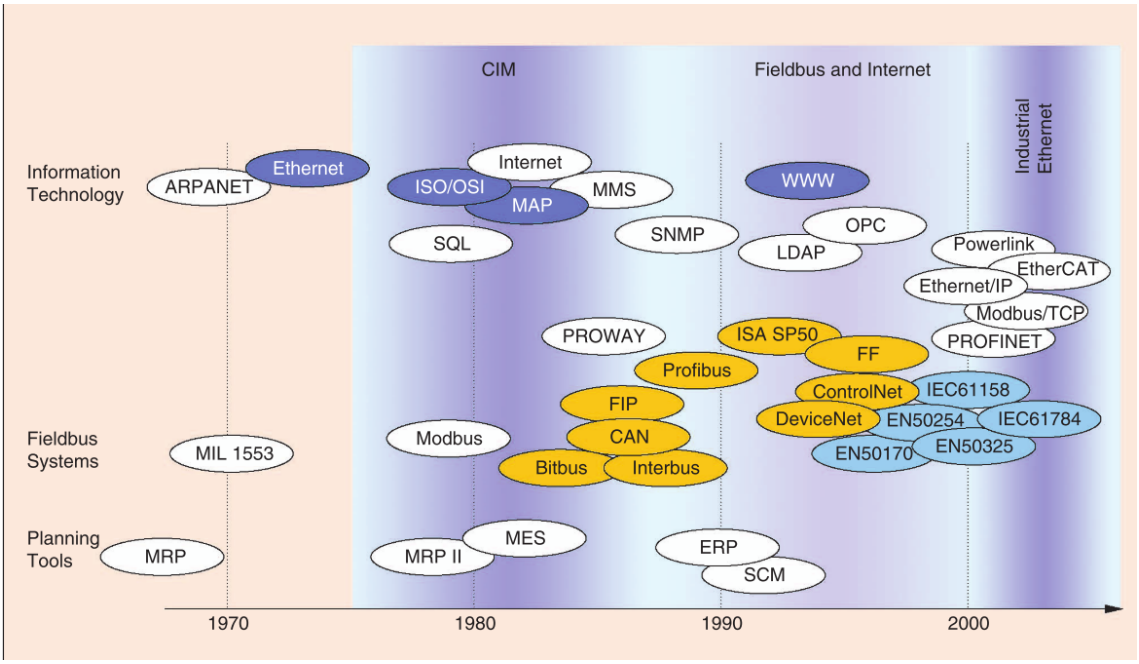


Figura 2.1. Cronología de avances tecnológicos en tecnología de redes en el contexto de la automatización. Imagen de [5]

La arquitectura de integración tradicional presenta varias desventajas claras. Se trata de sistemas cerrados que soportan una cantidad limitada de dispositivos y normalmente de un fabricante o un grupo limitado de estos. Al tratarse de un sistema cerrado, las nuevas necesidades de la arquitectura difícilmente pueden verse satisfechas por la implementación inicial, y tampoco existe la posibilidad de ampliar las capacidades de esta pues no es posible realizar modificaciones sobre el sistema cerrado. La integración finaliza en un sistema SCADA, lo que dificulta o imposibilita el uso de otras aplicaciones para la monitorización o el control del proceso en tiempo real, además de no estar asegurado el soporte de bases de datos o estar limitado a la decisión de qué base de datos soportar por parte del fabricante.

A día de hoy, las redes y protocolos usados en automatización todavía son un campo en evolución [6]. Por ejemplo, un protocolo tan longevo como *Modbus* [7], en concreto su variante TCP, tiene diferentes aproximaciones que eliminan limitaciones como la de 8 *bytes* por registro, permitiéndose así la transmisión de variables reales o más complejas sin necesidad de adaptaciones en emisor y receptor [8]. Otro ejemplo es la adición de mecanismos de seguridad en los mensajes y en el establecimiento de la comunicación. Estas mejoras no están estandarizadas o implementadas ampliamente por los fabricantes. Por otra parte, mientras que *EthernetIP* [9] es un estándar cada fabricante lo desarrolla a su manera, haciendo que aprovechar librerías genéricas que implementan este protocolo para su uso no sea baladí y requiera trabajo de adaptación para cada fabricante.

En general un proceso integrador se encuentra caracterizado por tres aspectos principales [10]:

1. Integración horizontal dentro de los niveles individuales de la jerarquía.
2. Integración vertical entre distintas capas.
3. Integración temporal durante el ciclo de vida entre elementos del sistema y sus componentes.

Más en concreto estos tipos de integración se pueden descomponer en dos aspectos tecnológicos: el cómo se comunican y para qué (aplicación). El primero se refiere a las distintas redes que forman el flujo de información, mientras que el segundo hace referencia a cómo se procesa y qué utilidad se le da a esa información.

Desde el punto de vista de una arquitectura, hay dos posibilidades para lograr la interconexión entre buses de campo y redes *Ethernet*:

- Encapsular un protocolo en otro. Este es el caso de *EthernetIP* el cual encapsula tramas Protocolo industrial común (CIP) o *ModbusTCP*, que básicamente encapsula tramas *Modbus* en el protocolo TCP.
- Mediante una puerta de enlace (*gateway*). Esta es la aproximación que se propone en este trabajo, conectando protocolos a nivel de campo (*Modbus*, p.ej.) y usando *OPC* como *gateway*. El intermediario entre estos es la herramienta desarrollada y explicada en la sección 4.3.

2.1 Tecnologías *middleware*. OPC

El *software middleware* es el pegamento entre comunicación y aplicación, así como entre las propias aplicaciones. En general, se encarga de las tareas de gestión de datos, servicios de aplicaciones, mensajería, autentificación y gestión de API [11].

En el caso de los sistemas de bus de campo, un paso importante hacia la integración horizontal fue la definición de perfiles y normas, es decir, la capa de usuario que permite la interoperabilidad entre dispositivos de distintos proveedores. En un principio, se adaptaron a los distintos sistemas de bus de campo (por ejemplo, HART DDL, PROFIBUS GSD, CANopen EDS, FF DDL) y sentaron las bases para una configuración y herramientas accesibles (basadas en XML).

La tecnología más usada en esta categoría para el intercambio de información entre bus de campo y aplicaciones de capas superiores (SCADA, *MATLAB*, etc) es OLE para Control de Procesos (OPC). Pero no sólo está limitado a integración vertical, pues también se puede usar para integración horizontal a nivel de campo o en capas superiores (como

se propone en la sección 6.2). Además su viabilidad en el futuro está asegurada con la versión Arquitectura Unificada (UA), pues integra servicios *web*.

2.2 *Edge, Fog, Cloud*

Con el desarrollo de la capacidad de computación se ha tendido en muchas industrias a dedicarse a analizar los datos de los distintos sensores y elementos del sistema para obtener patrones e información útil mediante análisis estadístico y otros algoritmos [12]. Esto es especialmente útil por ejemplo en el campo del mantenimiento predictivo y análisis de la producción en la industria manufacturera, por mencionar algunos. El problema de la transmisión en ambas direcciones en este paradigma de Internet de las cosas (IoT) de toda esta información es que produce ralentizaciones y limita la operación en tiempo real al tener que transmitirse hasta un centro de procesamiento central.

Dado que la mayoría de los sensores y dispositivos de adquisición de datos en los sistemas Internet de las cosas industrial (IIoT) operan en la periferia de la red, se tiende a producir más datos en esta capa, lo que implica que el procesamiento de los datos en el borde de la red sería más eficiente [13]. Se usarán los términos anglosajones *edge*, *fog* y *cloud* por no haberse encontrado alternativa clara en castellano al conjunto. Un diagrama de este tipo de arquitectura se muestra en la Figura 2.2.

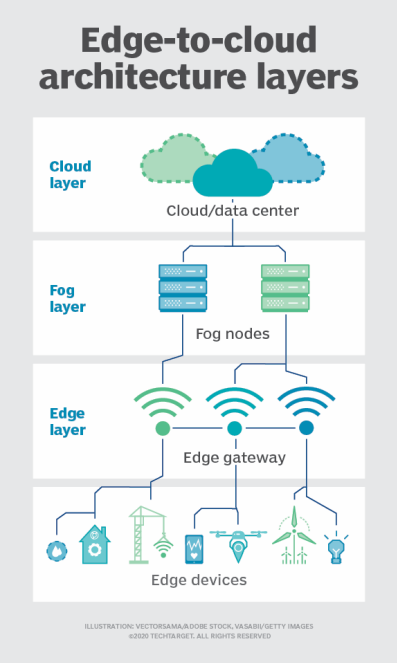


Figura 2.2. Diagrama del paradigma *Edge, Fog, Cloud*. Obtenido de [14]

2.2.1 Capa *Edge*

Hace referencia a intentar que los cálculos se realicen en el borde de la red en lugar del núcleo de la red. En este escenario, el "borde" se refiere a cualquier recurso situado en cualquier ruta de la red entre dispositivos de adquisición de datos (situados cerca de la periferia de la red) y el centro de datos en la nube (situado en el núcleo de la red) [13]. La base del paradigma de la computación de borde es que los cálculos deben realizarse en el "borde", pues está en la proximidad de las fuentes de datos y esto evita la latencia asociada a la transferencia de datos al núcleo de la red.

2.2.2 Capa *Fog*

Es similar a la capa *Edge*, su función es limitar la necesidad de computación en la capa superior. Se orienta más hacia una red local y se sitúa cerca de los dispositivos de campo de la capa *Edge*. Esta capa asiste cuando los dispositivos o servicios de la capa inferior no tienen capacidad de procesamiento. De esta manera la diferencia fundamental entre la capa *Edge* y *Fog* es la localización de la capacidad de procesamiento e "inteligencia", si a nivel de campo (*Edge*) o fuera de este (*Fog*) pero no centralizado (*Cloud*) [13].

2.2.3 Capa *Cloud*

Con esta filosofía la capa *Cloud* pasa de ser el nodo central con toda la capacidad de procesamiento para pasar a cumplir idealmente únicamente una función de repositorio de almacenamiento de toda la información. Esta es accesible tanto por la capa *Edge* como la *Fog* [15].

En la arquitectura propuesta en este TFM no se intenta rehuir completamente de la capa *Cloud* y se tiende más a presentar una arquitectura flexible que permita ambas aproximaciones: tener una centralización en la capa *Cloud* si así se requiere para facilitar el acceso a aplicaciones y usuarios, así como permitir la distribución de los elementos de la arquitectura y su acceso, limitando intermediarios y aliviando el nodo central.

Capítulo 3

Arquitectura propuesta para integración de sistemas

En este capítulo se expone conceptualmente la arquitectura propuesta. Para ello se muestra un diagrama de la arquitectura aplicada para un invernadero experimental compuesto por varios subsistemas de distinta índole. La idea es que tras esta sección queden claros los roles de cada uno de los elementos que forman la arquitectura y que se desarrollan en el siguiente capítulo donde se explica en detalle su implementación. Para este caso particular se mostrará a nivel de *hardware* cómo se distribuyen los dispositivos así como las posibilidades de configuración y *software* usado.

3.1 Concepto de arquitectura para integración de sistemas y dispositivos industriales

Se puede observar en la Figura 3.1 la arquitectura base. Se trata de una estructura clasificada en capas interconectadas, formando la unión de todas ellas la arquitectura integrada objetivo. Se distinguen tres capas en orden de abstracción ascendente: la capa *edge*, *fog* y *cloud* [12]. En la capa *edge* se encuentran todos los elementos que realizan tareas de bajo nivel en el sistema, es decir, estaciones de medida proveyendo información acerca de las condiciones climáticas, datos del suministro de energía, etc. Estos elementos transmitirán información de los distintos sistemas mediante protocolos industriales como *ModbusTCP* o *ModbusRTU*. Esto también ocurre con elementos como las estaciones comerciales, pero

haciendo uso de su propia API, al igual que con servicios externos como la predicción climática de AEMET.

Esta información se transmite a la capa *fog*, la cual se encarga de procesar dicha información en bruto, sin tratar (*raw*), de los distintos sensores y variables del sistema y realizar su gestión. Se pueden aquí encontrar los sistemas SCADA locales con interfaz Interfaz hombre - máquina (HMI) para manipulación in situ por parte del operario. Cada uno de los elementos de esta capa forma su propio entorno de variables y clases del elemento al cual dan servicio, es ya una decisión de diseño decidir agrupar varios dispositivos en un mismo grupo, únicamente agrupar las variables de un dispositivo, agrupar por funcionalidad, etc. Estos grupos pueden formar campos dentro del servidor OPC-UA, donde se gestionará su acceso y modificación. En esta capa intermedia se reciben o envían tramas entendibles por la capa inferior (p. ej., *Modbus*), y se comunican con la capa superior mediante el protocolo OPC.

Toda la información del sistema, es clasificada y gestionada (ETL, Extract, Transform and Load) desde una nube privada que forma parte de la capa *Cloud*. A partir del servidor OPC se dispone de una unificación de los datos en tiempo real y a corto plazo. Gracias a este se puede, de manera sencilla, ampliar la disponibilidad de la información a datos históricos mediante el uso de una base de datos permanente. Esta información (en tiempo real y datos históricos) está disponible a los usuarios internos del sistema que se podrán conectar mediante distintas aplicaciones cliente: *MATLAB*, *Citect SCADA*, clientes genéricos OPC como *UAExpert*, etc, así como clientes que puedan acceder a la base de datos (*MATLAB*, *SQLectron*, *Excel*, etc). Con estas aplicaciones se podrán realizar tareas de analítica, optimización, automatización, planificación, redacción de informes, etc. En esta misma capa, y con las convenientes medidas de seguridad para limitar el nivel de acceso, se ofrecerá el mismo servicio en una nube pública para el acceso de usuarios invitados.

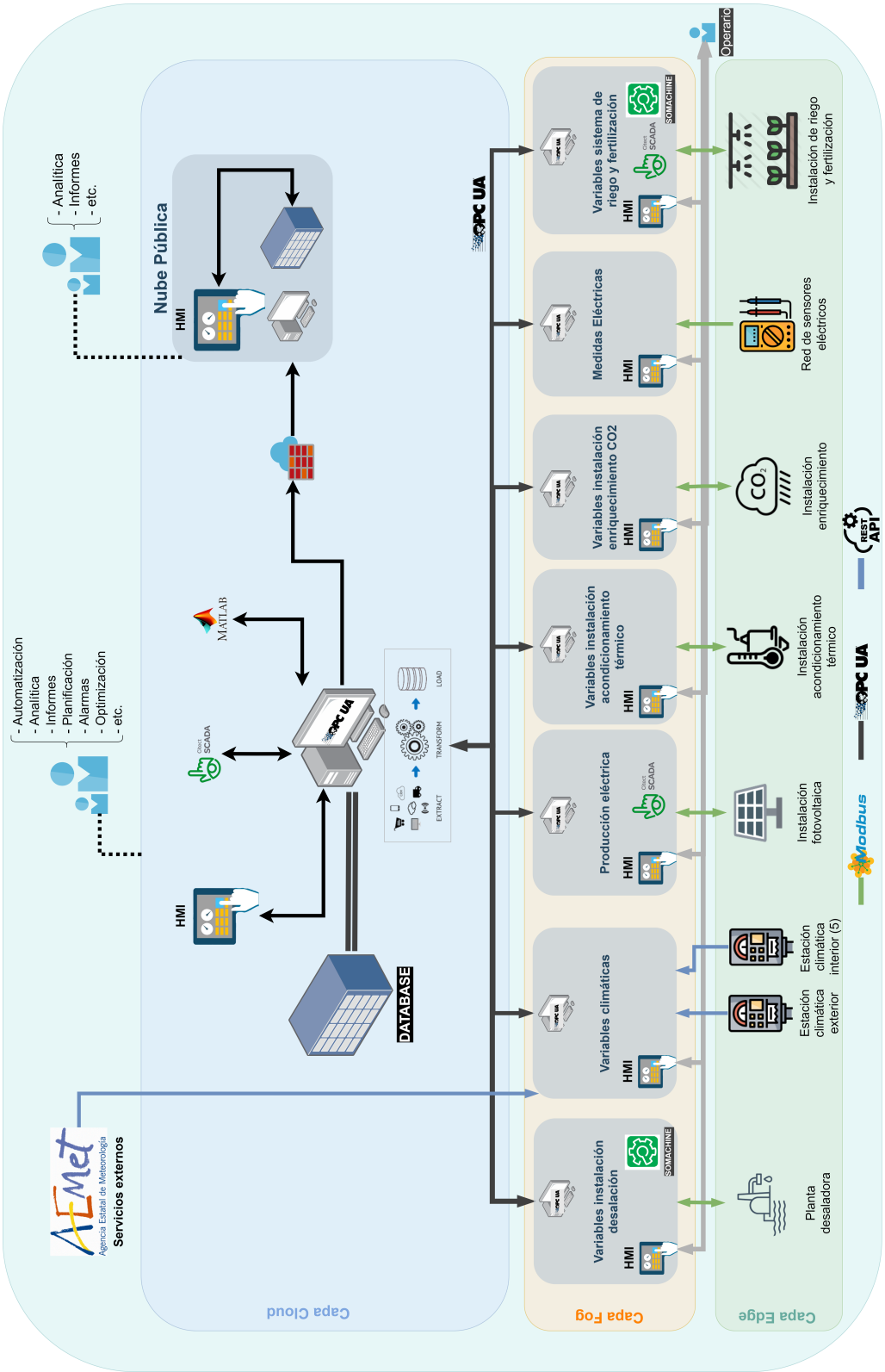


Figura 3.1. Diagrama de la arquitectura propuesta aplicada a un invernadero experimental.

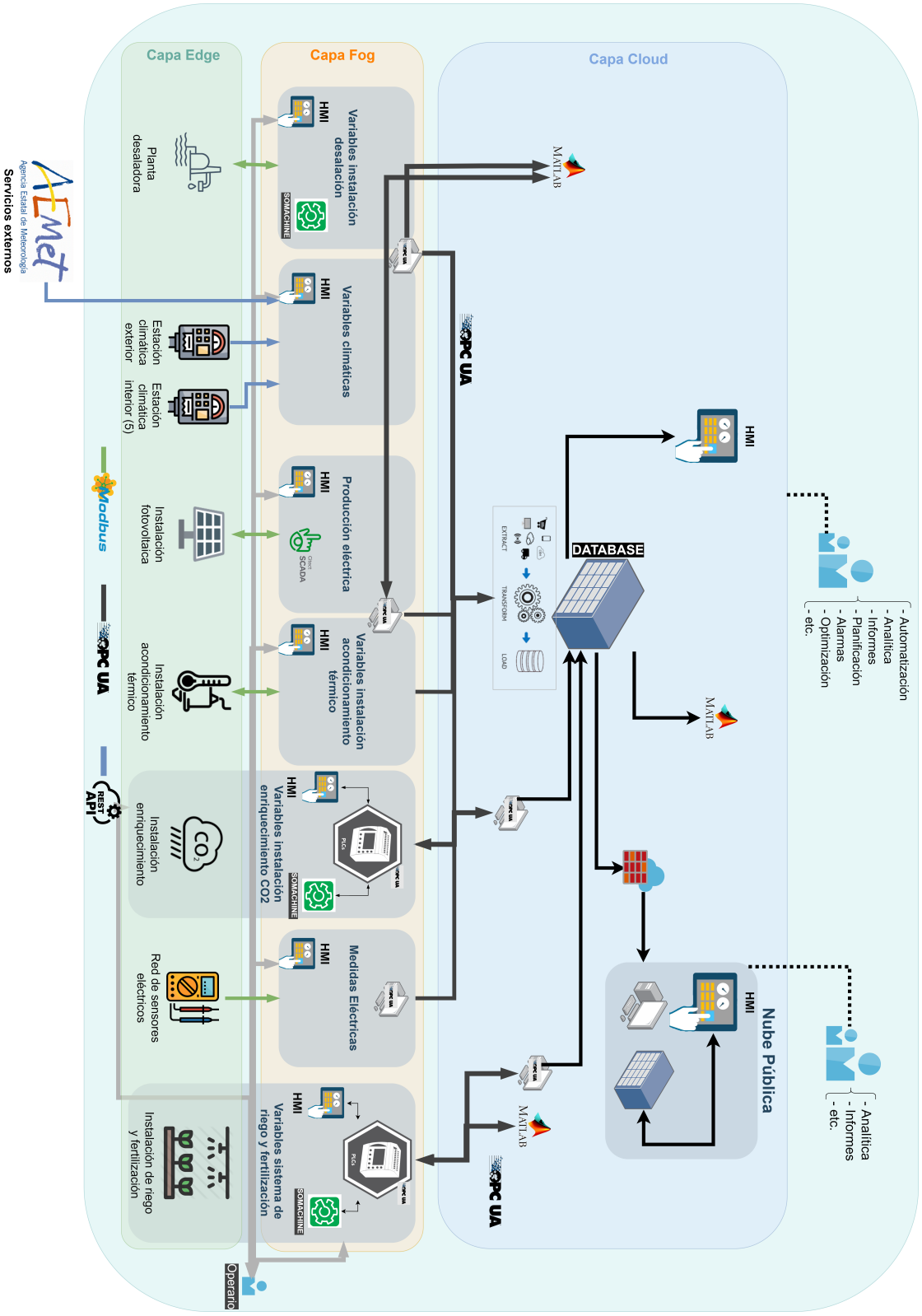


Figura 3.2. Ejemplo de concepto de arquitectura distribuida

Esta es la arquitectura base con un enfoque tradicional. Una de las ventajas de las herramientas desarrolladas es que se puede transferir capacidad de procesamiento de una capa a otra de manera flexible. Por ejemplo, un mismo equipo puede estar comunicándose con el dispositivo de bajo nivel (capa *edge*), o varios de ellos, y a la vez ser el que genera un servidor OPC (pasaría a situarse en la capa *fog*). Se tendría así una arquitectura más distribuida donde un equipo central no sería el que soportase toda la gestión de información del sistema (capa *cloud*). Los distintos equipos unificadores igualmente accederían a la base de datos común (capa *cloud*) para registrar toda la información del sistema. Así mismo, las aplicaciones podrían, o bien acceder al equipo integrador de la capa *cloud*, o bien comunicarse directamente con las *islas* de interés de la capa *edge*/*fog*, reduciendo así la carga computacional y obteniéndose en conjunto una arquitectura más distribuida. Se podría dar también en el caso en el que el propio dispositivo de la capa *edge* tenga capacidad de computación y *software* para generar su propia *isla*, haciendo así innecesario la capa intermedia. El concepto aquí explicado se puede observar en la Figura 3.2.

3.2 Arquitectura *hardware*

En la capa base (*edge*) se hallan las instalaciones o dispositivos físicos que realizan las tareas específicas del sistema. En el esquema de la Figura 3.1 se trataría, por ejemplo, de una estación comercial con su batería de sensores para medir condiciones climáticas ambientales, con su sistema de tratamiento que adapta las medidas analógicas hasta hacerlas disponibles mediante la API implementada por el fabricante. También forman parte de esta capa los PLCs encargados de gestionar el sistema de riego y fertilización, los *powertag* (medidores de magnitudes eléctricas) encargados de registrar las variables eléctricas de los distintos subsistemas para tener una monitorización del consumo de estos, la instalación de paneles fotovoltaicos con su sistema de monitorización, un módulo de ósmosis inversa, una caldera y campo de concentración solar para generación de energía térmica, etc. A nivel de comunicación debe existir, o bien una instalación cableada bajo algún estándar como CAT5[1] y conexiones RJ45 para formar una red LAN, o bien una red inalámbrica con distintos repetidores o enrutadores distribuidos por el terreno de la instalación. Los dispositivos que hagan uso de conexiones serie requerirán de cables que puedan transmitir señales usando protocolos como, por ejemplo, RS232 [16].

Para la capa inmediatamente superior es fundamental la existencia de una red local donde distintos equipos ejecuten la adaptación de las distintas señales para hacerlas compatibles con el estándar OPC. Con la solución propuesta en este TFM, se pueden integrar varios dispositivos en un mismo equipo. En esta capa se encontrarían las HMI que los operarios usarían para gestionar y monitorizar el desempeño de los distintos subsistemas a

[1]CAT 5: `https://en.wikipedia.org/wiki/Category_5_cable`

nivel de campo. Para sistemas más complejos también pueden existir equipos ejecutando sistemas SCADA locales para el control y monitorización de los mismos.

En la capa *cloud* a nivel de *hadware* se presenta una situación similar, se requieren equipos para ejecutar el servidor, así como una o varias computadoras dedicadas a generar y mantener la base de datos. Cada usuario requerirá de su propio equipo donde se ejecuten las aplicaciones mencionadas anteriormente (*MATLAB*, *Excel*, etc).

3.3 Arquitectura *software*

La capa inferior (*edge*) se relaciona con la programación de bajo nivel, fundamentalmente encargada de realizar la transformación de señales analógicas de los sensores, o de transformar instrucciones en señales de salida para los actuadores. Los dispositivos más integrados, como las estaciones comerciales, sí cuentan con un sistema capaz de realizar solicitudes y enviar información al servicio web encargado de agrupar las medidas de la estación.

En la capa intermedia es donde se encuentra el programa encargado de recibir la información de los protocolos de bajo nivel y transmitir y/o recibir del servidor. En la arquitectura propuesta en este TFM sería el programa `cliente genérico`, explicado en el capítulo 4.3. Cada dispositivo a integrar requiere ejecutar una instancia diferente de este programa. El archivo de configuración sí puede contener información sobre parámetros de varios dispositivos, pero el programa sólo accede a un único dispositivo por ejecución.

En la capa superior se sitúa el programa encargado de generar el servidor OPC (ap. 4.4) así como la base de datos (apdo. 4.5). Ambas funcionalidades se hallan integradas en un mismo programa, por una decisión de diseño, pero son dos módulos distintos que pueden funcionar independientes el uno del otro (eso sí, sin el servidor, la base de datos no puede tener nuevas entradas).

3.4 Clientes OPC y posibilidades

Puesto que el protocolo OPC, y en concreto la versión UA (Arquitectura Unificada, *Unified Architecture*), es un estándar usado extensivamente y de manera generalizada en la industria así como en el ámbito de investigación, la mayoría de aplicaciones desarrolladas en este ámbito soportan la conexión e interacción con servidores de este tipo. Esto incluye *software* científico como *MATLAB*, el cual incluye el *OPC Toolbox* [17]. Este *toolbox* ofrece soporte para la conexión con servidores OPC UA. En concreto, soporta la autentificación mediante usuario y contraseña así como mediante certificado. En cuanto a la seguridad de los mensajes, ofrece la posibilidad de comunicación cifrada mediante por ejemplo `Basic256Sha256`, usada por el servidor.

Distintos *software* SCADA comerciales también tienen soporte para comunicación mediante este protocolo como *LabVIEW* [18]. *Citect SCADA* de *Schneider Electric* ofrece la posibilidad de generar un servidor así como conectarse como cliente a otro ya existente. Un ejemplo de esto es el servidor para *Citect SCADA* de *Matrikon* [19]. Según *Matrikon*, ofrece ciertas ventajas respecto a la implementación de *Schneider Electric* y demuestra que el servidor con el que este *software* puede interactuar es independiente de que sea propio del fabricante o no.

Finalmente, existen otros clientes genéricos que sirven para consultar el estado del servidor en tiempo real así como para modificar variables (siempre que estas sean de escritura y se acceda como administrador). Un ejemplo es el *software* comercial pero gratuito como el proveído por *Matrikon UA Expert* [20], o programas de código abierto como, por ejemplo, el desarrollado para la librería usada en este TFM *FreeOpcUa*, [21].

3.5 Clientes para conectarse a base de datos y posibilidades

Tal y como está programada la base de datos, no se requiere nada especial en cada cliente para acceder a ella, tan sólo tener soporte para el módulo *SQLite*. Si en un futuro se amplía la base de datos para hacer uso, por ejemplo, de *Postgres*, sí hace falta un *driver* para conectarse a este tipo de base de datos. La base de datos usada genera únicamente un archivo en formato `.db`, al cual se puede acceder con clientes de bases de datos genéricos como *SQLectron* [22] o *HeidiSQL* [23]. *MATLAB* provee soporte para bases de datos tanto relacionales como no relacionales con *Database Toolbox* [24]. Este *toolbox* incluye además una aplicación con interfaz gráfica para interactuar con la base sin tener que escribir código y exportar o importar datos al espacio de trabajo de *MATLAB*. A su vez, este *software* presenta soporte directo para *SQLite* [25]. Otra aplicación importante con posibilidad de acceso a este tipo de bases de datos es *Excel*, que permite importar datos de distintas maneras [26].

Finalmente, cabe mencionar que aunque la aplicación a usar por un usuario no presente soporte directo a un motor de base de datos como los mencionados siempre se puede hacer el paso intermedio de usar un cliente como *SQLectron*, hacer la solicitud (*query*) de los datos deseados y directamente exportarlos en formato `json` o `csv`, ambos formatos universales que la mayoría de aplicaciones son capaces de procesar.

Capítulo 4

Implementación de la arquitectura usando *Python*

Este capítulo del Trabajo de Fin de Máster es donde se explica la implementación de los dos programas desarrollados en *Python* que permiten la aplicación de la arquitectura propuesta en el capítulo 3. Primero se mostrarán las clases desarrolladas para ofrecer soporte a los distintos protocolos industriales así como a servicios web. A continuación, se explicará en detalle el funcionamiento del programa encargado de comunicar dispositivo con un servidor para después describir los distintos aspectos de la implementación del servidor OPC. Finalmente, se mostrará cómo se ha añadido la generación y actualización de la base de datos integrándola en el programa del servidor.

4.1 Especificaciones de diseño

Los distintos programas que se explican en este capítulo se han desarrollado en base a las siguientes especificaciones:

- Soporte para protocolos industriales acorde con la norma IEC 61784. Esto incluye *ModbusTCP* y *ModbusRTU*.
- Soporte para API de estaciones meteorológicas comerciales.
- Lectura de parámetros desde archivo de configuración, con lo que no se requiere modificar el programa para cada cliente.
- Que permita la interacción de aplicaciones como *MATLAB*, *LabView*, *Citect SCADA* con las distintas variables de una instalación.

- Soporte para variables complejas (ver apartado 4.4.2).
- Renovación automática de `token` cuando este caduca para API.
- Cuando la lectura a API no cambia, se descarta la lectura, se informa de ello y se espera un tiempo establecido para realizar otro intento de lectura.
- Cuando se reconecta un cliente al servidor OPC, si existen dos clientes con el mismo nombre, se da la opción de borrarlo y crear uno nuevo, o mantenerlo e: intentar recuperar las variables existentes en servidor y listado de variables de configuración, borrar las que ya no se usen y añadir las variables nuevas añadidas al listado.
- En caso de pérdidas de conexión, reintento automático de conexión.
- Que se base en herramientas de código abierto.
- Que el producto se halle correctamente documentado de manera que con esta memoria y los comentarios en el código alguien con experiencia en programación pueda entender el funcionamiento del programa.
- Que tenga un carácter modular que facilite la modificación y ampliación de las características y funcionalidades de la arquitectura así como el soporte a otros protocolos y modos de comunicación.

4.2 Clases propuestas para dar soporte a diversos servicios y protocolos industriales

Con el objetivo de construir un *software* que tenga una estructura modular, se ha decidido crear clases para los distintos modos de comunicación con dispositivos y servicios. Cada clase tiene distintos métodos que realizan distintas funciones, de manera que el programa que gestiona la conexión con el cliente se pueda abstraer de las instrucciones concretas que requieren la comunicación con el dispositivo. En general, todas las clases desarrolladas requieren de métodos para iniciar la conexión con el dispositivo así como realizar las operaciones de lectura y escritura. Para cada uno de los modos de comunicación se han usado distintas librerías de código abierto disponibles en repositorios públicos (*GitHub*, p. ej.).

4.2.1 *ModbusTCP*

Modbus es un protocolo desarrollado por *Modicon* en los años 70 inicialmente pensado para usarse con comunicación serie (RTU o American Standard Code for Information Interchange (ASCII)) pero diseñado de manera que fácilmente se puede adaptar a cualquier

medio (*Ethernet*). Su gran ventaja es que es un estándar abierto que tiene un uso muy extendido en la industria [7]. En el anexo B.2 se comenta más en detalle las características de este y en este apartado la atención se dirige más hacia cómo se ha realizado la implementación.

Para *ModbusTCP*, la librería usada ha sido *Modbus-tk* [27]. Hay varias librerías que ofrecen soporte para la comunicación con este protocolo, entre ellas *PyModbus* [28], *MinimalModbus* [29] y *uModbus* [30], entre otras. Probablemente se pueda usar cualquiera de estas librerías y ofrecerían un rendimiento satisfactorio. De hecho, durante gran parte del desarrollo del programa se trabajó con *PyModbus*. Finalmente, se decidió usar *Modbus-tk* por ser una librería ligera y tener un rendimiento excelente. Esto se compara en un hilo de *StackOverflow* [31], donde*Modbus-tk* es la que mejor rendimiento ofrece de todas las alternativas. Además, con esta librería puede usarse para comunicación con dispositivos a través de *ModbusRTU*, así como *ModbusRTU* a través de TCP (*ModbusRTU over TCP*), lo que permite que ambas clases sean muy similares, facilitando el desarrollo y mantenimiento de las mismas.

La clase tiene varias propiedades que son necesarias para poder establecer la conexión con el dispositivo, en concreto requiere de una dirección IP, un puerto y un identificador, todas ellas características del protocolo. Con esta información almacenada en la clase se puede llamar al método `crearCliente` sin ningún argumento de entrada. Este método lo único que hace es llamar al método adecuado de la librería, en concreto es de la siguiente manera: `self.client = tkTcp.TcpMaster(host=self.dirIP, port=self.puerto, timeout_in_sec=2.0)`. Hecho esto la clase tiene dos formas de leer valores del dispositivo:

- Leer una única variable. Esta función toma como entrada el registro inicial que apunta al dato de interés, el número de registros a acceder y el tipo de dato de la variable a leer. Probablemente sería únicamente necesario conocer el tipo de dato pues, en base a esto, en general se tiene una longitud determinada. Por ejemplo, un registro tiene un tamaño de 16 *bits* o 2 *bytes*, mientras que una variable tipo *Float* requiere de 32 *bits* o 4 *bytes*. Por lo tanto, hace falta acceder a dos registros para leer el valor de una variable de este tipo. La función completa se muestra a continuación:

Código 4.1. Método leer variable de la clase `clienteModbusTCP`

```
1 def leerVariable(self, registro, longitud, tipo_dato):
2         if tipo_dato in self.dic_simbolos_lectura:
3                 s = self.dic_simbolos_lectura[tipo_dato]
4                 valor = self.client.execute(self.ID,
                  ↪  tkCst.READ_HOLDING_REGISTERS, registro, longitud,
                  ↪  data_format=f'{s}')
```

```
5         else:
6                 valor = self.client.execute(self.ID,
                  ↪  tkCst.READ_HOLDING_REGISTERS, registro, longitud)
7         return valor
8
```

Hay un argumento opcional que es `data_format`. Este especifica a la librería el tipo de dato, de manera que automáticamente une los registros y los decodifica en una variable del tipo de dato especificado, ahorrando el tener que hacer esto con el vector de datos devueltos para cada registro accedido. La variable `tipo_dato` es una de las opciones que se deben introducir en el archivo de configuración. Como propiedad de clase se define una variable tipo diccionario. Este contiene entradas para todos los tipos de dato que se pueden introducir en el archivo de configuración, asignando el símbolo adecuado que se debe introducir en `data_format`. Los diccionarios para lectura y escritura se muestran a continuación:

Código 4.2. Diccionarios con símbolos en base al tipo de variable para lectura y escritura

```
1                   dic_simbolos_lectura = {
2                         'Float': '>f',
3                         'Int64': '>q',
4                         'Double': '>d',
5                         'Int32': '',
6                         'Int16': '',
7                         'Int': '',
8                         'Boolean': '',
9                         'Bool': ''
10                  }
11
12                  dic_simbolos_escritura = {
13                          'Float': '>f',
14                          'Int64': '>q',
15                          'Double': '>d',
16                          'Int32': '>i'
17                  }
18
```

- Leer un grupo de variables. La implementación es muy similar a la que se realiza para una variable, la diferencia es que se trabaja con listas de valores en lugar de con valores individuales. También se genera un vector con las marcas de tiempo de lectura para cada variable:

Código 4.3. Extracto de método para leer grupos de variables

```
for registro, longitud, tipo_dato in zip(lista_registros,
↪  lista_longitudes, lista_tipo_dato):
        if tipo_dato in self.dic_simbolos_lectura:
                s = self.dic_simbolos_lectura[tipo_dato]
                valores.append(self.client.execute(self.ID,
                ↪  tkCst.READ_HOLDING_REGISTERS, registro, longitud,
                ↪  data_format=f'{s}'))
        else:
                valores.append(self.client.execute(self.ID,
                ↪  tkCst.READ_HOLDING_REGISTERS, registro, longitud))
        sourceTimestamp.append(datetime.now()) # Registrar tiempo en el
        ↪  que se ha producido la lectura

```

El método restante es el que se llama cuando se detecta un cambio de valor en el servidor. Éste es el encargado de mandar ese valor en el registro (o registros) adecuado. La función tiene una estructura similar, simplemente se cambia el código *Modbus* y se añade el argumento `output_value`, al que se le asigna el valor a escribir:

Código 4.4. Extracto de método para escribir valor en dispositivo

```
        self.client.execute(self.ID, tkCst.WRITE_MULTIPLE_REGISTERS,
        ↪  starting_address=registro, output_value=[valor],
        ↪  data_format=f'{s}')
```

4.2.2 *ModbusRTU*

Como se ha comentado en el apartado anterior, una de las ventajas de usar la librería *Modbus-tk* es que provee soporte para *ModbusRTU*. Por lo tanto, esta clase es prácticamente idéntica. En este caso la conexión en lugar de requerir una dirección IP y un puerto lógico, requiere del *baudrate* (tasa con la que se transmite información por el puerto serie) y el puerto serie donde está conectado el dispositivo al ordenador:

Código 4.5. Inicio de método para clase `clienteModbusRTU`

```
        def crearCliente(self):
        self.client = tkRtu.RtuMaster(serial.Serial(port=self.puerto,
        ↪  baudrate=self.baudrate))
```

4.2.3 APIs

Mientras que los protocolos anteriores definen un estándar de comunicación, el concepto API hace referencia a una interfaz que permite a dos programas distintos comunicarse [32]. No es un estándar y por lo tanto cada implementación particular de una API puede (y suele) ser distinta de otra. Cada API define una serie de reglas para acceder al servicio y a los datos que puede devolver. No obstante, se puede definir un procedimiento básico común para las API tipo Transferencia de Estado Representacional (REST), que se usan para comunicarse con servicios web mediante solicitudes HTTP. El procedimiento es el siguiente:

1. Iniciar sesión en el servicio para conseguir un *token*.
2. Solicitar información al servicio acompañando a la solicitud del *token*.
3. Procesar información recibida.

En concreto, para las APIs a las que se les va a dar soporte se va a usar la instrucción `GET` para realizar solicitudes al servicio. La referencia seguida para la integración con Python ha sido [33]. El módulo en *Python* que da soporte para la comunicación con tramas HTTP en *Python* se llama `Requests` (ver anexo B.3).

Se muestra la definición e inicialización de la clase en el código 4.6. Cuando se crea una instancia de la clase se debe suministrar el nombre de usuario y contraseña que permiten la autentificación en el servicio, así como la clase que contiene un nuevo tipo de dato definido para almacenar la información de interés devuelta por la API, que se explica con más detalle en 4.4.2.

Código 4.6. Definición de la clase API

```
1         class API():
2         def __init__(self, usuario, password, claseEstacionComercial, api =
          ↪  'manuel'):
3         self.usuario = usuario
4         self.password = password
5         self.api = api
6         self.logger = logging.getLogger(__name__)
7         self.valores = []
8         self.msgTipo_EstacionComercial = claseEstacionComercial     # Una
          ↪  instancia que modifico
9         self.claseEstacionComercial = claseEstacionComercial        # Otra
          ↪  que no modifico
```

Finalmente, se puede incluir como argumento opcional la API en concreto que se quiere usar. Es un argumento opcional porque por defecto se hace uso de la API desarrollada

por Manuel Muñoz en el marco del proyecto *IoF2020* [34] con el nombre clave `manuel`. Se ha decidido hacer así pues esta API da soporte y hace de intermediario para otro tipo de estaciones y servicios web, de manera que existe la posibilidad de integrar nuevos dispositivos sin tener que extender el soporte desde esta clase. La otra API soportada es la de la estación comercial de la empresa *Hispatec*, con el nombre clave `nazaries`.

Como se ha comentado anteriormente, el procedimiento es similar para ambas APIs pero la implementación es distinta. Ambas requieren de acompañar un token con cada solicitud para validar dicha solicitud. Para ello, en ambas es necesario iniciar sesión y la API devuelve un token, el cual se almacena como dato de la clase:

Código 4.7. Extracto de método para iniciar sesión en API

```
    # Solicitar token mediante usuario y contraseña
        if direccion == 'https://iof2020.ual.es:3790/api/login':
        direccion =
        ↪  'https://decidecrop.nazaries.cloud/public_api/v1/users/sign_in'
        try:
        response = requests.post(direccion,
        headers = {
                'accept': 'application/json',
                'Content-Type': 'application/x-www-form-urlencoded',
        },
        data = {
                'user[login]': self.usuario,
                'user[password]': self.password
        })
        if response.status_code == 200:
        self.token = response.json()['auth_token']
        self.logger.info(f'Autentificación correcta (API Nazaries), token
        ↪  conseguido')
        else:
        self.logger.warning(f'Problema en la autentificación (API
        ↪  Nazaries), comprobar datos: {response.status_code}')
```

A continuación se debe solicitar y procesar la información del servicio. Para ello se han construido dos métodos de clase: uno que sólo se ejecuta una vez y toma información de las variables contenidas en la estación, así como de sus identificadores; y otro método que se llama cíclicamente para leer los valores de los sensores. Esta separación se hace necesaria, pues el servidor OPC queda configurado al inicio de la ejecución del programa con la lectura inicial que se hace de la API. Si en cada iteración se solicitase tanto la información de los sensores contenidos como sus valores, y estos se modificasen (eliminación o adición de nuevas medidas), entonces a la hora de intentar publicar la nueva información en el

servidor se produciría un error al no existir nodo para las nuevas variables. Se muestra el extracto de código en este caso para la API de *IoF2020*.

Código 4.8. Extracto de método para leer datos de variables en API

```
        def leerDatos(self, station_id):
        self.station_id = station_id

                elif self.api == 'manuel':
                response = requests.get(f'https://iof2020.ual.es:3790/
                ↪  api/getSensorIdstation/{station_id}',
                headers = {
                        'accept': 'application/json',
                        'Authorization': self.token
                })

                # Obtener id de sensor + id estacion
                self.ids = []
                self.id_abs = ''
                for station in response.json()['Station_types']:
                self.ids.append(station['id_sensor_station_absolute'])
                self.id_abs = self.id_abs +
                ↪  (station['id_sensor_station_absolute']) + ','

                # Obtener nombre de cada sensor
                response = requests.get(f'https://iof2020.ual.es:3790/api/
                ↪  getDataSensorGreenhouseLastDataV4/{self.id_abs}',
                headers = {
                        'accept': 'application/json',
                        'Authorization': self.token
                })
                self.nombres = []
                sensores = response.json()['DatagreenhouseRecuperado']
                for sensor in sensores:
                self.nombres.append(sensor['medidas'][0]['sensor_type'][0]
                ↪  ['name_comun'])
```

El método requiere como entrada el ID de la estación a leer, que se almacena en la clase y así a la hora de leer los valores no es necesario introducirlo como entrada. Primero se lee el identificador del sensor, que posteriormente se usará para saber qué valor corresponde a qué variable. El nombre de estas variables se obtiene con la segunda solicitud dando como entrada la lista de los identificadores de cada sensor. Se puede apreciar como a cada solicitud se le añade el `token` obtenido anteriormente.

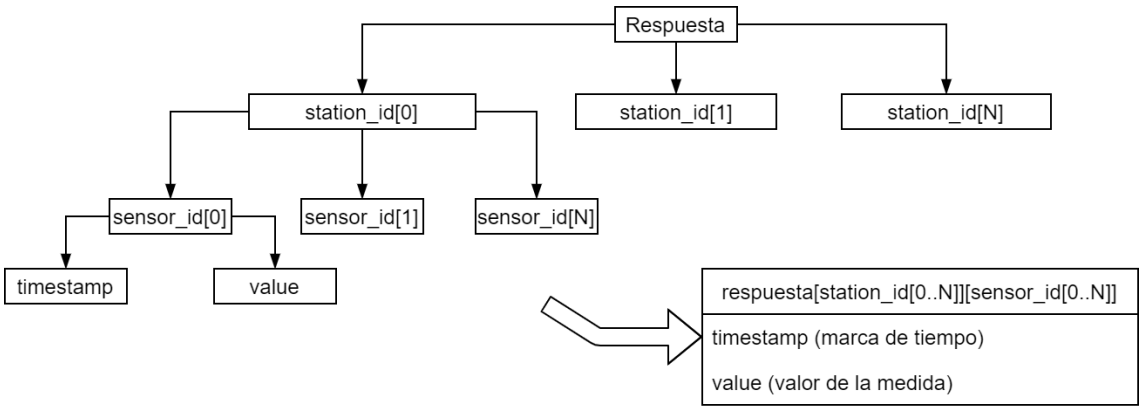


Figura 4.1. Esquema de la estructura del diccionario obtenido como respuesta a la solicitud a la API `nazaries`

Para la API `nazaries` sólo es necesario realizar una solicitud que devuelve los nombres de las distintas variables así como sus identificadores. Una vez más, la información obtenida se almacena en la clase, aunque también se da como salida por si el método que lo llama requiere los valores.

4.2.3.1 Lectura de valores

Con los identificadores de cada API se pueden leer los valores de los distintos sensores haciendo una solicitud similar a la anterior con las direcciones que se muestran a continuación ((1) para `nazaries` y (2) para `manuel`), donde `self.station_id` y `self.id_abs` son los identificadores:

Código 4.9. Extracto de método para lectura de valores de las distintas variables

```
1        https://decidecrop.nazaries.cloud/public_api/v1/stations/
         ↪  {self.station_id}/data/latest
2        https://iof2020.ual.es:3790/api/getDataSensorGreenhouse
         ↪  LastDataV4/{self.id_abs}
```

Cambia la estructura de la respuesta de cada una. Para `nazaries` la respuesta es de la forma `[str(self.station_id)][str(Id)]['value']`, es decir, Identificador estación → Identificador medida → Valor de interés. Para `manuel` la estructura es: `['medidas'][0]['sensor_type'][0]['name_comun']` para el nombre de la medida y `['medidas'][0]['attrValue']` para los valores de la medida, lo que viene a significar Vector Medidas para sensor particular (un único valor en general) → Valor. En la Figura 4.1 se muestra esquemáticamente la estructura para `nazaries` y en la Figura 4.2 para `IoT2020`.

Por otro lado, ambas marcas de tiempo se presentan como una cadena de caracteres, pero mientras que el valor de la marca de tiempo para `nazaries` es un *timestamp* que se puede convertir directamente, para `manuel` se presenta conun formato más visual pero

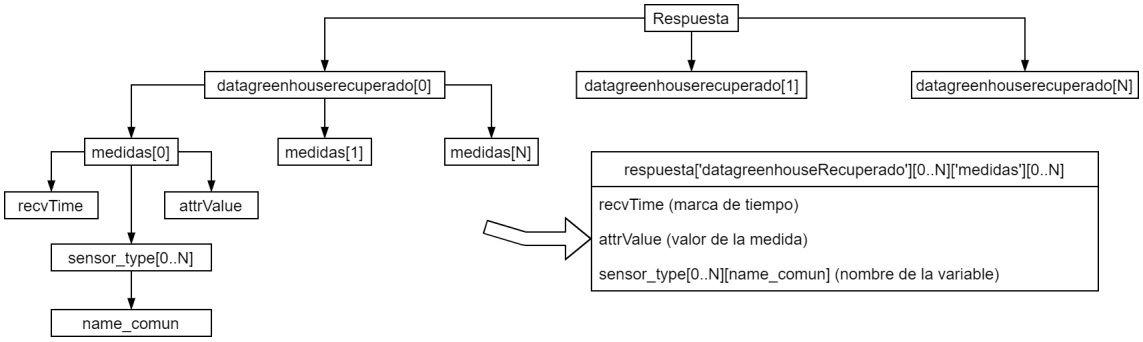


Figura 4.2. Esquema de la estructura del diccionario obtenido como respuesta a la solicitud a la API `manuel`

que requiere de aplicar una tranformación en base a la estructura de la fecha y el tiempo. Esta adaptación se muestra a continuación:

Código 4.10. Procesamiento de marcas de tiempo para cada una de las API soportadas

```
1         # importar timestamp nazaries
2         datetime.fromtimestamp(response.json()[str(self.station_id)]
          ↪  [str(Id)]['timestamp'])
3
4         # importar timestamp manuel
5         float(response.json()[str(self.station_id)][str(Id)]['value'])
6         t = datetime.strptime(t, '%Y-%m-%dT%H:%M:%S.%fZ')
```

Estas marcas de tiempo en formato *timestamp* son las que entiende el servidor OPC, así como la base de datos.

Finalmente, en este método, antes de la lectura se guardan en una variable los valores de la medida anterior, para al final compararla con la última medida obtenida, si la lectura no ha cambiado se genera un error que informará y deberá ser gestionado por el cliente de que la medida no es válida. Si no es así, devuelve el valor obtenido. Aunque nuevamente se almacenan en la propia clase pues a diferencia de las implementaciones anteriores, será la propia clase quien gestione la publicación en el servidor OPC.

4.2.3.2 Publicar valores en el servidor OPC

Puesto que se trata de un nuevo tipo de dato se ha decidido, en lugar de crear un nuevo método en el programa del cliente, realizarlo directamente desde la clase. En la creación de una instancia (un nuevo objeto de la clase definida) se lee una clase que es el nuevo tipo de dato para almacenar las medidas de la estación meteorológica. En este método se crea una instancia de esta clase y se asignan los valores en los campos adecuados. Para que en el servidor OPC aparezcan correctamente las marcas de tiempo, en lugar de establecer sólo el valor del nodo, se establece el *datavalue*, que es un tipo de dato que permite establecer

manualmente los datos de tiempo de servidor y fuente. Este método se puede apreciar a continuación:

Código 4.11. Método propio de la clase `API` para publicar valores en servidor OPC

```
def publicarOPC(self, variables_OPC):
        for Id, timeStamp, nombre, valor, variable_OPC in zip(self.ids,
        ↪  self.tiempos, self.nombres, self.valores, variables_OPC):
        if nombre == variable_OPC.get_display_name().Text:

        self.logger.info(f'Valor leído -> {nombre}|| ID: {Id}, Time:
        ↪  {timeStamp}, Valor: {valor}')
        # self.logger.info(f'Tipo  leído -> {type(nombre)}|| ID:
        ↪  {type(Id)}, Time: {type(timeStamp)}, Valor: {type(valor)}')

        self.msgTipo_EstacionComercial.ID = str(Id)
        self.msgTipo_EstacionComercial.timeStamp = timeStamp
        self.msgTipo_EstacionComercial.nombre = nombre
        self.msgTipo_EstacionComercial.valor = valor

        # Escribir usando datavalue para así poder establecer
        ↪  ServerTimeStamp y SourceTimeStamp
        dv = ua.DataValue(
        variant=self.msgTipo_EstacionComercial,
        sourceTimestamp=timeStamp,
        serverTimestamp=datetime.now()
        )

        variable_OPC.set_value(dv)
```

En resumen, esta clase da soporte a dos APIs tipo REST que se comunican con un servicio web para la lectura de variables climáticas en estaciones meteorológicas comerciales. La clase gestiona la autentificación con el servicio, almacena y es capaz de renovar el `token` necesario para la realización de las distintas solicitudes, y dispone de métodos para gestionar la lectura de los datos, así como de su publicación en el servidor OPC.

4.3 Cliente genérico

Se ha desarrollado un programa en *Python* con el objetivo de que, mediante un archivo de configuración con los parámetros específicos necesarios para cada cliente, se pueda gestionar de manera automática la conexión al servicio o dispositivo industrial, sus entradas/salidas e interacción con el servidor OPC. En el apartado 4.1 se han definido unas especificaciones generales para la arquitectura. Mientras que a continuación se mencionan

algunas específicas que atañen al programa para los clientes:

- Comprobación de la configuración para intentar evitar errores comunes en la entrada de datos de la configuración del programa.
- Registro de mensajes con información e incidencias mostrado en terminal así como en archivo.
- Gestión de lectura y escritura de cada variable en paralelo, con la posibilidad de establecer un tiempo de muestreo específico para cada variable, o común para todas las del cliente.
- La lectura en paralelo requiere de la apertura y gestión de muchos hilos o *threads*, lo cual no es óptimo cuando se tienen dispositivos de los que únicamente se requiere acceder a sus valores para lectura con un tiempo común de muestreo. Para este caso particular es preferible que la lectura se haga de manera secuencial en un único *thread*.
- El cliente debe ser capaz de conectarse a un servidor OPC con distintos niveles de seguridad, es decir, tanto para un servidor sin seguridad, como para el caso en que se requiera autentificación y/o los mensajes estén cifrados.
- El cliente debe ser capaz de importar las definiciones de nuevos tipos de datos y a su vez, cuando lo requiera, generar sus propias definiciones y hacerlas públicas a nivel de servidor.

El procedimiento de ejecución del programa es el siguiente:

1. Se ejecuta introduciendo en una terminal: `python .\clienteGenerico.py --archivo_configuracion .\configuracion.yaml --usuario nombre_usuario --contraseña --usuario_API contraseña nombre_cliente`

   Los parámetros `archivo_configuracion`, `usuario`, `contraseña`, `usuario_API` son parámetros opcionales mientras que `nombre_cliente` es un argumento posicional necesario. El parámetro `archivo_configuracion` sirve para indicar el directorio y nombre del archivo donde el programa debe leer la configuración del dispositivo. Si no se introduce, se usa la definida por defecto. Los parámetros `usuario` y `contraseña`, se usan para la autentificación en el servidor OPC, y si no se introduce la opción `usuario`, por defecto se realiza la conexión usando el perfil de administrador. La opción `contraseña` sólo ha de incluirse sin argumentos para que cuando se inicie el programa pregunte por ella, ya que si no se incluye en la ejecución se entiende que el servidor no tiene autentificación. La opción `usuario_API`

se usa cuando el cliente a conectarse es una API y no se desea escribir las credenciales en el archivo de configuración. Al ejecutarse, el programa pedirá como entrada una contraseña para la identificación en la API. Finalmente, puesto que el archivo de configuración puede contener la configuración de múltiples clientes, se debe especificar a qué cliente en concreto se desea acceder a sus parámetros, sustituyendo `nombre_cliente` por el nombre de la sección en el archivo de configuración.

Se puede obtener ayuda sobre la ejecución del programa usando: `python .\clienteGenerico.py --help`

2. Tras un logo y mensaje de bienvenida se accede al archivo de configuración y se realiza una comprobación inicial de su contenido.

3. Se establece la conexión con el servidor OPC.

4. En base al protocolo especificado en la configuración se crea e inicializa una instancia de la clase respectiva que da soporte a ese protocolo. En caso de API además se solicita el listado de variables al servicio web.

5. Se configura el espacio para el cliente dentro del servidor a partir de las variables contenidas en la configuración o del listado obtenido de la solicitud al servicio web.

6. Para cada variable se ejecuta en paralelo la lectura cíclica del valor en el dispositivo o escritura de valor en el dispositivo al cambiar éste en el servidor OPC.

7. En caso de pérdida de conexión con dispositivo o servidor, se espera un tiempo y se reintenta la operación automáticamente.

En la Figura 4.3 se puede apreciar un diagrama de flujo de la lógica completa del programa descrito en esta sección. Se distingue una primera parte de configuración y conexión inicial donde, si ocurren fallos, se finaliza la ejecución del programa informando al usuario de manera que se puedan solucionar. A continuación, se desarrolla un proceso repetitivo cíclico para la lectura de variables y por cambio de estado para la escritura. Esta segunda parte se ha implementado de manera que los fallos se gestionen de forma automática, ya que se supone que se deben a situaciones excepcionales y puntuales que se paliarán con el tiempo, sin necesidad de intervención por parte del usuario.

4.3.1 Archivo de configuración. Estructura

Como se ha comentado en las características del programa, se ha decidido que los parámetros de configuración para la conexión con el dispositivo o servicio, así como otros parámetros, sean dados mediante un archivo de configuración en formato `yaml` [35]. Esto

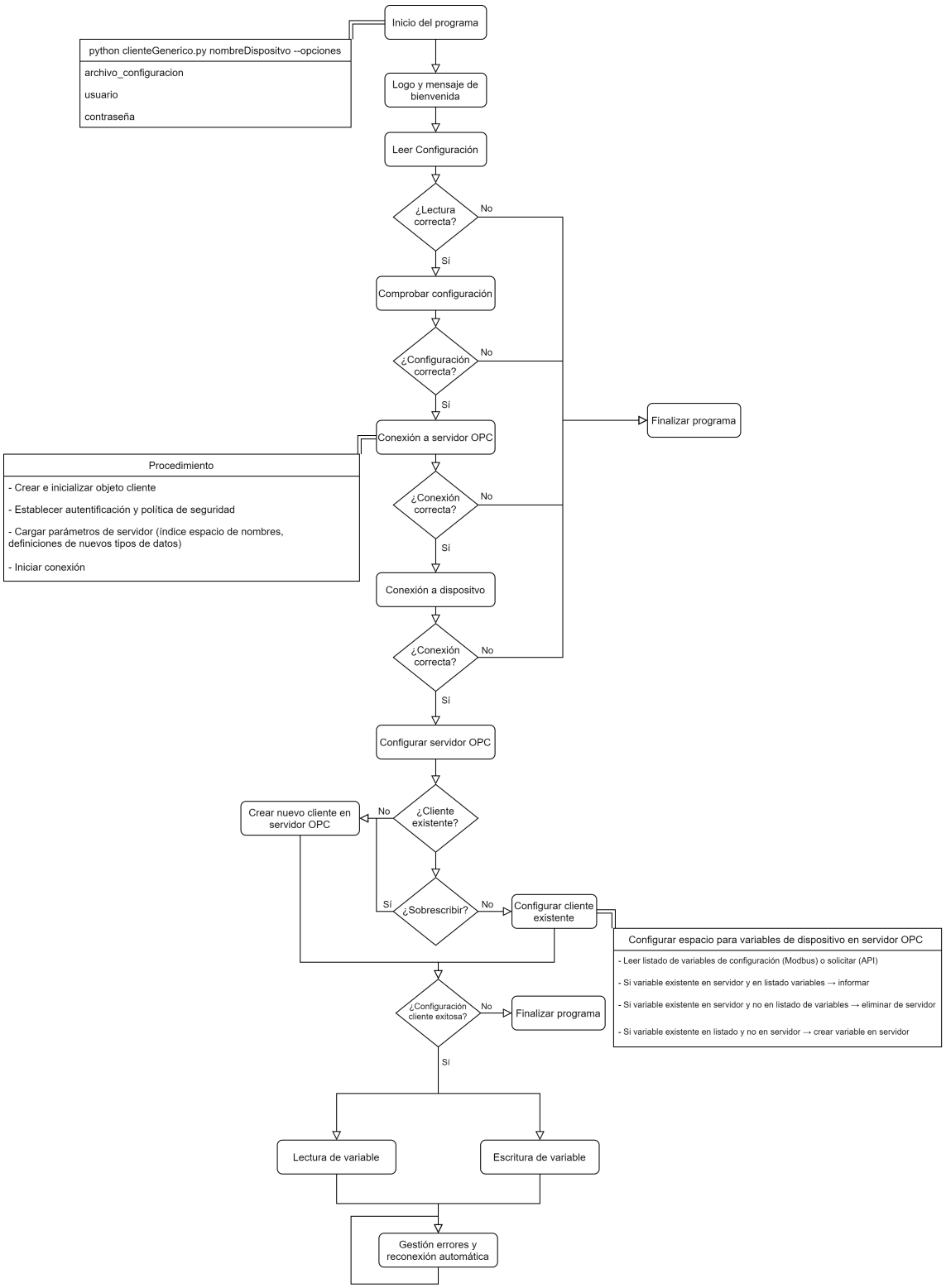


Figura 4.3. Diagrama de flujo de la lógica del programa para la integración de dispositivos en servidor OPC

ahorra al usuario tener que modificar el código del programa, con los problemas que esto puede acarrear, además de requerir que el usuario sepa dónde ésta debería ser introducida. Este lenguaje es bastante flexible, ya que de manera transparente entiende el tipo de variable que se le introduce, sin necesidad especificarlo, y tiene un formato similar al de *Python*. Las distintas secciones de la configuración se delimitan por indentación. A continuación, se muestra una configuración a modo de ejemplo que contiene parámetros para los tres protocolos.

Código 4.12. Ejemplo de configuración para cada uno de los protocolos soportados

```
# Ejemplo de configuración para múltiples dispostivos en un mismo archivo
↪  de configuración
clienteModbusTCP:
  protocolo: ModbusTCP
  ID: 1
  registros:        [0,          1,              20    ]
  longitud:         [1,          1,              1     ]
  variables:        ['Var1',     'Var2',         'Var3' ]
  tipo_datos:       ['Int32',    'Boolean',      'Float']
  tipo_acceso:      ['read',     'read',         'write']
  tiempo_muestreo:  [2,          2,              1     ]
  IP:               127.0.0.1
  puerto:           502

clienteModbusRTU:
  protocolo: ModbusRTU
  ID: 1
  registros:        [0,          1,              20    ]
  longitud:         [1,          1,              1     ]
  variables:        ['Var1',     'Var2',         'Var3' ]
  tipo_datos:       ['Int32',    'Boolean',      'Float']
  tipo_acceso:      ['read',     'read',         'write']
  tiempo_muestreo:  [2,          2,              1     ]
  baudrate:         9600
  puerto:           COM5

clienteAPI_nazaries:
  protocolo: API
  tipo_api: nazaries
  usuario: XXXX@ual.es
  password: XXXXXXXX
  tiempo_muestreo: 20 # 20seg para pruebas, 900seg o 15min
  station_id: 191
```

Se puede apreciar cómo, por ejemplo, la dirección IP se puede introducir directamente sin necesidad de comillas para tranformarlo en una cadena de caracteres. Algo similar ocurre con direcciones web o de correo. Las variables que requieren listas se delimitan con corchetes, estando los corchetes separadados por comas. Junto con los programas se adjunta un archivo de configuración a modo de plantilla donde en el encabezado se encuentra una documentación detallada de las opciones disponibles y su formato.

4.3.2 Lectura y comprobación de la configuración

Se han creado dos métodos para la lectura y la comprobación de la configuración. El primero toma como entrada el nombre del archivo de configuración y construye la ruta hasta el mismo (asumiéndose que se encuentra en el directorio del programa o en un subdirectorio del mismo). Se intenta leer; si la lectura es exitosa, se devuelve la configuración en una variable tipo diccionario que permite acceder a los distintos campos con: `IP = conf['clienteModbusTCP']['IP']` para acceder a la dirección IP del primer cliente, por ejemplo. Si falla la lectura o no se encuentra el cliente especificado en la ejecución dentro de la configuración se termina la ejecución del programa informando del problema al usuario tanto en la consola como en el *log*. Por defecto, se busca un archivo de configuración llamado `config_cliente.yaml` a menos que otro se especifique en la ejecución del programa.

Inicialmente, cada método que requería acceder a algún parámetro de la configuración aplicaba su propia comprobación y generaba excepciones en consecuencia. Esta aproximación no resultaba eficiente, pues distribuía la posible aparición de errores en muchas partes del programa, complicaba la gestión de errores, además de aumentar la complejidad de los métodos y reducir la claridad del código. Es por ello que se decidió centralizar todas las comprobaciones en un método llamado `comprobarConfiguracion(conf)`, el cual toma como entrada la configuración leída con el método anterior y aplica las comprobaciones mostradas a continuación.

1. Comprobaciones comunes a todos los protocolos:
   (a) Comprobar que existe el campo `protocolo`.
   (b) Comprobar que el campo `protocolo` forma parte de alguno de los soportados, informar al usuario de los protocolos soportados.
   (c) Comprobar que existe el campo `tiempo_muestreo`.
2. Comprobaciones específicas de *Modbus*:
   (a) Comprobar la existencia de los campos:

- `ID`. *Unit ID*, en *ModbusTCP* no es necesaria y suele ser 1 por defecto, pues el dispositivo viene identificado a través de su dirección IP. En *ModbusRTU* sirve para identificar un dispositivo cuando varios comparten el puerto serie.
- `registros`. Número de los registros a leer para acceder a cada variable.
- `longitud`. Cantidad de registros a leer desde el registro inicial anterior. Cierta redundancia con tipo de variable.
- `variables`. Identificador para las variables.
- `tipo_datos`. Tipo de los datos, por ej.: *Float, Int32, Boolean*, etc. El tipo de dato podría probablemente definir la cantidad a registros a leer. Se comprueba que pertenezca a uno de los tipos definidos en una lista: `tipos_soportados = ['Float', 'Double', 'Int64', 'Int32', 'Int16', 'Int', 'Boolean', 'Bool']`.
- `tipo_acceso`. Acceso a la variable, puede ser de lectura o escritura.

(b) Comprobar que `registros`, `longitud`, `variables`, `tipo_datos` y `tipo_acceso` tengan el mismo número de elementos.

(c) Comprobar que `tipo_acceso` sea bien `read` o `write`.

3. Comprobaciones específicas de TCP: Existencia de `IP` y `puerto`.

4. Comprobaciones específicas de RTU: Existencia de `baudrate` y `puerto`.

5. Comprobaciones específicas de API:

   (a) Existencia de:

   - `tipo_api`. Identificador de las APIs soportadas.
   - `station_id`. Número asociado a la estación comercial.
   - `usuario` y `contraseña`. Para la autentificación, deben existir aunque `contraseña` se puede dejar vacío y entonces se pedirá al inicio de la ejecución del programa.

   (b) Comprobar que `tipo_api` forma parte del listado de APIs soportadas.

4.3.3 Conexión al servidor OPC

El `cliente genérico` se conecta por defecto como administrador al servidor, pues necesita permisos de escritura a la hora de publicar los valores de las distintas variables, aunque como se ha comentado en el apartado 4.3 se puede introducir otro perfil de acceso con la opción `--usuario`. La conexión se realiza creando primero un objeto cliente, el cual se inicializa introduciendo la dirección del servidor `opc.tcp://perfil_acceso@(ip o host):puerto/ruta/hasta/servidor`. A continuación se establece la configuración de

seguridad; que en el caso del servidor implementado tiene autentificación y encriptación, por lo que se deben indicar el certificado y la clave privada. Estos deben ubicarse en una carpeta en el mismo directorio del programa llamada `certificados`, y dentro de esta, en una llamada `clientes`. Se ha decidido hacer así para que en caso de que en un mismo equipo se contengan clientes y servidor, los certificados queden claramente organizados e identificados. En cuanto a los perfiles de acceso existen dos: un `administrador` que tiene permisos para todo tipo de acciones en el servidor y uno de `usuario` que sólo puede consultar la información del servidor sin realizar modificaciones.

4.3.4 Creación del cliente para comunicarse con el dispositivo

En base al protocolo especificado en la configuración se crea una instancia de alguna de las clases definidas en el apartado 4.2. En el caso de *Modbus*, tras inicializar una instancia, se asignan las propiedades necesarias de la clase a partir de la configuración:

Código 4.13. Ejemplo de asignación de propiedades en clase cliente

```
1         cliente = libreriaClientes.clienteModbusTCP()
2         cliente.dirIP = conf_cliente['IP']
3         cliente.puerto = conf_cliente['puerto']
4         cliente.ID = conf_cliente['ID']
5         cliente.registros = conf_cliente['registros']
6         cliente.longitud = conf_cliente['longitud']
7         cliente.variables = conf_cliente['variables']
8         cliente.tipo_acceso = conf_cliente['tipo_acceso']
9         cliente.tipo_datos = conf_cliente['tipo_datos']
```

Tras esto se inicia la conexión con el dispositivo. Para la API es un ligeramente diferente. En primer lugar se intenta leer la contraseña de la configuración; si el campo se halla vacío entonces se pide al usuario que la introduzca. Posteriormente, se crea una instancia de la clase `API`, introduciéndose directamente las entradas necesarias que se almacenarán como propiedad internamente en la clase. Una de estas entradas es la clase del nuevo tipo de dato definido para dar soporte a las estaciones comerciales. Esta clase viene definida en el servidor, por lo que al conectarse al servidor se le solicitan todas las definiciones de tipo de datos de las que disponga. Posteriormente en la configuración de la API, haciendo uso de la función `get_ua_class()` y dando como entrada el nombre del tipo definido, se obtiene la clase del tipo de dato solicitado.

Otra opción habría podido ser definir el nuevo tipo de dato desde el propio cliente. La ventaja de esta aproximación es que simplifica el servidor y es el propio cliente que hace uso de la variable el que la crea y la gestiona. Esta es la única opción en caso de que se use un servidor distinto al propuesto (y que soporte definición de nuevos tipos de datos), pues

no se tendría acceso para modificar el código del mismo. En este caso se tiene el control del programa del servidor, por lo tanto, se ha decidido integrarlo aquí puesto que este también contiene la creación y actualización de la base de datos, la cual requiere acceder al tipo de datos. De cualquier manera, el proceso es el mismo independientemente de si se hace en cliente o en servidor, por lo que es una decisión de diseño que puede modificarse fácilmente en un futuro.

4.3.5 Gestión de fallos de conexión, intentos automáticos de re-conexión

Uno de los retos cuando se desarrolla una arquitectura como la propuesta desde cero no es sólo que cumpla su función, sino que lo haga de manera robusta. Cuando se pretende tener dispositivos comunicándose entre sí continuamente, hay que preveer que habrá momentos en los que alguno de los dispositivos pierda la conexión, lo haga el servidor, o lo haga la propia vía de comunicación.

No parece razonable depender de una persona que se encargue de volver a establecer la conexión y configuración de todos los dispositivos afectados cada vez que algún incidente ocurra. Es por ello que parece interesante la implementación de alguna estrategia de reconexión automática. En la Figura 4.4 se muestra un diagrama de flujo de la lógica de reconexión. El proceso se describe a continuación.

Una vez se ha establecido satisfactoriamente la conexión y configuración inicial con el servidor, se genera para cada variable un *thread* que gestiona la lectura y escritura en paralelo, esto se detalla en el apartado 4.3.6. El proceso de lectura en condiciones normales se repite de manera cíclica con un tiempo de muestreo especificado en la configuración. Esto ocurre así indefinidamente a menos que, o bien falle la comunicación con el dispositivo, o bien el servidor. En este caso se comprueba entonces si se ha superado el máximo número de fallos consecutivos (puesto que cada lectura-publicación satisfactoria resetean el contador). Si no se ha superado este límite entonces se espera un tiempo arbitrario (10 segundos) y se intenta de nuevo realizar la lectura. En el caso de que se supere el máximo número de intentos, la estrategia es distinta dependiendo de si el causante es el dispositivo a o es el servidor. Si es el dispositivo, se cierran los *threads*, se espera un tiempo arbitrario mayor (1 minuto en el diagrama) y se vuelven a abrir, repitiéndose de nuevo el proceso. Si el problema viene provocado por la conexión con el servidor entonces se produce la desconexión del mismo, se espera un tiempo y se repite el procedimiento de conexión y configuración inicial del servidor a excepción de que en esta nueva situación automáticamente se mantiene el nodo existente del cliente en el servidor y sus variables asociadas.

Con esta lógica se espera que, mientras que se haya producido la conexión inicial y no cambien sustancialmente servidor o dispositivo, cualquier error que pueda producirse sea puntual y paliable simplemente dando tiempo al sistema para que se recupere.

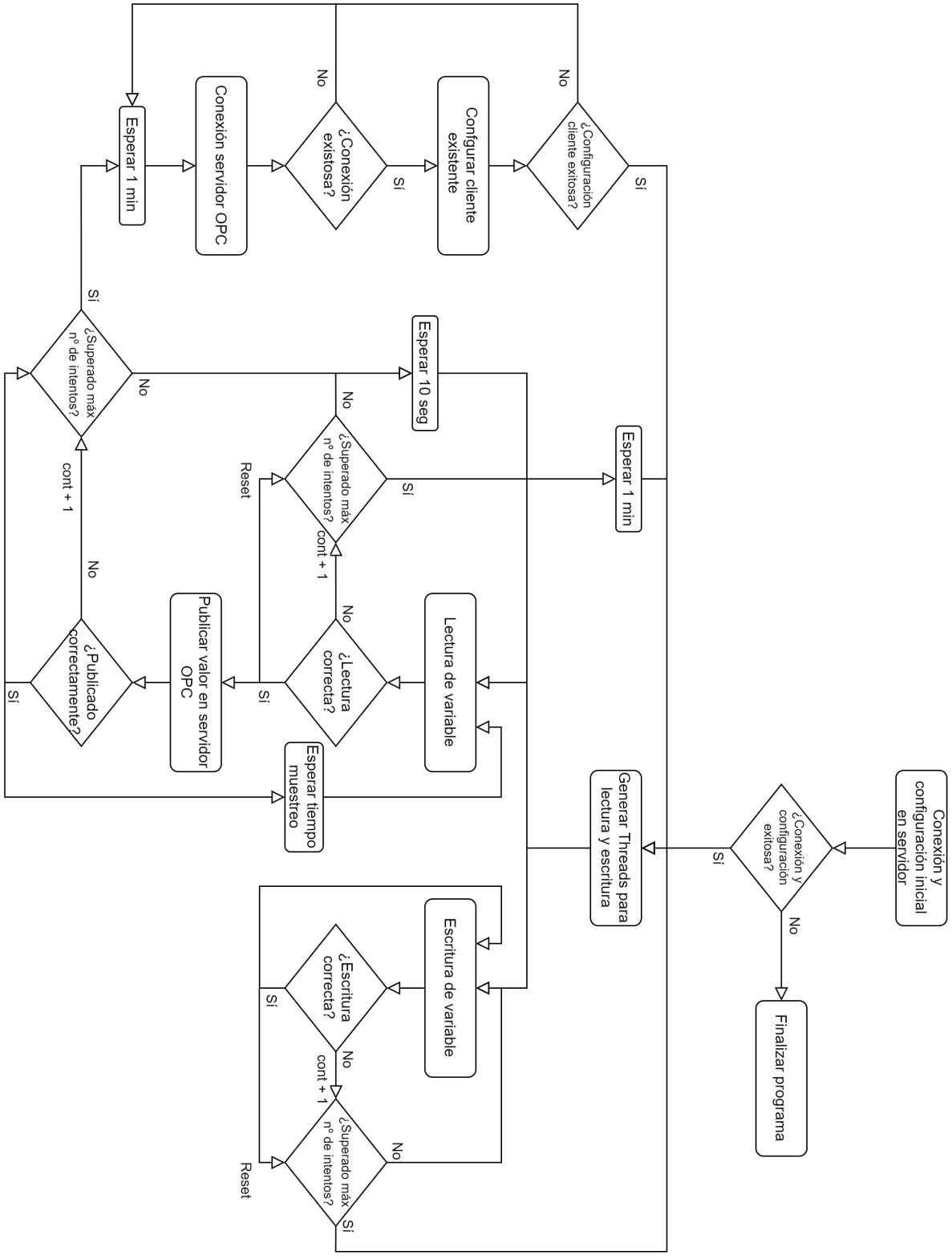


Figura 4.4. Diagrama de flujo de la lógica de reconexión

4.3.6 Configuración del servidor OPC, gestión de variables

Uno de los primeros pasos al iniciarse el programa es comprobar si ya existe un nodo con el mismo nombre del cliente a importar y si no es así, se crean los nodos correspondientes para cada variable. Esta comprobación inicial se hace asignando a una variable los *hijos* del nodo `objetos` (para entender mejor la estructura del estándar OPC léase el apartado 4.4 o el anexo A). Se comprueba el nombre de cada uno y, si coincide con el del cliente, se pregunta al usuario si desea sobreescribir el cliente (eliminar el existente y crear uno nuevo) o mantener la configuración existente en el servidor y hacerla compatible con la del archivo de configuración. Al final el resultado es el mismo con la diferencia de que los nodos cambian. Probablemente sea preferible mantener siempre que sea posible los nodos asociados a las variables, pues puede darse el caso de que alguna aplicación conectada esté leyendo las variables a partir del identificador del nodo. Es por ello que a menos que el usuario escriba explícitamente la letra `S` (mayúscula), no se sobreescribe.

El proceso de creación de un nodo para el cliente y su posterior población con nodos para cada variable se realiza en dos métodos: `configurarOPC` y `gestionarVariablesOPC`. El primero es el encargado de comprobar la existencia del cliente en el servidor y gestionar su eliminación o creación según sea necesario. Es quien llama a `gestionarVariablesOPC`, el cual tiene un argumento de entrada opcional `cliente_existente` (por defecto falso) que indica al método si ya existía el cliente o no.

El procedimiento es similar para el caso de dispositivos con comunicación por *Modbus* o API. La diferencia radica en que el listado de variables para *Modbus* viene definido en el archivo de configuración mientras que para las API hay que hacer una solicitud al servicio para obtener el listado. Otra diferencia importante es que, mientras que las variables de *Modbus* son un único valor, las variables de la estación comercial son multicampo. Esto se realiza en las primeras líneas del método:

Código 4.14. Extracto de método donde se gestiona el espacio de trabajo del cliente dentro del servidor

```
1 def gestionarVariablesOPC(conf_cliente, servidor, idx, cliente_OPC,
  ↪  cliente, cliente_existente = False):
2         """ Método que se encarga de añadir las nuevas variables en el
          ↪  servidor con los permisos de lectura/escritura y tipo de datos
          ↪  especificados en configuración """
3
4         if conf_cliente['protocolo'] == 'API':
5         # Importar tipo de datos propio definido para usarlo al añadir las
          ↪  variables
6         binary = servidor.get_node('ns=0;i=93')
7         for nodo in binary.get_children():
8                 if nodo.get_display_name().Text == 'EstasionComersial':
```

```
                # (clase definida, consultar con UAExpert en: Types ->
                ↪ DataTypes -> OPC Binary)
                nodo_estasionComersial = nodo
                # print(nodo_estasionComersial)
                # print('Tipo de datos EstasionComersial encontrado')
        # Extraer los nombres de las variables para compararlas con las del
        ↪ servidor OPC
        variables,_ = cliente.leerDatos(conf_cliente['station_id'])
```

4.3.7 Publicación de valores en servidor OPC

Esta parte del código se trata de un método que recibe como entrada la variable (realmente el nodo de la variable según la estructura del estándar OPC, explicado en anexo A), el tipo de dato de la misma y la marca de tiempo de cuándo se ha producido la lectura. El método podría tratarse de una línea en la que se usa la función `variable_OPC.set_value(valor)`. El problema con esta aproximación es que no especifica los datos temporales de la medida ni tampoco especifica explícitamente el tipo de dato del valor. Cuando se trabaja con dispositivos como PLCs es importante determinar clara y consistentemente el tipo de datos con los que se trabaja. Otro problema es que se encontraron inconsistencias o directamente marcas de tiempo erróneas (en *serverTimestamp*).

Código 4.15. Método para publicar valores en servidor OPC

```
def publicarOPC(variable_OPC, valor, tipo, tiempoLectura):
        """ Método que actualiza el valor de variable_OPC en el servidor
        ↪ OPC    """
        # Escribir usando datavalue para así poder establecer
        ↪ ServerTimeStamp y SourceTimeStamp
        dv = ua.DataValue(
        variant=ua.Variant(valor, eval('ua.VariantType.' + tipo)),
        sourceTimestamp=datetime.now(),
        serverTimestamp=datetime.now()
        )
        variable_OPC.set_value(dv)
```

La forma de tener control sobre todos los aspectos del valor publicado en el servidor es haciendo uso de la clase `datavalue`. En el fragmento de código mostrado (código 4.15) se genera una instancia de esta clase que se asigna a la variable `dv` y en ella se especifican tres campos: el valor acompañado del tipo de dato (`variant`) y las marcas de tiempo de la fuente y servidor (source/server Timestamp). Estas últimas usan el estándar UTC y, mientas que la primera viene generada en el momento que se produce la lectura, la

segunda se genera en el instante que se ejecuta este método, que es cuando se produce la publicación el servidor. La diferencia entre ambas es despreciable.

4.3.8 Lectura de valores desde dispositivo o servicio web

El método encargado de la lectura de valores se ha denominado *leerValor*. En este método se han distinguido tres formas de realizar la operación de lectura: lectura de una única variable, lectura de un grupo de variables y lectura de valores de servicio web. Para todos los casos, la lectura se produce con un tiempo de muestreo fijo, definido en el archivo de configuración. El tiempo de muestreo puede ser un vector conteniendo el tiempo de muestreo para cada variable individual o puede ser un único valor para todas las variables. Esto último es la única opción para API. Es precisamente el tiempo de muestreo el que define cómo se acceden a las variables; si se dispone de un valor único para cada variable, se realiza la lectura de la variable individual (argumento de entrada al método), mientras que si es un único valor (y no hay variables de escritura), se realiza la lectura en grupo.

La lectura de una variable es bastante sencilla una vez se identifica que la operación a realizar es esta (y no una de las otras dos opciones), simplemente se llama al método de lectura de la clase adecuada en base al protocolo escogido y se guarda el tiempo en que se ha producido la lectura:

Código 4.16. Extracto de método para leer valores. Caso lectura de una única variable

```
# Lectura de una única variable
if isinstance(conf_cliente['tiempo_muestreo'], list) or ('write' in
↪  conf_cliente['tipo_acceso']):
        # Leer es único para cada tipo de protocolo
        if conf_cliente['protocolo'] == 'ModbusTCP':
        valor,*_ = cliente.leerVariable(conf_cliente['registros'][i],
        ↪  conf_cliente['longitud'][i], conf_cliente['tipo_datos'][i])
        # print(valor)

        elif conf_cliente['protocolo'] == 'ModbusRTU':
        valor,*_ = cliente.leerVariable(conf_cliente['registros'][i],
        ↪  conf_cliente['longitud'][i], conf_cliente['tipo_datos'][i])

        # tz=timezone.utc
        sourceTimestamp = datetime.now() # Registrar tiempo en el que se ha
        ↪  producido la lectura
```

La lectura de grupos de variables es similar, únicamente en lugar de introducir una variable (especificada con el índice i), se envían las listas de registros, tipo de datos, etc. Otra diferencia es que mientras que para una variable desde el propio método se genera

la marca de tiempo, para este caso es el método que se llama el que genera este vector y lo devuelve como salida.

La lectura de datos desde API sí es diferente. No se conocen *a priori* las variables a leer, y cuando se realiza una solicitud se devuelven listas de valores. Se distinguen dos solicitudes: una en la que se devuelve información de las medidas disponibles con sus identificadores, y otra en la que se devuelven los valores de las lecturas asociadas a esos identificadores. El modo de proceder es entonces realizar una solicitud de los datos disponibles de la estación (una única vez), y entonces, en base al tiempo de muestreo, solicitar los valores. Este proceso puede dar cuatro resultados:

1. Lectura correcta. Se sigue con la publicación de los valores en el servidor.
2. Lectura incorrecta por fallo de conexión. Gestión de fallos (ver apartado 4.3.5).
3. Lectura incorrecta por *token* caducado. Llamar al método renovar *token*.
4. Lectura incorrecta por valor repetido. Los valores devueltos no han cambiado desde la última lectura, descartar lectura.

En los dos últimos casos es la clase `API` la que genera errores específicos y desde el programa cliente se gestionan:

Código 4.17. Gestión de valores repetidos o `token` caducado para API

```
try:
        datos_nuevos = True
        cliente.leerValor()
        except AssertionError as e:
                logging.warning(f'Fallo en la lectura: {e}') # Datos
                ↪ repetidos
                datos_nuevos = False
        except RuntimeError as e:
                logging.warning(f'Fallo en la lectura: {e}. Renovando Token
                ↪ para solucionarlo')
                datos_nuevos = False
                cliente.renovarToken()
    except Exception as e:
                logging.warning(f'Fallo en la lectura: {e}') # (cualquier
                ↪ otro caso)
```

La variable `datos_nuevos` se usa para decidir el tiempo a esperar hasta el próximo intento de lectura. Si se han recibidos valores nuevos entonces se espera el tiempo especificado en la configuración, mientras que si no se espera un tiempo inferior (1 min).

4.3.9 Escritura de valores en dispositivo

La escritura de valores en el dispositivo se produce cuando ocurren cambios en el valor de la variable asociada en el servidor OPC. Para ello, se realiza una suscripción al nodo de la variable usando la función `create_subscription`. Esta función recibe dos entradas: el período de comprobación de cambios en el nodo y el objeto a llamar cada vez que se produzca un cambio. El objeto es una instancia de la clase `SubHandler`.

La clase gestiona la acción a realizar cuando se produce un cambio y tiene dos métodos. El primer método se ejecuta cuando se crea la instancia (`__init__`) y almacena alguno de los objetos definidos en el apartado 4.2 y la configuración del cliente para posteriormente poder llamar al método de `escribirVariable` del objeto mencionado y disponer de los argumentos de entrada necesarios para este. El segundo método se llama `data_change_notification`. Se ha creado en base a la especificación de la librería y por ello recibe varios argumentos posicionales, en concreto el que es de utilidad para la funcionalidad que se busca es el de `val`, que únicamente almacena el valor de la variable. Como se ha comentado este es el encargado de llamar al método `escribirVariable` otorgándole los argumentos de entrada necesarios. Este método existe para las clases *ModbusTCP* y *RTU*, ya que las API con las que se han trabajado son sólo de lectura, aunque para darle soporte sería cuestión de crear un método `escribirVariable` en la clase API y añadir un condicional en `data_change_notification` para que se considere este caso.

4.3.10 Resumen

Se ha descrito en este capítulo de la memoria uno de los dos elementos claves desarrollados para la integración en una arquitectura unificada de servicios y dispositivos industriales: el programa encargado de realizar la conexión y comunicación a bajo nivel y la interacción con el servidor. Se ha diseñado de manera que un usuario sólo requiera introducir una configuración básica y se pueda abstraer del funcionamiento interno del programa. Además, con la estrategia usada para la implementación, la ampliación de las capacidades y protocolos soportados se puede hacer desarrollando nuevos módulos que se integren en el programa, sin necesidad de alterar o afectar a los existentes.

4.4 Servidor OPC

El segundo elemento clave de la arquitectura propuesta es el encargado de integrar toda la información recogida por el programa descrito en el apartado anterior de manera que cualquier aplicación (*MATLAB*, *SCADA*, etc) pueda abstraerse de la comunicación a bajo nivel. El estándar OPC es un protocolo de gran recorrido en aplicaciones industriales y con un amplio soporte por todo tipo de fabricantes (ver anexo A). De manera similar al

cliente genérico, la estrategia ha sido estudiar cómo funciona el protocolo a nivel teórico, investigar las librerías disponibles en *Python* que implementan este estándar y, tras entender su funcionamiento, programar un servidor propio que implemente las características propuestas en los objetivos.

4.4.1 Detalles de implementación

La librería *FreeOpcUa* [36] provee de una clase para el servidor que otorga todas las funciones necesarias para el despliegue del mismo. La única modificación que se desea hacer a esta clase servidor es proveer el soporte a los nuevos tipos de datos que se deseen implementar. Esto podría hacerlo cada cliente, pero daría lugar a cierta incertidumbre de si un tipo de dato ya está implementado o no a la hora de ejecutar un nuevo cliente que desee usar ese tipo de datos. Puesto que a modo de demostración de concepto se ha decidido implementar un nuevo tipo de dato, se ha creado directamente a nivel de servidor una nueva clase que hereda la clase `server` de la librería y que lo único que hace es extender la capacidad de esta, añadiendo soporte para el nuevo tipo de dato definido.

Una vez creado el objeto servidor se deben establecer unos parámetros básicos. El primero es `endpoint`, el cual establece la dirección que usan los clientes para conectarse al servidor; incluye una dirección IP seguida del puerto y una ruta que puede ser arbitraria, por ejemplo:

Código 4.18. Establecimiento de parámetros básicos del servidor

```
1        server.set_endpoint("opc.tcp://127.0.0.1:4841/SERVIDOR_OPC/")
2        server.set_server_name("Nombre arbitrario de servidor")
```

A continuación se aprovecha el método de la clase personalizada definida y se añade soporte al nuevo tipo de dato para las estaciones comerciales (explicado en el apartado 4.4.2), siendo necesario únicamente una llamada a método con los parámetros adecuados: `server.create_structure(server, idx, uri, name='tipoEstacionMeteo')`. Con esto resta establecer la seguridad del servidor y activar de manera automática la historización de los nodos que se conecten al servidor. El primero se explica en el apartado 4.4.3 y a continuación se desarrolla el segundo.

Puesto que la librería usada soporta de forma nativa la opción de mantener datos históricos en una ventana temporal, la activación de esta funcionalidad no requiere más de una línea de código: `server.historize_node_data_change(variable, period=timedelta(days=7), count=1000)`, donde `variable` es el nodo de la variable para la que se activa la característica, `period` es la ventana temporal durante la que se almacenan valores y `count` es el número de entradas máximas a almacenar antes de comenzar a descartar. Se entiende, por tanto, que ,en base al tiempo de publicación de nuevos valores, será más limitante, o bien la ventana temporal, o bien el número máximo

de valores almacenados. La cuestión se centra entonces en acceder a los nodos adecuados para activar la característica. Para ello se ha diseñado el siguiente bucle:

Código 4.19. Bucle para activar historización en cada nodo conectado al servidor

```
1 for i,cliente in enumerate(objetos.get_children(), 1):
2         if cliente.get_display_name().Text != 'Server':
3                 for variable in cliente.get_children():
4                         estado = variable.get_attribute(ua.AttributeIds.His
                          ↪  torizing).Value.Value
5                         # Si historización no activa, activar
6                         if estado == False:
7                                 server.historize_node_data_change(variable,
                                  ↪  period=timedelta(days=7),count=1000)
8                                 logging.info(f'Nueva variable:
                                  ↪  {cliente.get_display_name().Text} ->
                                  ↪  {variable.get_display_name().Text},
                                  ↪  activada historización')
```

Este bucle se repite con un tiempo de muestreo alto, pues su objetivo es comprobar la activación de la característica. Que se active 1 segundo después de su creación en el servidor a que lo haga 5 minutos después, no parece prioritario, pues una vez hecho en la ocasión inicial, ya no se requiere volver a actuar sobre la variable durante su estancia en el servidor. Recorrer cada variable en el cliente es un proceso computacionalmente costoso, por lo tanto, es preferible limitar el número de veces que se ejecuta. La variable que determina la activación o no de la historización en un nodo es la mostrada en la línea 7 del fragmento de código anterior (4.19). Una vez asignada a `estado`, sólo resta comprobar su valor y, en base a ello, activar la opción.

4.4.2 Soporte para definición de nuevos tipos de dato usando estructuras

Aunque en la mayoría de casos no es necesario, una variable puede tener más campos de interés además de la pareja tiempo-valor. Es el caso de las medidas proveídas por las estaciones meteorológicas, donde además de estas medidas básicas se proveen otras como desviación, un identificador, una descripción de la medida, etc. El estándar OPC ofrece la posibilidad de dar soporte a este tipo de variables multi-campo, aunque requiere de una implementación manual.

Cabe comentar que las posibilidades no acaban ahí. Se podría crear un nuevo tipo de dato a partir de otro tipo definido previamente, creándose así un nuevo tipo de dato más complejo (estructuras anidadas). Además, se pueden crear también objetos similares a una clase donde ya no se tendría sólamente una estructura simple o anidada, sino que también se podrían definir funciones dentro del propio objeto.

A modo de demostración de concepto, en este trabajo se ha implementado un nuevo tipo de dato para recoger ciertos parámetros devueltos por las estaciones comerciales, definiéndose para ello una estructura. La idea es establecer una plantilla para que si, en un futuro se quiere añadir soporte a nuevos tipos de variable, los cambios a realizar en el código queden claramente identificados y se pueda implementar de forma sencilla.

En concreto, se ha definido un tipo de dato que es básicamente una estructura simple con cuatro campos: ID, marca de tiempo, nombre de la variable y valor. La definición se ha realizado desde el servidor, aunque también se puede realizar desde el cliente. Esto permitiría que si se decide usar el programa desarrollado para los clientes, pero hacer uso de un servidor OPC comercial que tenga soporte para definir nuevos tipos de datos, entonces se podría realizar un proceso similar desde el cliente para dar soporte a este tipo de variables más complejas.

Para la implementación realizada, las estaciones comerciales comunicadas mediante un servicio web proveen la información encapsulada en una variable tipo diccionario. En este diccionario se incluyen muchos parámetros de interés para la medida, tales como la desviación, el identificador del sensor, un nombre para la medida, la marca de tiempo de cuando la estación produjo la lectura, etc. A modo de demostración de la capacidad del servidor, se ha decidido recuperar los siguientes campos de la información proveída (nombres de campos usados los de API del proyecto *IoT2020*, ver apartado 4.2.3):

1. `id_sensor_station_absolute`. Identificador de la estación como `ID`.
2. `recvTime`. Tiempo en que la estación realizó la medida como `timeStamp`.
3. `sensor_type.name_comun`. Nombre de la variable medida como `nombre`.
4. `Attvalue`. Valor de la medida como `valor`.

una vez entendido el funcionamiento funciones adecuadas a usar, la manera de implementar este nuevo tipo de dato es bastante sencillo. Como se ha comentado anteriormente, se ha integrado la definición del nuevo tipo de dato en el propio objeto servidor, heredando la clase `Server` de la librería y definiendo para esta un nuevo método `create_structure`, que crea un diccionario en el que se almacenan los nuevos tipos de dato definidos:

Código 4.20. Clase servidor heredada y ampliada con la creación de estructura para estación comercial

```
1 class CustomServer(Server):
2         """Clase para añadir soporte a variables con más de un campo, en
          ↪  concreto para las variables de la estación climática, donde se
          ↪  desea tener: {ID, nombre, valor, timeStamp} siguiendo ejemplo
          ↪  de: https://github.com/FreeOpcUa/python-opcua/blob/master/examp
          ↪  les/server-create-custom-structures.py"""
```

```
3
4         def create_structure(self, server, idx, uri, name='MiDiccionario'):
5                 self.dict_builder = DataTypeDictionaryBuilder(server, idx,
                  ↪  uri, name)
6                 nuevo_tipo = self.dict_builder.create_data_type(name)
7
8                 nuevo_tipo.add_field('ID', ua.VariantType.String)
9                 nuevo_tipo.add_field('timeStamp', ua.VariantType.DateTime)
10                nuevo_tipo.add_field('nombre', ua.VariantType.String)
11                nuevo_tipo.add_field('valor', ua.VariantType.Double)
12
13                self.dict_builder.set_dict_byte_string()
14
15                return nuevo_tipo
```

4.4.3 Seguridad

Hay que distinguir dos aspectos de seguridad: uno es la necesidad de un usuario de autentificarse para poder acceder al servicio, y otra es la seguridad de los mensajes. Una capacidad añadida interesante en un servidor OPC es el nivel de acceso en función del perfil de usuario, es decir, que existan tipos de usuario (administrador, usuario, mantenedor, etc) que puedan tener permisos distintos para realizar modificaciones en el servidor. Por ejemplo, sería interesante que el administrador tuviera acceso total de modificación del servidor, incluyendo modificar el valor de las variables de escritura, o cambiar si una variable es de lectura, escritura, etc. En cambio, si se quiere dar acceso al servidor a otro tipo de usuarios donde sólo se quiere que puedan leer los valores de las distintas variables pero sin tener oportunidad de manipular sus valores, es interesante tener un perfil distinto al de administrador. En cuanto a la seguridad de los mensajes, la técnica usada para todo tipo de comunicaciones es la encriptación de los mensajes [37]. Existen distintos tipos de encriptación, entre ellos, los soportados por el estándar OPC son: `Basic 256 SHA256, AES128 SHA256 RSA-OAEP and AES256 SHA256 RSA-PSS` [38]. es el usado en la implementación `Basic 256 SHA256`.

De un modo simplificado, consiste en la modificación del mensaje enviado mediante distintos métodos, de manera que alguien que capture el mensaje en su camino al receptor no pueda entender su contenido, mientras que el destinatario sí podrá. En el inicio de la comunicación ambos intercambian sus claves públicas y, cada vez que se envía un mensaje, se usa la clave pública de la otra parte para codificar el mensaje. Entonces el receptor hace uso de su clave única secreta (clave privada) para decodificar el mensaje. Para que la comunicación pueda establecerse, ambas partes deben confiar inicialmente una en otra

para intercambiar sus claves públicas incluidas en un certificado digital. Para que esto se produzca es necesario que ambos certificados estén firmados por una entidad certificadora de confianza para ambos. En la Figura 4.5 se puede observar un diagrama del proceso en función del tipo de claves usadas.

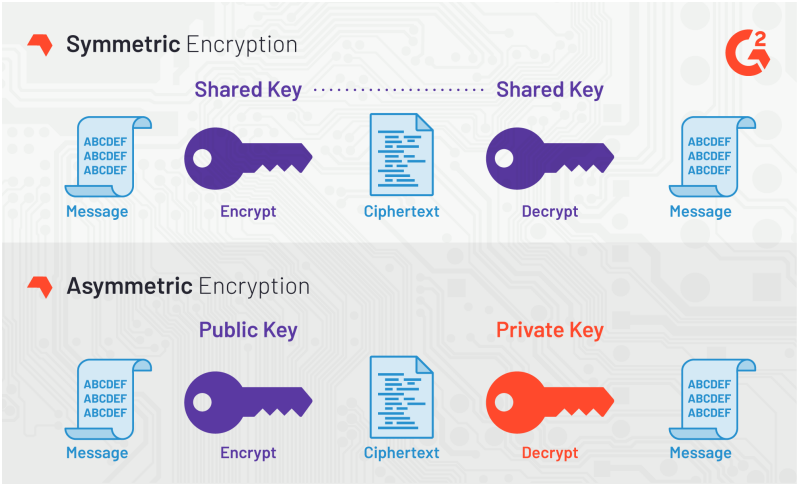


Figura 4.5. Diagrama del proceso de intercambio de mensajes encriptados. Imagen de `https://www.g2.com/articles/what-is-encryption`

Mientras que, como se ha explicado anteriormente, los certificados de emisor y receptor requieren estar firmados por una entidad en la que ambos confíen para que se acepten uno a otro, esto no funciona así en la implementación de la librería *FreeOpcUa*. En este caso, a la hora de realizar la conexión el cliente al servidor, estos intercambian directamente sus claves públicas confiando automáticamente uno en otro. Esto supondría un problema de seguridad, pues cualquier intruso podría leer los mensajes del sistema con cualquier certificado. Este agujero de seguridad es paliado con la autentificación. Para conectarse al servidor es necesario hacer uso de un usuario y contraseña, con lo cual, a menos que el intruso sea conocedor de esta, no podrá acceder al servidor y tampoco servirá capturarlos en su camino, pues no podrá decodificarlos. Si existiese la opción de usar certificados firmados por entidades certificadoras de confianza, se podría usar para realizar la autentificación sin necesidad de hacer uso de usuario y contraseña.

Cabe comentar que en este trabajo se soportan dos perfiles de acceso: administrador y usuario. El administrador tiene acceso total al servidor y puede realizar modificaciones tanto en las variables contenidas en este como en parámetros del mismo. El perfil de usuario sólo provee de acceso de lectura de las variables contenidas en el servidor, pero no permite realizar cambios en estas ni sobre el servidor.

4.4.4 Generación de certificados

Para la generación de los certificados para el servidor y para cada cliente que desee conectarse, aunque no se controla, se podría usar el mismo certificado para todos los clientes. Se ha desarrollado un *script* haciendo uso del lenguaje *sh* (*Shell Script*), que es un lenguaje de los sistemas operativos basados en *Linux*. La librería a usar se encuentra disponible de manera nativa en este tipo de sistemas operativos, aunque también se puede instalar en *Windows*[1]. Con la instrucción mostrada en el código 4.22 se genera una clave privada y un certificado en formato `.der` para un cliente genérico en el subdirectorio `certificados_clientes`, haciendo uso de la configuración contenida en `configuracion_SSL.conf`

Código 4.21. Configuración contenida en `configuracion_SSL.conf`

```
openssl req -x509 -newkey rsa:2048 -days 600 -keyout
↪  certificados\clientes\clave_privada_cliente.der -out
↪  certificados\clientes\certificado_cliente.der  -config
↪  certificados\configuracion_SSL.conf -addext
↪  "subjectAltName=IP:192.168.1.95" -subj "/organizationName=clienteGeneri
↪  co/commonName=CertificadoParaClienteGenerico"

Contenido configuracion_SSL.conf:
[ req ]
default_bits = 2048
default_md = sha256
distinguished_name = subject
req_extensions = req_ext
x509_extensions = req_ext
string_mask = utf8only
prompt = no
[ req_ext ]
basicConstraints = CA:FALSE
nsCertType = client, server
keyUsage = nonRepudiation, digitalSignature, keyEncipherment,
↪  dataEncipherment, keyCertSign
extendedKeyUsage= serverAuth, clientAuth
nsComment = "OpenSSL Generated Certificat"
subjectKeyIdentifier=hash
[ subject ]
countryName = ES
stateOrProvinceName = Almeria
localityName = Almeria
emailAddress = luismiguel@gymail.com
```

[1] *OpenSSL*: `https://www.openssl.org/`

4.5 Generación automática de base de datos

El servidor OPC es una herramienta muy potente para la integración de dispositivos y para acceso y lectura de variables de sistemas en tiempo real. También, dependiendo del servidor, se dispone de datos históricos limitados a ventanas temporales. Por ejemplo, la librería usada *FreeOpcUa* haciendo uso de *SQLite* ofrece la opción de activar la historización con datos de los últimos siete días por defecto. Esto no es suficiente si se requiere tener un registro de los datos históricos del sistema desde el inicio de la operación. Es por ello que parece interesante, una vez se han unificado todos los datos en el servidor, aprovechar esta capa de integración y ampliarla generando la base de datos permanente.

La forma de implementar esta funcionalidad ha sido el desarrollo de una clase que, en este caso, está definida en el mismo programa del servidor, pero bien se podría extraer en un archivo separada de igual manera a como se ha hecho para *libreriaClientes.py*, donde se han agrupado las clases para el soporte de los distintos protocolos. De esta forma, la base de datos se genera a través de las publicaciones en el servidor, y al integrarlo en el mismo no es necesario gestionar la autentificación y conexión. Igualmente no sería complicado partiendo de esta clase añadir un método para la conexión con el servidor y usarla en un programa separado, siendo el programa de base de datos a ojos del servidor un cliente más. O incluso integrarla en `clienteGenerico` y aprovechar el acceso al archivo de configuración para disponer de más información a la hora de generarla. El problema de esta última aproximación es que se tendrían distintas bases de datos. Siendo el objetivo de este TFM la integración, habiéndose usado un servidor OPC para ello, parecía más adecuado realizar también la integración de los datos históricos en una base de datos común para todo lo que esté conectado al servidor OPC.

4.5.1 Creación y actualización de tablas para cada variable

Como se ha comentado, la generación de la base de datos podría ser un cliente más que se conecta al servidor, pues la forma que tiene de extraer la información del mismo es suscribiéndose a los nodos cada vez que estos se crean. Para ello, primero sería necesario detectar la aparición de nuevos nodos, pero esto es bastante simple aprovechando el proceso explicado anteriormente para la activación de la historización de los distintos nodos (esta es otra ventaja de implementarlo junto al programa del servidor). Cuando se activa la historización, también se crea un *thread* que gestiona la suscripción al nodo y crea el objeto *base de datos* para la creación de la tabla:

Código 4.22. Método para la creación de nuevas tablas en la base de datos

```
1 def nuevaTabla(self, nodo_cliente, nodo_variable):
2 """Crear una nueva tabla para la variable en caso de que no exista"""
```

```
3
4           _, cliente, variable = self.crearNombreTabla(nodo_cliente,
            ↪  nodo_variable)
5
6           ...
7
8           tipo = type(nodo_variable.get_value())
9           if 'EstasionComersial' in str(tipo): # EstasionComersial
10                  c.execute(""" CREATE TABLE {} (
11                  SourceTimestamp TIMESTAMP NOT NULL,
12                  ServerTimestamp TIMESTAMP NOT NULL,
13                  ID TEXT,
14                  nombre TEXT,
15                  valor TEXT
16                  )""".format(self.nombre_tabla))
17
18          else:
19                  c.execute(""" CREATE TABLE {} (
20                  SourceTimestamp TIMESTAMP NOT NULL,
21                  ServerTimestamp TIMESTAMP NOT NULL,
22                  Value TEXT,
23                  VariantType TEXT
24                  )""".format(self.nombre_tabla))
25
26          self.logger.info(f'Creada nueva tabla para [{cliente} ->
            ↪  {variable}], con nombre {self.nombre_tabla}')
```

El primer paso es generar el nombre de la tabla, para lo que se ha creado un simple método que toma el nombre del cliente y el nombre de la variable y las une de forma `nombreCcliente_nombreVariable`. Antes se aplica un filtro a los nombres para asegurarse de que no hay caracteres incompatibles con la sintaxis de SQL. Acto seguido se lee el tipo de dato de la variable; si se trata de la clase definida para las estaciones comerciales entonces la tabla a generar es diferente. Ambas contienen los tiempos de lectura y publicación en el servidor y después se diferencian en que las variables simples sólo incluyen su valor, mientras que la estación comercial incluye alguno más.

Con la suscripción al nodo se genera una notificación cada vez que se detecta un cambio en el valor del nodo, con lo que es suficiente hacer que esta notificación llame al método de `actualizarTabla`. Este toma el valor (*datavalue*) del nodo y, de forma similar a la creación de la tabla en función de si se trata de una variable tipo estación comercial o una variable simple, genera una nueva entrada en la tabla:

Código 4.23. Extracto de método para la adición de nuevas entradas en tablas de la base de datos

```
def actualizarTabla(self, datavalue):
""" Añadir nueva fila en la tabla con el valor de la variable """
        data = datavalue.monitored_item.Value
        ...
        tipo = type(data.Value._value)
        if 'EstasionComersial' in str(tipo): # EstasionComersial
                # La escritura tiene que ser distinta cuando es una
                ↪ variable tipo estacion
                c.execute(""" INSERT INTO {} VALUES (?, ?, ?, ?, ?)
                """.format(self.nombre_tabla), (
                tiempo,
                tiempo_server,
                str(data.Value._value.ID),
                str(data.Value._value.nombre),
                str(data.Value._value.valor)
                ))
        else:
                c.execute(""" INSERT INTO {} VALUES (?, ?, ?, ?)
                """.format(self.nombre_tabla), (
                tiempo,
                tiempo_server,
                str(data.Value._value),
                str(data.Value._variantType._name_)
                ))
        ...
```

Para ello se usa la instrucción `INSERT INTO`; y para eliminar la complejidad de gestionar el tipo de dato de cada campo se introducen como variables de tipo `TEXTO`. La entrada a la función es `datavalue`, el cual tiene una estructura compleja. Por un lado tiene dos campos: `monitored_value` y `subscription_data`. El primero es el que incluye los datos de interés mientras que el segundo almacena información acerca de la suscripción. A su vez `monitored_item` posee varios campos, siendo el de interés `Value`. Este a su vez incluye `SourceTimestamp`, `ServerTimestamp` y `Value` (otra vez). Finalmente, para acceder a la variable que almacena el valor de la variable dentro de `Value` hay que acceder a `_value` (bastante redundante).

4.5.2 Gestión de la llamada a los distintos métodos para la creación de tabla cliente

El procedimiento es similar al explicado para comprobar si está activo el registro de datos históricos temporales en el servidor (apartado 4.4.1). La idea es, cada vez que transcurra un tiempo determinado (podrían ser 20 segundos, 1 minuto, etc) recorrer la lista de clientes conectados al servidor, comprobar si ya se ha activado la generación de la base de datos para el cliente y si no es así, activarla. Si un cliente se desconecta también debe de eliminarse el objeto asociado para la creación de su tabla, permitiendo que, si vuelve a conectarse, vuelva a generarse normalmente.

La manera de implementar esto ha sido mediante la creación de una lista de clientes existentes, es decir, clientes a los que ya se les ha activado la generación de la tabla, estando esta inicialmente vacía. A su vez se trabaja con otra lista que almacena los objetos creados para objeto `baseDeDatos` (el encargado de generar y actualizar las tablas). Entonces, para cada cliente, si no forma parte de esta lista, se realiza el siguiente proceso:

Código 4.24. Llamada al método para creación de tablas de agrupación de variables por cliente

```
if cliente not in clientes_exixtentes: # Nuevo cliente en servidor
        clientes_exixtentes.append(cliente)
        if args.nombre_database is not None:
                lista_database[clientes_exixtentes.index(cliente)] =
                ↪ baseDeDatos(args.nombre_database)
        else:
                lista_database.append(baseDeDatos())
                lista_database[clientes_exixtentes.index(cliente)].nuevaTab
                ↪ laCliente(cliente) # Crear
                ↪ tabla
```

En cambio, si el cliente ya forma parte de la lista de clientes existentes, entonces lo que se debe hacer es actualizar la tabla común con las nuevas lecturas disponibles desde la última llamada:

Código 4.25. Llamada a método para generación de nuevas entradas en tabla de agrupación de cliente

```
 # Actualizar valores en base de datos para todo el cliente
if cliente in clientes_exixtentes:
        try:
                lista_database[clientes_exixtentes.index(cliente)].actualiz
                ↪ arTablaCliente(cliente)
        except Exception as e:
                print(traceback.print_exc())
```

Finalmente sólo resta gestionar la conexión de clientes actualizando y manteniendo

las listas de clientes y objetos lo que se resuelve recorriendo las listas y comparando con los clientes conectados en momento de ejecución. Se puede apreciar cómo se ha podido implementar en pocas líneas de código:

Código 4.26. Actualización del listado de clientes

```
# Si se elimina algún cliente quitarlo de la lista de clientes existentes
# Listado de clientes en servidor:
clientes_servidor = set(objetos.get_children())
# Lista de índices de los clientes desconectados del servidor
indices_eliminados = [clientes_exixtentes.index(x) for x in
↪  clientes_exixtentes if x not in clientes_servidor]
# Actualizar listado de objetos tipo baseDeDatos para que sólo estén las
↪  que sigan en el servidor
lista_database = [database for x, database in enumerate(lista_database, 1)
↪  if x not in indices_eliminados]
# Actualizar listado de clientes existentes
clientes_exixtentes = [x for x in clientes_exixtentes if x in
↪  clientes_servidor]
```

4.5.3 Generación de tablas que agrupen clientes con mismo tiempo de muestreo

Se ha conseguido generar una tabla para cada variable en base a su tiempo de muestreo. Con esto ya se dispone de un registro de todos los datos del sistema. Esto podría ser suficiente, pero hay un uso particular al que le es de gran utilidad que la propia base de datos genere una tabla común para todas las variables de un cliente. Este es el caso para cuando se dispone de un tiempo de muestreo común. Si el soporte se limita a generar una tabla para cada variable individual, se requiere un trabajo posterior por parte del usuario de unir las tablas en una común. Con *SQLite* el procedimiento se describe a continuación. Primero se crea la tabla para el cliente, si es que esta no existe ya:

Código 4.27. Código SQL para creación de una tabla cualquiera de agrupación

```
create table if not exists cliente(
        SourceTimestamp TIMESTAMP NOT NULL UNIQUE,
        ServerTimestamp TIMESTAMP NOT NULL UNIQUE,
        S0 TEXT,
        V0 TEXT,
        V1 TEXT,
        V2 TEXT
);
```

En este caso se define la tabla *cliente* con las variables de marcas de tiempo y otras variables arbitrarias. El siguiente paso es seleccionar estas variables provenientes de las distintas tablas individuales:

Código 4.28. Selección de las variables de interés de cada tabla individual

```
INSERT INTO cliente
select
        cliente.SourceTimestamp,
        cliente_S0.ServerTimestamp,
        cliente_S0.Value as S0,
        cliente_V0.Value as V0,
        cliente_V1.Value as V1,
        cliente_V2.Value as V2
```

Hecho esto se produce la unión de las columnas seleccionadas partiendo de una tabla que en este caso es la de la primera variable:

Código 4.29. Parte del código para la unión de las tablas individuales

```
from cliente_S0
        inner join cliente_V0 on cliente_S0.ServerTimestamp =
        ↪  cliente_V0.ServerTimestamp
        inner join cliente_V1 on cliente_S0.ServerTimestamp =
        ↪  cliente_V1.ServerTimestamp
        inner join cliente_V2 on cliente_S0.ServerTimestamp =
        ↪  cliente_V2.ServerTimestamp
```

Se puede apreciar que la condición para la unión es que *ServerTimestamp* sea igual. De esto se ha asegurado el programa cliente genérico, pues cuando se produce la lectura se asigna una marca de tiempo individual en *SourceTimestamp* para cada variable, mientras que para el tiempo de publicación en el servidor (*ServerTimestamp*) se asigna la misma para el grupo de variables. Finalmente se añade una condición, y es que no se produzcan entradas repetidas en la tabla común lo que pasaría si se producen ejecuciones múltiples de esta lista de instrucciones. Para prevenir esto se usa:

Código 4.30. Filtración de entradas ya incorporadas

```
WHERE NOT EXISTS
(
        SELECT 1
        FROM cliente WHERE
        cliente_S0.ServerTimestamp = cliente.ServerTimestamp
);
```

Simplemente se compara el *ServerTimestamp* de la tabla común con la misma variable

de la primera tabla individual escogida y se asegura de que no exista ya en la común. Esta solución requeriría de realizar esta operación manual sustituyendo cada nombre de variable en cada uso particular que se haga. Por tanto, parece interesante que esto se haga de manera automática en la generación de la base de datos. El reto está entonces en generar las líneas de código mostradas en este apartado de manera automática para cualquier número y nombres de variable - tabla.

El primer paso es la generación de la tabla si esta no existe, lo que se hace en:

Código 4.31. Código de generación automática de tabla en *Python*

```
def nuevaTablaCliente(self, nodo_cliente):
        """ Tabla que agrupa todas las variables cuando tienen el mismo
        ↪  tiempo de muestreo.
        Llamar cada vez que se cree un nuevo cliente"""
        cliente = nodo_cliente.get_display_name().Text
        variables = nodo_cliente.get_children()

        texto = ''
        self.nombres_tablas = []
        for variable in variables:
        self.nombres_tablas.append(self.crearNombreTabla(nodo_cliente,
        ↪  variable))
        texto = texto + f'{variable.get_display_name().Text} TEXT,'
        texto = texto.rstrip(texto[-1])

        a = """ CREATE TABLE {nombre_tabla} (
        SourceTimestamp TIMESTAMP NOT NULL UNIQUE, ServerTimestamp
↪  TIMESTAMP NOT NULL UNIQUE, {txt})""".format(nombre_tabla=cliente,
↪  txt=texto)
```

La instrucción a generar se almacena en la variable *a*, donde `nombre_tabla` es simplemente el nombre del cliente en el servidor OPC. Las columnas de la tabla se dividen en una parte fija para las marcas de tiempo y en una parte variable para los valores de las distintas variables. Esta parte variable se genera en el bucle `for` anterior. Se inicia con una variable tipo cadena de caracteres vacía a la que en cada iteración se le añade la entrada para cada una de las variables. La estructura es `nombre_variable TEXT`, donde `TEXT` indica el tipo de dato.

El segundo paso es el de la actualización de las entradas de la tabla en base a las tablas individuales, el grueso del código `SQLite` (mostrado en códigos 4.27 - 4.31). De esto se encarga el método `actualizarTablaCliente`. Igual que en el paso anterior, en la variable *a* se almacena el texto con el código a introducir en *SQLite*:

Código 4.32. Selección, unión y filtración de tablas individuales en *Python*

```
a = """ INSERT INTO {nombre_tabla}
        select
                {txt_1}
                from {nombre_subtabla_1}
                {txt_2}
        where not exists
        (
        select 1
        from {nombre_tabla} where
        {nombre_subtabla_1}.ServerTimestamp = {nombre_tabla}.ServerTimestamp
        ) """.format(nombre_tabla=cliente, txt_1=texto,
↪  nombre_subtabla_1=self.nombres_tablas[0][0], txt_2=texto_2)
```

Es necesario entonces generar `cliente`, `texto`, `nombre_tablas` y `texto_2`. La variable `cliente` es simplemente el nombre del nodo, `nombre_tablas` se genera llamando al método `crearNombreTabla` para cada variable y almacenando su valor como propiedad de la clase, lo que se realiza durante la creación de la tabla, por lo que ya se halla disponible para cuando se llama a la actualización de la tabla. Resta entonces generar las cadenas de texto que seleccionan las columnas de las distintas tablas individuales y posteriormente la condición de unión determinada por la marca de tiempo de la publicación en el servidor:

Código 4.33. Generación de valores necesarios para variables de cod. 4.32

```
texto = f'{self.nombres_tablas[0][0]}.SourceTimestamp,
↪  {self.nombres_tablas[0][0]}.ServerTimestamp, '
texto_2 = ''
for variable in variables:
        nombre_subtabla = self.nombres_tablas[variables.index(variable)][0]
        texto = texto + f'{nombre_subtabla}.Value as
        ↪  {variable.get_display_name().Text},'
        if variables.index(variable) < len(variables)-1:  # Para que no
        ↪  llegue a la última
                nombre_subtabla_sig =
                ↪  self.nombres_tablas[variables.index(variable)+1][0]
                texto_2 = texto_2 + f'inner join {nombre_subtabla_sig} on
                ↪  {self.nombres_tablas[0][0]}.ServerTimestamp =
                ↪  {nombre_subtabla_sig}.ServerTimestamp '
texto = texto.rstrip(texto[-1])
```

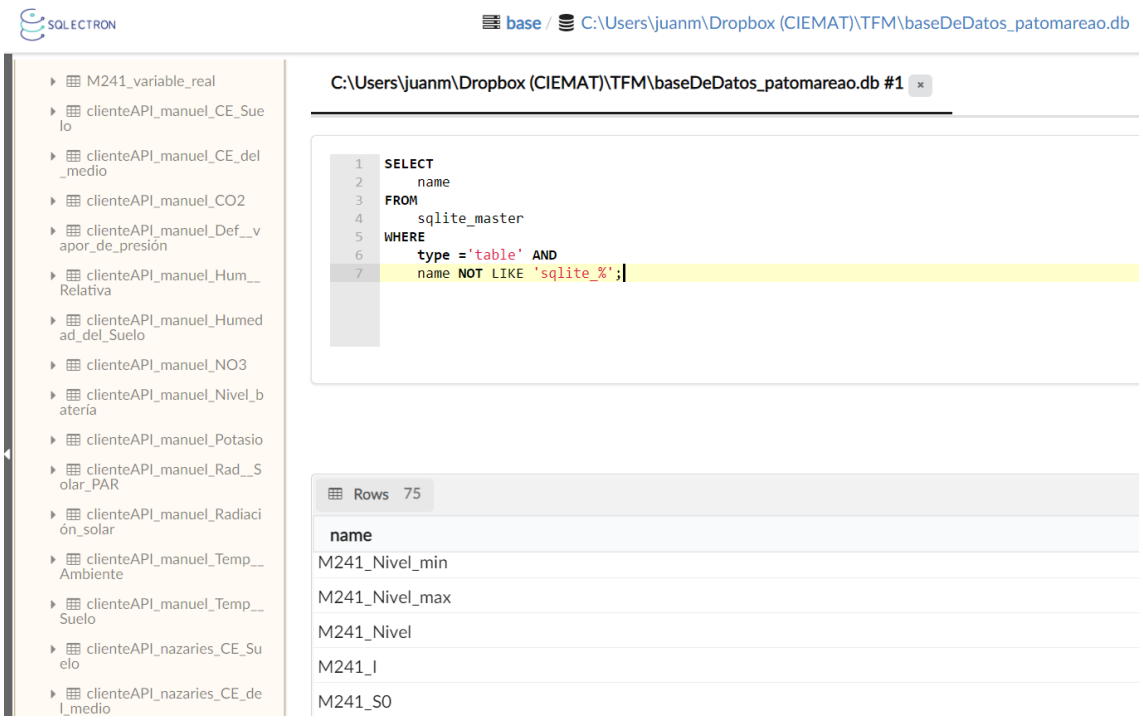


Figura 4.6. Fragmento de listado de tablas disponibles en base de datos en momento de ejecución

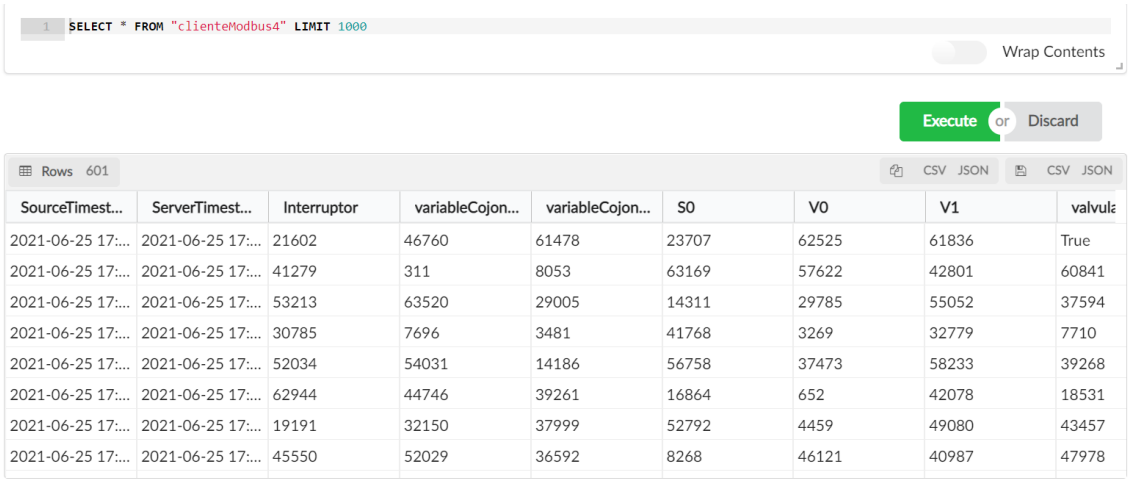
4.5.4 Ejemplo de resultado

Usando el cliente para bases de datos *SQElectron* se puede acceder a la base de datos generada. Las distintas pruebas han populado la base de datos con las tablas mostradas en la Figura 4.6. En la Figura 4.7 se puede apreciar el resultado de la creación de la tabla con todas las variables unificadas para un *clienteModbus*.

4.5.5 Limitaciones de la implementación actual

Con la implementación actual se tienen ciertas limitaciones; fundamentalmente se trata de que *SQLite* es una base de datos simple, adecuada para usar durante el desarrollo, pruebas iniciales o aplicaciones de pequeña escala, pero una vez se dispone de una solución suficiente y se procede al uso extendido de esta, es preferible hacer uso de bases de datos más completas y complejas como lo puede ser *Oracle*, *MySQL* o preferiblemente *PosgreSQL*. Esta última es preferible por ser de código abierto y por estar *SQLite* inspirada en ella en varios aspectos, por lo que la migración de una base de datos implementada usando *SQLite* a *PostgreSQL* requiere cambios mínimos de código (o ninguno) [39].

Por otro lado, *MATLAB* cuando realiza la conexión bloquea la base de datos mientras no se cierre esta. Esto no es un problema si el usuario que accede a la base de datos realiza



```
1 SELECT * FROM "clienteModbus4" LIMIT 1000
```

Wrap Contents

Execute or Discard

Rows 601 CSV JSON CSV JSON

SourceTimest...	ServerTimest...	Interruptor	variableCojon...	variableCojon...	S0	V0	V1	valvula
2021-06-25 17:...	2021-06-25 17:...	21602	46760	61478	23707	62525	61836	True
2021-06-25 17:...	2021-06-25 17:...	41279	311	8053	63169	57622	42801	60841
2021-06-25 17:...	2021-06-25 17:...	53213	63520	29005	14311	29785	55052	37594
2021-06-25 17:...	2021-06-25 17:...	30785	7696	3481	41768	3269	32779	7710
2021-06-25 17:...	2021-06-25 17:...	52034	54031	14186	56758	37473	58233	39268
2021-06-25 17:...	2021-06-25 17:...	62944	44746	39261	16864	652	42078	18531
2021-06-25 17:...	2021-06-25 17:...	19191	32150	37999	52792	4459	49080	43457
2021-06-25 17:...	2021-06-25 17:...	45550	52029	36592	8268	46121	40987	47978

Figura 4.7. Ejemplo de tabla unida para un cliente cualquiera

una conexión y, una vez extraídos los datos necesarios, cierra la misma. Sin embargo, si el usuario mantiene la conexión abierta, el programa encargado de acceder y actualizar la base de datos con la información del servidor no puede realizar su trabajo y se pierde esta funcionalidad mientras no se libere el bloqueo.

Una breve explicación de *SQLite* y las diferencias con otro motor de base de datos como *PostgreSQL* se desarrolla en el anexo C.

Capítulo 5

Resultados

En este capítulo se presentan los resultados de la aplicación del trabajo realizado. En primer lugar, se exponen los casos de estudio con los que se ha puesto en marcha la arquitectura desarrollada, explicando su planteamiento, funcionamiento y conclusiones en base al desempeño observado. A continuación, se ofrecen resultados adiciones fruto de la necesidad de su implementación para la realización de los casos de estudio. Estos son, por un lado, una plantilla para la lectura y escritura de variables reales mediante *Modbus* en los controladores lógicos y, por otro lado, conocido el amplio uso de la aplicación *MATLAB* dentro del entorno de investigación, se aporta una plantilla para la conexión y manipulación de variables en tiempo real usando esta aplicación, así como un ejemplo de acceso y extracción de información de la base de datos. Finalmente, se presenta el presupuesto del trabajo técnico realizado.

5.1 Casos de estudio

Se plantean dos casos de estudio. En el primero se dispone de una configuración experimental compuesta por dos controladores lógicos de la empresa *Schneider Electric*, en concreto el modelo M241. Estos han sido programados para simular un sistema de riego alimentado por un tanque. Adicionalmente, se ha integrado una estación meteorológica. En el segundo caso de estudio se han integrado medidas eléctricas provenientes de distintos subsistemas de un invernadero experimental situado en La Mojonera (Almería), además de un caudalímetro comunicándose por *ModbusRTU*.

5.2 Integración de dos PLCs y estación comercial

Se dispone de un conjunto compuesto por dos sistemas de riego controlados por dos controladores lógicos. Cada uno de ellos abastece a dos sectores de cultivo que deben ser regados periódicamente. Se dispone de un tanque que provee del agua para cubrir esta necesidad. El tanque inicia su llenado cuando se activa un interruptor, lo que provoca la apertura de la válvula de llenado. Cuando el tanque completa su carga, automáticamente inicia el riego de los dos sectores. Durante el riego, cuando el tanque alcanza un nivel determinado, se limita el riego a uno de los sectores. Este sistema dispone de dos señales de control manipulables: `S0` es una señal de seguridad que desactiva el llenado incluso aunque el interruptor esté activo e `I` que es el interruptor que controla la operación. Las salidas para activar actuadores se corresponden con el estado de apertura de las válvulas. El caso de estudio se ha programado en la herramienta para programación de controladores lógicos *SoMachine*, de *Schneider Electric*. Para ello se ha usado lenguaje estructurado (ST, *Structured Text*). Ambos controladores lógicos se integran usando el protocolo *ModbusTCP*. Un esquema del caso se muestra en la Figura 5.1.

La activación del sistema se puede realizar desde el propio servidor, así como desde *MATLAB*. Se ha programado la representación gráfica donde se mostrará la evolución en tiempo real del sistema así como usando medidas provenientes de los datos históricos del sistema. Adicionalmente se integrará la estación meteorológica.

La configuración física para realizar la emulación del sistema descrito se puede observar en la Figura 5.2. Se ha empleado un equipo con el *software SoMachine*, el cual se usa para la programación de los dos PLC (ver Figura 5.3) que se aprecian en la foto. También se usa para ejecutar el programa para generar el servidor OPC. En un equipo portátil se conectan los dos PLC a través de la red local. Además, se ejecutan dos instancias del programa de `cliente genérico` para integrarlos en el servidor. En este equipo también se ejecuta el *script* de *MATLAB* así como el cliente OPC.

En la Figura 5.6 se muestra una configuración de las entradas y salidas en el módulo *software* que permite la comunicación vía *ModbusTCP*. Además, es necesario configurar la conexión introduciendo la dirección IP del dispositivo en el que se va a ejecutar el programa `cliente genérico`, es decir, el equipo que vaya a interactuar con el PLC. Para ello, se asigna la IP como se muestra en la Figura 5.4. Se hace uso de variables pasarela tipo `WORD` que posteriormente se transforman según necesidad, esta adaptación se explica en un apartado más adelante. El primer paso es cargar el programa en el PLC y poner cada PLC en modo ejecución. En la Figura 5.5 se puede ver el programa durante la ejecución y cómo funciona el proceso de transformación de los distintos tipos de datos. Posteriormente, se debe ejecutar el programa adaptador, especificando la configuración que se muestra a continuación:

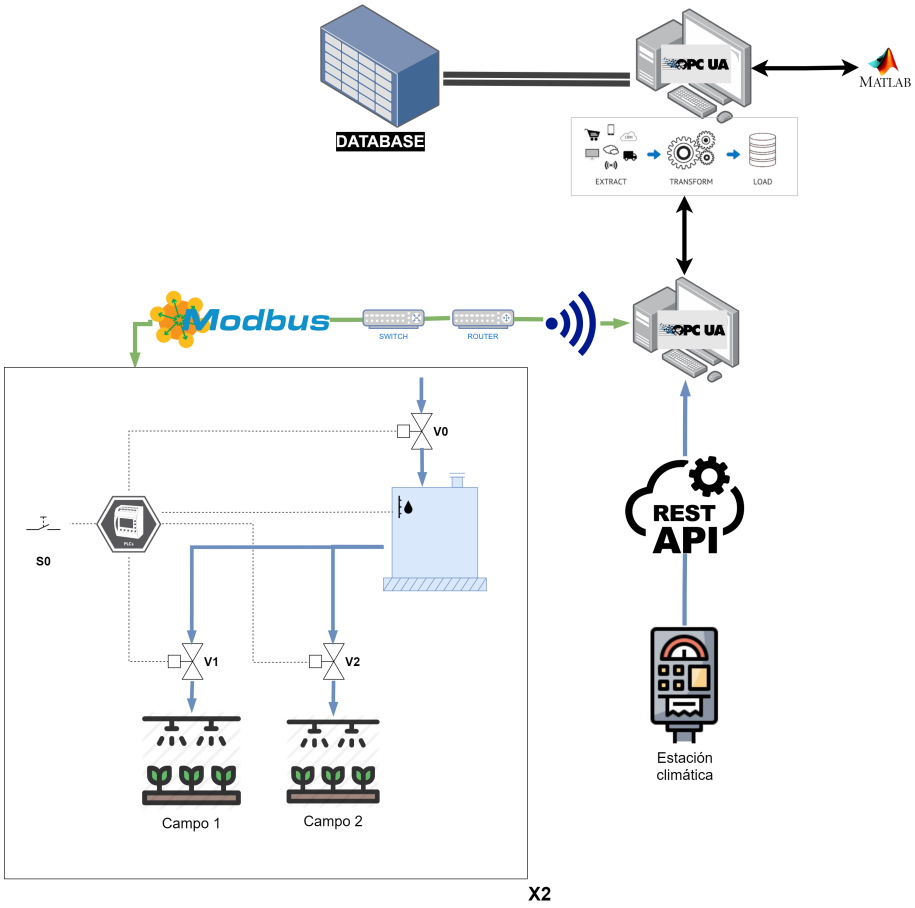


Figura 5.1. Esquema conceptual del primer caso de aplicación propuesto

(a) Imagen de los dos PLC, *switch* y *router*.

(b) Otro ángulo. Gato para escala.

Figura 5.2. Configuración experimental para el primer caso de estudio.

ConnectionMode	Controller	ProjectName	IP_Address	TimeSinceBoot	NodeName	ProjectAuthor	FW_Version
ETH	TM241CE24R	Invelnadelo2	192.168.1.142	03h 31m 01s	C2	juanm	V4.0.6.38
ETH	**TM241CE24R**	**Invelnadelo2**	**192.168.1.143**	**03h 31m 01s**	**C3**	**juanm**	**V4.0.6.38**
PC	CODESYS Control Win V3				JUAXMIX		V3.5.3.83

Figura 5.3. PLCs conectados a la red local y visibles desde *SoMachine* para su programación.

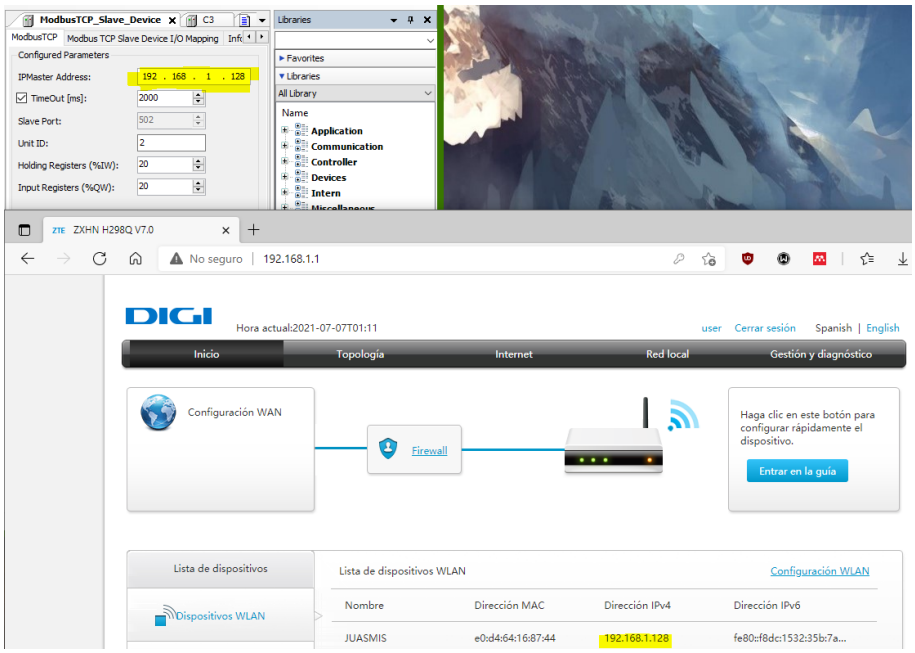


Figura 5.4. Configuración de la conexión en el módulo esclavo de comunicación *ModbusTCP*.

```
1  // Entradas
2  Evolucion.I TRUE := WORD_TO_BOOL(word_I 1 );
3  Evolucion.S0 FALSE := WORD_TO_BOOL(word_S0 0 );
4  // Entrada variable REAL
5  variable_real 0 :=DW_TO_REAL( X:=DWORD_OF_WORD(W1:=word4 0 , W0:=word44 0 ) );
6
7  // Salidas
8  word_V0 1 := BOOL_TO_WORD(Evolucion.V0 TRUE );
9  word_V1 0 := BOOL_TO_WORD(Evolucion.V1 FALSE );
10 word_V2 0 := BOOL_TO_WORD(Evolucion.V2 FALSE );
11
12 // Intento salida REAL
13 dword_level 1061284422 := REAL_TO_DW(X:= Level 0.758 );
14 word_level 59974 := WORD_OF_DWORD(in:=dword_level 1061284422 , N:= 0);
15 word_level2 16193 := WORD_OF_DWORD(in:=dword_level 1061284422 , N:= 1);
16 variable_real2 0.757 :=DW_TO_REAL( X:=DWORD_OF_WORD(W1:=word_level2 16193 , W0:=word_level 59974 ) );
17
18 dword_nivelmax 1065353216 := REAL_TO_DW(X:= NivelMax 1 );
19 word_nivelmax 0 := WORD_OF_DWORD(in:=dword_nivelmax 1065353216 , N:= 0);
20 word_nivelmax2 16256 := WORD_OF_DWORD(in:=dword_nivelmax 1065353216 , N:= 1);
21
22 dword_nivelmin 0 := REAL_TO_DW(X:= NivelMin 0 );
23 word_nivelmin 0 := WORD_OF_DWORD(in:=dword_nivelmin 0 , N:= 0);
24 word_nivelmin2 0 := WORD_OF_DWORD(in:=dword_nivelmin 0 , N:= 1);
25 RETURN
```

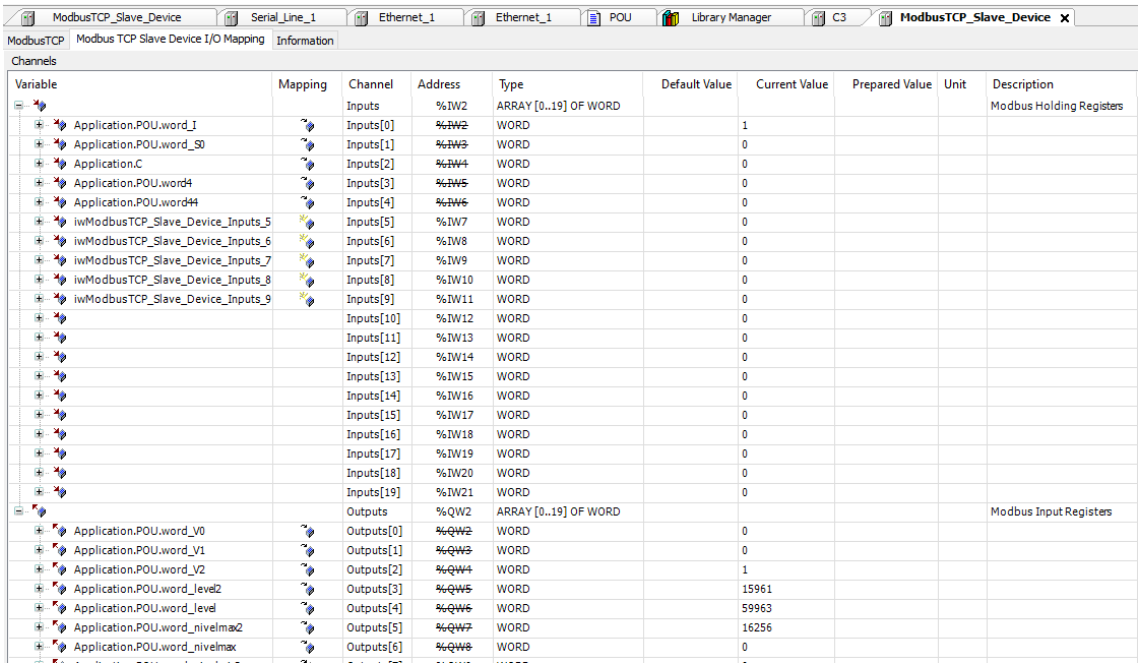
Figura 5.5. Programa en ejecución en uno de los PLCs.

Código 5.1. Configuración para comunicación con PLCs via *Modbus*

```
M241_C2:
  protocolo: ModbusTCP
  ID: 2
  registros:        [0,  1, 20, 21, 22,  3, 25, 27, 23]
  longitud:         [1,  1,  1,  1,  1,  2,  2,  2,  2]
  tiempo_muestreo:  [2,  2,  1,  1,  3,  4,100,100,0.5]
  variables: ['I','S0','V0','V1','V2','variable_real','Nivel_max',
  ↪  'Nivel_min', 'Nivel']
  tipo_datos: ['Boolean','Boolean','Boolean','Boolean','Boolean','Float',
  ↪  'Float','Float','Float']
  tipo_acceso: ['write','write','read','read','read','write','read',
  ↪  'read','read']
  IP: 192.168.1.142
  puerto: 502

M241_C3:
  protocolo: ModbusTCP
  ID: 247
  registros:        [0,  1, 20, 21, 22,  3, 25, 27, 23]
  longitud:         [1,  1,  1,  1,  1,  2,  2,  2,  2]
  tiempo_muestreo:  [2,  2,  1,  1,  3,  4,100,100,0.5]
  variables: ['I','S0','V0','V1','V2','variable_real','Nivel_max',
  ↪  'Nivel_min', 'Nivel']
  tipo_datos: ['Boolean','Boolean','Boolean','Boolean','Boolean','Float',
  ↪  'Float','Float','Float']
  tipo_acceso: ['write','write','read','read','read','write','read',
  ↪  'read','read']
  IP: 192.168.1.143
  puerto: 502
```

Se puede apreciar que las variables que forman parte del apartado entradas de la Figura 5.6 ocupan los primeros números de registro, siendo cada variable `WORD` un registro. Las variables pertenecientes a las salidas parten de un número de registro *n+m-1*, donde n es el número de registros de entrada (*Holding Registers*) y *m* es es la posición que ocupa en el registro de salida (*Input Registers*)[40].

De manera similar, se muestra la configuración para la estación meteorológica para las APIs soportadas. La contraseña se puede introducir en la configuración o por seguridad dejar el parámetro en blanco, con lo que el programa lo pedirá tras su ejecución (código 5.2).



Variable	Mapping	Channel	Address	Type	Default Value	Current Value	Prepared Value	Unit	Description
		Inputs	%IW2	ARRAY [0..19] OF WORD					Modbus Holding Registers
Application.POU.word_I		Inputs[0]	~~%IW2~~	WORD		1			
Application.POU.word_S0		Inputs[1]	~~%IW3~~	WORD		0			
Application.C		Inputs[2]	~~%IW4~~	WORD		0			
Application.POU.word4		Inputs[3]	~~%IW5~~	WORD		0			
Application.POU.word44		Inputs[4]	~~%IW6~~	WORD		0			
iwModbusTCP_Slave_Device_Inputs_5		Inputs[5]	%IW7	WORD		0			
iwModbusTCP_Slave_Device_Inputs_6		Inputs[6]	%IW8	WORD		0			
iwModbusTCP_Slave_Device_Inputs_7		Inputs[7]	%IW9	WORD		0			
iwModbusTCP_Slave_Device_Inputs_8		Inputs[8]	%IW10	WORD		0			
iwModbusTCP_Slave_Device_Inputs_9		Inputs[9]	%IW11	WORD		0			
		Inputs[10]	%IW12	WORD		0			
		Inputs[11]	%IW13	WORD		0			
		Inputs[12]	%IW14	WORD		0			
		Inputs[13]	%IW15	WORD		0			
		Inputs[14]	%IW16	WORD		0			
		Inputs[15]	%IW17	WORD		0			
		Inputs[16]	%IW18	WORD		0			
		Inputs[17]	%IW19	WORD		0			
		Inputs[18]	%IW20	WORD		0			
		Inputs[19]	%IW21	WORD		0			
		Outputs	%QW2	ARRAY [0..19] OF WORD					Modbus Input Registers
Application.POU.word_V0		Outputs[0]	~~%QW2~~	WORD		0			
Application.POU.word_V1		Outputs[1]	~~%QW3~~	WORD		0			
Application.POU.word_V2		Outputs[2]	~~%QW4~~	WORD		1			
Application.POU.word_level2		Outputs[3]	~~%QW5~~	WORD		15961			
Application.POU.word_level		Outputs[4]	~~%QW6~~	WORD		59963			
Application.POU.word_nivelmax2		Outputs[5]	~~%QW7~~	WORD		16256			
Application.POU.word_nivelmax		Outputs[6]	~~%QW8~~	WORD		0			

Figura 5.6. Configuración de las entradas y salidas del adaptador *ModbusTCP* para uno de los controladores.

Código 5.2. Configuración para comunicación con estación comercial mediante APIs

```
clienteAPI_nazaries:
  protocolo: API
  tipo_api: nazaries
  usuario: correoprivado@ual.es
  password:
  tiempo_muestreo: 20 # 20seg para pruebas, 900seg o 15min
  station_id: 191

clienteAPI_manuel:
  protocolo: API
  tipo_api: manuel
  usuario: correoprivado@ual.es
  password:
  tiempo_muestreo: 900 # 20seg para pruebas, 900seg o 15min
  station_id: 191
```

El resultado obtenido tras la consulta usando un cliente cualquiera es el mostrado en la Figura 5.7. Se puede ver cómo el tiempo registrado por la estación comercial para la lectura del valor es distinto del momento en el que ha sido publicado en el servidor. Para ver el valor de cada una de las variables, se puede acceder y obtener lo mostrado

Data Access View

#	Server	Node Id	isplay Nam	Value	Datatype	Source Timestamp	Server Timestamp	Statuscode
1	Servidor ...	NS2|Num...	CE Suelo	Double cli...	Extension...	14:13:53.000	14:20:42.398	Good
2	Servidor ...	NS2|Num...	CE del m...	Double cli...	Extension...	14:13:53.000	14:20:42.378	Good
3	Servidor ...	NS2|Num...	CO2	Double cli...	Extension...	14:13:53.000	14:20:42.812	Good
4	Servidor ...	NS2|Num...	Def. vapo...	Double cli...	Extension...	14:13:53.000	14:20:42.820	Good
5	Servidor ...	NS2|Num...	Hum. Rel...	Double cli...	Extension...	14:13:53.000	14:20:42.358	Good
6	Servidor ...	NS2|Num...	Humedad...	Double cli...	Extension...	14:13:53.000	14:20:42.484	Good
7	Servidor ...	NS2|Num...	Humedad...	Double cli...	Extension...	17:10:33.000	14:20:42.910	Good
8	Servidor ...	NS2|Num...	Nivel bate...	Double cli...	Extension...	14:13:53.000	14:20:42.692	Good
9	Servidor ...	NS2|Num...	RT CE Su...	Double cli...	Extension...	17:10:33.000	14:20:42.898	Good
10	Servidor ...	NS2|Num...	RT CE del...	Double cli...	Extension...	17:10:33.000	14:20:42.882	Good
11	Servidor ...	NS2|Num...	RT CO2	Double cli...	Extension...	17:10:33.000	14:20:43.033	Good
12	Servidor ...	NS2|Num...	RT Def. v...	Double cli...	Extension...	17:10:33.000	14:20:42.864	Good
13	Servidor ...	NS2|Num...	RT Hum. ...	Double cli...	Extension...	17:10:33.000	14:20:42.981	Good
14	Servidor ...	NS2|Num...	RT Rad. S...	Double cli...	Extension...	17:10:33.000	14:20:42.950	Good
15	Servidor ...	NS2|Num...	RT Radiac...	Double cli...	Extension...	17:10:33.000	14:20:42.848	Good
16	Servidor ...	NS2|Num...	RT Temp. ...	Double cli...	Extension...	17:10:33.000	14:20:43.001	Good
17	Servidor ...	NS2|Num...	RT Temp. ...	Double cli...	Extension...	17:10:33.000	14:20:42.926	Good
18	Servidor ...	NS2|Num...	RT_BAT	Double cli...	Extension...	17:10:33.000	14:20:42.966	Good
19	Servidor ...	NS2|Num...	Rad. Sola...	Double cli...	Extension...	14:13:53.000	14:20:42.564	Good
20	Servidor ...	NS2|Num...	Radiación...	Double cli...	Extension...	14:13:53.000	14:20:42.840	Good
21	Servidor ...	NS2|Num...	Temp. A...	Double cli...	Extension...	14:13:53.000	14:20:42.366	Good
22	Servidor ...	NS2|Num...	Temp. Su...	Double cli...	Extension...	14:13:53.000	14:20:42.528	Good

Figura 5.7. Variables de la estación meteorológica disponibles desde un cliente conectado al servidor OPC

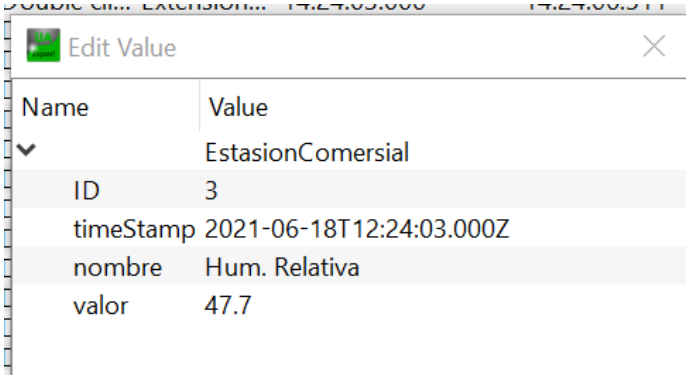


Figura 5.8. Valor de una de las variables de la estación meteorológica

```
09-Jul-21 12:11:52 - [WARNING] - Fallo en la lectura: Los valores no
 han cambiado desde la última lectura
09-Jul-21 12:12:55 - [INFO] - Valor leído -> CE Suelo|| ID: 2_191, T
ime: 2021-07-09 10:08:45, Valor: 2.61
09-Jul-21 12:12:55 - [INFO] - Valor leído -> CE del medio|| ID: 3_19
1, Time: 2021-07-09 10:08:45, Valor: 0.23
09-Jul-21 12:12:55 - [INFO] - Valor leído -> CO2|| ID: 15_191, Time:
 2021-07-09 10:08:45, Valor: 494
09-Jul-21 12:12:55 - [INFO] - Valor leído -> Def. vapor de presión||
 ID: 8_191, Time: 2021-07-09 10:08:45, Valor: 4.274833291
09-Jul-21 12:12:55 - [INFO] - Valor leído -> Hum. Relativa|| ID: 6_1
91, Time: 2021-07-09 10:08:45, Valor: 34.9
09-Jul-21 12:12:55 - [INFO] - Valor leído -> Humedad del Suelo|| ID:
 5_191, Time: 2021-07-09 10:08:45, Valor: 17.4
09-Jul-21 12:12:55 - [INFO] - Valor leído -> NO3|| ID: 17_191, Time:
 2021-07-09 10:08:45, Valor: 1999.1
09-Jul-21 12:12:55 - [INFO] - Valor leído -> Nivel batería|| ID: 1_1
91, Time: 2021-07-09 10:08:45, Valor: 4.18
09-Jul-21 12:12:55 - [INFO] - Valor leído -> Potasio|| ID: 21_191, Time: 2021-07-09 10:08:45, Valor: 1891.7
09-Jul-21 12:12:55 - [INFO] - Valor leído -> Rad. Solar PAR|| ID: 43_191, Time: 2021-07-09 10:08:45, Valor: 216.070298077
09-Jul-21 12:12:55 - [INFO] - Valor leído -> Radiación solar|| ID: 42_191, Time: 2021-07-09 10:08:45, Valor: 449.42622
09-Jul-21 12:12:55 - [INFO] - Valor leído -> Temp. Ambiente|| ID: 392_191, Time: 2021-07-09 10:08:45, Valor: 37.6
09-Jul-21 12:12:55 - [INFO] - Valor leído -> Temp. Suelo|| ID: 390_191, Time: 2021-07-09 10:08:45, Valor: 40.556
09-Jul-21 12:13:23 - [WARNING] - Fallo en la lectura: Los valores no han cambiado desde la última lectura
09-Jul-21 12:14:24 - [WARNING] - Fallo en la lectura: Los valores no han cambiado desde la última lectura
09-Jul-21 12:14:45 - [INFO] - Voy a intentar limpiar este desastre (Thread Leer)
09-Jul-21 12:14:45 - [INFO] - Thread Leer cerrado..
09-Jul-21 12:14:45 - [INFO] - Programa Terminado
PS C:\Users\juanm\Dropbox (CIEMAT)\TFM>
```

Figura 5.9. Estrategia de gestión de lecturas duplicadas en API en funcionamiento

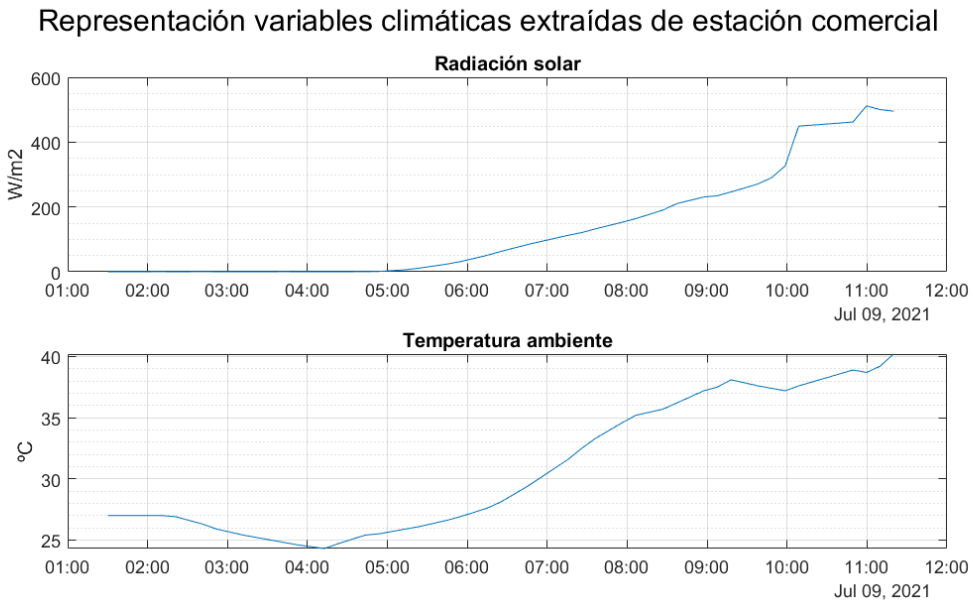


Figura 5.10. Representación de variables extraídas de una estación comercial durante una noche.

en la Figura 5.8. Se demuestra por tanto el correcto funcionamiento de la definición de nuevos tipos. Otra funcionalidad comentada en el apartado 4.2.3 es que si el tiempo de muestreo no es lo suficientemente grande, cabe la posibilidad de que se intente leer dos veces el mismo valor. Cuando esto ocurre, la estrategia consiste en descartar la lectura y esperar un tiempo menor que el de muestreo normal (más de 10 min) para realizar otro intento, como se aprecia en la Figura 5.9. Finalmente, se dejó el sistema recolectando información durante una noche de manera que con *MATLAB* y accediendo a la base de datos se ha podido realizar la representación mostrada en la Figura 5.10. Se puede apreciar la evolución de varias variables como la radiación solar y la temperatura ambiente. Como cabe esperar la radiación es nula durante la noche y crece por la mañana. De manera similar, la temperatura es menor por la noche y alcanza casi 40 °C al mediodía. El siguiente paso se efectuó para integrar *MATLAB* en esta arquitectura y poder tanto controlar como monitorizar los dos sistemas de riego. Desde la perspectiva de *MATLAB*, la manera en la que se comuniquen los PLCs será transparente y sólo deberá gestionar la conexión al servidor y extracción de las variables de interés. Para esto último, el proceso es bastante sencillo, tan sólo es necesario disponer del espacio de nodos del servidor, y después, mediante los nombres de nodo, se puede llegar a la variable de interés:

Código 5.3. Ejemplo de cómo acceder a variables de interés a través de la jerarquía del protocolo OPC

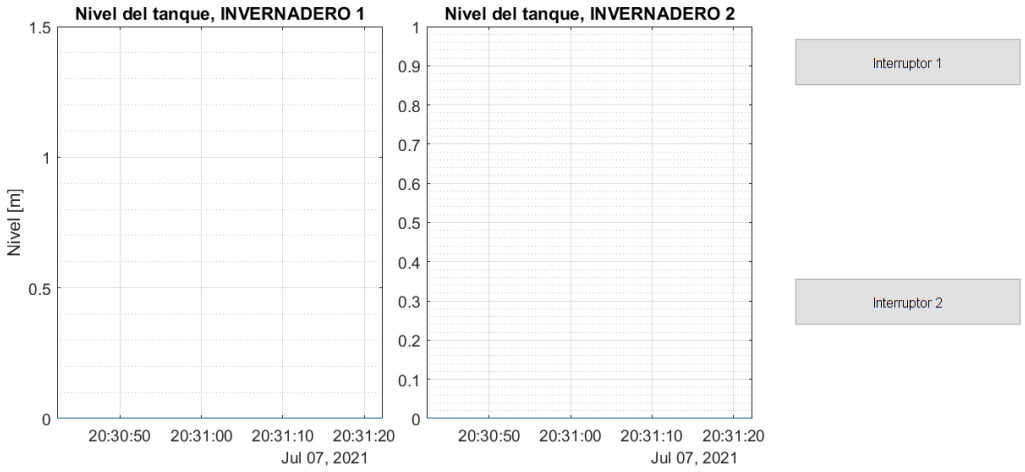
```
%   Root
%   |-- Objects
%               |-- M241_C2
%                           |-- Nivel
%                           |-- I
%                           |-- etc
%               |-- M241_C3
%               |-- Server
%   |-- Types
%   |-- Views

Nodos = uaClient.getNamespace;
nodo_cliente = Nodos.findNodeByName('M241');
Nodo_Nivel = nodo_cliente.findNodeByName('Nivel');

Nivel = Nodo_Nivel.readValue;
```

No se presenta el código de la aplicación desarrollada en *MATLAB* pues se ha realizado a modo de demostración para probar el funcionamiento. Lo interesante es el resultado mostrado en la Figura 5.11.

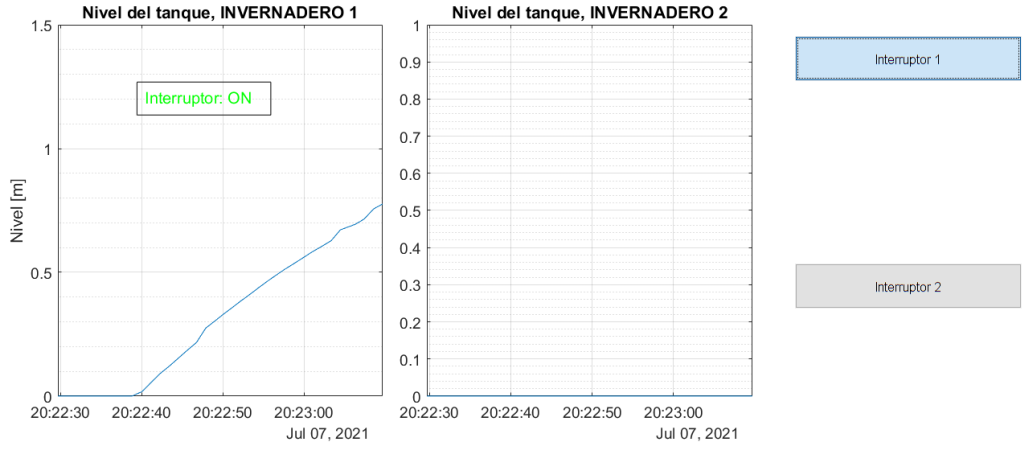
En la Figura 5.11a se puede apreciar ambos sistemas de riego antes de comenzar el llenado, en la Figura 5.11b el proceso de llenado de uno de ellos tras pulsar el interruptor,

y en la Figura 5.11c, mientras que el primero comienza el riego se inicia el llenado del segundo. Finalmente, en la Figura 5.11d se muestra cómo el primero tras finalizar no reinicia el proceso, pues se ha desactivado el interruptor mientras que el segundo lo repite.

Por último, se desea demostrar la correcta generación y funcionamiento de la base de datos, importándose en *MATLAB* para su posterior representación. El proceso de importar los datos se explica en el apartado 5.5.2, por lo que aquí sólo se mostrará el resultado obtenido, el cual se puede apreciar en la Figura 5.12. En la gráfica superior se muestra la evolución dinámica de ambos sistemas para uno o pocos ciclos de llenado. En contraste, en la gráfica inferior se representa la evolución del sistema para cada sistema de riego durante un período superior con muchos ciclos de llenado-regado, apreciándose cómo el proceso se conecta y desconecta en distintos momentos.

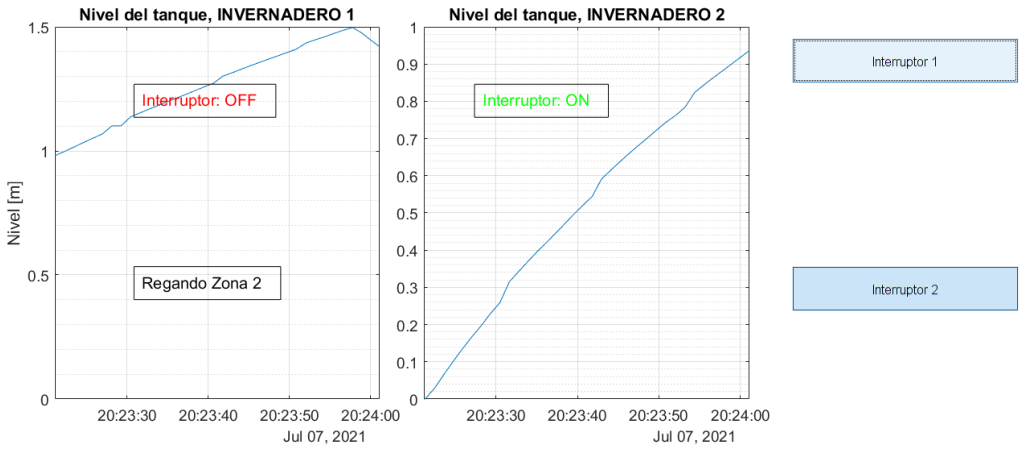


(a) Estado inicial

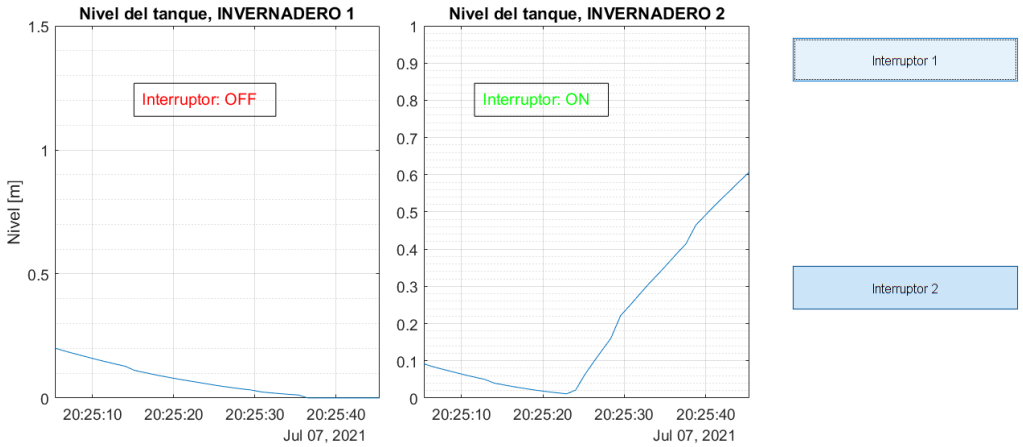


(b) Llenado de tanque 1

Figura 5.11. Evolución de dos sistemas de riego



(c) Vaciado de tanque 1 y llenado de 2



(d) Primer sistema parado, segundo en funcionamiento

Figura 5.11. Evolución de dos sistemas de riego

5.3 Integración de medidas de un invernadero industrial (medidores de magnitudes eléctricas y caudalímetro)

En el centro de Instituto Andaluz de Investigación y Formación Agraria, Pesquera, Alimentaria y de la Producción Ecológica (IFAPA) situado en La Mojonera [4], con el objetivo de aumentar el rendimiento de los cultivos intensivos, ha surgido la necesidad de analizar el consumo del proceso productivo. En concreto, se monitoriza un invernadero industrial de 1.120 m^2. Este dispone de distintos sistemas:

1. Sistema de calefacción por red de rubo radiante a baja temperatura.
2. Sistema de captura de CO_2.

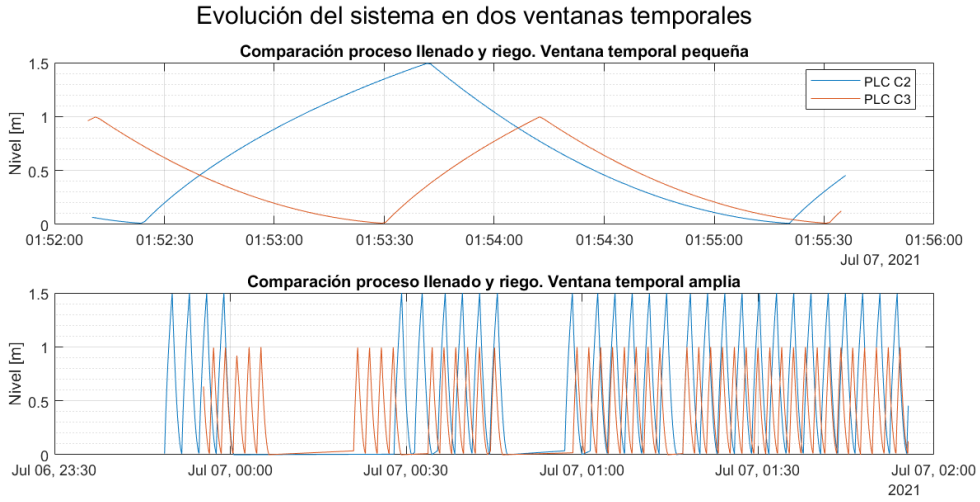


Figura 5.12. Resultado de acceder a la base de datos desde *MATLAB* y representar la evolución del nivel para dos ventanas temporales.

3. Caldera de biomasa.
4. Sistema de riego.
5. Instalación eléctrica.
6. Instalación meteorológica.

A modo de demostración de las capacidades del trabajo realizado, se han integrado cinco medidores de magnitudes eléctricas, pertenecientes a las distintas instalaciones mencionadas. Estos medidores (denominados *PowerTags*, son unos dispositivos fabricados por la empresa *Schneider Electric* que se colocan en un cuadro eléctrico y realizan lecturas de las distintas variables eléctricas de la instalación a la que se conectan. Cada uno de los *PowerTag* registra medidas y es capaz de mostrarlas mediante un servicio web o darles salida por *ModbusTCP*. Además, se dispone de una bomba con caudalímetro integrado y se usa para medir el caudal de riego. Este se comunica por medio de *ModbusRTU*. Un diagrama de este segundo caso se muestra en la Figura 5.13. Con la aplicación del desarrollo de este trabajo se integrarán todos los elementos de la figura 5.13 menos el *software Hortimax*. Para ello, en `ordenador A` se ejecutarán instancias de `cliente genérico` para cada uno de los dispositivos, así como una instancia del programa `servidor`.

Tras la puesta en marcha de la arquitectura desarrollada (ver Figura 5.14), se pueden acceder a los datos del sistema en tiempo real con algún cliente como se muestra en la Figura 5.15 usando el cliente *UAExpert*. En la Figura 5.16 se muestra una captura del cliente para conectarse a la base de datos generada, en la que se pueden apreciar las

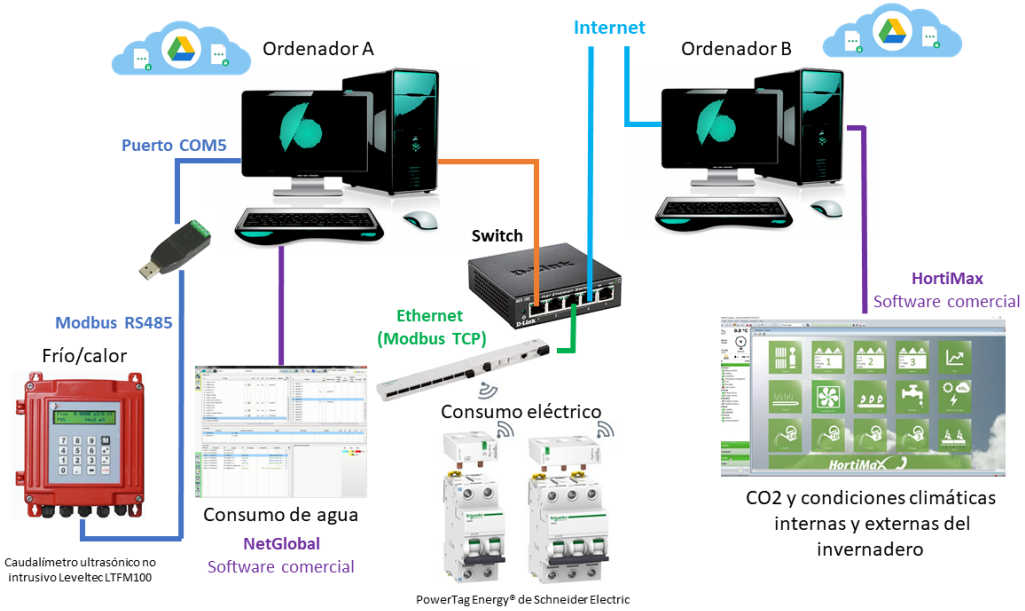


Figura 5.13. Esquema de segundo caso de aplicación

```
05-Jul-21 11:38:16 - [INFO] - Archivo de configuración leído correctamente
05-Jul-21 11:38:16 - [INFO] - Configuración encontrada para Powertag_Agua en archivo de configuración None
05-Jul-21 11:38:16 - [INFO] - Comprobación configuración. La configuración parece ser correcta.
05-Jul-21 11:38:16 - [INFO] - Cliente creado, protocolo: ModbusTCP || IP: 172.20.168.82, puerto: 502, ID: 15
4
05-Jul-21 11:38:16 - [INFO] - Nuevo cliente importado en servidor OPC: Powertag_Agua
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "CorrienteFaseA"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "CorrienteFaseB"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "CorrienteFaseC"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "TensionAB"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "TensionBC"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "TensionCA"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "TensionAN"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "TensionBN"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "TensionCN"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "PotenciaActA"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "PotenciaActB"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "PotenciaActC"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "PotenciaTotal"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "fdp"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "EnergiaTotal"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "EnergiaParcial"
05-Jul-21 11:38:16 - [INFO] - Añadida nueva variable en cliente: "Powertag_Agua" -> "ValorReinicio"
```

Figura 5.14. *Log* del programa cliente genérico tras iniciarlo.

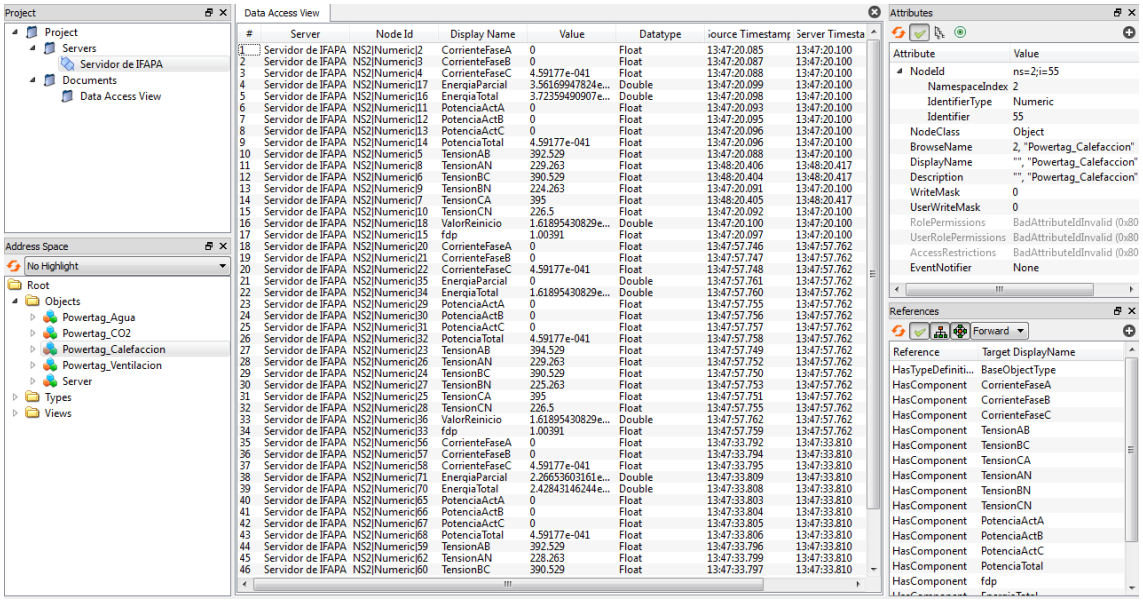


Figura 5.15. Cliente OPC mostrando los valores en tiempo real de muchas variables del sistema para las distintas instalaciones.

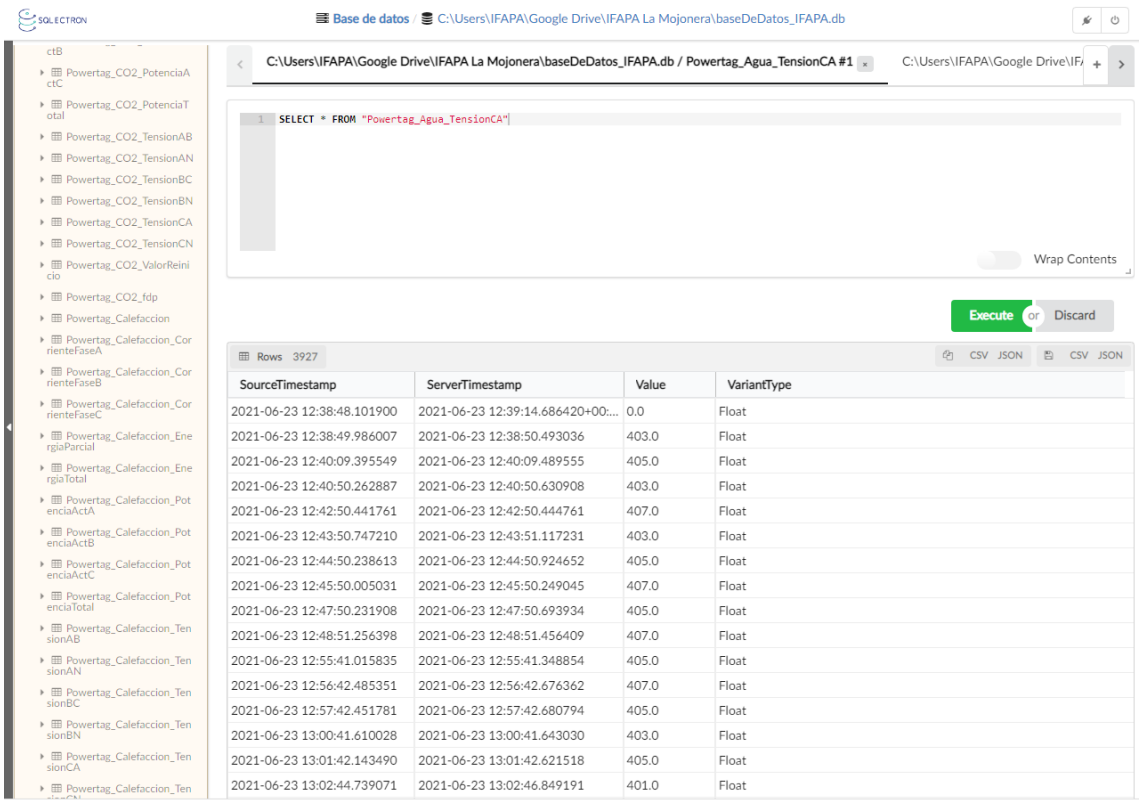


Figura 5.16. Base de datos tras la ejecución del programa cliente genérico para cada uno de los dispositivos, En concreto, se muestra el contenido del *PowerTag* de la instalación de agua, la variable tensión de línea CA.

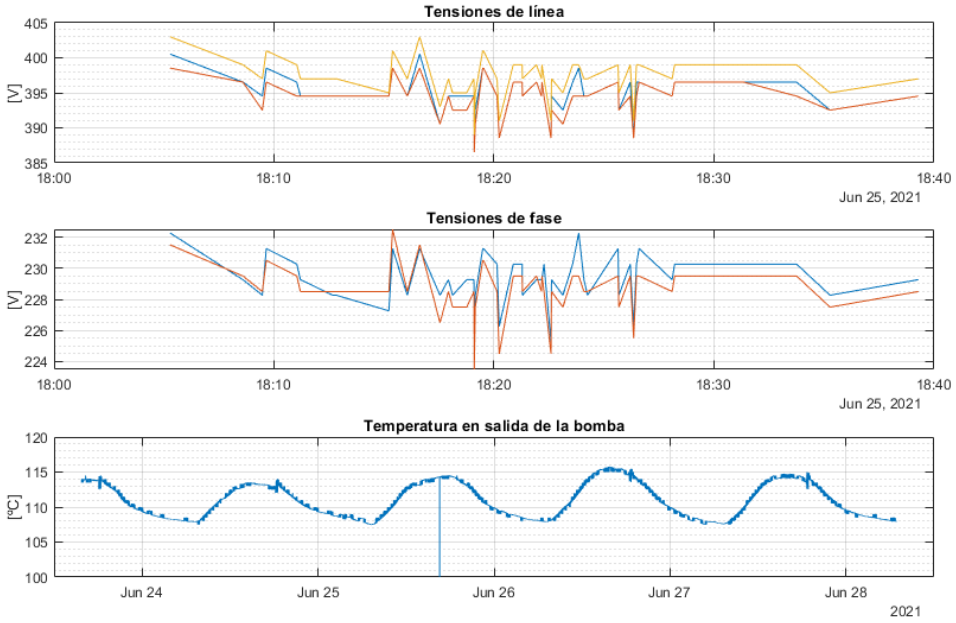


Figura 5.17. Representación gráfica de varias variables del sistema tras haber sido importadas desde la base de datos y tratadas en *MATLAB*.

tablas para las variables del sistema, así como una tabla que recupera 3927 medidas de la variable *Tensión CA*. Finalmente, haciendo uso de *MATLAB* y la plantilla explicada en el apartado 5.5.2, se ha realizado la conexión y extracción de datos de interés de la base de datos y generado la gráfica mostrada en la Figura 5.17. Esta representación muestra la evolución del valor de la tensión a lo largo del tiempo así como otras medidas de temperatura provenientes del caudalímetro.

5.4 Lectura y escritura de variables enteras y reales mediante *Modbus* en controladores lógicos

Antes de explicar el procedimiento concreto de cómo se ha logrado comunicar el valor de distintos tipos de variables mediante el protocolo *Modbus* en controladores lógicos industriales, en concreto el modelo M241 de *Schneider Electric*, es necesario explicar el tamaño de los distintos tipos de letra y las limitaciones del protocolo.

Mediante el protocolo *Modbus* se pueden enviar mensajes de tamaño 2 *bytes* (16 *bits*). Una variable de tipo entero requiere un tamaño de 2 también, mientras que una real es de 4 *bytes* (32 *bits*). El tipo real de doble precisión requiere de 8 *bytes* (64 *bits*). Esto se traslada a los PLC de la gama *Modicon* donde el espacio de registros se divide en variables

de tipo palabra (`WORD`), las cuales tienen el tamaño de 2 *bytes*. Posteriormente, es tarea del usuario transformar esta variable en el tipo de interés aplicando las transformaciones necesarias. En el caso de variables reales, puesto que un registro no es lo suficientemente grande para acoger el valor, se deben dividir en dos palabras, las cuales posteriormente desde el programa deben de agruparse correctamente.

Para aplicar las transformaciones es conveniente importar y hacer uso de librerías como *Oscat* [41]. Esta librería provee de ciertas funciones que facilitan mucho la tarea de realizar conversiones y decodificar distintos tipos de datos. Las variables que se pueden asociar a los distintos registros deben ser de tipo `WORD`, por lo tanto independientemente del número de registros se hace necesario para cada variable de interés crear una variable auxiliar de este tipo, por ejemplo, para una variable entera `I` de entrada, `V0` entera de salida, `level` real de salida y `variable` real de entrada:

Código 5.4. Programa en PLC. Definición de variables

```
PROGRAM POU
VAR
        // Definición de variables de entrada
        word_I: WORD;
        variable_real: REAL;
        word_real: WORD;
        word_real2: WORD;

        // Definición de variables de salida
        word_V0: WORD;
        dword_level: DWORD;
        word_level: WORD;
        word_level2: WORD;
END_VAR
```

Entonces, para las entradas se toma la variable auxiliar y se convierte al tipo adecuado. En el caso de varias palabras, se deben usar ambas variables en el orden adecuado y convertirlas en una variable tipo `DWORD` (*Double Word* o palabra doble) colocando el primer registro en la posición 1 y el segundo en la posición 0. Con esto se puede aprovechar otra función y transformar la variable tipo `DWORD` en la deseada `REAL`. Esto probablemente se deba a que `W1` es la palabra más significativa mientras que `W0` la menos significativa.

Código 5.5. Programa en PLC. Transformación de entradas

```
// Entradas
Evolucion.I := WORD_TO_BOOL(word_I);

// Entrada variable REAL
variable_real:=DW_TO_REAL(X:=DWORD_OF_WORD(W1:=word_real , W0:=word_real2));
```

Para el caso de las salidas, el proceso es el inverso; se transforma del tipo deseado `BOOL` o `INT` a `WORD`, o en el caso de variables más grandes se debe primero transformar la variable en una variable tipo `DWORD` para posteriormente disgregarla en dos. Al contrario que antes, la variable que ocupe la primera posición de registro lleva asociada la posición N=0, mientras que la segunda la posición N=1.

Código 5.6. Programa en PLC. Tranformación de salidas

```
// Salidas
word_V0 := BOOL_TO_WORD(Evolucion.V0);

// Salida REAL
dword_level := REAL_TO_DW(X:= Level);
word_level := WORD_OF_DWORD(in:=dword_level , N:= 0);
word_level2:= WORD_OF_DWORD(in:=dword_level , N:= 1);
```

Para poder hacer uso de estas funciones primero hay que instalar e importar la librería mencionada en *SoMachine* o el entorno de programación usado. El procedimiento es el que se muestra en la Figura 5.18. El archivo de librería se puede descargar desde la página oficial[1], y para instalarlo hay que seguir los pasos indicados en fig.5.18a.

A modo de resumen, en la Figura 5.19 se muestra un esquema de las transformaciones necesarias para la comunicación mediante el protocolo entre PLC y equipo.

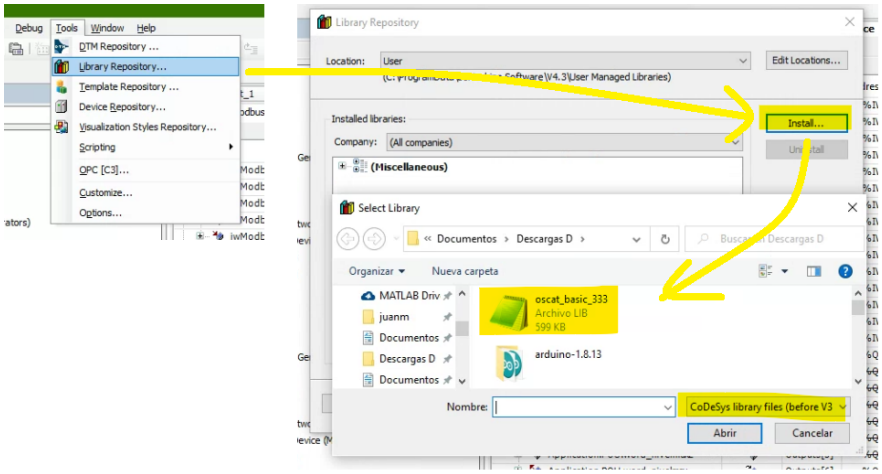
5.5 Plantilla de conexión y manipulación de variables usando *MATLAB* como cliente

Dado el amplio uso del *software MATLAB* dentro del entorno de investigación y puesto que este trabajo se ha planteado inicialmente para la integración de la instalación de un invernadero experimental con objeto de probar el desempeño de resultados de investigaciones, parece importante ofrecer la base para hacer uso de esta aplicación con las herramientas desarrolladas. Para ello, en dos apartados se explicará la interacción de *MATLAB* con el servidor, así como con la base de datos. Un esquema conceptual de la comunicación se muestra en la Figura 5.20.

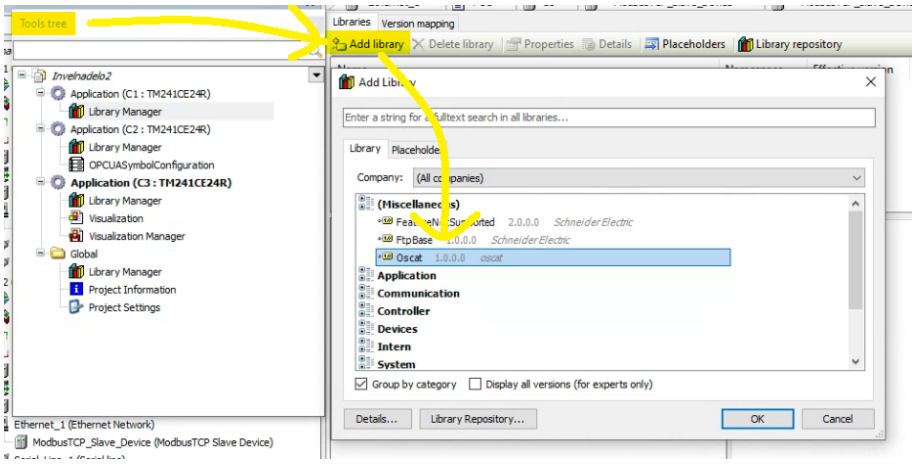
5.5.1 Conexión a servidor OPC

Como se ha mencionado anteriormente, se ha hecho uso del *OPC Toolbox*. Esta herramienta dispone de distintas instrucciones que cubren las necesidades de comunicación con la arquitectura propuesta. El primer paso es hacer uso de la función `opcuaserverinfo`

[1] `http://www.oscat.de/dlmanager/7-news/69-downloads.html`



(a) Proceso de instalación de la librería



(b) Proceso para importar la librería

Figura 5.18. Procedimiento para instalar e importar una librería en *SoMachine*

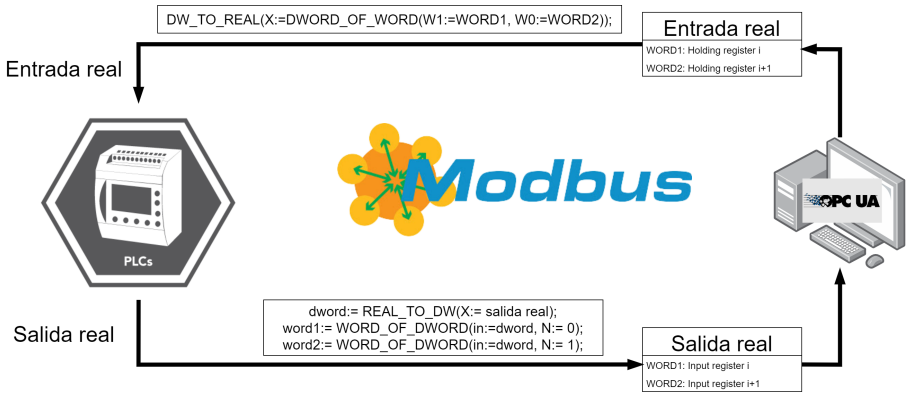


Figura 5.19. Esquema de la comunicación entre PLC y equipo usando *Modbus*

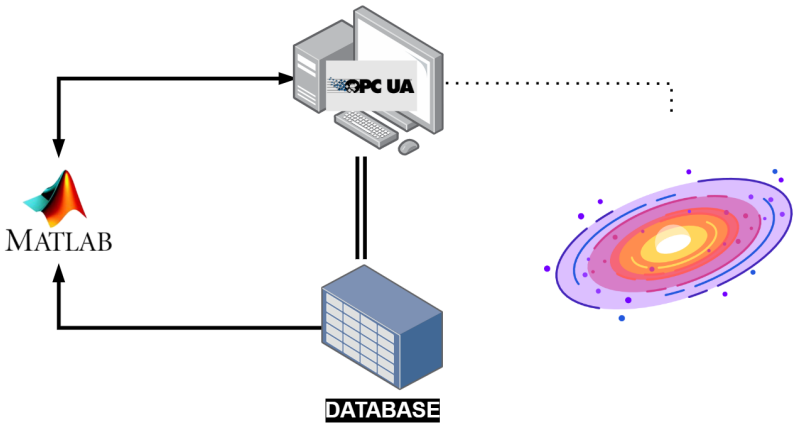


Figura 5.20. Esquema de comunicación *MATLAB* - dispositivos o servicios y base de datos

para obtener información del servidor: su política de seguridad, modos de autentificación, etc. Esta función se puede usar sin especificar la dirección de un servidor concreto y devuelve un vector de objetos con información de todos los servidores encontrados (ver código 5.7).

Código 5.7. *Script* de *MATLAB*. Conexión a servidor OPC

```
% Obtener información del servidor
s = opcuaserverinfo('opc.tcp://localhost:4841/SERVIDOR_OPC/');

% Crear objeto tipo opcua
uaClient = opcua(s);

% Establecer modelo de seguridad (autentificación y encriptación)
setSecurityModel(uaClient,'Best');
% setSecurityModel(uaClient, 'Sign', 'Basic256Sha256')

% Establecer conexión
% connect(uaClient);          % sin seguridad
connect(uaClient, 'administrador', 'admin') % acceso con usuario y
↪  contraseña

if isConnected(uaClient) == 1
        disp("Conectado correctamente al servidor");
else
        ME = MException('servidorOPC:conexionFallida', ...
        'Intento de conexión a servidor fallido');
        throw(ME)
end
```

Una vez conocida la política de seguridad, se establece y se procede a la conexión en este caso con usuario y contraseña. Se puede dar el caso en el que se encuentre el servidor pero falle el intento de conexión; para ello se añade el condicional del final del fragmento de código anterior donde se informa al usuario del resultado del intento.

El siguiente paso es encontrar los nodos de las variables de interés. En primer lugar se asigna a la variable `Nodos` el objeto que almacena la definición de todos los nodos en el servidor; a partir de este se pueden ir accediendo a los *hijos* de los distintos nodos.

Código 5.8. *Script* de *MATLAB*. Identificación de nodos de interés

```
Nodos = uaClient.getNamespace;
nodo_cliente = Nodos.findNodeByName('clienteModbus');
Nodo_Nivel = nodo_cliente.findNodeByName('Level');
Nodo_V0 = nodo_cliente.findNodeByName('V0');
```

Por ejemplo, en el extracto de código anterior se accede al nodo del cliente `ClienteModbus` y posteriormente se accede a determinadas variables de este: `Level` y `V0`. Estos nombres de nodo se pueden encontrar fácilmente haciendo uso de alguno de los clientes mencionados en el apartado 3.3. Se pueden usar alguno de los métodos disponibles y que se pueden encontrar usando `help opc.ua.Client`:

Código 5.9. Instrucciones soportadas por `OPC Toolbox` para interacción con nodos de variables

```
readValue          - Read current data values from the server
readHistory        - Read historical data from the server
readAtTime         - Read historical data at specified times from the server
readProcessed      - Read processed (aggregated) data from the server
writeValue         - Write current values to the server
getNodeAttributes - Retrieve node attributes
```

5.5.2 Conexión a base de datos

De manera similar se presenta directamente a continuación el *script* completo para la plantilla propuesta para conexión y extracción de datos de base de datos implementada en *SQLite* haciendo uso de *MATLAB*. Existen dos modos de hacerlo [42]. La primera es una manera simple que no requiere ningún *driver*, pero tiene ciertas limitaciones. La segunda requiere hacer uso de un *driver* pero habilita el uso del *Database Toolbox*, el cual provee una herramienta gráfica para interactuar con una base de datos[2] además de ser más completa.

[2] `https://www.mathworks.com/videos/using-the-database-explorer-app-71881.html`

5.5.2.1 Versión simple

Para la primera aproximación, el proceso es bastante sencillo, se ha basado en [43]. Tan sólo se debe indicar dónde se halla el archivo base de datos (`.db`) generado por el programa y establecer una conexión. Acto seguido, se realizan las solicitudes de los distintos datos de interés, mostrándose algunos ejemplos comentados en el extracto de código 5.10. Cabe mencionar que la implementación de *MATLAB* requiere tener cuidado con la gestión de recursos, ya que si se produce algún error durante la solicitud o el usuario no realiza correctamente el cierre, no se ejecutará el cierre de la conexión y por lo tanto permanecerá abierta, bloqueando la base de datos como se ha mencionado en el apartado 4.5.5. Es por ello que se introducen las intrucciones en un bloque `try - catch` y en caso de que se produzca el error se cierra la conexión y se vuelve a lanzar. El código completo se muestra a continuación:

Código 5.10. Versión simple. Conexión y extracción datos de base de datos.

```
clc
clear all
close all
%% Ejemplo de cómo importar datos de una base de datos

% Conexión a base de datos
path = 'C:\Users\juanm\Dropbox\TFM'; % una ruta cualquiera

dbfile = fullfile(path,'baseDeDatos.db');

% Leer datos de tablas, se recomienda usar algún cliente gráfico para
% explorar la base de datos y ver comandos de cómo realizar operaciones
% Para Windows, Linux: Sqlectron, https://sqlectron.github.io/
% Para Mac: psequel, http://www.psequel.com/ o Sqlectron

% Importante hacer así, si no se usa conn.close() MATLAB bloquea la base de
% datos y no se puede ser actualizada por el servidor OPC

try
        conn = sqlite(dbfile, 'readonly');
        %sqlquery = 'SELECT * FROM clienteModbus_VO where ServerTimestamp
        ↪  IS NOT NULL';
        %sqlquery = 'SELECT * FROM clienteModbus_VO LIMIT 1000';
        sqlquery = 'SELECT * FROM clienteModbus4';
        datos = fetch(conn, sqlquery);
        conn.close()
catch e
```

```
27          conn.close()
28          rethrow(e)
29  end
30
```

Cabe recordar la limitación de no poder extraer los nombres de columna junto con sus valores como se ha mencionado en el apartado 4.5.5. Además, de esta manera se obtienen variables tipo célula, siendo necesario tratar los datos posteriormente realizando las transformaciones necesarias.

5.5.2.2 Versión completa

La segunda manera de proceder se basa en [44] y [45]. Este método requiere la instalación de un driver JDBC (*Java Database Connectivity*). La documentación de *MATLAB* recomienda el uso de un *driver* mantenido y publicado por Taro L. Saito en un repositorio de *GitHub*[3]. Todo el proceso se puede realizar haciendo uso de la herramienta *Database Explorer* (ver fig.5.21), pero en esta memoria se explica el *script* desarrollado para poder importar información de manera programática. En este ejemplo se extraerán las lecturas de caudal de un caudalímetro accediendo a la tabla adecuada dentro de la base de datos `IFAPA`.

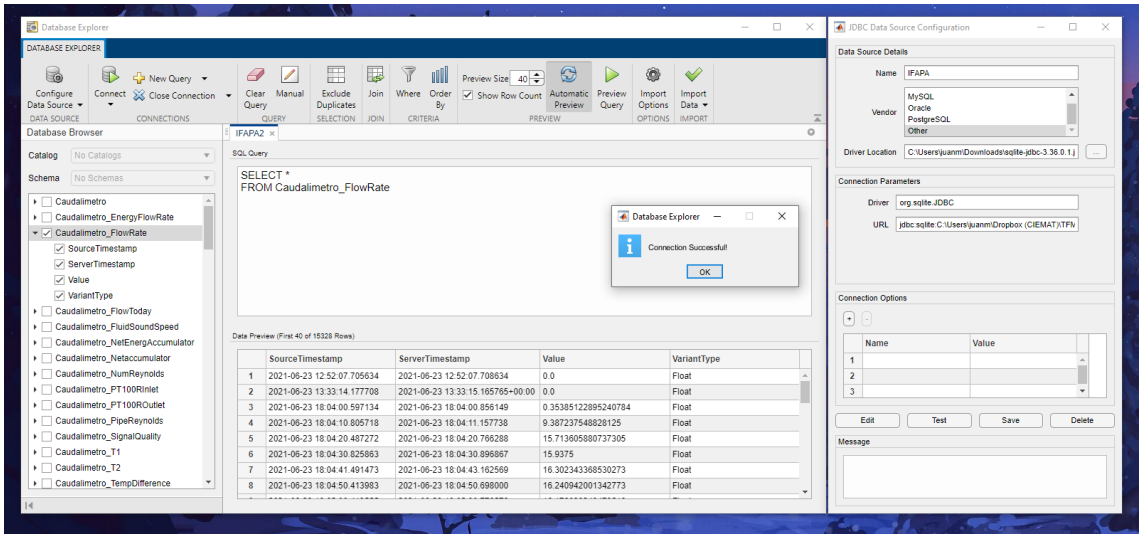


Figura 5.21. Ejemplo de configuración de conexión y solicitud de datos usando *Database Toolbox*

El primer paso es configurar la conexión, para lo que se deben establecer parámetros como la ubicación del driver, el protocolo a usar y especificar el directorio donde se en-

[3] `https://github.com/xerial/sqlite-jdbc`

cuentra el archivo de base de datos:

Código 5.11. Versión con *driver*. Conexión

```
%% Configuración de la conexión
javaaddpath('sqlite-jdbc-3.36.0.1.jar');
dbpath = 'C:\Users\juanm\Dropbox (CIEMAT)\TFM\baseDeDatos_IFAPA.db';

user = '';
password = '';
driver = 'org.sqlite.JDBC';
protocol = 'jdbc';
subprotocol = 'sqlite';
resource = dbpath;
url = strjoin({protocol, subprotocol, resource}, ':');

%% Conexión
conn = database(dbpath, user, password, driver, url);
```

A continuación, se realiza la solicitud especificando la tabla y variables a leer. A la hora de importar la lectura al espacio de trabajo de *MATLAB* se pueden aplicar ciertas transformaciones. Concretamente, se puede directamente transformar las entradas de tipo marca de tiempo (que se leen como cadenas de caracteres) en variables tipo `datetime`. Además, se puede establecer la política de qué hacer con las medidas inválidas.

Código 5.12. Versión con *driver*. Extracción de datos

```
try
        query = ['SELECT * ' ...
        'FROM Caudalimetro_FlowRate'];

        % Configuración de los datos a importar
        opts = databaseImportOptions(conn,query);
        opts =
        ↪  setoptions(opts,{'SourceTimestamp','ServerTimestamp','Value'},
        'Type',{'datetime','datetime','string'});
        opts = setoptions(opts,{'Value'},'FillValue',{"NaN"});

        data = fetch(conn,query,opts);
        conn.close()
catch e
        % Cerrar conexión en caso de error para no bloquear base de datos
        conn.close()
        rethrow(e)
end

data.Value = str2double(data.Value);
```

Como se ha hecho para el caso anterior, se previene el no cierre correcto de la conexión en caso de que ocurra algún error en la solicitud. Finalmente, ya que no existe la opción de hacerlo al importar los datos se transforma el campo `Values` manualmente a una variable numérica. A modo de ejemplo como se ha mencionado anteriormente se ha leído el valor de caudal de un sensor de la base de datos y se ha representado en la Figura 5.22.

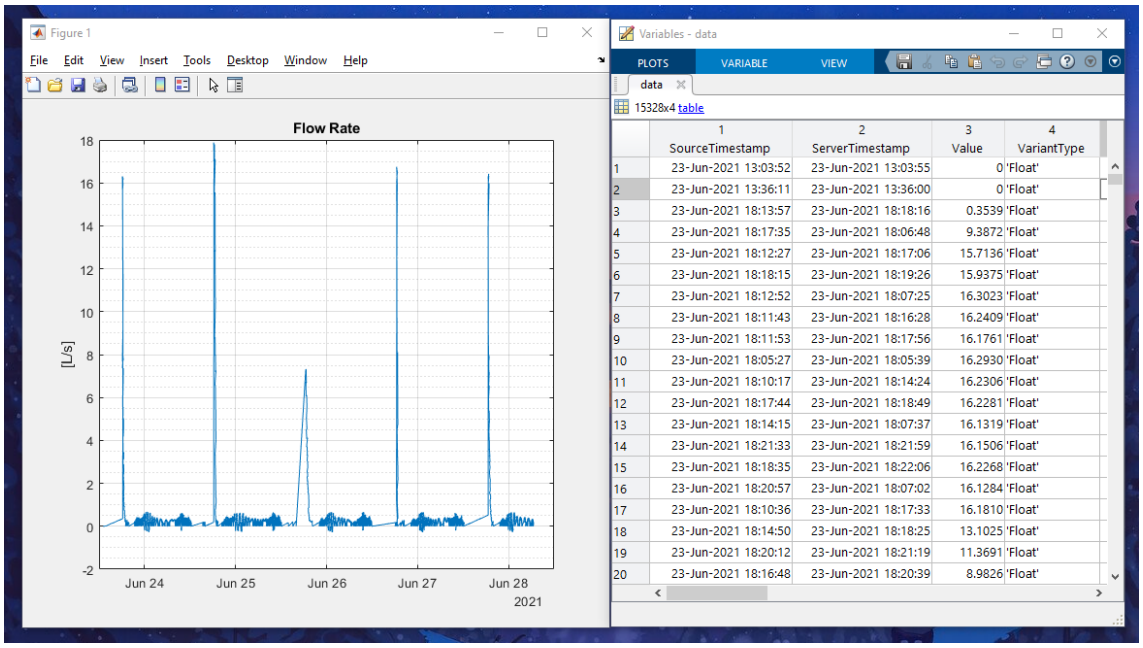


Figura 5.22. Resultado de importar información de base de datos usando *driver*

5.6 Presupuesto

Para terminar el capítulo de resultados se presenta un presupuesto en el que se desgranan y detallan los costes asociados a este proyecto. Al no tratarse de un proyecto tradicional de ingeniería, los costes de materiales no contribuyen demasiado al coste total. Aún así existen y se pueden apreciar en la Tabla 5.1, donde se puede observar una de las ventajas de utilizar *software* de código abierto y es que no hay costes asociados a la adquisición de licencias. En las tablas sucesivas se disgregan los costes de los distintos aspectos del proyecto: diseño (Tabla 5.2), desarrollo (Tabla 5.3), entrega (Tabla 5.4), administración (Tabla 5.5) y otros costes (Tabla 5.6). Sumando todos los costes resulta un presupuesto total de quince mil novecientos quince euros, 19 915 €.

Tabla 5.1. Disgregación de costes de material

Coste de materiales	Descripción material	Cantidad	Precio unitario [€]	Total [€]
	Equipo donde realizar desarrollo	1	1500	1500
	PLC Modicon M241	2	300	600
	SUBTOTAL			2100 euros

Tabla 5.2. Disgregación de costes de diseño del proyecto

	Tareas	Horas mano de obra	Coste mano de obra [€]	Otros costes [€]	Total tarea [€]
Diseño del proyecto	Desarrollar especificaciones funcionales	10	10	0	100
	Desarrollar arquitectura del sistema	30	10	0	300
	Desarrollar especificaciones de diseño preliminares	10	10	0	100
	Desarrollar especificaciones de diseño detalladas	10	10	0	100
	SUBTOTAL	60 horas		0	600 euros

Tabla 5.3. Disgregación de costes de desarrollo del proyecto

	Tareas	Horas mano de obra	Coste mano de obra [€]	Otros costes [€]	Total tarea [€]
Desarrollo del proyecto	Desarrollar componentes principales	700	10	0	7000
	Desarrollar componentes auxiliares	250	10	0	2500
	Desarrollar pruebas de funcionamiento	50	10	0	500
	Realizar pruebas de funcionamiento	30	20	0	600
	SUBTOTAL	1030 horas		0	10600 euros

Tabla 5.4. Disgregación de costes de entrega del proyecto

Entrega del proyecto	Tareas	Horas mano de obra	Coste mano de obra [€]	Otros costes [€]	Total tarea [€]
	Instalar el sistema	15	10	0	150
	Entrenar a los clientes	5	10	0	50
	Realizar prueba de aceptación	5	10	0	50
	Realizar revisión posterior al proyecto	3	10	0	30
	Proporcionar soporte técnico bajo garantía	50	10	0	500
	Archivar material	2	10	0	20
	SUBTOTAL	80 horas		0	800 euros

Tabla 5.5. Disgregación de costes de administración del proyecto

Administración del proyecto	Tareas	Horas mano de obra	Coste mano de obra [€]	Otros costes [€]	Total tarea [€]
	Reuniones/informes de progreso con el cliente	5	0	0	150
	Administración de la configuración	10	0	0	50
	Control de calidad	0	0	75	50
	Administrador global del proyecto	0	0	100	20
	SUBTOTAL	15 horas		175	175 euros

Tabla 5.6. Disgregación de costes de otros costes.

Otros costes	**Tareas**	**Horas mano de obra**	**Coste mano de obra [€]**	**Otros costes [€]**	**Total tarea [€]**
	Licencias de programas	-	-	-	-
	Desplazamiento	-	-	50	50
	Costes de oficina	-	-	1500	1500
	Otros costes (contratación tarifa internet, etc.)	-	-	90	90
	SUBTOTAL	-	-	1640 euros	1640 euros

Capítulo 6

Conclusiones

Presentados los resultados de la aplicación práctica de la arquitectura propuesta, se presentan en este capítulo final de la memoria las conclusiones alcanzado el término de este trabajo. Asimismo, se enumeran trabajos pendientes que por diferentes motivos (necesidad de establecer objetivos alcanzables dadas las restricciones de tiempo disponible, trabajo realizable razonable para un trabajo de fin de estudios, etc.) han quedado pendientes de implementar. También se presentan otras propuestas de ampliación que resultan de interesante aplicación para mejorar las características y utilidad de la arquitectura implementada.

6.1 Conclusiones

Para comenzar las conclusiones, es de especial interés comparar los resultados alcanzados con los objetivos inicialmente propuestos. Como se propuso, se ha diseñado una arquitectura que permite la integración de servicios y dispositivos industriales, y gracias a una serie de casos de aplicación, se ha podido demostrar su flexibilidad para agrupar información de formatos y tipos distintos, así como de una variedad heterogénea de dispositivos: sensores, PLCs, estaciones meteorológicas, etc. Todas las herramientas usadas son abiertas, de uso libre y gratuitas, tanto a nivel de cliente, servidor y todos los servicios auxiliares utilizados (base de datos). Es una excepción la programación de los controladores lógicos, para la que se ha usado *software* comercial, pero no es objeto de este trabajo ofrecer una alternativa a esto.

Se ha programado un servidor OPC en base a librerías existentes para el lenguaje de programación *Python*. Para ello, se ha comprendido su funcionamiento, y se han realizado

modificaciones en el código de la misma para adaptarla a las necesidades encontradas, así como integrarla en un programa con el resto de características implementadas. Al servidor se le han implementado mecanismos de seguridad en el proceso de autentificación, con la existencia de perfiles de acceso, así como en la seguridad de los mensajes, estando estos encriptados. Una ventaja de haber usado una librería abierta es que todo es configurable según las necesidades de la arquitectura, lo que permite que los programas desarrollados en el presente trabajo posean un carácter flexible, facilitando futuras modificaciones.

En cuanto al programa para la interacción con servicios y dispositivos, se ha resuelto satisfactoriamente el soporte para el protocolo *Modbus*, así como para las APIs de estación meteorológica. La única tarea de un usuario final es configurar la conexión y los parámetros necesarios en un archivo de configuración. Como el programa tiene una arquitectura modular, el soporte a nuevos protocolos (*EthernetIP*, API de Agencia Estatal de Meteorología (AEMET), etc.) sólo requiere generar una nueva clase y añadir la llamada a los métodos siguiendo el ejemplo de los protocolos ya implementados.

El último objetivo principal era la validación del trabajo desarrollado en dos casos de estudio. El desempeño del trabajo ha sido satisfactorio habiéndose obtenido los resultados esperados, es decir, se ha observado la habilidad de interactuar desde aplicaciones con los distintos sistemas en tiempo real, como disponer de un registro de los datos históricos de distintos sistemas. Los programas desarrollados se han dejado en funcionamiento durante semanas en el segundo caso de aplicación y se ha podido comprobar que funcionan de manera robusta.

En cuanto a los objetivos propuestos de manera complementaria a los principales de este trabajo, queda cubierta la generación de la base de datos, habiéndose visto necesario resolver esta necesidad en el campo de aplicación de este desarrollo. Se han creado plantillas para la interacción entre *MATLAB* y servidor, así como con bases de datos. También se han realizado plantillas para la comunicación vía *Modbus* con controladores lógicos supliéndose, limitaciones de este protocolo.

Asimismo, se pueden extraer ciertas conclusiones a nivel técnico. Frente a la existencia de multitud de protocolos para comunicación a nivel de campo con las particularidades de cada uno, sus limitaciones y sus procedimientos específicos para cada dispositivo, es de agradecer la existencia de *middleware* como OPC, en concreto en su variante UA. Se trata de un estándar soportado y reconocido por la mayoría de fabricantes y aplicaciones comerciales. Debido a su carácter abierto, existen multitud de librerías en la mayoría de lenguajes de programación populares. Esto ha permitido simplificar extensamente la capa *fog* de la arquitectura, con versatilidad incluso para eliminar la necesidad de esta capa en determinadas circunstancias. Por otra parte, es necesario reconocer la importancia del lenguaje *Python* por haber permitido este trabajo. *Python* tiene desventajas conocidas como su peor rendimiento frente a lenguajes como C o C++, pero hay que reconocer que

es un lenguaje accesible, con soporte para multitud de aplicaciones con miles de librerías, una comunidad activa y extensa así como material didáctico disponible para cualquier aspecto de su implementación.

Otra aspecto a valorar de la labor realizada es el hecho de que este trabajo está probado en condiciones reales de funcionamiento. Muchos de los fallos propios de implementación ya han sido identificados durante la puesta en marcha y corregidos; de igual forma que posibles mejoras también han sido identificadas y se comentarán en los siguientes párrafos como propuestas para trabajos futuros. Esto hace de este proyecto no únicamente un trabajo académico, si no un desarrollo con potencial utilidad en el contexto de aplicación a instalaciones industriales de distinta escala (véase el apartado 1.3).

En cuanto al presupuesto, cabe reflexionar sobre los elementos que componen este y en qué magnitud, así como comparar la solución propuesta con alternativas comerciales. Si se analiza el apartado 5.6 se puede ver que el principal coste es el asociado al desarrollo del proyecto. Esto es debido a que para la programación de todos los componentes que forman la arquitectura se han requerido una cantidad importante de horas de ingeniero. En concreto han sido necesarias más de 900 horas. Suponiendo una jornada laboral de 8 $h/jornada$ equivale a más de 100 días o algo más de 4 meses de trabajo. También es interesante observar que al tratarse de un proyecto *software* el coste asociado a materiales es inapreciable comparado con el resto, únicamente el de equipo que podría no existir si se proveyera, así como el los controladores lógicos pues son equipos que sólo se requieren temporalmente durante el desarrollo para realización de pruebas.

Es difícil comparar el precio del desarrollo con soluciones comerciales pues el mayor aportador al coste es el desarrollo del *software*. Este debería dividirse en los distintos proyectos de aplicación, con lo que con el tiempo y el número de aplicaciones el coste asociado se distribuiría. Por ejemplo, si sólo se usa en un único proyecto supondría un coste algo inferior a 20 000 €, mientras que si se aplica en 5 tendría un coste por proyecto de en torno a 4000 €. Otra dificultad es que en general fabricantes no hacen públicos sus precios y sólo responden a consultas. Esto, sumado a la tendencia de cambiar el paradigma de producto con un coste único, al de servicio que tiene un coste de suscripción que se prolonga indefinidamente en el tiempo. Esto hace que el coste vaya asociado al tiempo de explotación, haciéndolo difícil de establecer. Se han encontrado precios para soluciones HMI en el catálogo de *Schneider* [46], donde únicamente para el *software* el precio oscila entre 2500 y 5000 €. Un fabricante que sí provee de un catálogo de precios para su *software* [47] se sitúa entre 10 000 y 20 000 €. El coste de uso del *software* desarrollado es nulo ya que no tiene asociado ningún tipo de licencia o dependencia de *software* comercial.

Estudiados en la medida de lo posible los precios de *software* comercial, no parece precipitado suponer que en un período de tiempo corto la rentabilidad sería mayor con la solución propuesta. Siendo este ahorro más notable cuanto mayor sea el tiempo de

explotación. Si este es suficientemente grande el ahorro puede llegar a suponer una cantidad muy importante, teniendo en cuenta siempre que la alternativa propuesta también conllevará costes menores de mantenimiento o mejora.

Finalmente, se mencionan algunas conclusiones de carácter personal. Uno de los objetivos personales principales que se se propuso el autor de este trabajo es que el producto de este proyecto no sea únicamente una herramienta para cumplir los requisitos docentes. Hay un deseo de que realmente se encuentre utilidad en la aplicación del trabajo desarrollado en aplicaciones industriales. Probablemente si esto sucede, con la variedad de casuísticas y el escalado del uso ocurran imprevistos o retos a los que el autor ofrece su compromiso a colaborar en solventar. Como se aprecia en el apartado de planificación temporal (1.5) el inicio del trabajo de desarrollo se da por la necesidad de solventar un problema con la comunicación de una configuración experimental en el invernadero de IFAPA de La Mojonera. Puesto que la arquitectura se ha desarrollado y probado ya en condiciones reales durante meses (al menos una versión primitiva o *beta*) se intuye así su potencial utilidad y capacidad para satisfacer las necesidades de comunicación con servicios y protocolos industriales, al menos a pequeña escala.

Como se ha mencionado anteriormente, otra clave de este trabajo es que está completamente basado en herramientas y librerías de código abierto. Esto permite que tenga un coste de explotación asociado nulo, tan sólo el coste energético del equipo donde se ejecute y la amortización del mismo.

Este proyecto se ha desarrollado partiendo de conocimientos limitados de programación de alto nivel, siendo la procedencia del autor el grado de *Ingeniería Electrónica Industrial* en el que tan sólo se cursa una asignatura de programación y algo más en asignaturas de control. Con esto se quiere expresar que otro alumno o programador podría partir de este trabajo y aprender fácilmente su funcionamiento, o usarse como apoyo para el desarrollo de otro tipo de aplicaciones similares. Para ello, se ha intentado documentar de la mejor manera posible todo el código y conforme se han aprendido nuevas técnicas para realizar distintas tareas de manera más eficiente se ha revisado el código anterior para incluirlas.

6.2 Trabajos futuros

Algo que suele ocurrir en trabajos de desarrollo de aplicaciones *software* es que la lista de características y nuevas funcionalidades a integrar en el desarrollo, o nuevas maneras de implementar funcionalidades actuales, siguen surgiendo durante toda la vida del proyecto, tratándose por tanto de un trabajo dinámico en constante evolución a lo largo del tiempo. A día de redacción de este capítulo de la memoria se ha ejecutado lo mencionado en los apartados 1.4 y 5, pero es intención del autor de este trabajo a lo largo del tiempo mejorarlo si se encuentra aplicación e interés en el uso del trabajo realizado. En concreto,

se plantean las siguientes ampliaciones:

1. Portar la base de datos de *SQLite* a *PostgreSQL*. Esta tarea no debería suponer demasiado trabajo pues hay una gran compatibilidad entre ambas implementaciones de bases de datos [48].

2. Acceso y lectura de variables climáticas haciendo uso de la API de AEMET.

3. Aprovechar la posibilidad de ciertos modelos de PLC Modicon (M241, p. ej.) de generar un servidor OPC desde el propio PLC para integrarlos en la arquitectura. Para ello será necesario en la librería de clientes crear una nueva clase para este tipo de PLC y gestionar la conexión y comunicación con el servidor creado desde el PLC. De esta manera se puede facilitar la gestión de las variables del PLC al no tener que lidiar con números de registros, transformación de variables, separación de variables en varios registros, etc. Además que se supera también el límite en el tamaño del número de registros totales disponibles.

4. Otra variante de *Modbus* es la comunicación de *ModbusRTU* a través de TCP. Hay dispositivos que se comunican de esta manera y la librería usada para dar soporte a *Modbus* [27] parece que lo ofrece. Es interesante entonces añadir una nueva clase en la librería de clientes y así dar soporte a un rango mayor de dispositivos.

5. Ha quedado pendiente dar soporte en este trabajo a dispositivos que se comunican por *EthernetIP*, este protocolo a diferencia de *Modbus* tiene variaciones en su implementación específica por los fabricantes. Haciendo uso de varias librerías [49] y [50] se ha logrado establecer comunicación con el PLC M241 pero no se ha logrado el intercambio de valores de variables. Debido a la necesidad de concentrar esfuerzos en terminar este trabajo en fecha de entrega, se ha dejado este protocolo como trabajo futuro.

6. Conforme el sistema crezca probablemente sería interesante separar base de datos y servidor en programas distintos; por defecto hacer uso de *PostgreSQL* y la generación de las tablas para variables con más campos debería de realizarse automáticamente sin necesidad de modificar el programa de base de datos.

7. Portar servidor y cliente a la versión *Asyncio* [51] de la librería usada. Esta versión de la librería implementa programación asíncrona [52], una programación más eficiente que tiene el potencial de simplificar el código y sobre todo mejora de rendimiento conforme los programas se escalan. Mientras que la librería usada en este TFM es perfectamente funcional y recibe mantenimiento, la versión *Asyncio* de la misma es donde se encuentran los esfuerzos actuales de desarrollo y presenta nuevas funcionalidades que no lo hace la usada actualmente.

8. Una de estas ventajas es una mejor implementación del sistema de gestión de usuarios con distintos niveles de acceso. Con la implementación actual sólo se tienen dos niveles de acceso: administrador y usuario. El administrador tiene acceso y permisos totales mientras que el usuario sólo permisos de lectura. Sería interesante tener un rango mayor de opciones, como hacer que ciertos usuarios tengan permiso de acceso y modificación a un grupo de variables, mientras que de manera restringida o sin acceso a otras.

9. Mejor implementación de la seguridad en los mensajes. Actualmente el cifrado de los mensajes del servidor funciona de la siguiente manera: en la solicitud de conexión el servidor y cliente intercambian sus claves públicas y automáticamente estos confían el uno en el otro. Esto combinado con la necesidad de iniciar sesión con usuario y contraseña ofrece un nivel de seguridad aceptable pues un atacante capturando los mensajes no podría entenderlos sin poseer las claves privadas. Siendo esto así la implementación debería ser como lo es para cualquier estándar como HTTPS, donde cada uno de los certificados, tanto servidor como cliente debería estar firmado por una entidad certificadora (la cual podría ser cualquier equipo de confianza configurado para ello). Cada nuevo cliente debería solicitar la firma de su certificado por la entidad, así como las distintas aplicaciones estar configuradas para confiar en certificados firmados por esta entidad. Con ello se lograría un nivel de seguridad superior, el de la mayoría de comunicaciones en la web.

Anexos

Anexos A

Estańdar OPC

La *OPC Foundation* ha hecho públicos a lo largo del tiempo varios estándares para uso en procesos industriales. El último miembro en llegar fue la versión UA [53]. Este integra las capacidades de los estándares anteriores: Acceso a datos (DA) para acceso a datos en tiempo real, Acceso a datos históricos (HDA) para acceso a datos históricos y Alarmas y Eventos (AE) para alarmas y eventos. Además incorpora nuevas funcionalidades, como la capacidad de acceder al histórico de eventos, la capacidad de definir métodos así como definición de nuevos modelos de información complejos (tipos de datos) de alto nivel, no simplemente hacer uso de los tipos de datos básicos (real, entero, etc). Otra característica interesante es la capacidad de identificar nodos por nombres comunes, haciendo más sencillo la gestión e interacción con el servidor.

La arquitectura de este estándar se puede apreciar en la Figura A.1. Se puede ver que sólo están estandarizados cliente y servidor. Estos se comunican entre ellos haciendo uso del *stack* de comunicación OPC UA a través de una API. Esta no está estandarizada y es única para cada implementación, aquí es donde entra el trabajo de la librería usada para este TFM (ver anexo B.1). Mientras que en la figura mencionada se aprecian fuentes de información que se comunican con el servidor, en este trabajo se han implementado estas fuentes como otro cliente más que gestiona y actualiza los valores de los nodos a los que atañe.

En la figura A.2 se puede apreciar la estructura del espacio de direcciones del servidor. En verde se pueden apreciar los nodos que forman parte del estándar y por lo tanto en cualquier implementación siempre son iguales y con los mismos identificadores. En el nodo *Objetcs* es donde se incluyen los nodos creados por la aplicación o el usuario. Según la estructura seguida en este TFM, el nodo *Boiler* de la imagen haría referencia a un dispositivo concreto y directamente en el siguiente escalón de la jerarquía se tendrían todas las variables de este. Otra opción (que se puede implementar *a posteriori*) es agrupar ciertos clientes en un nodo, y entonces realizar lo mencionado.

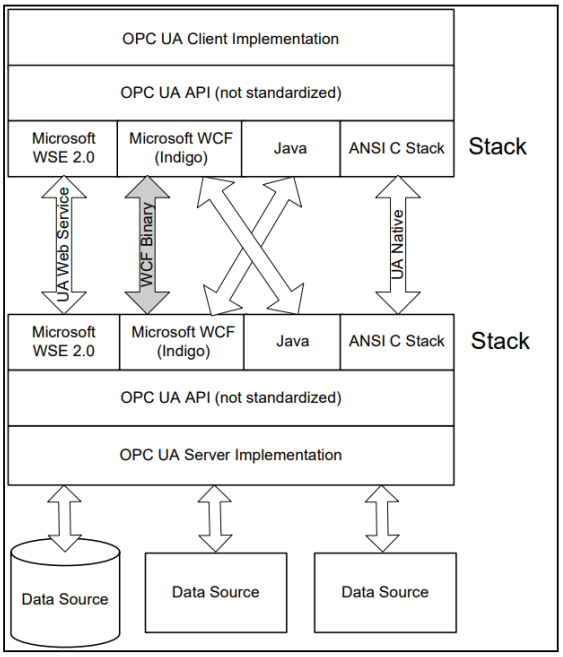


Figura A.1

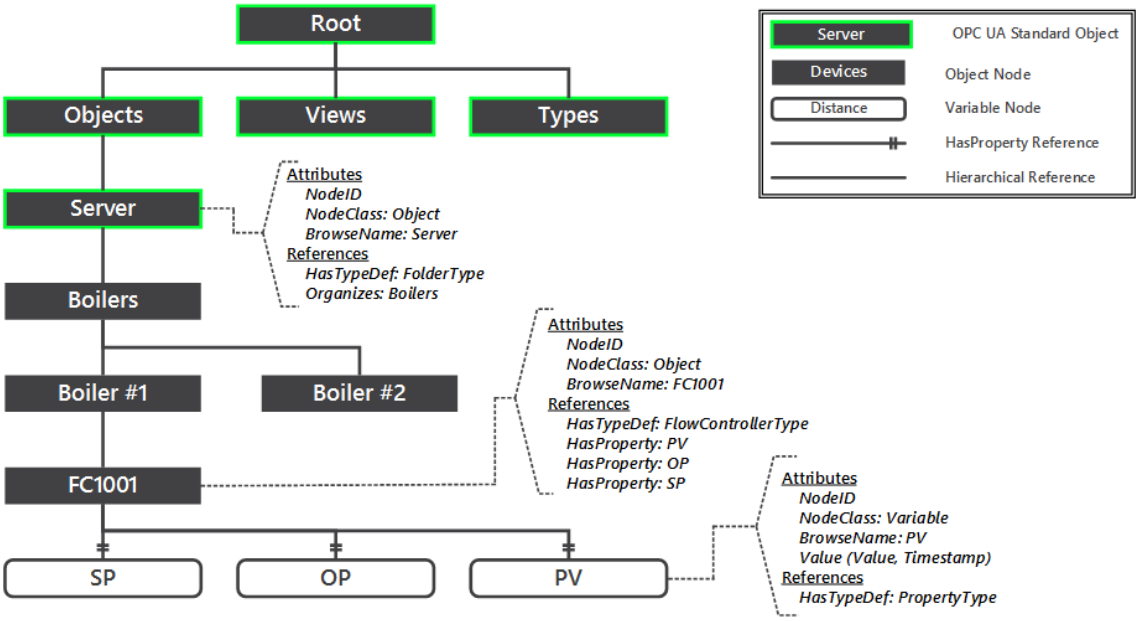


Figura A.2. Espacio de direcciones de OPC UA. Imagen de [54].

Anexos B

Librerías de Python

B.1 Servidor OPC. FreeOpcUa

FreeOpcUa es la librería usada para el desarrollo del cliente y servidor OPC [36]. Esta implementa prácticamente todo lo definido en el estándar OPC explicado en el anexo A.

En concreto para clientes que se conectan a un servidor soporta:

- Conexión al servidor, anónima o segura.
- Navegación y lectura de atributos del servidor.
- Obtención de nodos por ruta e identificadores de nodo.
- Creación de suscripciones.
- Suscripción a nodos por cambio de valor.
- Suscripción a eventos.
- Añadir nodos.
- Autentificación mediante usuario y contraseña.
- Acceso a valores históricos de nodo.
- Inicio de sesión con certificado.
- Encriptación de la comunicación.
- Eliminar nodos.

A nivel de servidor soporta:

- Crear canales y sesiones.
- Leer/establecer atributos y navegar por los mismos.
- Obtención de nodos por ruta e identificadores de nodo.
- Autogenerar el espacio de direcciones a partir de especificaciones.
- Añadir nodos al espacio de direcciones (*address space*).
- Generar eventos frente a cambio de valores.
- Soporte a eventos.
- Soporte de métodos.
- Implementación básica de usuarios (un usuario existente llamado admin, que puede ser desactivado, todos los demás son de sólo lectura).
- Cifrado.
- Gestión de certificados.
- Eliminación de nodos.
- Soporte para valores históricos y eventos en ventana temporal configurable.

Cabe mencionar que esta librería está en modo de mantenimiento pues los esfuerzos se han centrado en el desarrollo de la versión *Asyncio* de la misma [51]. *Asyncio* [52] es un diseño de programación concurrente que ha recibido soporte dedicado en *Python*. Se trata de un diseño de un solo hilo y un solo proceso, que aparenta concurrencia a pesar de no serlo. Las *coroutines* (una característica central de *Asyncio*) pueden ser programadas de forma concurrente, pero no son inherentemente concurrentes. Las ventajas potenciales de esta aproximación son la simplificación del código al eliminar la necesidad de usar bloqueos así como la mejora del rendimiento al disminuir la complejidad en la gestión de *threads* o hilos.

B.2 Comunicación por *Modbus*. *Modbus-tk*

Modbus, según la página web de la organización encargada de su mantenimiento [7], es un protocolo cuya estructura de mensajería fue desarrollada por *Modicon* en 1979 (ahora *Schneider Electric*). Se utiliza para establecer una comunicación cliente-servidor entre dispositivos inteligentes. Es un estándar abierto, a pesar de ser una marca registrada, y uno de los protocolos de red más utilizados en el entorno de la fabricación industrial.

Ha sido implementado por cientos de proveedores en miles de dispositivos diferentes para transferir datos de registro y E/S discretas/analógicas entre dispositivos de control. Inicialmente fue concebido para transmisión por líneas haciendo uso de protocolos serie, pero ha evolucionado y también puede hacer uso de *Ethernet*.

Hay dos tipos de *Modbus* en serie, RTU y ASCII. Los modos de transmisión RTU y ASCII determinan la forma en que se codifican los mensajes *Modbus*. En *ModbusRTU*, los *bytes* se envían consecutivamente sin espacio entre ellos, con un espacio marcado por el carácter: `3-1/2` delimitando así los mensajes. Esto permite al *software* de la interfaz *Modbus* saber cuándo empieza un nuevo mensaje. Para cada *byte* de ocho *bits*, se envían un *bit* de inicio, ocho *bits* de datos, un *bit* de paridad y un *bit* de parada, para un total de 11 *bits* por *byte*. Cada mensaje *ModbusRTU* termina con una suma de comprobación de errores Comprobación de redundancia cíclica (CRC). *Modbus TCP* es básicamente *ModbusRTU* empaquetado en un paquete *Ethernet* (IEEE 802.3) con la dirección de destino como una IP usando el protocolo Protocolo de Control de Transmisión / Protocolo de internet (TCP/IP).

Modbus-tk [27] implementa la pila completa del protocolo *Modbus* (*full-stack*) ofreciendo soporte para comunicación haciendo uso de TCP, RTU y RTU a través de TCP. Ello permite escribir tanto maestros como esclavos (o cliente-servidor). Tiene una cantidad de dependencias mínima, tan sólo *PySerial* para RTU. Esto y otros factores llevan a que esta implementación entre otras existentes para *Python* sea la que tenga mejor rendimiento [31]. Además es una librería que una vez se entiende su implementación permite en muy pocas líneas, con una única librería y de manera elegante, incorporar la comunicación en sus distintas variantes de todo el protocolo *Modbus*.

B.3 Comunicacioń con APIs. *Requests*

Requests [55] es una librería que ofrece soporte para el uso de Protocolo de transferencia de hipertexto (HTTP) con un enfoque de ser de uso intuitivo. Permite abstraerse de la complejidad del protocolo para permitir centrase en la interacción con la aplicación [56]. Soporta entre otras cosas:

1. *Keep-Alive* y *Connection Pooling*
2. Dominios y URLs internacionales
3. Sesiones con persistencia de *cookies*
4. Verificación Capa de Conexiones Seguras (SSL) al estilo del navegador
5. Descodificación automática de contenidos

6. Autenticación básica/digestada
7. Soporte de *cookies* de tipo clave/valor (*key/value*)
8. Descompresión automática
9. Cuerpos de respuesta *Unicode*
10. Soporte de *proxy* HTTP(S)
11. Carga de archivos en varias partes
12. Descargas en *streaming*
13. Tiempos de espera de la conexión
14. Solicitudes fragmentadas

Anexos C

Base de datos. *SQLite*

Una base de datos es una colección de información organizada que se puede almacenar y acceder de manera electrónica. Para su creación, gestión e interacción con el usuario o aplicación se hace uso de Sistema de gestión de bases de datos (DBMS). Hay dos tipos dominantes de bases de datos: las relaciones y las no relacionales. El modelo relacional organiza la información en filas y columnas en una serie de tablas, la mayoría hacen uso del lenguaje Lengua de Consulta Estructurada (SQL). El otro tipo hace referencia a una manera de gestionar y almacenar la información con métodos distintos al tabular, esto es por ejemplo relaciones *key-value* como puede ser la variable tipo diccionario en *Python*. También existen el tipo gráfico o documento [57].

SQLite es una base de datos de tipo relacional, la palabra *lite* contenida en su nombre ya da una idea de su orientación. Se trata de una base de datos ligera en términos de configuración, administración y necesidad de recursos [58].

C.1 Caracterísicas principales

Las características principales del DBMS usando en este TFM se enumeran a continuación:

- Sin servidor o *Serverless*. Normalmente los DBMS requieren de un servidor donde ejecutarse, realizándose la comunicación con la base de datos mediante el protocolo TCP/IP. Es decir, tienen una arquitectura cliente-servidor. Una base de datos de tipo *SQLite* es un archivo con el cual se interactúa directamente.
- Contenida en sí misma. Está programado en C, para importarla a un proyecto basta con incluir el código fuente (un archivo llamado `sqlite3.c`) y su cabecera (`sqlite3.h`). Esto ha facilitado mucho su uso en todo tipo de plataformas como las móviles (*Android*, *iOS*) o su integración en otros lenguajes como *Python* o *C++*.

- Requerimientos de configuración mínimos. Como consecuencia de no requerir un servidor, no hay proceso de conexión y configuración del mismo, tampoco requiere de ningún archivo de configuración.
- De tipo transaccional. La comunicación con una base de datos es complaciente con el estándar Atomicidad, Consistencia, Aislamiento y Durabilidad (ACID).
- Soporte para tablas con tipos de datos dinámicos (coexistencia de distintos tipos de datos en la misma tabla).
- Una única conexión para distintas bases de datos, lo que facilita la interacción entre las mismas.
- Se pueden crear bases de datos en memoria RAM, sin necesidad de tener un archivo en memoria física.

C.2 Limitaciones de *SQLite*

A continuación se enumeran las limitaciones principales de la DBMS empleada en este TFM, según [59]:

1. Concurrencia limitada. Varios procesos pueden realizar solicitudes a la base de datos al mismo tiempo, pero sólo uno puede realizar cambios al mismo tiempo.
2. No existe gestión de usuarios. Esto quiere decir que no existen distintos perfiles de usuario con distintos permisos de acceso a la base.

C.3 Ventajas de *PostgreSQL*

Otro DBMS importante que merece la pena mencionar y comparar con *SQLite* es *PostgreSQL* [60], pues *SQLite* está inspirado en muchos aspectos en esta, lo que hace sencillo la transición de un tipo de base de datos a la otra. Se trata de una base de datos completa al nivel de cualquier solución comercial (*MySQL*, *Oracle*, etc.) pero se trata de un desarrollo de código abierto y gratuito. Además de ser estar enfocada en una arquitectura relacional, también tiene soporte a objetos.

1. Manejo de un gran volumen de datos. Aunque en teoría un archivo de base de datos *SQLite* puede almacenar hasta 140 TB, la documentación del proyecto no recomienda más de 1 TB, siendo preferible usa base de datos cliente-servidor como *PostgreSQL*.

2. Escritura de gran volumen de datos. Si en la arquitectura se implementan una gran cantidad de dispositivos cada uno con su serie de variables, muy rápidamente el número de accesos de escritura a la base de datos puede suponer un problema pues, como se ha comentado en las limitaciones, sólo un proceso puede acceder a la base de datos para un tiempo dado. *PostgreSQL* implementa de manera eficiente una alta capacidad de concurrencia haciendo uso de Control de concurrencia multi-versión (MVCC), por lo que esto no supone un problema.

3. Necesidad de acceso a la red. *SQLite* no provee acceso nativo a la red, para suplir esto es necesario colocar el archivo en un disco compartido en la red, lo que conlleva un uso ineficiente y extenso de ancho de banda. *PostgreSQL* es una arquitectura cliente-servidor, así que tiene intrínseca la capacidad de comunicación por la red.

4. Soporte multitud de tipos de dato. *SQLite* sólo tiene soporte para unos pocos tipos de dato: `NULL, TEXT, DATETIME, REAL, INT` mientras que *PosgreSQL* tiene un soporte mucho más extenso.

Bibliografía

[1] K. Hitomi, "Automation—its concept and a short history," *Technovation*, vol. 14, no. 2, pp. 121–128, 1994.

[2] T. Sauter, "The three generations of field-level networks—evolution and compatibility issues," *IEEE Transactions on Industrial Electronics*, vol. 57, no. 11, pp. 3585–3595, 2010.

[3] M. Felser, "Real-time ethernet-industry prospective," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1118–1129, 2005.

[4] "CHROMAE Project. Industrial Greenhouse." `http://www2.ual.es/chromae/facilities/industrial-greenhouse/`. Consultado: 23/07/2021.

[5] T. Sauter, "The continuing evolution of integration in manufacturing automation," *IEEE Industrial Electronics Magazine*, vol. 1, no. 1, pp. 10–19, 2007.

[6] R. Moraes, F. Vasques, P. Portugal, and J. A. Fonseca, "Real-time communication in 802.11 networks: The virtual token passing vtp-csma approach," in *Proceedings. 2006 31st IEEE Conference on Local Computer Networks*, pp. 389–396, IEEE, 2006.

[7] "Modbus FAQ: about Modbus Organization and the protocol." `https://modbus.org/faq.php`. Consultado: 26/06/2021.

[8] "Enron Modbus." `https://www.simplymodbus.ca/Enron.htm`. Consultado: 20/07/2021.

[9] P. Brooks, "Ethernet/ip-industrial protocol," in *ETFA 2001. 8th International Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No. 01TH8597)*, vol. 2, pp. 505–514, IEEE, 2001.

[10] T. Sauter, "Integration aspects in automation-a technology survey," in *2005 IEEE conference on emerging technologies and factory automation*, vol. 2, pp. 255–263, IEEE, 2005.

[11] D. Bakken, "Middleware," *Encyclopedia of Distributed Computing*, vol. 11, 2001.

[12] G. S. S. Chalapathi, V. Chamola, A. Vaish, and R. Buyya, "Industrial internet of things (iiot) applications of edge and fog computing: A review and future directions," *Fog/Edge Computing For Security, Privacy, and Applications*, pp. 293–325, 2021.

[13] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things journal*, vol. 3, no. 5, pp. 637–646, 2016.

[14] S. J. Bigelow, "What is edge computing? Everything you need to know." `https://searchdatacenter.techtarget.com/definition/edge-computing`. Consultado: 23/07/2021.

[15] N. Mohan and J. Kangasharju, "Edge-fog cloud: A distributed cloud for internet of things computations," in *2016 Cloudification of the Internet of Things (CIoT)*, pp. 1–6, IEEE, 2016.

[16] "RS232. Basic specifications." `https://circuitdigest.com/article/rs232-serial-communication-protocol-basics-specifications`. Consultado: 11/07/2021.

[17] "MATLAB OPC Toolbox." `https://www.mathworks.com/products/opc.html#opc-ua`. Consultado: 11/06/2021.

[18] "LabVIEW. Introduction to OPC." `https://www.ni.com/innovations/white-papers/08/introductison-to-opc.html`. Consultado: 11/06/2021.

[19] "OPC Server for Citect SCADA. Matrikon." `https://www.matrikonopc.com/opc-drivers/237/index.aspx`. Consultado: 11/06/2021.

[20] "UaExpert. Full-featured OPC UA Client which is capable of several OPC UA Profiles and features." `https://www.unified-automation.com/downloads/opc-ua-clients.html`. Consultado: 2/07/2021.

[21] "Simple open source OPC-UA GUI client." `https://github.com/FreeOpcUa/opcua-client-gui`. Consultado: 2/07/2021.

[22] "SQLectron. A simple and lightweight SQL client desktop/terminal with cross database and platform support." `https://sqlectron.github.io/`. Consultado: 2/07/2021.

[23] "HeidiSQL. Free and powerful client for MariaDB, MySQL, Microsoft SQL Server, PostgreSQL and SQLite." `https://www.heidisql.com/`. Consultado: 2/07/2021.

[24] "Database Toolbox. Exchange data with relational and nonrelational databases." `https://www.mathworks.com/products/database.html`. Consultado: 2/07/2021.

[25] "Import Data Using MATLAB Interface to SQLite." `https://www.mathworks.com/help/database/ug/import-data-using-the-matlab-interface-to-sqlite.html`. Consultado: 2/07/2021.

[26] "Connecting to SQLite from Microsoft Excel using ODBC Driver for SQLite." `https://www.devart.com/odbc/sqlite/docs/excel.htm`. Consultado: 2/07/2021.

[27] "ljean/modbus-tk: Create Modbus apps easily with Python.." `https://github.com/ljean/modbus-tk`. Consultado: 26/06/2021.

[28] "riptideio/pymodbus: A full modbus protocol written in python." `https://github.com/riptideio/pymodbus/`. Consultado: 26/06/2021.

[29] "pyhys/minimalmodbus: Easy-to-use Modbus RTU and Modbus ASCII implementation for Python.." `https://github.com/pyhys/minimalmodbus`. Consultado: 26/06/2021.

[30] "AdvancedClimateSystems/uModbus: Python implementation of the Modbus protocol.." `https://github.com/AdvancedClimateSystems/uModbus`. Consultado: 26/06/2021.

[31] "Python modbus library - Stack Overflow." `https://stackoverflow.com/questions/17081442/python-modbus-library`. Consultado: 26/06/2021.

[32] "Application Programming Interface — HubSpire." `https://www.hubspire.com/resources/general/application-programming-interface/`. Consultado: 15/06/2021.

[33] "How to Use an API with Python (Beginner's Guide) [Python API Tutorial]." `https://rapidapi.com/blog/how-to-use-an-api-with-python/`. Consultado: 15/06/2021.

[34] M. Muñoz, J. D. Gil, L. Roca, F. Rodríguez, and M. Berenguel, "An iot architecture for water resource management in agroindustrial environments: A case study in almería (spain)," *Sensors 2020, Vol. 20, Page 596*, vol. 20, p. 596, 1 2020.

[35] O. Ben-Kiki and C. Evans, "Yaml ain't markup language (yaml™) version 1.2 yaml ain't markup language (yaml™) version 1.2 3 rd edition, patched at 2009-10-01," 2001.

[36] "FreeOpcUa/opcua: OPC UA library for python." `https://github.com/FreeOpcUa/python-opcua`. Consultado: 04/07/2021.

[37] G. C. Kessler, "An overview of cryptography." http://www.garykessler.net/library/crypto.htmlhttp://www.garykessler.net/library/crypto.html, 1998.

[38] "Staying Secure with the Latest OPC UA Encryption. Opcfoundation." https://opcconnect.opcfoundation.org/2019/06/staying-secure-with-the-latest-opc-ua-encryption/. Consultado: 27/07/2021.

[39] "SQLite to PosgreSQL database migration. Medium." https://medium.com/djangotube/django-sqlite-to-postgresql-database-migration-e3c1f76711e1. Consultado: 13/06/2021.

[40] "Schneider Electric. Modicon M241 Logic Controller - Programming Guide." www.schneider-electric.com, 2018. Consultado: 11/05/2021.

[41] "Manual OSCAT Basic." http://www.oscat.de/images/OSCATBasic/oscat_basic333_en.pdf. Consultado: 22/06/2021.

[42] "Working with MATLAB Interface to SQLite - MATLAB and Simulink." https://www.mathworks.com/help/database/ug/working-with-the-matlab-interface-to-sqlite.html. Consultado: 05/07/2021.

[43] "Import Data Using MATLAB Interface to SQLite - MATLAB and Simulink." https://www.mathworks.com/help/database/ug/import-data-using-the-matlab-interface-to-sqlite.html. Consultado: 05/07/2021.

[44] "SQLite JDBC for Windows - MATLAB and Simulink." https://www.mathworks.com/help/database/ug/sqlite-jdbc-windows.html. Consultado: 05/07/2021.

[45] "SQLite with MATLAB." https://gist.github.com/cbcunc/e2bc3ef170544e4bf0f0. Consultado: 05/07/2021.

[46] "Schneider Electric. Precios software HMI." https://www.se.com/es/es/download/document/ESMKT02055B18_9/. Consultado: 26/07/2021.

[47] "VTSCADA. Euro Pricing for Small to Medium Systems." https://www.vtscada.com/documents/pricing/VTScada_Euro_Pricing_Web.pdf. Consultado: 26/07/2021.

[48] "Migrating a SQLite database to PostgreSQL — pgloader 3.4.1 documentation." https://pgloader.readthedocs.io/en/latest/ref/sqlite.html. Consultado: 04/07/2021.

[49] "Home - pycomm3 1.2.0 documentation." `https://docs.pycomm3.dev/en/latest/index.html`. Consultado: 20/06/2021.

[50] "pjkundert/cpppo: Communications Protocol Python Parser and Originator – EtherNet/IP CIP." `https://github.com/pjkundert/cpppo`. Consultado: 04/07/2021.

[51] "FreeOpcUa/opcua-asyncio: OPC UA library for python ¿ 3.6 asyncio." `https://github.com/FreeOpcUa/opcua-asyncio`. Consultado: 04/07/2021.

[52] "Async IO in Python: A Complete Walkthrough – Real Python." `https://realpython.com/async-io-python/`. Consultado: 04/07/2021.

[53] S.-H. Leitner and W. Mahnke, "Opc ua–service-oriented architecture for industrial applications," *ABB Corporate Research Center*, vol. 48, no. 61-66, p. 22, 2006.

[54] "OPC Unified Architecture." `https://inmation.com/docs/system/1.76/opc-connectivity/opc-unified-architecture.html`. Consultado: 11/07/2021.

[55] "Requests: HTTP for Humans™ — Requests 2.25.1 documentation." `https://docs.python-requests.org/en/master/`. Consultado: 15/06/2021.

[56] "Python's Requests Library (Guide) – Real Python." `https://realpython.com/python-requests/`. Consultado: 11/07/2021.

[57] A. Hughes, "What is a Database? - Definition from WhatIs.com." `https://searchdatamanagement.techtarget.com/definition/database`. Consultado: 11/07/2021.

[58] M. Owens and G. Allen, *SQLite*. Springer, 2010.

[59] "SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems — DigitalOcean." `https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-database-management-systems`. Consultado: 11/07/2021.

[60] M. Stonebraker and G. Kemnitz, "The postgres next generation database management system," *Communications of the ACM*, vol. 34, no. 10, pp. 78–92, 1991.

UNIVERSIDAD DE ALMERÍA

Resumen/Abstract

La integración de servicios y dispositivos industriales es una necesidad básica de cualquier aplicación en un entorno industrial o de investigación desde la extensión del uso de señales para la monitorización y automatización de procesos. Existen diversos protocolos abiertos y probados para comunicación a bajo nivel con dispositivos concretos. También existen soluciones comerciales para la integración de capas superiores y la unificación de protocolos, pero estas suelen estar limitadas al uso de *hardware* del fabricante específico. Surge por tanto la idea de suplir esta necesidad de integración diseñando e implementado una arquitectura abierta y modular para la comunicación de una diversidad heterogénea de dispositivos y que permita la interoperabilidad de distintos agentes, tanto a nivel de campo como en un nivel de abstracción superior.

Con este objetivo se ha diseñado, implementado y posteriormente verificado el funcionamiento de una propuesta de arquitectura para comunicación de servicios y dispositivos. Para ello se ha aprovechado la existencia de protocolos abiertos tanto a nivel de campo (*Modbus*, p. ej.) como a mayor nivel de abstracción (*OPC*). Haciendo uso de librerías existentes para el lenguaje de programación *Python*, se han desarrollado dos programas que conforman la base de la arquitectura. Uno de ellos es el encargado de generar y gestionar el servidor OPC, así como automáticamente gestionar una base de datos que almacene los registros de todas las variables del sistema. Por otra parte, se ha programado un cliente genérico, el cual puede establecer una comunicación bidireccional entre dispositivo o servicio industrial y el servidor OPC. Además, se han desarrollado plantillas para la interacción con los diferentes elementos de la arquitectura con el *software* científico *MATLAB* así como la programación de un *PLC* para comunicación vía *Modbus* haciendo uso de un entorno de desarrollo basado en *CODESYS*.

The integration of industrial services and devices is a basic need for any application in an industrial or research environment since the widespread use of signals for process monitoring and automation. There are several open and proven protocols for low-level communication with specific devices. Tthere are also commercial solutions for the integration of higher layers and unification of protocols, but these are usually limited to the use of specific manufacturer's hardware, and these solutions are usually rigid and limited to the manufacturer's approach. Therefore, the idea arises to supply this need for integration by designing and implementing an open and modular architecture for the communication of a heterogeneous diversity of devices and to allow the interoperability of different agents, both at field level and at a higher level of abstraction.

With this objective in mind, the architecture has been designed, implemented and later on tested. To this end, the existence of open protocols at both field level (e.g. *Modbus*) and at a higher level of abstraction (e.g. *OPC*) has been taken advantage of, and using existing libraries for the programming language *Python*, two programs have been developed that make up the architecture. One of them is in charge of generating and managing the *OPC* server, and also responsible for automatically generating a database that stores the records of all the system variables. On the other hand, a generic client has been programmed which can establish a bidirectional communication between the device or industrial service and the *OPC* server. In addition, templates have been developed for the interaction with the different elements of the architecture with the *MATLAB* scientific software as well as the programming of a *PLC* for communication via *Modbus* using a development environment based on *CODESYS*.