Facultad de Ciencias

Estudio y extensión de una biblioteca de comunicaciones para dispositivos PLC

(Study and extension of a library used in PLC communications)

Trabajo de fin de grado
para acceder al título de

GRADUADO EN INGENIERÍA INFORMÁTICA

Autor: Raúl Zamora Martínez

Director: Enrique Vallejo Gutiérrez

Septiembre de 2021

Estudio y extensión de una biblioteca de comunicaciones para dispositivos PLC

Resumen

palabras clave: SLMP, Protocolo, PLC, Biblioteca

Dado el auge de las tecnologías *Internet of things* (IoT) y el incremento en la cantidad de información que se procesa en el sector industrial, hay una necesidad evidente de conectar herramientas de automatización y recolección de datos, que típicamente consisten en *Programable Logical Controllers* (PLC), con soluciones *software*. Dado que muchas de esas soluciones son propietarias y poco flexibles, en este proyecto se implementa un protocolo de comunicaciones entre dispositivos PLC y PC. Tras un estudio de las diferentes opciones disponibles, se ha propuesto una solución basada en el protocolo SLMP (*Seamless Message Protocol*). Esta solución ayuda a simplificar el número de dependencias necesarias en el proceso de recopilación de datos, además de obtener unas tasas de transmisión de información más altas que las alternativas analizadas.

(Study and extension of a library used in PLC communications)

Abstract

keywords: SLMP, Protocol, PLC, Library

The growth in Internet of things (IoT) technologies and rise in the amount of information processed in the industry sector, forces the need for connecting automation and data recolection tools, typically consisting in Programable Logical Controllers (PLC), with software based solutions. Since many of these solutions are propietary and inflexible, this project implements a communication protocol between PCs and PLC devices. After evaluating other available options, a SLMP (Seamless message protocol) communications solution has been offered. The resulting implementation helps simplify needed dependencies in the data recolection process in addition to obtaining higher information transmission rates than the alternatives analyzed.

Índice

1 Introducción

Esta sección presenta la motivación y el contexto en el que se ha realizado el trabajo de fin de grado.El presente proyecto ha formado parte de las prácticas realizadas en SILECMAR, una empresa de ingeniería naval. Entre sus principales áreas de desarrollo se encuentra la automatización y control de diversos sistemas de barcos, como largado y recogida de redes automáticos y monitorización de estos procesos.

Para efectuar estos trabajos de automatización y control se usan dispositivos *Programmable Logical Controller* (PLC), que proporcionan servicios de adquisición de datos y actuación sobre elementos *hardware*. Estos elementos abarcan desde sensores analógicos hasta componentes más complejos que pueden enviar señales digitales con información variada, por ejemplo, la centralita de un motor. La extracción de todos estos datos a sistemas como un PC requiere de herramientas adicionales, normalmente *software* y/o *hardware* propietario.

Este proyecto se centra en el el desarrollo de una solución que permita exponer la capa de adquisición de datos que proporcionan los PLC a otras superiores, en particular el trasiego de datos desde un PLC a un PC, u otros dispositivos, donde puedan correr aplicaciones *software* de mayor complejidad. Para lograr esto, se ha realizado un análisis de las tecnologías disponibles.

1.1. Estado del Arte

Los dispositivos PLC constituyen una solución reconocida para el control automático de maquinaria en entornos industriales. SILECMAR emplea PLC de Mitsubishi, que utilizan el sistema de comunicaciones propietario CC-Link. Si bien son altamente fiables y robustos, estos dispositivos presentan la desventaja de usar dichos protocolos propietarios, que no están generalizados en la industria y suelen requerir de *hardware* específico. Un ejemplo de estos equipos son las pantallas empleadas para la visualización de datos, normalmente referidas como *Human-Machine Interface* (HMI).

Este *hardware* suele presentar un diseño cerrado, sin la posibilidad de realizar cambios a la manera en que se muestra la información. En SILECMAR, las comunicaciones entre PLC y PC se han llevado a cabo usando HMI. Estas pantallas requieren de una programación relativamente rígida a la hora de la edición de interfaces; por ejemplo, la colocación de elementos suele ser absoluta, es decir, con posiciones de píxeles (x, y). Esto complica la portabilidad de interfaces de una pantalla a otra, obligando a tener plantillas específicas para cada pantalla.

Los sistemas SCADA (*Supervisory Control And Data Acquisition*) suelen ser una parte fundamental de la mayoría de procesos de automatización y monitorización. Dichos sistemas consisten en un componente *software* que interactúa con las interfaces de comunicación y obtención de datos. No solo eso, sino que muestran esta información en algún tipo de *Graphical User Interface* (GUI) [1]. Así, los PLC suelen ser los encargados

de realizar tareas de muestreo y comunicación con los sensores y demás dispositivos *hardware*. Las necesidades del sector, por la parte de IoT (*Internet of things*), han impulsado cambios al *software* de control supervisado para que dé más funcionalidades de visualización del estado de una planta industrial desde cualquier parte, sin necesidad de estar *in situ*. Al alejar las visualizaciones del lugar físico donde se están tomando los datos, se facilita mostrar información a clientes que antes estaba reservada a los técnicos encargados de controlar un entorno industrial.

Con el auge de las tecnologías asociadas a la Industria 4.0, cada vez es más relevante aportar soluciones que consigan facilitar la unión entre las diferentes partes que entran en juego en cualquier proceso de carácter industrial. En este caso, la industria marítima presenta una serie de dificultades extra por tener condiciones más hostiles. Al no ser una planta típica ubicada en un lugar fijo, esto hace que las comunicaciones con el exterior sean menos estables y normalmente dependientes de conexiones por satélite.

El desarrollo hecho en este proyecto se centra en el aspecto de las comunicaciones, valorando diferentes protocolos, para permitir crear una biblioteca que sea usable en una aplicación de tipo SCADA. Existen protocolos de dominio público, como *Seamless Message Protocol* (SLMP) o Modbus, que permiten organizar las comunicaciones de manera propia, eliminando el *hardware* extra que era necesario previamente. Mediante la biblioteca desarrollada, se simplifica el uso de estos protocolos y se consigue aumentar las posibilidades de personalización y disminuir costes asociados a los productos propietarios.

1.2. Objetivos

Este proyecto busca generar una solución *software* que sea accesible para usuarios de conocimiento medio en programación, en particular los ingenieros de automatización que trabajan en el departamento técnico de SILECMAR.

La biblioteca proporciona una API en diferentes lenguajes, entre ellos C y Python, para facilitar la comunicación con otros módulos. Basándose en esta interfaz, se facilitan una serie de funciones que sirven como punto de acceso a otros componentes software.

Como el proyecto se ha enfocado con la idea de permitir su uso a cualquier usuario, conozca o no el protocolo SLMP, se exponen varias llamadas útiles para realizar comunicaciones o conocer la organización de memoria del PLC. A partir de dicha información, se ha definido el tipo de consulta que se quiere hacer. Alternativamente, se puede integrar con el sistema de procesado de datos de la empresa, extendiéndolo con las funcionalidades nuevas y permitiendo la automatización de las consultas al PLC, que antes dependía de pantallas de visualización propietarias.

Los objetivos definidos al inicio del proyecto son los siguientes:

- Analizar las diferentes opciones para desarrollar una alternativa al mecanismo actual de comunicación entre PC y PLC.
- Generar un módulo que pueda ser integrado con el resto del código de empresa.
- Extraer información sobre las variables del PLC, que pueda ser usada en procesos posteriores.
- Facilitar el uso de la biblioteca para que mantenga el flujo de datos de la empresa.

1.3. Estructura del documento

En el capítulo 2 se explica el marco de trabajo del proyecto; sus tres secciones entran en detalle sobre sobre los apartados que se han considerado más importantes: metodología, lenguajes y herramientas. Se comenta la planificación general del trabajo junto a las tecnologías que se han usado durante el desarrollo.

El capítulo 3 centra la atención sobre el sistema hardware sobre el que se va a trabajar. Inicialmente, en la sección 3.1, se hace una comparación entre los protocolos más relevantes para resolver el problema planteado, explicando las ventajas y desventajas de cada uno. Se justifica el uso de SLMP con base en este análisis. A continuación, se proporciona una visión general de un PLC, incluyendo aspectos como la comunicación con otros dispositivos o su modo típico de ejecución de rutinas. Finalmente, se explica en detalle el protocolo SLMP, observando sus características más significativas.

En el capítulo 4 se explican las características y partes diseñadas para la biblioteca SLMP, desde los requisitos iniciales a la descripción de las llamadas que se proporcionan en la API creada. Adicionalmente, se muestra una aplicación que integra la biblioteca en su estructura.

El capítulo 5 muestra las metodologías para hacer tests y los resultados de los mismos. Las dos primeras secciones explican diferentes tipos de pruebas, ejemplificando cómo se han llevado a cabo. Por último, en la sección 5.3 se analizan los resultados obtenidos, enfrentando la biblioteca a otras similares que se han valorado como posibles alternativas.

2 Metodología y tecnologías empleadas

Esta sección detalla la planificación propuesta al principio del proyecto y el seguimiento de la misma durante sus etapas. Adicionalmente, se hace un listado exhaustivo con una breve explicación de las herramientas que se han usado durante el proyecto.

2.1. Planificación y metodología

La planificación del proyecto se ha divido en tres etapas principales, con una duración total de unos ocho meses. El diagrama de la figura 1 muestra una idea general de las tareas principales o consideradas más relevantes.

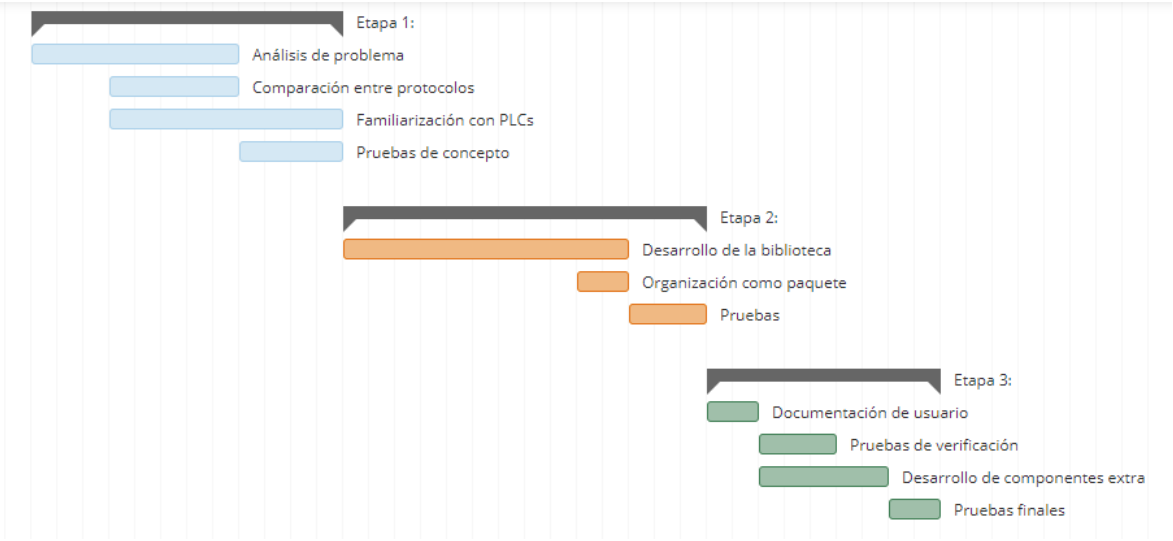


Figura 1: Planificación del proyecto

Las dos primeras etapas de desarrollo se corresponden aproximadamente con el contenido de los capítulos 3 y 4, respectivamente. Los componentes extra forman una implementación de una aplicación completa, usando tecnologías existentes junto a la biblioteca desarrollada. A pesar de no ser el foco principal del proyecto, se ha considerado conveniente añadirlo para dar un contexto de uso de la biblioteca en una aplicación típica de *software*. Todo esto se comenta con mayor detalle en el capítulo 4.

Integración continua

La integración continua (CI por sus siglas en inglés, *Continuous integration*) es una técnica de desarrollo de *software* que permite gestionar la integración del código proveniente de diferentes fuentes de manera automatizada. Para esto, se definen varias etapas con tareas como la compilación del código o la ejecución de baterías de test. Estas etapas se lanzan en entornos aislados cada vez que hay un cambio en las ramas principales del proyecto.

Esta técnica facilita el trabajo cuando se cuenta con un equipo de desarrollo formado por varios contribuyentes de código. Si bien este proyecto se ha desarrollado de forma autónoma, el dejar una base sólida para desarrollos futuros se ha valorado como parte importante durante las etapas posteriores del ciclo de vida del mantenimiento.

Existen diversas aplicaciones *software* que permiten integrar estas técnicas en un proyecto. Inicialmente, plataformas como Jenkins han sido consideradas para implementar la integración continua, ya que esta herramienta ofrece muchas posibilidades en procesos de automatización de desarrollo de *software*. Sin embargo, no se han considerado necesarias tantas funcionalidades. Por este motivo, se ha optado por las utilidades que ofrece GitLab de forma nativa, por ser más aplicables a la escala de este trabajo. Se ha usado una instalación local de Gitlab, en un servidor de la empresa, para almacenar el código fuente del proyecto y mantener su control de versiones.

La implementación de CI se hace por medio de un *runner* configurado en la aplicación de GitLab. Se trata un servicio que se ejecuta en un computador con acceso al repositorio. Este *runner* es el encargado de detectar si se hace un *push* de las ramas principales, normalmente la rama *master* o *develop*. Cuando esto ocurre, se hace uso de un archivo de configuración que se sube junto al proyecto, *.gitlab-ci.yml*. Este archivo es interpretado por el *runner* y, a continuación, se lanza algún tipo de entorno encargado de realizar los pasos deseados relacionados con la integración continua.

El proceso de CI diseñado para este proyecto cuenta con varias fases. El primer paso que se da es la compilación del código. En el caso de Python, implica crear un paquete que se pueda distribuir e instalar con la utilidad nativa por medio de '*pip install <paquete>*'. A continuación, se lanzan las baterías de test. Junto a los tests, se ejecuta un paquete de Python llamado 'coverage'. Este paquete indica la cantidad de código que los tests han alcanzado durante la ejecución, dando una buena idea de si hay casos básicos que no se están teniendo en cuenta. Estos tres pasos generan un reporte en Gitlab, avisando en caso de que haya fallado alguno. Así, se tiene un flujo básico que permite automatizar la integración de características nuevas con el resto del código.

2.2. Lenguajes

Los lenguajes usados han sido C para la parte básica de la biblioteca y Python 3.8 para generar la API principal y la interfaz que interactúa con C. Para una aplicación que se base en comunicaciones que pueden ser en directo, el uso de Python no suele ser el más adecuado, pues no tiene tanta velocidad como lenguajes de bajo nivel, C concretamente. Sin embargo, esto no afecta al proyecto, pues la parte más limitante, el PLC, es mucho más lento en comparación a un PC.

Adicionalmente, como se explica en la sección 1.2, el alcance del proyecto no busca dar características de tiempo real, sino simplemente desarrollar una herramienta para

realizar configuraciones iniciales sobre el PLC, mostrar gráficas durante procesos de pruebas, etc. Finalmente, el uso de Python ha facilitado la integración con el resto del código de empresa, minimizando dependencias de diferentes lenguajes.

2.3. Herramientas y tecnologías

Visual Studio Code

Se ha usado durante el desarrollo este entorno ligero y rápido. Las extensiones específicas de Python de las que dispone y su sencilla integración, hacen este *Integrated Development Environment* (IDE) una opción idónea para la tarea.

Compilador GCC 7.5.0

GNU Compiler Collection (GCC), nativo en los sistemas UNIX, ha sido el elegido para realizar la compilación del código de C. El compilador guarda la biblioteca generada como un objeto compartido (extensión .dll en Windows o .so en sistemas UNIX), que se carga en tiempo de ejecución por el módulo de Python.

Git

Git es el sistema de control de versiones considerado como lengua franca en la informática. Desarrollado por Linus Torvalds en 2007, es uno de los modos más comunes de gestionar repositorios de *software*. Se ha usado junto a la plataforma de GitLab.

Sonarqube

Para obtener un análisis sobre la calidad del código se ha usado el paquete de Python Pylint, que da avisos sobre sintaxis y aspectos de 'buenas prácticas', como pueden ser la longitud de las líneas o nombres de variables que no se ajusten a los estándares. Profundizando un poco más y como apoyo a lo anterior, se usa el *software* SonarQube [2], que entra en mayor detalle sobre múltiples aspectos del código, como son los riesgos de seguridad o posibles defectos que encuentre. Todo esto se tiene en cuenta durante el análisis del código.

JSON

El formato de archivo JSON (*JavaScrip Object Notation*) es una manera de mandar información, ya sea a través de la web por API o como forma de guardar información para ser usada por otros procesos [3].

XML

XML (*Extensible Markup Language*) es un lenguaje de marcado que permite almacenar información en texto plano, siguiendo una estructura [4]. Los PLCs permiten extraer sus variables internas en este formato, por lo que es necesario un *parser* selectivo para guardar los datos relevantes para el desarrollo de la aplicación.

Jupyter-Notebook

Ya sea para hacer pruebas de concepto, o para hacer exploraciones de datos, *Jupyter-Notebook* permite ejecutar código en bloques, manteniendo el intérprete de Python activo [5]. Ha sido especialmente útil el desarrollo de la parte de visualización,

ya que facilita las pruebas con diferentes maneras de visualización sin necesidad de volver a cargar la biblioteca o crear nuevas conexiones al PLC.

Grafana
Aplicación de código abierto usada para generar paneles de visualización de información a partir de una conexión a base de datos.

InfluxDB
Base de datos de estructura no SQL. Usada principalmente para guardar métricas de series temporales.

Paquetes auxiliares de Python

Logging
Una de las bibliotecas estándar de Python, encargada de llevar a cabo tareas relacionadas con el registro de sucesos durante la ejecución del código. Se ha usado para definir diferentes niveles de importancia en los *logs*.

Unittest
Otra biblioteca estándar de Python. Inspirada en JUnit, tiene las funcionalidades generales de cualquier biblioteca para realizar casos de prueba unitarios.

Coverage
Este paquete permite monitorizar las líneas de código que se ejecutan al correr un programa. Al terminar, puede volcar la información que ha recogido en un archivo XML, que más tarde se puede integrar con el *software* de Sonarqube. Normalmente, se usa junto a los tests para obtener un informe sobre qué partes del código fuente son accedidas por los tests y cuáles no.

Pylint
Se ha usado durante el desarrollo y sirve para generar informes sobre la calidad del código. Aporta información sobre el estilo, como nombres de variables, y complejidad cognitiva, como funciones muy largas o con muchas variables.

Venv
Herramienta básica de desarrollo para crear entornos virtuales ligeros. Permite separar el entorno de trabajo del general de Python, consiguiendo 'congelar' las actualizaciones de bibliotecas fuera de este entorno.

Numpy
Esta biblioteca matemática tiene, entre otras cosas, múltiples funcionalidades para trabajar con listas multidimensionales [6]. Dadas las características de este proyecto, ha permitido simplificar hasta cierto punto la serialización y deserialización de los mensajes del protocolo.

InfluxDB-Python

Este paquete de Python es el recomendado oficialmente para hacer conexiones a las bases de datos de Influx. Ha sido desarrollado principalmente por trabajadores de InfluxData Inc. y, más adelante, por miembros de la comunidad.

Bokeh

Permite lanzar un servidor de Javascript en una pestaña del navegador con las gráficas que se quiera, mientras el código principal está sirviendo datos a las mismas. Se ha usado esta biblioteca, en vez de otras, por generar paneles altamente personalizables e interactivos en tiempo de ejecución [7]. Matplotlib, por ejemplo, genera visualizaciones estáticas.

Panel

Biblioteca de composición de dashboards [8]. Se ha usado junto con Bokeh para generar de manera dinámica paneles de gráficas.

3 Estudio del protocolo y características del sistema

Este capítulo hace una introducción al protocolo que se ha usado y justifica su elección como solución para el problema planteado. Para ello, se hace una comparativa inicial con otros protocolos usados para intercambio de datos entre PLC. Seguidamente, se explica el funcionamiento básico de un PLC y las características que son relevantes para este proyecto. Por último, se hace un resumen de las partes más importantes del protocolo SLMP.

3.1. Comparativa de protocolos

A continuación se presenta una comparativa entre tres protocolos candidatos para resolver el problema planteado. Estos protocolos son Modbus, SLMP y MQTT. Modbus y MQTT han sido valorados por su carácter general, mientras que SLMP por su relación concreta con los PLC con los que se trabaja en la empresa.

Las necesidades de extracción de datos de sistemas de monitorización son evidentes en la proliferación de nuevas tecnologías que buscan facilitar estos procesos. Herramientas ETL (*Extract, Transform and Load*) como PLC4X [9] buscan salvar la distancia que hay entre los PLC y dispositivos de mayor complejidad.

La problemática principal que existe con dispositivos PLC es que todo su *stack* de tecnologías suele ser propietario. Cada empresa que fabrica estos dispositivos tiene control sobre la manera que tienen de comunicarse y el soporte que dan a otros protocolos. Así, ofrecen soluciones propias para extraer datos o visualizarlos.

Existen estimaciones que muestran Modbus como un protocolo muy extendido en sistemas de transmisión basados en bus de campo [10]. De esta forma, los PLC suelen incluir en sus módulos de comunicaciones estándar una parte de Modbus, ya sea por Ethernet o por conexión serie. No solo eso, sino que dan facilidades en la programación de los PLC para gestionar estos tipos de comunicaciones. Así, se ofrece una manera estandarizada de comunicar múltiples PLC entre sí, ya sean iguales o de diferentes fabricantes. Además, Modbus ya tiene bibliotecas desarrolladas por la comunidad, que permiten su uso desde una computadora cualquiera. Sabiendo todo esto, Modbus podría ser una solución al problema de cómo extraer datos.

El número máximo de registros, de 16 bits cada uno, que se pueden pedir en una petición Modbus es 125. El tamaño máximo de trama son 256 bytes. Esta característica tiene sentido en el contexto en el que Modbus nació, a finales de los años 70, para comunicar PLC. A pesar de esto, por las limitaciones que tiene un PLC para tratar cada trama que le llega (explicado en mayor detalle en la sección 3.2), Modbus no es una solución adecuada.

Otro protocolo que se ha valorado es el MQTT. Este protocolo implementa un modelo basado en suscripción a datos, con una parte encargada de mantener actualizados estos datos. El principal inconveniente observado en este protocolo es que, al ser relativamente más moderno que los otros (de 1999), tiene un soporte menos extendido. Es importante hacer notar que la adopción de protocolos como estos, para que sean estándar en la industria, son procesos largos en el tiempo. En este caso, existen algunas soluciones que implementan MQTT para los PLCs usados durante el desarrollo, como módulos que convierten comunicaciones de PLC de múltiples marcas a protocolos IoT. Esto no se alinea con la idea inicial de limitar las dependencias de *hardware* adicional.

Por último, SLMP es una solución perteneciente a la familia CC-Link que proporciona una capa de abstracción sobre otros protocolos subyacentes. Los PLC usados por la empresa implementan de manera nativa este protocolo, por lo que el único desarrollo implicaría la parte *software* que vaya instalado en una computadora. No solo eso, sino que este protocolo permite solicitar hasta 960 registros en una misma petición de lectura, lo cual minimiza tiempos extras empleados en operaciones de E/S.

3.2. Descripción de un dispositivo PLC

Un PLC es una computadora diseñada para ser usada en entornos industriales como dispositivo de automatización de procesos. Estos dispositivos suelen tener una serie de entradas y salidas tanto analógicas como digitales. Están construidos para que puedan soportar condiciones adversas, como entornos con altas temperaturas o vibraciones.

Normalmente no implementan un sistema operativo como tal, sino que se basan en una arquitectura monolítica con una rutina principal (*main*) que se ejecuta periódicamente. Esta rutina puede tener funcionalidades como lecturas de los búferes de entrada del módulo de Ethernet, si tuviese alguna conexión activada. A este *main*, los programadores añaden rutinas propias encargadas de hacer las operaciones necesarias para un proceso cualquiera. Todas estos procesos se van ejecutando de forma secuencial, repitiéndose cíclicamente de forma indefinida.

El sistema de memoria está compuesto normalmente por registros de tipo palabra y bit, aunque puede haber variaciones de un PLC a otro. Por ejemplo, pueden existir registros de doble palabra, los cuales ocupan 16 bits. Los diferentes tipos de registros que hay en un PLC suelen tener funciones específicas, siendo algunos dedicados a acceso más rápido, otros para contadores, etc. Cabe decir que, a pesar de tener un uso recomendado, no hay restricciones concretas en la mayoría de ellos, pudiendo usar un registro tipo word para guardar un valor binario. El Anexo II contiene una lista exhaustiva de los registros usados por el PLC.

El direccionamiento puede ser en decimal o hexadecimal, dependiendo del registro. Por ejemplo, en el PLC que se va a usar durante el desarrollo, a los registros 'D' se accede usando posiciones en decimal (D0, D1, …, D9, D10, …) y los registros W de manera hexadecimal (W0, W1, … , W9, W0A, W0B, …)

El PLC usado durante el desarrollo pertenece a la familia de los iQ-F de Mitsubishi. En la figura 2 se ve uno, con varios componentes extra conectados para generar valores en los registros. En particular, se puede apreciar un potenciómetro y un interruptor NO (*Normally Open*).

Si se define una función que lea ciertos valores de las entradas, que haga cálculos con ellos y que guarde un resultado en un registro de memoria, esto se ejecutará cada vez que el programa vuelva a iterar. A todo el tiempo de ejecución de rutinas se le

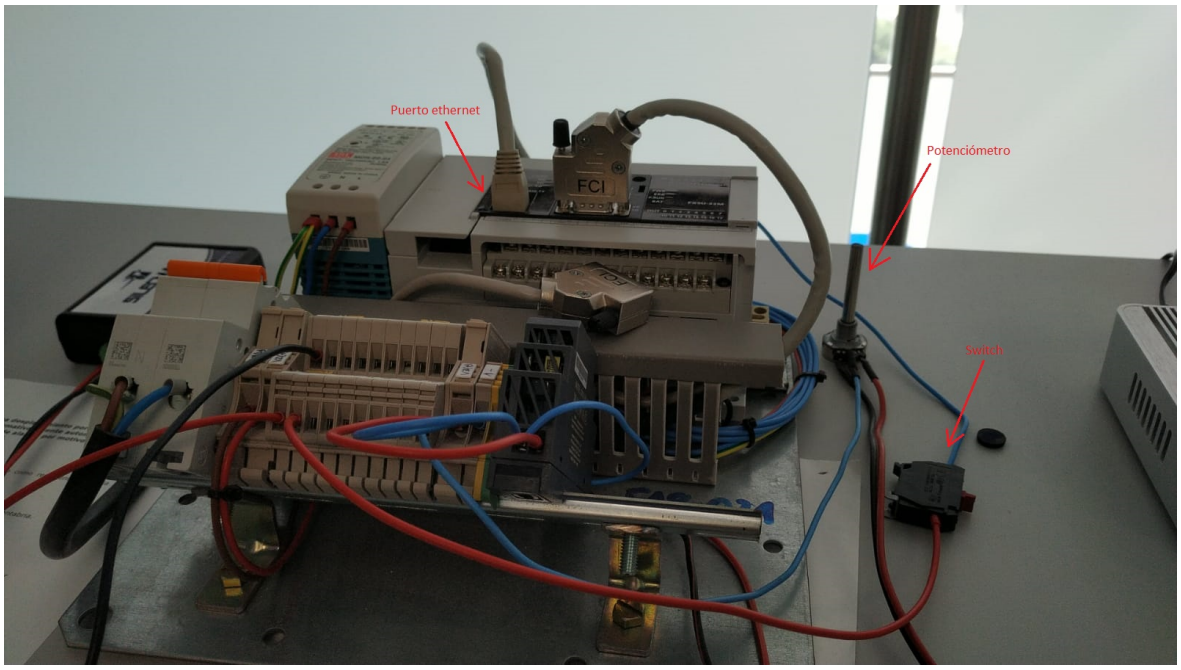


Figura 2: *Hardware* usado durante el desarrollo

llama ciclo de programa. Este ciclo de programa es variable y depende de la longitud de las rutinas escritas y de las operaciones que se hacen en estas rutinas: cuanto más complejo sea un programa, más tiempo tardará en finalizar cada ciclo.

El ciclo de programa es un concepto importante, pues determina el tiempo de respuesta que tiene el PLC ante una petición, o una transmisión de tramas por la red a otro PLC/sensor. Esto se debe a que solo lee peticiones del módulo de Ethernet durante un intervalo de tiempo después de acabar cada ciclo de programa. En consecuencia, es una métrica que se debe tener en cuenta para determinar ciertos aspectos de la conexión que se explica en detalle en la siguiente sección.

La figura 3 muestra el flujo de petición y respuesta que tiene el PLC. Cada ciclo de programa viene determinado por el intervalo de tiempo entre *Step0* y *END*. En el contexto de la programación de PLCs, cada línea de la rutina principal se llama *step* (paso). Al seguir una estructura secuencial, los programas se miden en pasos, es decir, el número de instrucciones que se van a ejecutar en cada ciclo. Por lo tanto, *END* representa el último paso del ciclo. Entre el final de un ciclo y el principio del siguiente está el tiempo de *END processing*. Este espacio es el dedicado a resolver peticiones que hayan llegado al puerto de red durante el ciclo. Las peticiones de lectura o escritura, por ejemplo, se resuelven en este intervalo. Una vez ejecutada, se envía el resultado de la operación en los casos en los que haya respuesta. En la figura 3 se pueden ver los dos ACK correspondientes a la comunicación TCP. En caso de que fuese a través de un socket UDP, no serían necesarios.

3.3. Pruebas de rendimiento y funcionamiento

Durante las fases iniciales del desarrollo, se han efectuado varias pruebas del protocolo junto al PLC para generar métricas de rendimiento y funcionamiento. Todo esto se ha llevado a cabo por medio de la herramienta de programación de PLCs de Mitsubishi, GXWorks [12]. Esta aplicación permite acceder a funciones como visualización

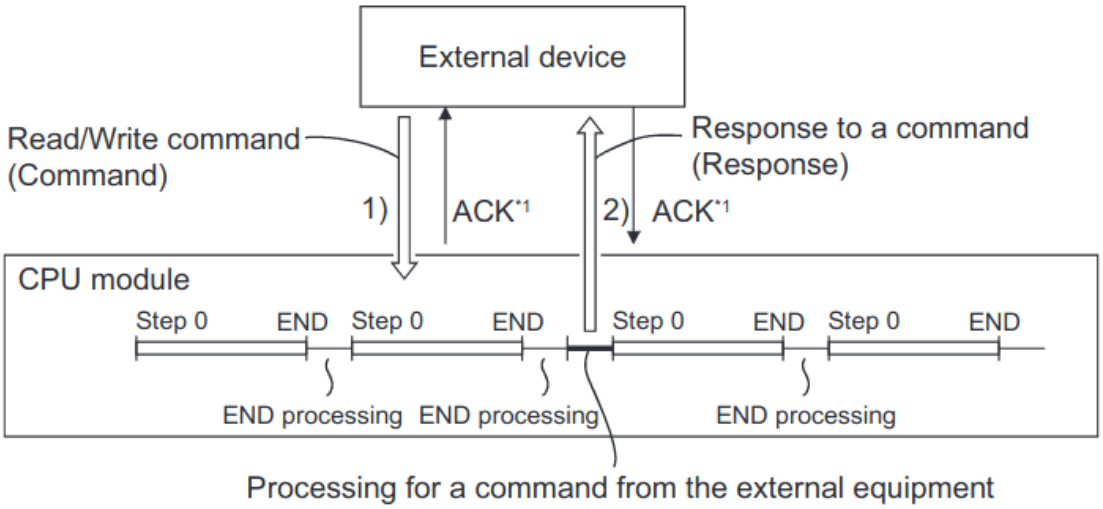


Figura 3: Ciclo de petición-respuesta [11]

de memoria (Figura 4).

En la primera etapa del proyecto se han seguido los manuales de usuario de este *software* para familiarizarse con el funcionamiento del mismo. Adicionalmente, esta aplicación permite modificar la configuración de comunicaciones, como el número de *sockets* habilitados para SLMP u otros protocolos.

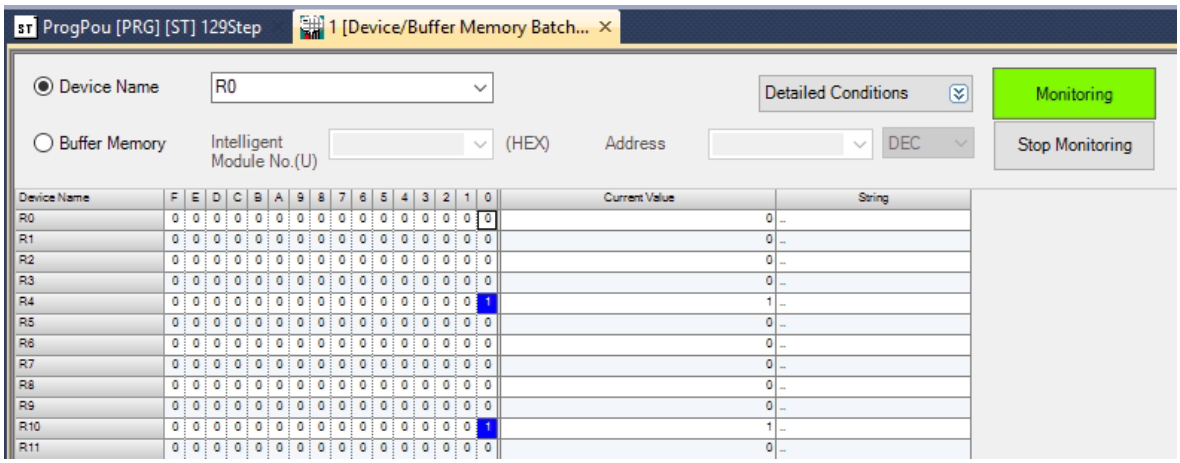


Figura 4: Zona de memoria del PLC

Por ejemplo, permite ajustar el tiempo de ciclo a un valor en concreto. Así, aunque la rutina principal del PLC tarde menos tiempo en completarse, se extiende el tiempo hasta el definido en la configuración. Haciendo uso de esta funcionalidad, se han explorado los tiempos que el PLC dedica a resolver comunicaciones tras el tiempo de ciclo, entre otras cosas.

El modelo usado durante el proyecto permite configurar hasta ocho conexiones diferentes, por lo que se pueden abrir un máximo de ocho *sockets*. La figura 5 muestra un ejemplo de configuración. En este caso, se han habilitado tres conexiones de tipo SLMP, una de tipo Modbus y otras tres de MELSOFT. Estas últimas son específicas de Mitsubishi, no se han usado durante el desarrollo.

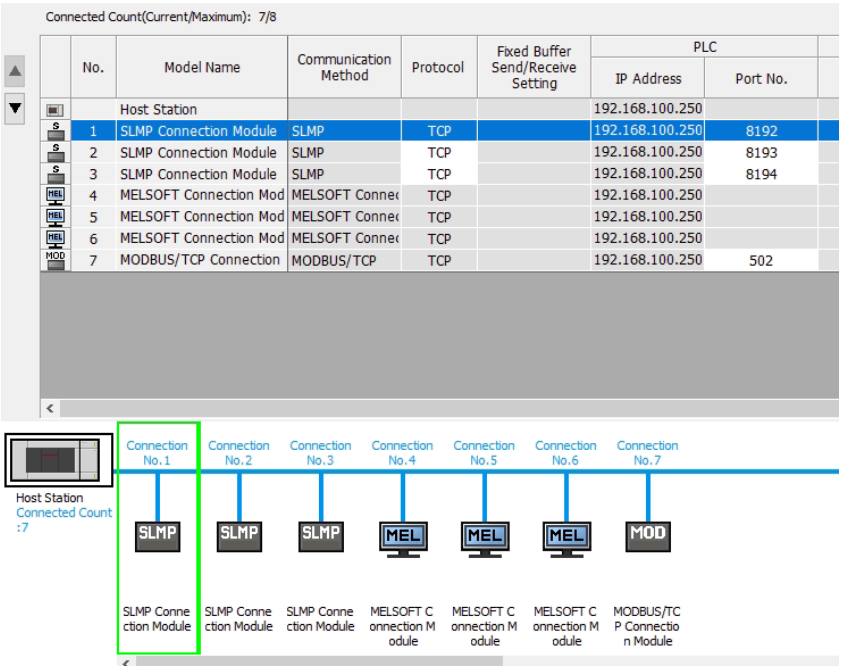


Figura 5: Pantalla de conexiones del PLC

El objetivo de estas pruebas es explorar la manera que tiene el PLC de responder a una petición cualquiera. En la sección 3.2 se explica este proceso; no obstante, se han llevado a cabo estas pruebas a modo de comprobación, para comprender mejor su funcionamiento.

Las pruebas realizadas han usado desde uno hasta tres *sockets* simultáneamente. Conociendo la explicación teórica de cómo es el proceso de lectura de peticiones, se ha supuesto que usar varias conexiones simultáneas no conllevan una mejora, ya que el PLC responde a una petición por ciclo. Sin embargo, como se puede ver en la tabla 1, los resultados no han sido los esperados. Estos valores son los obtenidos tras hacer el mismo número de peticiones por *socket*, usando el módulo *threading* de Python.

Iteración \ **Conexiones**	1	2	3
1	38,769	55,436	75,423
2	38,991	55,967	75,539
3	38,348	55,639	76,302
4	37,948	55,998	75,794
5	38,230	55,244	76,022
Media	**38,457**	**55,656**	**75,81**

Tabla 1: Tiempos de respuesta del PLC ante varias conexiones (segundos)

El PLC parece ser capaz de responder a más de una petición simultánea por ciclo de programa. No obstante, los resultados no mejoran de manera proporcional al número de conexiones, sino que se aprecia un *speedup* mayor.

Cabe añadir que inicialmente las pruebas se habían hecho usando el módulo de *threading* de Python. Por su diseño, el GIL (*Global Interpreter Lock*) evita que los hilos se repartan entre las CPU del computador. En esencia, aquellas tareas en las que

predominen operaciones de CPU será mejor usar el módulo de multiproceso. Por el contrario, si la principal tarea conlleva esperas de entrada/salida, el módulo *threading* será el apropiado. En este caso, por ser esperas de red, se ha empleado *threading*. A pesar de esto, al cambiar la prueba para usar la librería de *multiprocess*, se han obtenido tiempos ligeramente más bajos que usando *threading*. Esto se ha achacado a este funcionamiento específico de Python.

3.4. Protocolo SLMP

SLMP es un protocolo de comunicaciones desarrollado por CC-Link para englobar todos los diferentes protocolos específicos que usan los dispositivos entre sí. Durante el desarrollo de este trabajo, se ha usado como un protocolo de capa de aplicación, aunque tiene la capacidad de funcionar en la capa de transporte. En los casos en los que el despliegue de red que se implementa entre los PLC y demás dispositivos sea usando protocolos de la familia de CC-Link, SLMP puede ser implementado en la capa de transporte. Su implementación está basada en el paradigma de cliente/servidor.

Este protocolo permite comunicaciones con dispositivos, independientemente de la jerarquía de red que esté desplegada. Como se ve en la figura 6, un dispositivo PLC usa un protocolo de CC-Link cuando se comunica con otro PLC. Para acceder a los registros de memoria no hace falta saber cómo se comunican los dispositivos entre sí, o cuál es su configuración de protocolos en las capas inferiores.

Esto se traduce en una mayor flexibilidad ante cambios, como actualización de los modelos de PLC o la posibilidad de que comunicaciones que típicamente se hacen de manera directa entre dos PLC pasen a ser gestionadas por el *software* de SLMP.

El formato de trama que se usa para peticiones es de longitud variable, pues permite usar ASCII o binario para generar la trama. En nuestro caso se usará binario, pues permite mandar más información por trama. Al usar ASCII se ocupa el doble de tamaño por unidad de información: un campo que en binario se especifica como '0x50 0x00' (dos *bytes*), en ASCII es 0x35 0x30 0x30 0x30, siendo cada *byte* la representación hexadecimal del caracter correspondiente. Este aumento del tamaño del mensaje también implica que el tiempo de transmisión del mismo se duplica, pues se tienen que mandar aproximadamente el doble de *bytes* para pedir lo mismo.

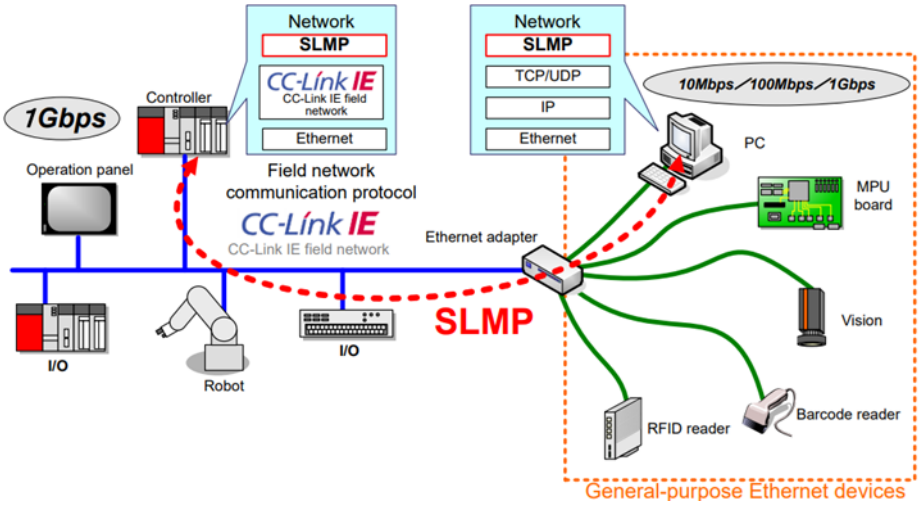


Figura 6: Ejemplo de arquitectura [13]

Por otra parte, usar ASCII no conlleva ninguna ventaja aparente de cara a la comprensión del protocolo, pues es necesario conocer la representación binaria de un campo igualmente.

El protocolo SLMP permite realizar una comunicación de dos maneras diferentes: en modo de transmisión única o múltiple. La transmisión única consiste en ir respondiendo a las peticiones de forma ordenada, esperando a la respuesta tras cada petición. Cuando se manda un mensaje, el PLC lo recibe, procesa la respuesta y contesta. Para el modo de transmisión múltiple, se añaden dos *bytes* en un campo, con el número que identificará al mensaje. Este modo permite enviar varios mensajes de petición consecutivos y escuchar las respuestas de manera asíncrona. El uso de este modo requiere mayor complejidad a la hora de mantener un control sobre los mensajes que se mandan, y no tiene ninguna ventaja significativa respecto a mandarlo de uno en uno. Esto se debe al propio funcionamiento del PLC: si, por ejemplo, existiesen peticiones de lectura que necesitasen varios ciclos de programa del PLC, y otras que se respondiesen en un solo ciclo, el uso de envío múltiple tendría más sentido, ya que se evitaría el bloqueo del PC mientras espera la respuesta, minimizando los tiempos de espera a E/S (Entrada/Salida). En peticiones de lectura/escritura no hay ninguna que aumente el tiempo de respuesta más que otra, 'bloqueando' el PC mientras está esperando una respuesta.

Por estos motivos, se usará el modo de transmisión única en binario para establecer comunicaciones.

3.4.1. Mensaje de petición

Todas las longitudes de los campos de las tramas se especificarán para mensajes usando el código binario, no ASCII. La figura 7 expone un diagrama con los campos listados.

Header	Subheader	Request destination network No.	Request destination station No.	Request destination module I/O No.	Request destination multidrop station No.	Request data length	Monitoring timer	Request data	Footer

Figura 7: Mensaje de petición [14]

- ***Header:*** Este campo se añade automáticamente. Normalmente se corresponde con la cabecera de la capa previa; en caso de que SLMP vaya sobre TCP, sería la cabecera de TCP.

- ***Subheader:*** (2 *bytes*, 6 en el caso de que el envío sea con transmisión múltiple) Indica que el mensaje es de petición. Si el modo de transmisión de mensajes es múltiple, se puede añadir un índice en este campo para que el PLC haga lo propio y responda usando ese mismo índice. De esta manera, se pueden identificar los mensajes aunque se manden/reciban en desorden.

- ***Network No.:*** (1 *byte*) Red donde se encuentra el dispositivo. Esta red no se corresponde con las direcciones IP, sino con el mapeado propio que tienen los dispositivos. Tiene valores entre 1 y 239. El 0 se reserva para la pareja network/station

básicas, que son las que se usarán para este proyecto (las comunicaciones suelen ser con un solo PLC).

- ***Station No.:*** (1 *byte*) Dispositivo (PLC) dentro de la red al que se van a mandar las peticiones. 255 indica que es la primera estación conectada a la red.
- ***Module I/O No.:*** (2 *bytes*) CPU del PLC que se encargará de gestionar la petición. Existen PLC con más de un módulo de CPU, por lo que se pueden direccionar usando este campo. Al igual que los campos anteriores, se dejará en 0, pues no hay múltiples CPU en los PLC con los que se ha trabajado.
- ***Multidrop station No.:*** (1 *byte*) Al igual que el campo *Network No.*, se corresponde con una configuración de red propia de los PLC. Se usa para casos en los que a la red en la que está el PLC al que se quiere acceder solo se puede llegar saltando a través de otro PLC, una especie de *daisy chain* específico a estos mapeados de red. Por los motivos expuestos anteriormente, no es necesario usar este campo.
- ***Request data length:*** (2 *bytes*) Tamaño de los dos siguientes campos, *timer* y *request data*, en *bytes*.
- ***Monitoring timer:*** (2 *bytes*) Tiempo que el dispositivo al que le llega una petición debe dedicar a responder. Si el tiempo excede el marcado en este campo, no se continua con el procesado, por lo que la petición queda sin responder. Para tiempo ilimitado, es decir, lo que el PLC tarde en procesar la petición, se puede indicar 0. En caso de querer un tiempo específico, cada incremento se corresponde con 250 ms. Poner este campo con un valor de 4 haría que se espere un máximo de 1 segundo.
- ***Request data:*** (*n bytes*) Este campo está compuesto de otros, dependiendo del tipo de petición que se esté haciendo. Típicamente contiene un comando, subcomando y datos. Se explica con mayor detalle en la sección 3.4.3.
- ***Footer:*** Al igual que la cabecera, se añade automáticamente en una capa posterior.

3.4.2. Mensaje de respuesta

Para el mensaje de respuesta hay dos variantes: la respuesta normal a una petición, cuando se ha ejecutado correctamente, y la respuesta de error. La figura 8 tiene los campos correspondientes, esta vez, al mensaje de respuesta.

Header	Subheader	Request destination network No.	Request destination station No.	Request destination module I/O No.	Request destination multidrop station No.	Response data length	End code	Response data	Footer

Figura 8: Mensaje de respuesta [14]

Los campos *Header*, *Network*, *Station*, *Module*, *Multidrop Station* son iguales que en el mensaje de petición.

- ***Subheader:*** (2 *bytes*, 6 en el caso en que el envío sea de transmisión múltiple). Indica que el mensaje es de respuesta. De forma análoga a la petición, si el modo de transmisión es múltiple, llevará en los dos *bytes* centrales el índice que identifica este mensaje como respuesta a una petición con ese mismo índice.
- ***Response data length:*** (2 *bytes*) Análogo a la petición, tamaño de los dos campos siguientes *End Code* y *Response data*.
- ***End code:*** (2 *bytes*) Código de finalización de petición. Si ha habido un error en el formato de la petición o durante el procesado de la misma, se indica en este campo. Si la petición se ha realizado correctamente, este campo vale 0. Cuando hay errores, el campo siguiente contiene la información referente al comando y subcomando que han causado el error, como el tipo de comunicación que se va a usar es simple (pregunta, respuesta) ya sabemos de antemano el mensaje que ha generado el error, por lo que estos campos pierden su utilidad.
- ***Response data:*** (*n bytes*) Datos de respuesta a una petición. En caso de error, esta zona tiene los datos de red de la estación que ha dado error, junto al comando y subcomando que se mandaron en la petición.

3.4.3. Comandos implementados

El campo de datos de un mensaje de petición está compuesto normalmente por un comando, un subcomando y una sección con los datos referentes a los registros o partes del PLC con las que se quiere interactuar. De todos los comandos que SLMP dispone, se van a usar los de escritura/lectura en bloques. Para explicar este comando hay que entender primero cómo se accede a una zona de registros del PLC. A cada uno de los accesos que requieren de estos campos se le llamará 'bloque'. Un bloque tiene las siguientes partes:

- **Registro**: nombre del registro.
- **Posición inicial**: número de registro a partir del cual se va a leer.
- **Nº de registros**: número de posiciones de memoria consecutivas a partir del primer registro.
- **Datos** (opcional): solo aparece en el caso de peticiones de escritura. En este campo se añaden los valores que se deben guardar en los registros.

Los comandos que se han usado se basan en esos campos básicos para juntar varios bloques y hacer una petición a más de un registro cada vez. Al añadir un número arbitrario de bloques en cada petición, son necesarios dos campos nuevos (1 byte cada uno) para indicar el número de bloques que va a haber para registros de tipo bit y de tipo *word*.

El protocolo SLMP impone una serie de restricciones a la cantidad de registros que se pueden pedir. Para una petición de lectura se pueden acceder a un máximo de 960 registros repartidos en, como mucho, 120 bloques. Para una escritura, el número de bloques se mantiene, pero el total de registros que se escriben no puede superar los 770. En la sección 4.1.5 se explican las implicaciones de estas restricciones.

Para hacer lecturas/escrituras a un solo registro, hay un comando específico que permite pedir n posiciones de memoria consecutivas a partir de un punto inicial en un

registro. Por ejemplo, si se quieren leer los registros R100, R101 y R102, los datos que van en el mensaje de petición son el código del registro R, la posición inicial, 100, y el número de posiciones consecutivas que se quieren leer, en este caso, 3.

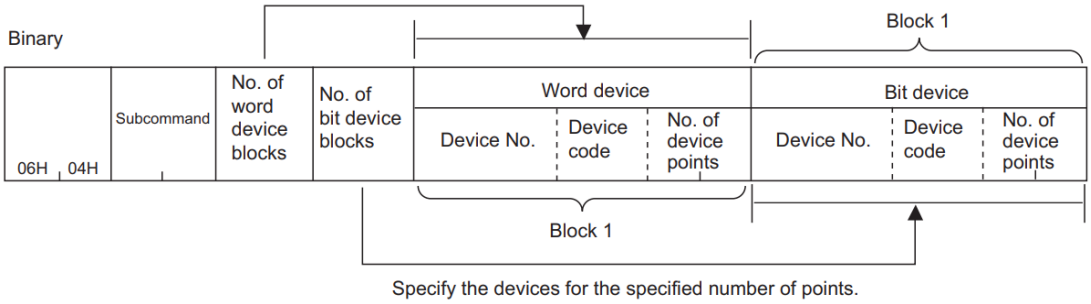


Figura 9: Datos de una petición de lectura [14]

En la figura 9 se especifica la estructura de esta petición. Usando el ejemplo anterior, la tabla 2 muestra un ejemplo de cómo quedan los campos. Al no querer pedir ningún registro de tipo bit, el mensaje termina con el campo *No. of device points*.

Campo	**Valor**	**Descripción**
No. of word device blocks	1	Un solo bloque de tipo word
No. of bit device block	0	Ninguno de tipo bit
Device No.	0	Posición 0
Device Code	90	El código del registro R
No. of device points	3	Registros, *points*, consecutivos

Tabla 2: Ejemplo de petición

En el caso de las peticiones de escritura, la estructura del mensaje es idéntica, con la única diferencia de que se añaden los datos que hay que escribir. Es necesario agregar un campo después de cada bloque que contenga los valores que van a ser escritos. La figura 10 muestra la estructura del mensaje. En ella se puede ver cómo se mantiene el mismo orden en los campos.

Siguiendo la misma configuración del ejemplo mostrado en la tabla 2, si se quiere escribir '1, 2, 3' el mensaje queda de la siguiente forma: al ser tres registros de 16 bits cada uno, hace falta añadir 6 *bytes* con los datos de cada registro (0x01, 0x00, 0x02, 0x00, 0x03, 0x00).

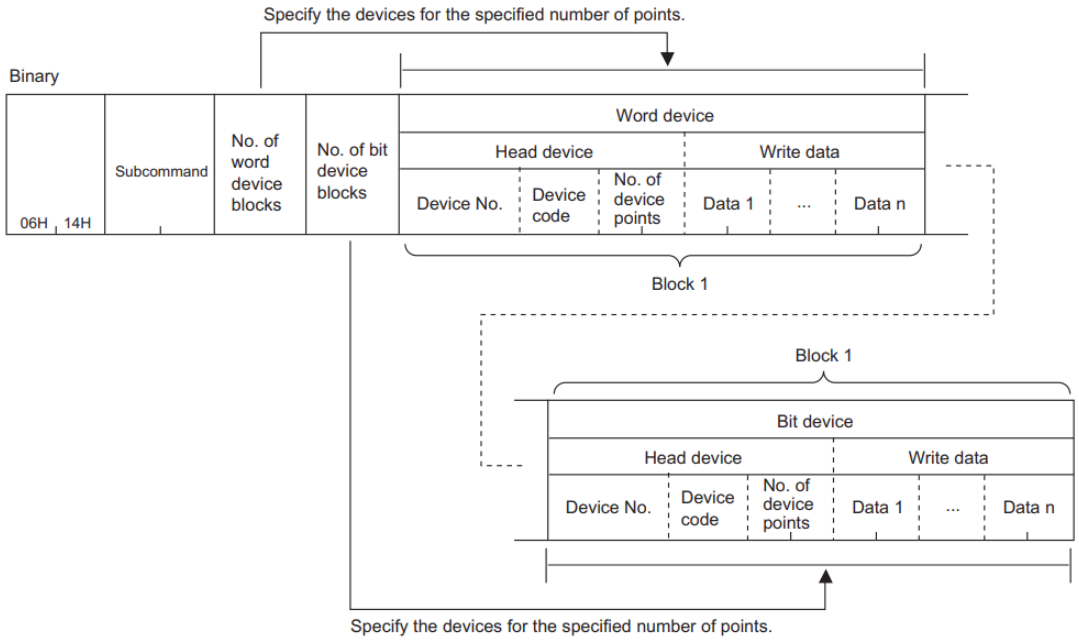


Figura 10: Datos de una petición de escritura [14]

4 Estructura y desarrollo

Este apartado se centra en el proceso seguido durante la etapa de desarrollo. La biblioteca ha sido diseñada siguiendo la estructura de paquetes de Python, lo cual está explicado con mayor detalle en la primera sección de este apartado.

El capítulo está dividido en dos partes diferenciadas: en la primera se detalla la biblioteca de SLMP en sí; posteriormente, se explica el desarrollo de una aplicación que incorpora Pyslmp como parte del flujo de datos. De esta manera, se pretende mostrar el funcionamiento en un entorno que simule una situación real de uso.

4.1. Pyslmp

Aunque no hay una norma específica al respecto, una gran proporción de paquetes de Python tienen '*py*' como sufijo o prefijo en su nombre: numpy, scipy, pytorch... Por darle un nombre en sintonía con este estilo, se ha elegido Pyslmp para la biblioteca.

4.1.1. Requisitos de la biblioteca

Tras exponer el sistema sobre el que se va a trabajar y las necesidades que busca cubrir, se han extraído una serie de requisitos. Para conseguir cubrir los objetivos, es necesario presentar dos funciones principales al usuario de la biblioteca: la de escritura y lectura. La tabla 3 lista los requisitos funcionales extraídos.

RF01	Las peticiones deben poder acceder a múltiples registros a la vez.
RF02	Se podrá escribir y leer usando variables mapeadas a un nombre cualquiera.
RF03	La biblioteca puede cambiar la conexión al PLC por medio de llamadas a la API.
RF04	Se podrán direccionar todos los tipos de dato básico que tiene el PLC.
RF05	Las propiedades de una conexión cualquiera se pueden cambiar en tiempo de ejecución, como *timeouts* o direccionamiento.

Tabla 3: Requisitos funcionales de la biblioteca

Para cumplir estos requisitos, hay una serie de funcionalidades extra que son necesarias. Entre ellas:

- Extracción de un archivo con el mapa de memoria del dispositivo.
- Creación de un sistema que permita realizar una suscripción a una serie de variables y que posibilite tener información actualizada sobre las zonas de memoria deseadas.

- Mostrado de datos de manera dinámica, por separado de la conexión en sí.

4.1.2. Estructura de Pyslmp

La organización de la biblioteca se ha inspirado en otras de uso común, buscando una estructura que se ajuste a los paquetes de Python típicos. Este enfoque facilita la distribución del código o su integración en otros programas. Para conseguir esto, la estructura de ficheros sigue unas directrices básicas definidas en la documentación oficial de Python [15]. Esta documentación define un archivo cualquiera con extensión *.py* como un módulo, y un directorio que contiene uno o más módulos como un paquete. Para que un directorio pueda ser interpretado como un paquete, es necesario añadir un archivo *___init___.py* en el que se pueden declarar ciertos parámetros, como las funciones o módulos que se quieren importar cuando se llama al paquete entero desde otro punto del código.

La biblioteca se puede distribuir como un paquete, permitiendo la importación de la misma desde otros módulos. Para conseguir esto se ha creado un directorio principal, *pyslmp/*. Así mismo, el punto de entrada de la biblioteca se encuentra en este nivel. El módulo se ha llamado *slmpClient.py*. Servirá como lugar donde se puedan añadir las llamadas de usuario que la biblioteca ha necesitado. La clase principal de Pyslmp está aquí definida. En ella se encuentran las funciones de la API como lectura, escritura o creación de la conexión, entre otras.

El proyecto se ha estructurado siguiendo una agrupación por directorios, según su relación lógica:

- **src/** Código fuente de la parte de Python. Contiene los módulos responsables de la lógica de la biblioteca. En particular forman la parte dinámica de las peticiones de escritura y lectura. En general, contiene todo el núcleo de la biblioteca, desde funciones auxiliares hasta la clase que realiza una conexión TCP.

- **c-src/** Código de la biblioteca base en C. Implementa la parte 'estática' del protocolo SLMP. Expone una API sobre la que hacer llamadas desde un *wrapper* en Python.

- **tests/** Módulo con la batería de tests creados para probar el funcionamiento de los casos de uso. Contiene archivos de configuración extra que permiten tener un entorno de pruebas controlado, minimizando elementos ajenos al núcleo de la bibioteca.

- **settings/** Archivos de configuración. Contiene los parámetros iniciales de configuración y las constantes que se usarán durante tiempo de ejecución. Adicionalmente, tiene el entorno de *logs* y el directorio donde se almacenan comprimidos. Este entorno permite configurar el tiempo que se guardan antes de ser eliminados.

La biblioteca base de C, ya proporcionada en la documentación de CC-Link, está compuesta por la parte más estática del protocolo. Define una estructura de datos con todos los campos de la petición, dejando el de datos, que es el más relevante para poder hacer peticiones de todo tipo, como un *array* a llenar por el usuario.

Los primeros campos de un mensaje están mapeados en un archivo de constantes, por lo que pueden estructurarse de manera sencilla. Sin embargo, la parte de datos puede variar enormemente, ya que depende enteramente de la petición de acceso a

memoria que quiera hacer un usuario. La lógica encargada de gestionar esta variabilidad es la que Pyslmp implementa, en Python. Una vez formados todos los campos, se mandan a un módulo que funciona como *wrapper* de la biblioteca en C. Esta biblioteca une todos los campos formando el paquete de datos a enviar posteriormente a través de una conexión TCP/IP.

Las dos funciones principales del módulo serán la de escritura y lectura, además de una serie de rutinas que se encarguen de realizar la conexión con el PLC y gestionar el envío y recepción de los paquetes. La figura 11 muestra el flujo de trabajo planteado para una petición de lectura. Como se puede apreciar, los procesos necesarios para llevar a cabo la petición requieren un estado, que se ha de mantener tras la respuesta, para poder 'descifrar' el mensaje resultante.

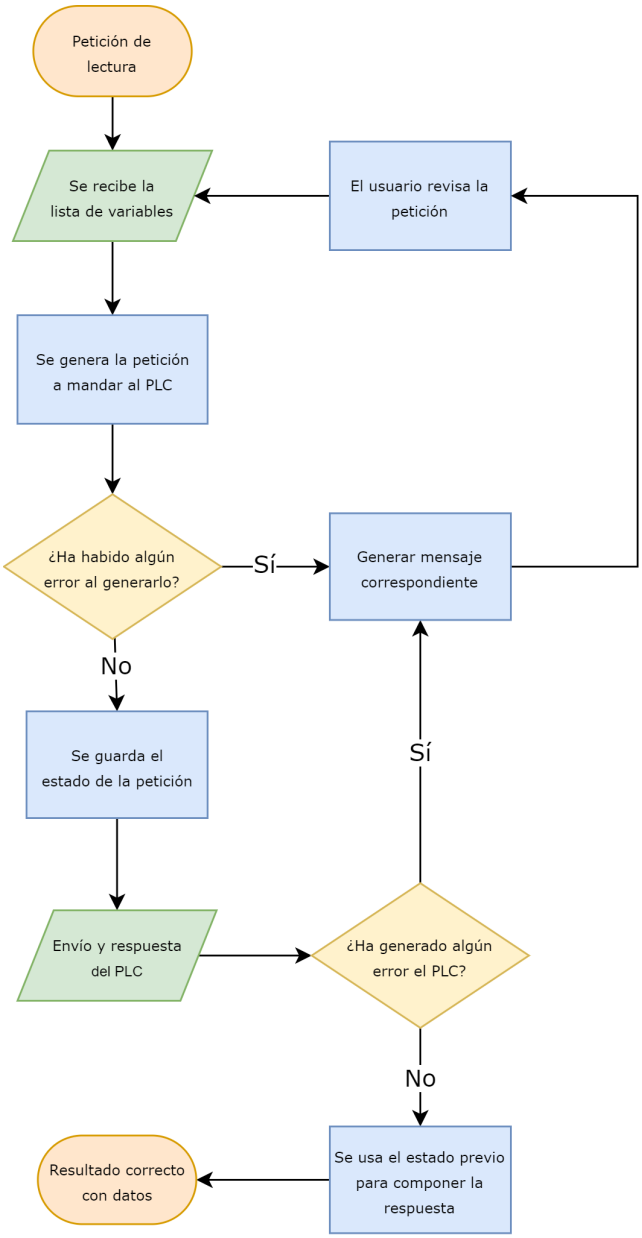


Figura 11: Flujo de trabajo planteado

La generación del mensaje que se quiere enviar al PLC requiere pasar los campos completos a la biblioteca de C, para que los ordene. Esta biblioteca se compila como un objeto compartido, *SLMP_lib.so* o *SLMP_lib.dll* en Linux y Windows, respectivamente. El compilador *GCC*, junto con los parámetros mostrados en la figura 12, ha sido usado para llevar a cabo esta tarea.

```
gcc -c -fPIC SLMP.c SLMP.h
gcc -c -fPIC python_reader.c python_reader.h
gcc -shared -o SLMP_lib.so SLMP.o python_reader.o
```

Figura 12: Compilación de la parte en C

Tras este proceso, ha sido necesario crear un *wrapper* en Python, que carga el objeto compartido en memoria. Así, se pueden acceder a las dos llamadas de la API:

- Ordenación de los campos para generar un mensaje de SLMP.
- Separación de un mensaje de respuesta de SLMP en los campos que lo componen.

Estos campos son los previamente explicados en los apartados 3.4.1 (para la ordenación), y 3.4.2 (para la separación).

4.1.3. Llamadas principales de Pyslmp

Pyslmp presenta una clase principal *SlmpClient*. Esta clase contiene las llamadas públicas de la biblioteca. Los parámetros básicos que se mandan a la clase son la dirección IP y el puerto TCP del PLC, o cualquier otro dispositivo que implemente un servidor SLMP. Una vez inicializada la clase se puede acceder a los métodos, además de modificar los atributos de la clase para apuntar a otras direcciones o añadir un *timeout*. A continuación se muestra un listado exhaustivo de las llamadas esenciales para cumplir los objetivos de la biblioteca:

- **connect(): bool**

  Tras la inicialización de la clase, crea una conexión TCP, devolviendo un valor booleano para el resultado de la función. Verdadero si ha sido exitosa y falso si ha habido algún problema.

- **disconnect()**

  Cierra la conexión y elimina el objeto *socket* correspondiente.

- **reading_call(list): dict**

  Crea una petición de lectura por medio de una lista de variables que la función recibe como parámetro. A continuación, manda esta petición al PLC y lee la respuesta, devolviendo los valores asociados a cada variable.

- **writing_call(dict): dict**

  Crea una petición de escritura. En este caso se envía un diccionario, en el que las claves son los nombres de las variables y los valores corresponden a los datos que se quieren escribir. Finalmente, devuelve otro diccionario con el mensaje resultante de la petición.

- **parse_xml(string): bool**

  A partir del nombre de un archivo xml generado por el PLC, extrae el mapa de variables asignadas a cada registro de memoria.

El modo de comunicación diseñado se limita al uso del protocolo TCP, por las características del sistema. Al ser la implementación principal para sistemas de monitorización, se ha considerado más importante hacer disponible este tipo de conexión frente a UDP. Si bien este protocolo añade algo más de *overhead*, las ventajas que ofrece en la integridad de los datos y calidad de la comunicación en general supone una mejora considerable.

En la figura 13 se puede ver un ejemplo básico de uso de la biblioteca: este fichero se llama *prueba.py*. Tras la inicialización del cliente, se comprueba que la conexión exista y se procede a leer una petición arbitraria. En este ejemplo, el PLC se encuentra en la dirección IP 192.168.100.250, y tiene un servidor TCP de SLMP habilitado en el puerto 8192.

```
1  import sys
2
3  from pyslmp import SlmpClient
4
5  client = SlmpClient('192.168.100.250', 8192)
6
7  if not client.connect():
8      print('Error while connectig!')
9      sys.exit(0)
10
11 reading_vars = ['R100', 'VariableMapeada']
12 response = client.reading_call(reading_vars)
13
14 print(response)
```

Figura 13: Ejemplo de uso de pyslmp

La petición consiste en dos accesos a memoria: uno directo al registro R en la posición 100 y otro por medio de una variable que ya está mapeada. Suponiendo que la variable contenga un *string* de 5 caracteres, con la cadena 'hola!' guardada, el resultado de la petición es el siguiente:

```
1  \$: python prueba.py
2  >>> {'response':{'R100':[10], 'VariableMapeada':['hola!']},
3       'missingVariables:[]',
4       'error':[]}
5  >>>
```

4.1.4. Mapeado de variables

Las variables guardadas en el programa del PLC se definen durante la compilación del mismo. Esto permite a los técnicos trabajar más cómodamente en la realización de

código, usando estructuras definidas previamente que evitan tener que usar zonas de memoria directamente por su nombre (R100, M23, …).

El PLC que se ha usado para el desarrollo, y todos los de la familia, tienen una serie de tipos de datos base que se pueden asignar a una zona de memoria. Estos tipos son los siguientes:

- **BOOL:** Booleano, los registros de tipo bit solo pueden contener este tipo de dato. Además un registro de tipo palabra (16 bits) puede contener un *array* de hasta 16 bits, que se corresponden con los bits menos significativos del registro al que esté asignado.
- **WORD / DOUBLE WORD:** Representa un entero sin signo. Ocupa uno o dos registros (16 o 32 bits), respectivamente.
- **INT / DOUBLE INT:** Representa un entero con signo. Ocupa uno o dos registros (16 o 32 bits), respectivamente.
- **REAL:** Representa un real de precisión simple (IEEE 754). Ocupa dos registros de tipo palabra.
- **STRING:** Cadena de caracteres, de longitud variable, en la que cada uno de ellos ocupa un byte.

Para crear un formato que sea utilizado por Pyslmp se han usado estos tipos básicos, junto con la información de variables del PLC. Esta información se extrae por medio de una opción que tiene el *software* de programación de los PLC. El archivo XML resultante es poco tratable, ya que puede contener cientos de miles de líneas. Tras invocar a la llamada correspondiente de pyslmp, se ha diseñado un formato JSON simplificado de estas variables.

Cuando se hace una petición de lectura/escritura al cliente de pyslmp, cada nombre de variable se resuelve consultando el archivo JSON que contiene toda la información extraída del PLC. Este archivo se carga en memoria al principio de la ejecución de la biblioteca. A pesar de usar este archivo como un método de configuración estática, se ha definido una clase de datos para poder modificar el valor de estas variables de manera dinámica en tiempo de ejecución. Para conseguir esto, se pueden exponer las variables que haya cargadas en memoria, permitiendo añadir, borrar o alterar las que se encuentran.

El archivo JSON sigue una estructura plana, por lo que todas las variables son una entrada única en el mapa de memoria. Se ha diseñado así, ya que se supone que el caso de uso típico de la biblioteca estará compuesto de accesos a variables mapeadas. El uso de un mapa supone una mejora de la eficiencia frente a soluciones que impliquen iterar hasta encontrar la variable. Por diseño, los mapas tienen tiempos de acceso a una pareja clave-valor de $O(1)$.

4.1.5. Cálculo del tamaño máximo de trama

Para calcular el tamaño máximo de trama que se va a mandar durante la conexión, se han usado los límites de mensaje establecidos por el protocolo. Este tamaño es relevante para la gestión de memoria en la parte de C, ya que define una cota máxima al mensaje que se quiere enviar/recibir. Esto ha permitido detectar errores durante el desarrollo de la biblioteca cuando los límites calculados se superan.

Petición de lectura

Sabiendo los tamaños de cada campo, definidos en el capítulo 3, tenemos que el tamaño de la parte común (*Subheader, NetNumber, DestinationNumber, DestinationModuleNumber, MultidropStation, DataLength, Command, Subcommand, Timer*) son 15 bytes. Al ser 120 bloques el máximo número que se puede enviar, dan igual las posiciones consecutivas que se pidan en cada uno de ellos, ya que por cada bloque siempre se añaden 7 bytes al mensaje (3 para la posición, 2 para el registro, 2 para el número de posiciones consecutivas de memoria). Es decir, da igual que pidamos la posición R100 que 10 posiciones a partir de R100: en ambos casos el mensaje sigue ocupando los mismos bytes.

De esta manera, siendo el tamaño en bytes $T_{total} = 15 + 7 * 120$, tenemos que un mensaje de lectura puede llegar a ocupar 855 bytes.

Petición de escritura

A diferencia de la lectura, una petición de escritura se maximiza cuando se envía el número máximo de registros individuales. Sabiendo que el máximo son 770 posiciones de memoria, repartidas en 120 bloques, podemos crear un mensaje que pida escribir un registro en 119 bloques, y una ristra de 651 registros en el último bloque.

El tamaño total del mensaje se puede calcular como $T_{total} = 15 + 120 * 7 + 770 * 2$, llegando a 2395 bytes.

4.1.6. Mantenimiento de estado

Para gestionar las peticiones de lectura, ha sido necesario crear una clase *Status*. Esta clase contiene la información crítica para poder interpretar la respuesta del PLC. Como se explica en la sección 3.4.3, los dos comandos principales que implementa la biblioteca tienen dos campos que especifican el número de registros tipo word y tipo bit, en ese orden, que se van a mandar en la petición.

En una petición cualquiera de usuario, las variables que pida no tienen por qué ir ordenadas siguiendo este formato. Durante la generación de la zona de datos, es necesario ordenar la entrada de usuario en dos listas, una para accesos a registros de tipo word y otra para accesos a registros de tipo bit. Una vez se tienen estas dos listas, se guardan como el estado de petición. Cuando llega la respuesta del PLC, se procede a leer los datos siguiendo el orden de variables guardado en estas listas de la clase *Status*. Entre otras cosas, el estado es el encargado de indicar a la función de lectura del *socket* el número de bytes que se esperan de la respuesta, en caso de que no haya errores.

4.1.7. Tratamiento de errores de Pyslmp

Inicialmente, el proyecto plantea dos capas de errores separadas: aquellos generados por el protocolo SLMP y los definidos como parte del módulo. Para errores del protocolo, la gran mayoría se pueden controlar antes de que se mande el mensaje, por lo que quedan más como apoyo al desarrollo y extensión de la biblioteca que como errores que se elevan hasta el usuario. No solo eso: estos errores son dependientes del dispositivo con el que se está comunicando, por lo que intentar mapear estos códigos se sale del alcance de este trabajo, teniendo en cuenta que cualquier fabricante puede crear su lista de códigos de error de manera arbitraria. En general, si el campo de la

respuesta *End code* es 0, no ha habido errores durante la ejecución. En caso de que sea diferente es necesario mirar el mensaje de error generado por el PLC.

Teniendo en cuenta lo comentado anteriormente, la biblioteca expone una serie de excepciones definidas y documentadas. Estas excepciones siguen el criterio típico, considerando los casos que constituyen un error debido a la mala formación de una petición por parte del usuario.

Varios de los errores que el protocolo define se pueden capturar previamente por la biblioteca desarrollada. Además, se han añadido algunos que son propios al proceso de serialización de variables. La tabla 4 muestra un listado de los errores que pyslmp devuelve en caso de que una petición no se haya podido completar correctamente. Además, se da una indicación de las acciones a tomar para corregir estos errores.

Nombre de error	**Descripción**	**Acción correctiva**
SLMPError	Clase principal de errores, sirve a modo de *catch all* para cualquier error.	-
EmptyRequestError	El usuario ha lanzado una petición sin variables que mandar al PLC.	Revisar la lista de variables que se ha pasado.
NotAStringError	La variable pasada en la petición de escritura/lectura no sigue el formato establecido para pedir datos.	Revisar el código que llama a la biblioteca con la lista de variables.
TooManyPointsError	Hay demasiadas variables para una sola petición.	Separar la petición en otras más pequeñas.
RegisterError	El registro pedido no está dentro de los soportados por la biblioteca/PLC.	Revisar la lista de variables que se han pedido.
ShapeMismatchError	En el caso de la petición de escritura, los datos que se deben escribir y el tamaño de la variable mapeada no concuerdan.	Revisar el dato que ha generado el error.
PackingError	Algún dato no se ha podido convertir a *bytes*.	Revisar el dato que ha generado el error.
ConnectorError	Error de conexión con el PLC.	Revisar el mensaje de error, comprobar el enlace entre ambos extremos.

Tabla 4: Listado de errores de la biblioteca

Como se puede apreciar, se ha creado un error principal, *SLMPError*, como clase de la que heredan los demás. Usar un bloque *try-catch* con esta clase permite capturar cualquier error que genere la biblioteca.

4.2. Aplicación completa

Para dar una visión más completa sobre cómo se puede integrar la biblioteca pyslmp en otros proyectos, se ha creado un flujo de datos completo. El modelo usado para el desarrollo es el de tres capas. Esta clasificación divide un proyecto de *software* en secciones con características diferenciadas del resto. La capa principal, que es la que integra pyslmp, es la capa lógica. Junto a esta, se han usado dos componentes más, para guardar la información en una base de datos y mostrarlos posteriormente. En las siguientes subsecciones se explican estas dos capas, y su integración entre sí.

4.2.1. Capa de datos

Como complemento y prueba al funcionamiento de la biblioteca, se ha usado una base de datos para guardar valores que se leen. Esto se corresponde con la capa de datos de una aplicación típica que siga el modelo de tres capas. La biblioteca por sí misma no presenta esta capa, ya que toda su configuración se puede crear en tiempo de ejecución.

Para este caso en particular, se ha optado por InfluxDB, al ser una base de datos específica para guardar series temporales. Sigue un esquema NoSQL, que usa la fecha, o *timestamp*, como índice de cada serie [16]. En este caso, la estructura de la base de datos es similar a la de un mapa, teniendo la información guardada por medio de una tupla (clave, valor). El tipo de dato que se guarda en Influx se denomina *point*. Cada una de las entradas tiene varios valores asociados, como *tags* o el *timestamp*. Estos valores, a su vez, forman otro mapa con sus claves propias. De esta manera, permite asociar información haciendo distinciones. La metainformación puede ser la misma para dos puntos (su nombre de campo, el timestamp...) pero provenir de dos fuentes diferentes; para hacer esa diferenciación se usa el campo *tag*, que añade una etiqueta definida por el usuario.

En particular para este proyecto hay dos motivos claros que han sido fundamentales para elegir esta tecnología:

- La definición de la base de datos es básica, no hay una complejidad alta en el uso de metainformación, como mucho el nombre del PLC desde el cual está llegando la información. Esto se traduce en pocos movimientos de la estructura, siendo el uso principal añadir filas a las tablas preestablecidas.

- El tipo de dato que se va a guardar es eminentemente temporal. Las variables de uso común del PLC son relativas al dato instantáneo de sensores y cálculos derivados de los mismos.

Para llevar a cabo esta implementación, ha sido necesario añadir un componente extra. La biblioteca no implementa conexiones a bases de datos en su versión actual. Este componente sigue el modelo DAO (Data Access Object).

Este modelo expone una capa de abstracción entre las funcionalidades de la biblioteca, capa lógica, y el acceso a bases de datos. Así, en caso de que haya un cambio en cualquiera de las dos partes, siguen separadas para minimizar los costes de mantenimiento y asegurar la separación lógica de las partes de la aplicación. Si alguna capa cambia, bien por uso de una base de datos diferente o por cambios en la manera de recibir información, se evita tener que modificar partes no relacionadas.

El DAO implementa una clase genérica que actúa como interfaz para ser implementada por clases hijas, específicas a cada base de datos que se quiera añadir. En este caso, solo es necesaria la conexión a una base de datos de Influx. Para ello, se ha usado el paquete oficial de Python que InfluxData mantiene. Este paquete tiene las funciones necesarias para conectarse a la API de Influx y mandar datos.

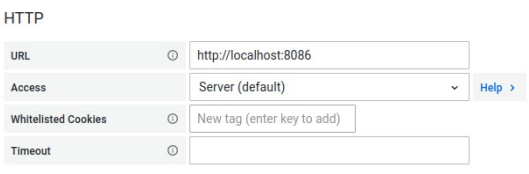
4.2.2. Capa de presentación

De manera análoga a la capa de datos, se ha usado Grafana, una herramienta de creación de paneles de gráficas, o *dashboards*.

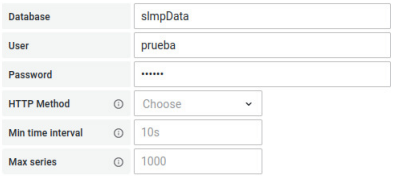
Al igual que la base de datos de Influx, Grafana puede ser instalada en una máquina local por medio de su distribución de código abierto. Esta versión se configura de manera local, quedando instalada como un servicio web al que conectarse desde el navegador.

Esta aplicación permite realizar una conexión a la base de datos de Influx, para más tarde proporcionar una interfaz gráfica de visualización de los datos. La figura 14 muestra la interfaz que presenta Grafana para introducir la información del servidor donde se encuentra la base de datos. Tras realizar la conexión, la base de datos queda accesible desde la página principal de visualización.

Un *dashboard* se puede configurar con las consultas básicas para acceder a las variables deseadas, ya que la aplicación ya da todas las facilidades para moverse por un rango de tiempo cualquiera, si hay datos para mostrar. Este tipo de visualización sirve para explorar los históricos en busca de variaciones interesantes, o simplemente para comprobar que el funcionamiento de una variable es el esperado.



(a) Conexión al servidor local de Influx



(b) Conexión a la base datos

Figura 14: Conexión de Grafana a Influx

Entre otras de sus funcionalidades, Grafana ofrece la posibilidad de explorar una base de datos. Ofrece un generador de consultas dinámico, que permite cambiar las partes de la misma. Esta funcionalidad facilita la exploración y verificación de los datos que se reciben, sin necesidad de generar *dashboards* específicos para ello.

En la configuración de Grafana, se puede modificar un parámetro que controla el tiempo de refresco mínimo de los *dashboards*. Tras cambiarlo a un valor bajo como 100 ms, las gráficas actualizan sus datos continuamente, dando la impresión de obtener información en tiempo real. La figura 15 muestra dos paneles con la query correspondiente a cada una de las variables que se han guardado en Influx. Estos paneles se han configurado para mostrar los últimos minutos, con lo que se consigue bastante resolución en los datos.

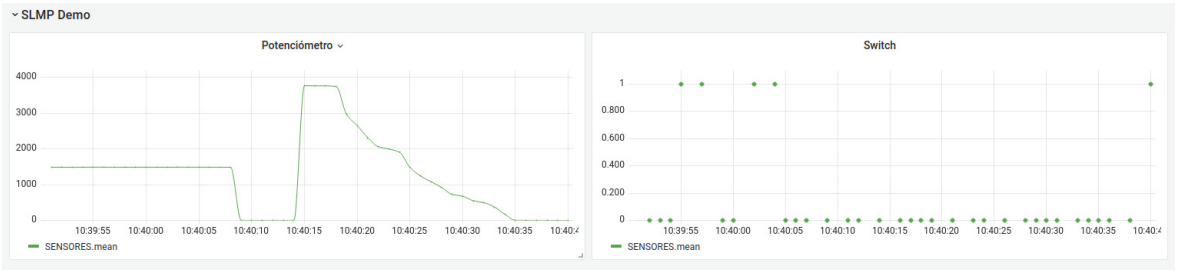


Figura 15: Ejemplo de dashboard de Grafana

5 Resultados y pruebas

Este capítulo tiene como objetivo mostrar el proceso de pruebas que se han implantado durante el desarrollo, y las baterías de test definidas para mantener la funcionalidad básica del proyecto cuando se le hagan cambios y/o extensiones. Las dos primeras secciones hablan sobre las pruebas unitarias y de aceptación, respectivamente.

Adicionalmente, se han llevado a cabo una serie de pruebas de rendimineto, *benchmarks*, para obtener métricas sobre los tiempos de respuesta de la biblioteca. Los resultados se puede encontrar en la última sección de este capítulo.

5.1. Pruebas unitarias

Las pruebas unitarias son aquellas que se ejecutan sobre partes concretas y aisladas del código, típicamente sobre clases o métodos que hacen una tarea concreta y acotada. Este tipo de pruebas tiene como objetivo detectar defectos en las funcionalidades programadas [17].

Entre las diferentes técnicas que hay para realizar pruebas, destacan las pruebas de caja negra y de caja blanca. Ambas se han implementado durante la fase de pruebas. Los test de caja negra son aquellos que prueban el funcionamiento de una parte del código sin saber exactamente cómo es la implementación. Sirven para comprobar las funcionalidades generales que se esperan de cada parte de un programa. Por el contrario, los test de caja blanca sí saben cuáles son los algoritmos usados en la implementación, por lo que suelen probar todos los casos de error definidos como parte del método.

Para poder implementar parte de las pruebas adecuadamente, hay varias cosas a tener en cuenta. Las pruebas de caja negra sobre las llamadas principales de E/S precisan de un PLC conectado para poder enviar y recibir. La manera ideal de estructurar esta parte consiste en desarrollar una clase auxiliar que simule una serie de respuestas concretas que el PLC genera. Esta clase que simula la interfaz de comunicación se llama *mock*. No obstante, para no extralimitar los recursos dedicados al proyecto, la fase de pruebas se ha centrado en la creación de pruebas del sistema en sí, usando un PLC configurado que tenga la comunicación SLMP habilitada.

Para el testeo de las funcionalidades se ha creado una suite de tests siguiendo el formato estándar de Python, con el paquete estándar *unittest*. Este paquete, inspirado en la biblioteca de Java *JUnit*, facilita la generación de pruebas, teniendo funciones encargadas de tareas generales. Por ejemplo, automatizar partes de las pruebas, o generar métodos de configuración y finalización comunes a todos los casos de prueba [18].

Todas las pruebas se han ejecutado sustituyendo el archivo que mapea las variables por uno estático y controlado, sobre el que se conocen de antemano los resultados esperados. Este proceso forma parte de la configuración inicial de la batería de pruebas.

La figura 16 muestra la declaración de la clase y el método *setUpClass*. Este método se ejecuta antes de lanzar todos los tests de esta clase.

```
1  import os
2  import json
3  import unittest
4
5  import pyslmp.settings.slmp_settings as slmp_settings
6
7  class TestModule(unittest.TestCase):
8      """
9      Black box suite of tests, mainly used to confirm that the
10     basic functionality of the module is met. (Readings, writings,
11     and handling of errors)
12     """
13
14     @classmethod
15     def setUpClass(cls):
16         f = open(os.path.join(slmp_settings.BASE_DIR,
17                               'tests',
18                               'test_variables.json'))
19         test_vars = json.load(f)
20         f.close()
21         slmp_settings.PLC_VARIABLES = test_vars
```

Figura 16: Declaración de una clase de tests

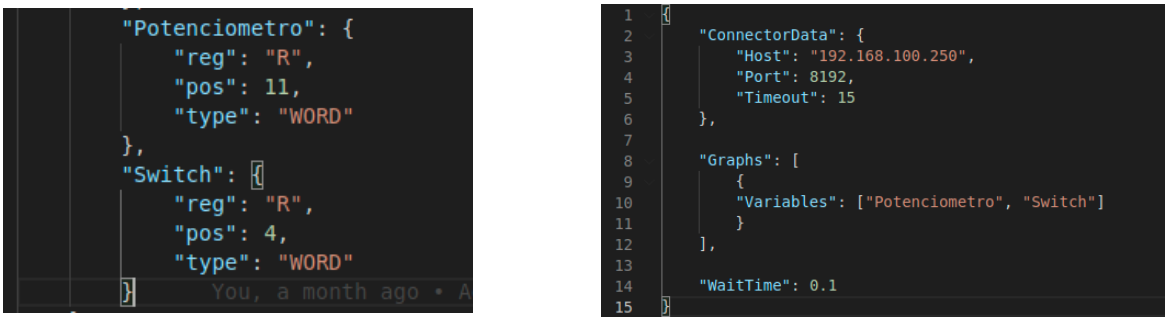
La clase TestModule, mostrada en el ejemplo, contiene los tests de caja negra. En general, las pruebas que se hacen se lanzan contra toda la biblioteca por ejemplo, la escritura de un número en un registro y su posterior lectura.

5.2. Pruebas de aceptación

Las pruebas de aceptación son aquellas en las que el usuario final es el encargado de comprobar si se cumplen los requisitos definidos. En este caso, al ser un proyecto interno de la empresa, es el departamento técnico el encargado de usar la biblioteca y comprobar su funcionamiento.

Para hacer esto, se ha proporcionado un manual de usuario que sirve como documentación interna de cómo usar el módulo. Un extracto se puede encontrar en el Anexo I. Usando este manual, se ha desarrollado un *script* sencillo que utiliza las bibliotecas gráficas Bokeh y Panel para crear un *streaming* de datos y mostrar información continuamente en gráficas.

Esta interfaz ha permitido probar la API de la biblioteca con casos concretos que usan todas las llamadas continuamente. En este ejemplo, se han usado dos variables generadas a partir de las variables internas del PLC: *Potenciómetro* y *Switch*. Estas dos variables se han obtenido inicialmente a partir del archivo XML que el *software* del PLC ha generado. Una vez se saben cuáles son, se han representado siguiendo la estructura de datos definida (Figura 17a). Para generar las gráficas, se ha creado como parte del *script* un fichero de configuración *ad hoc* que especifica las variables que se van a pedir, junto con algún otro parámetro (Figura 17b).



(a) Mapeado de variables (b) Parte de visualización

Figura 17: Configuración inicial de archivos

El paquete Bokeh genera un servidor web en el que se presenta un *dashboard* básico donde se pueden ver los valores que toman las variables pedidas durante el tiempo de ejecución. Un ejemplo de estos resultados, tras unos segundos de uso e interacción con el potenciómetro, se puede ver en la figura 18.

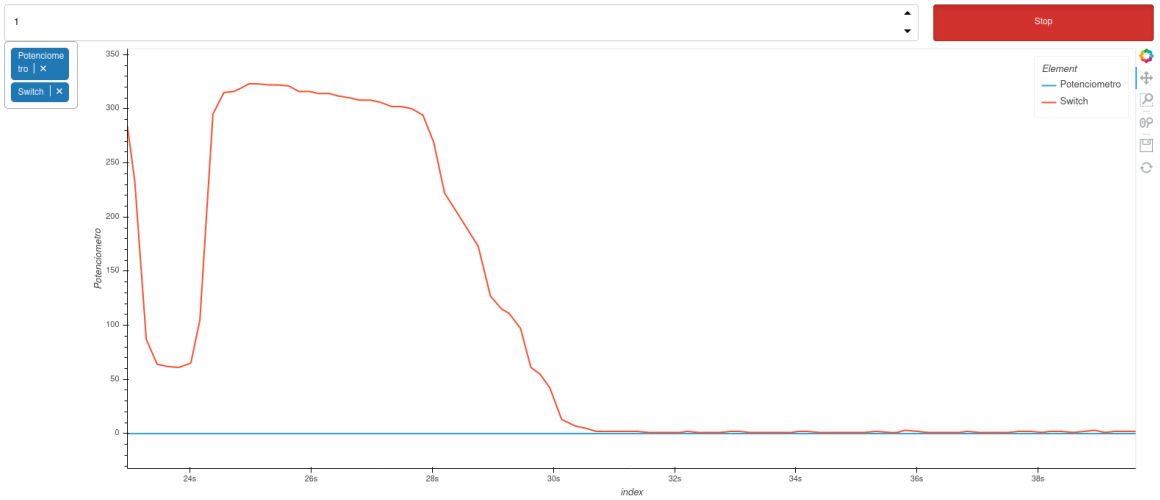


Figura 18: Datos obtenidos tras varios segundos

Esta pequeña aplicación, extendida para alcanzar más registros y mostrar datos más relevantes, ha sido usada por el departamento técnico para hacer seguimiento de configuraciones del PLC. El seguimiento se ha llevado a cabo a bordo de barcos, durante puestas en marcha de proyectos y en situaciones de asistencia por parte del departamento de servicios. Si bien no constituye una aplicación completa, ya que carece de maneras de generar históricos con los datos, representa una mejora sustancial respecto a las herramientas de monitorización de las que dispone el departamento cuando desarrolla sus tareas *in situ*. Estas herramientas suelen limitarse al propio *software* de programación, que solo muestra un valor instantáneo. Un ejemplo de esto se puede encontrar en la figura 4, en la sección 3.3.

El conector creado para esta parte es una versión ligeramente modificada del original de la biblioteca. En este caso, busca imitar funcionalidades similares a las de una suscripción a datos. Para lograr esto se ha empleado una clase específica del paquete *holoviews*, llamada *Pipe*. Esta clase implementa un objeto sobre el cual se puede hacer

un *stream* de datos. De manera periódica, se añaden a esta *Pipe* los datos obtenidos por medio de la petición de lectura. El pseudocódigo mostrado a continuación enseña el funcionamiento de esta parte.

Algoritmo 1: Suscripción a datos

```
Result: Void
request = createRequest();
deadline = currentTime + waitingTime;
while !(stopEvent) do
    if currentTime >= deadline then
        connector.send(request);
        response = connector.recv(request.Status);
        data = serializeResponse(response, request.Status);
        addToPipe(data)
    else
        deadline = currentTime + waitingTime;
    end
end
```

De esta forma, se han validado varios requisitos funcionales clave del proyecto. Todos los requisitos relacionados con el acceso a datos, RF01, RF02, RF04 se han comprobado en estas pruebas. Los requisitos referentes a la comunicación en sí no han sido necesarios durante las pruebas, ya que las conexiones han sido al mismo dispositivo continuamente.

5.3. Resultados obtenidos

Para comprobar las estimaciones iniciales sobre el rendimiento posible del protocolo en el sistema que se va a emplear, se ha diseñado un *benchmark* para concluir velocidades de transmisión en situaciones normales de transmisión. Estas pruebas se han realizado en un procesador Intel Core i5-5200U y una tarjeta de red Realtek 8111E.

De manera complementaria, se han comparado los resultados obtenidos con los equivalentes usando dos paquetes de Python desarrollados por la comunidad. Estos dos paquetes usan el protocolo Modbus sobre una conexión TCP. Se ha elegido este protocolo para realizar la comparación ya que ha constituido una de las alternativas a SLMP en en análisis inicial sobre otros protocolos.

Al haber una capa extra de implementación sobre el protocolo, permitiendo acceder a variables mapeadas, se han hecho dos pruebas diferentes. Se ha accedido a un registro por su nombre directo y, a continuación, por un nombre arbitrario mapeado en la configuración. De la misma manera, se han considerado los casos de tamaño mínimo y máximo de petición. De esta manera se puede sacar una velocidad máxima de transmisión de información. Los tiempos obtenidos se corresponden con diez mil peticiones consecutivas, manteniendo la conexión abierta durante todo el proceso.

Para minimizar impactos que otros procesos puedan haber hecho en los tiempos de ejecución, se ha repetido cada prueba cinco veces. En la tabla 5 se muestran las medias obtenidas tras ejecutar esta prueba de rendimiento.

Para realizar el cálculo de la velocidad, solo se tiene en cuenta la información de

	125 registros (R0_125)		Máx. registros (R0_960)	
Iteración	**Tiempo (s)**	**Velocidad (Kbps)**	**Tiempo (s)**	**Velocidad (Mbps)**
1	40,81	490,07	80,12	1,917
2	39,74	503,27	80,35	1,911
3	39,26	509,42	80,72	1,902
4	39,32	508,65	80,49	1,908
5	39,29	509,03	80,80	1,9
Media	39,68	504,03	80,49	1,908

Tabla 5: Accesos a registros directamente, usando SLMP

aplicación que se transmite, es decir, la que llega como parte de las respuestas. En este caso, siendo diez mil peticiones, se multiplica por el número de bits de cada una de ellas. El acceso a un registro son 16 bits por petición; para 125 y 960 registros son 2000 y 15360 bits respectivamente por petición. Por lo tanto, en la prueba de acceso a 125 registros, se reciben 20 Mbits. Al acceder a 960, 153,6 Mbits.

Como se puede ver en los datos obtenidos, el tiempo que el PLC dedica a responder las tramas añade un *overhead* claro. Al incrementar el número de registros que se piden al máximo, la tasa de transmisión de información se ve incrementada en varios órdenes de magnitud.

En la tabla 6 se pueden ver los tiempos obtenidos al hacer peticiones de lectura usando dos bibliotecas públicas de Python que implementan el protocolo Modbus. En este caso, se están pidiendo el máximo número de registros por trama, 125. Se puede apreciar como la velocidad para el mismo número de registros es ligeramente menor para pyslmp, algo que se puede asociar con el tiempo de computación extra por las funcionalidades extra que ofrece la biblioteca desarrollada. No obstante, para el número máximo de registros, la tasa de transmisión de datos es sustancialmente mayor que la que se puede conseguir usando el protocolo Modbus.

	pymodbus		pyModbusTcp	
Iteración	**Tiempo (s)**	**Velocidad (Kbps)**	**Tiempo (s)**	**Velocidad (Kbps)**
1	33,02	605,69	38,45	520,15
2	34,11	586,34	39,32	508,64
3	33,38	599,16	38,39	520,97
4	33,59	595,41	38,91	514,01
5	32,98	606,43	38,69	516,93
Media	33,42	598,44	38,75	516,13

Tabla 6: Lecturas con otras bibliotecas, usando Modbus

A partir de estos resultados, se visualiza cómo el PLC impone un límite a la cantidad de información que se puede extraer por unidad de tiempo. En este caso, al ser un dispositivo más lento en comparación a un PC, se pone de manifiesto la ventaja que presenta emplear un protocolo que permita pedir más registros por mensaje.

Al repetir varias de estas pruebas en otros sistemas, con procesadores más rápidos y tarjetas de red más modernas, se han obtenido resultados similares. Como es de esperar y de manera consistente con las conclusiones anteriores, se supone que esta diferencia se debe al tiempo dedicado a entrada/salida, siendo el PLC el extremo limitante.

6 Conclusiones y trabajo futuro

Este proyecto ha propuesto y desarrollado una solución, la biblioteca Pyslmp, para facilitar tareas de extracción de datos de dispositivos PLC. La elección de este protocolo se ha justificado tras valorar otras bibliotecas ya existentes. En base a esta decisión, se ha propuesto una ampliación que integra esta biblioteca en una aplicación para mostrar históricos de estos datos recogidos. De esta manera, se ha cumplido un objetivo fundamental del proyecto, que era no depender de elementos hardware propietarios para conseguir información almacenada en los PLCs.

Tras la implementación de la biblioteca, se han generado una serie de pruebas de rendimiento de la misma. Las comparativas resultantes muestran que, para accesos a pocos registros, Pyslmp tiene un rendimiento similar a las alternativas. No obstante, cuando se pide el máximo número de registros permitidos por el protocolo (por mensaje), se obtienen tasas de transmisión mucho más altas que estas alternativas. Esto hace que Pyslmp sea una opción mucho más atractiva para las tareas típicas de extracción de datos asociadas a IoT, que suelen conllevar accesos a gran cantidad de datos.

El paquete desarrollado todavía tiene muchas áreas en las que se pueden ir añadiendo características. Por ejemplo, la posibilidad de crear optimizaciones sobre cómo se hacen las peticiones, para ordenar los bloques de manera consecutiva. Todos estos aspectos forman parte de la etapa de mantenimiento de software, la cual permite la mejora constante de un producto como este.

7 Bibliografía

[1] Axel Daneels and Wayne Salter. What is SCADA? 1999.

[2] SonarSource. Sonarqube. https://www.sonarqube.org/. Accedido el 2021-06.

[3] Json. https://www.json.org/json-es.html. Accedido el 2021-06.

[4] Mozilla. Introducción a xml. https://developer.mozilla.org/es/docs/Web/XML/Introducción_a_XML. Accedido el 2021-06.

[5] Project Jupyter. Jupyter-notebook. https://jupyter.org/. Accedido el 2021-06.

[6] NumPy. Numpy. https://numpy.org/. Accedido el 2021-06.

[7] Bokeh contributors. Bokeh. https://bokeh.org/. Accedido el 2021-06.

[8] Panel contributors. Panel. https://panel.holoviz.org/. Accedido el 2021-06.

[9] Apache. Plc4x. https://plc4x.apache.org/. Accedido el 2021-05-17.

[10] HMS Networks. Industrial network market shares 2019 according to HMS. https://www.hms-networks.com/news-and-insights/news-from-hms/2019/05/07/industrial-network-market-shares-2019-according-to-hms. Accedido el 2021-06.

[11] Mitsubishi Electric Corporation. FX5 user's manual (SLMP). http://www.allied-automation.com/wp-content/uploads/2015/05/MITSUBISHI_manual_plc_fx5_slmp.pdf, January 2015. Version: JY997D56001B.

[12] Mitsubishi Electric Corporation. GXWorks. https://es3a.mitsubishielectric.com/fa/es/products/cnt/plceng/items/gx_works2/. Accedido el 2021-09-15.

[13] CC-Link Partner Association. SLMP reference manual, April 2016. Version: BAP-C3002-001.

[14] Mitsubishi Electric Corporation. SLMP reference manual. https://dl.mitsubishielectric.com/dl/fa/document/manual/plc/sh080956eng/sh080956engi.pdf, May 2020. Version: SH(NA)-080956ENG-I(1905)MEE.

[15] Python Software Foundation. Python docs, modules and packages. https://docs.python.org/3/tutorial/modules.html. Accedido el 2021-04.

[16] InfluxData Inc. InfluxDB. https://www.influxdata.com/products/influxdb/. Accedido el 2021-05-18.

[17] Ian Sommerville. Software Engineering. Pearson, 9th edition, 2011.

[18] Python Software Foundation. Python docs, unit testing framework. https://docs.python.org/3/library/unittest.html. Accedido el 2021-05-21.

8 ANEXO I - Manual de usuario

Este anexo muestra una versión reducida de la documentación proporcionada al Departamento Técnico de Silecmar. Su propósito es mostrar el uso típico de la biblioteca para ser integrada en otros sistemas.

En primer lugar, explica el formato seguido para guardar información sobre variables, incluyendo tipos de datos a los que se puede acceder. A continuación, se hace un resumen de cómo hacer lecturas/escrituras por medio de variables. Finalmente, se comenta el uso del acceso directo a las variables.

Campos de una variable

Reg

Se corresponde con el nombre del registro asociado del PLC. Estos registros tienen nombres preestablecidos que se guardan en el archivo 'slmp_constants.py'

Pos

Puntero a la posición de memoria donde comienzan los datos de la variable.

Type

En el campo "type" de cada variable. Conforman los tipos básicos del PLC, y tienen un valor correspondiente con lo que ocupan en registros.

BOOL	**Para registros de tipo bit. Booleano con valor 1 o 0**
WORD	1 registro (unsigned 16-bit)
DWORD	2 registros (unsigned 32-bit)
INT	1 registro (signed 16-bit)
DINT	2 registros (signed 32-bit)
REAL	2 registros (IEEE 754 simple precision 32-bit)
STRING	Longitud variable. Las variables cuyo tipo es string tienen un campo adicional llamado 'len'. Este campo especifica el número de caracteres (cada uno 8 bits) que tiene el string.

Para variables que estén definidas en el PLC como un array o matrices, se añade al tipo las dimensiones de la estructura. Así, si una variable tiene una estructura de dos reales, se definirá de la siguiente manera:

"type": "REAL_2"

En caso de que la variable contuviese una matriz de enteros con dimensiones 2x5:

"type": "INT_2_5"

Len

Este campo solo aparecerá cuando el tipo de la variable sea STRING. Indica el número de caracteres que tiene cada string. En el caso de que una variable defina un array de strings, todos tienen la misma longitud:

"type": "STRING_3",

```
"len": 4
```

Esto definiría un array de 3 strings, cada uno de ellos de 4 caracteres de longitud. Los strings van alineados al siguiente registro, por lo que si el número de caracteres es impar, el último ocupará la mitad de un registro. El siguiente string empezará en el siguiente registro.

Ciclo petición/respuesta

Las peticiones se realizarán a un método encargado de realizarlas, ubicado en el nivel raíz del módulo. Tanto para la petición de escritura como de lectura se realizará una llamada al método pertinente con los datos de la petición.

Petición de lectura

. reading_call(data: array) --> mapa

Al llamar al método de lectura, se le pasará un array de strings. Cada string es un nombre de variable, o grupo general de variables. Estos nombres serán buscados en el archivo de mapeo, dónde se encuentran todas las variables disponibles del PLC.

El formato de strings podrá tener las siguientes variantes:

- **"variableName"**: Nombre individual de variable. Puede servir para acceder directamente a una única variable con ese nombre, o referenciar a una estructura de la que caen más variables.
- **"variable.SubVariable":** Ruta a la variable mapeada en memoria, en caso de que pertenezca a una estructura lógica.
- **"variable[n]":** En el caso de variables que conforman una lista, se puede usar este nombre para indexarlas directamente.

Teniendo una estructura de variables de la siguiente forma:

```
{ "PLC_VARIABLES": {

            "MaquinillaSensor[0].AlarmaOn":{
                    "reg": "R",
                    "pos": 5700,
                    "type": "WORD"
            },
            "MaquinillaSensor[0].AlarmaSonido":{
                    "reg": "R",
                    "pos": 5701,
                    "type": "WORD"
            },
            "MaquinillaSensor[1].AlarmaOn":{
                    "reg": "R",
```

```
                "pos": 5702,
                "type": "INT"
        },
        "MaquinillaSensor[1].AlarmaSonido":{
                "reg": "R",
                "pos": 5703,
                "type": "INT"
        },
        "OtraVariable":{
                "reg": "R",
                "pos": 5706,
                "type": "STRING_2",
                "len": 2
        },
}
```

Ejemplos de petición:

```
>>> data = ['MaquinillaSensor[0].AlarmaOn']
>>> result = reading_call(data)
>>> result['response']
{'MaquinillaSensor[0].AlarmaOn':[10]}
```

```
>>> data = ['MaquinillaSensor[0]']
>>> result = reading_call(data)
>>> result['response']
{'MaquinillaSensor[0].AlarmaOn':[10], 'MaquinillaSensor[0].AlarmaSonido':[15],}
```

```
>>> data = ['MaquinillaSensor', 'OtraVariable']
>>> result = reading_call(data)
>>> result['response']
{'MaquinillaSensor[0].AlarmaOn':[10],          'MaquinillaSensor[0].AlarmaSonido':[15],
'MaquinillaSensor[1].AlarmaOn':[-2],           'MaquinillaSensor[1].AlarmaSonido':[1],
'OtraVariable':['ab', 'cd']}
```

Respuesta:

Las respuestas a una petición se conforman con un diccionario, cuyas parejas clave/valor se corresponden con el nombre de una variable atómica (ruta completa a la variable) y el valor leído en el PLC. Si la variable está definida en el PLC como un array (o matriz), se mostrará como tal en la respuesta.

{"other":[[n_{11}, … , n_{1j}], … , [n_{i1}, … , n_{ij}]]}

```
>>> data = ['Matriz2x3']
>>> result = reading_call(data)
>>> result['response']
{'Matriz2x3':[[1,2,3],[4,5,6]]}
```

La función responde un diccionario con campos preestablecidos.

Resultado	Key	Descripción
Lectura correcta	response	Diccionario con parejas variable;valores
	missingVariables	Lista de variables que no se han encontrado
Lectura incorrecta	error	String con el mensaje de error. Típicamente será uno de los definidos por la clase propia de excepciones.

```
>>> data = ['NoExisto', 'Existo']
>>> result = reading_call(data)
>>> result['response']
{'Existo':[2.6]}
>>> result['missingVariables']
['NoExisto']
```

Si ninguna de las variables se puede encontrar, la respuesta será de error:

```
>>> data = ['NoExisto', 'YoTampoco']
>>> result = reading_call(data)
>>> result['error]
EmptyRequestError: No variables to read. This may be because none of them were found or
it's an empty petition
```

Acceder a los otros campos resultará en un KeyError

```
>>> data = ['NoExisto', 'YoTampoco']
>>> result = reading_call(data)
>>> result['response']

Traceback (most recent call last):
  ...
KeyError: 'response'
>>> result['missingVariables']
Traceback (most recent call last):
  ...
```

Petición de escritura

.writing_call(data: mapa) --> mapa

La escritura se realizará de manera análoga a la respuesta de una lectura. Para mandar una escritura es necesario enviar un diccionario con los los nombres de variables como clave y los datos a introducir como valor. En este caso, no se puede enviar una variable que sea una estructura, es decir, de la que caigan más variables. Es necesario especificar variables únicas y específicas.

```
>>> data = {'MaquinillaSensor[0].AlarmaOn':[20]}
>>> result = writing_call(data)
>>> result['response']
Data writen
```

Respuesta:

Se devuleve una lista con las variables no encontradas.

Resultado	Key	Descripción
Escritura correcta	response	String 'Data writen'
	missingVariables	Lista de variables que no se han encontrado
Escritura incorrecta	error	String con el mensaje de error. Típicamente será uno de los definidos por la clase propia de excepciones.

Acceso directo a registros

Por una decisión de diseño, y para facilitar el uso de la librería a alguien que sepa cómo está estructurado el programa del PLC previamente, es posible acceder a la memoria del PLC sin necesidad de indicar nombres de variable. Para hacer esto, hay que mandar un string con un formato determinado:

'*xy[_m]*'

x: nombre de registro
y: posición inicial de registro
m: acceso en modo array

De esta manera se pueden pedir registros directamente al PLC. La respuesta será con el entero que haya guardado, sin signo, por lo que el tipo de dato que contenga deberá ser interpretado por el que lo pida.

Se permite también acceder a varias posiciones a partir de la inicial, en modo array (o multiarray). Simplemente se indicará con las dimensiones siguiendo el formato '_m_n...'

```
>>> data = ['R100', 'W0a_2', 'R200_3_2']
>>> result = reading_call(data)
>>> result['response']
{'R100': [34000], 'W0a_2':[1509, 2987], 'R200_3_2':[[1,2],[3,4],[5,6]]}
```

Nota: En el caso de que haya registros que se indexen en hexadecimal, tiene que haber un 0 precediendo a la dirección (si empieza por letra), para evitar confusiones con el nombre de registro. Por ejemplo acceder a la decimosegunda posición de memoria de un registro W se indicará *W0c.* El acceso a la octava posición sería *W8*.

9 ANEXO II - Listado de registros

Este anexo contiene un extracto del manual del PLC [11] sobre los registros que usa.

Device range

This section shows accessible CPU module device.
Specify the device and device number range that exist in the module targeted for data read or write.

In the case of FX5CPU

Classification	Device		Type	Device code*1 (Device specification format: Long)		Device No.		Applicable FX5CPU device*2
				ASCII code	Binary code			
Internal user device	Input		Bit	X* (X***)	9CH (9C00H)	Specify in the range of device numbers of the module to access.	Octal	○
	Output			Y* (Y***)	9DH (9D00H)		Octal	○
	Internal relay			M* (M***)	90H (9000H)		Decimal	○
	Latching relay			L* (L***)	92H (9200H)		Decimal	○
	Annunciator			F* (F***)	93H (9300H)		Decimal	○
	Edge relay			V* (V***)	94H (9400H)		Decimal	—
	Link relay			B* (B***)	A0H (A000H)		Hexadecimal	○
	Step relay			S* (S***)	98H (9800H)	—	Decimal	○
	Data register		Word	D* (D***)	A8H (A800H)	Specify in the range of device numbers of the module to access.	Decimal	○
	Link register			W* (W***)	B4H (B400H)		Hexadecimal	○
	Timer	Contact	Bit	TS (TS**)	C1H (C100H)		Decimal	○
		Coil	Bit	TC (TC**)	C0H (C000H)			○
		Current value	Word	TN (TN**)	C2H (C200H)			○
	Long timer	Contact	Bit	— (LTS*)	51H (5100H)		Decimal	—
		Coil	Bit	— (LTC*)	50H (5000H)			—
		Current value	Double Word	— (LTN*)	52H (5200H)			—
	Retentive timer	Contact	Bit	SS (STS*)	C7H (C700H)		Decimal	○
		Coil	Bit	SC (STC*)	C6H (C600H)			○
		Current value	Word	SN (STN*)	C8H (C800H)			○
	Long retentive timer	Contact	Bit	— (LSTS)	59H (5900H)		Decimal	—
		Coil	Bit	— (LSTC)	58H (5800H)			—
		Current value	Double Word	— (LSTN)	5AH (5A00H)			—
	Counter	Contact	Bit	CS (CS**)	C4H (C400H)		Decimal	○
		Coil	Bit	CC (CC**)	C3H (C300H)			○
		Current value	Word	CN (CN**)	C5H (C500H)			○

Classification	Device		Type	Device code*1 (Device specification format: Long)		Device No.		Applicable FX5CPU device*2
				ASCII code	Binary code			
Internal user device	Long counter	Contact	Bit	— (LCS*)	55H (5500H)	Specify in the range of device numbers of the module to access.	Decimal	○
		Coil	Bit	— (LCC*)	54H (5400H)			○
		Current value	Double Word	— (LCN*)	56H (5600H)			○
	Link special relay		Bit	SB (SB**)	A1H (A100H)		Hexadecimal	○
	Link special register		Word	SW (SW**)	B5H (B500H)		Hexadecimal	○
System device	Special relay		Bit	SM (SM**)	91H (9100H)	Specify in the range of device numbers of the module to access.	Decimal	○
	Special register		Word	SD (SD**)	A9H (A900H)		Decimal	○
	Function input		Bit	—	—	—	Hexadecimal	—
	Function output			—	—		Hexadecimal	—
	Function register		Word	—	—		Decimal	—
Index register			Word	Z* (Z***)	CCH (CC00H)	Specify in the range of device numbers of the module to access.	Decimal	○
Long index register			Double Word	LZ (LZ***)	62H (6200H)		Decimal	○
File register			Word	R* (R***)	AFH (AF00H)		Decimal	○
				ZR (ZR**)	B0H (B000H)		Decimal	—
Link direct device*3	Link input		Bit	X* (X***)	9CH (9C00H)		Hexadecimal	—
	Link output			Y* (Y***)	9DH (9D00H)		Hexadecimal	—
	Link relay			B* (B***)	A0H (A000H)		Hexadecimal	—
	Link special relay			SB (SB**)	A1H (A100H)		Hexadecimal	—
	Link register		Word	W* (W***)	B4H (B400H)		Hexadecimal	—
	Link special register			SW (SW**)	B5H (B500H)		Hexadecimal	—
Module access device*3	Link register		Word	W* (W***)	B4H (B400H)		Hexadecimal	—
	Link special register			SW (SW**)	B5H (B500H)		Hexadecimal	—
	Module access device			G* (G***)	ABH (AB00H)		Decimal	○

*1 [ASCII code]
If the device code is less than the specified character number, add "*" (ASCII code: 2AH) or a space (ASCII code: 20H) after the device code.
[Binary code]
When "Device code" is less than the size specified add "00H" to the end of the device code.

*2 ○: An FX5CPU device exists
—: No FX5CPU device

*3 "Device memory extension specification" for sub-commands must be turned ON (1).