UNIVERSIDAD MIGUEL HERNÁNDEZ DE ELCHE

ESCUELA POLITÉCNICA SUPERIOR DE ELCHE

GRADO EN INGENIERÍA INFORMÁTICA EN TECNOLOGÍAS DE LA INFORMACIÓN

"HERRAMIENTAS DE SIMULACIÓN E INTERCOMUNICACIÓN PARA LA INTEGRACIÓN DE SISTEMAS INDUSTRIALES BASADOS EN MODBUS"

TRABAJO FIN DE GRADO

Diciembre -2025

AUTOR: Jordi Segura Alonso

DIRECTOR: Héctor Francisco Migallón Gomís

ÍNDICE

Capítulo 1
Introducción

1.1 Justificación

El desarrollo de las herramientas de análisis y gestión de datos en las últimas décadas ofrece un gran potencial para la toma de decisiones basadas en factores medibles y objetivos a lo largo de todos los sectores de la economía. Este desarrollo ha sido consustancial con el desarrollo de las tecnologías de obtención de datos que han alimentado los algoritmos de toma racional de decisiones. Sin embargo, son muchos los sectores en el que el elevado coste del equipamiento, y por tanto, sus largos tiempos de amortización han supuesto un freno a esta dinámica.

Tal es el caso del sector naval, cuyos equipos, además, reúnen otra serie de características que dificultan su interoperatividad con el software moderno. Entre ellas se puede destacar: La naturaleza propietaria de la mayoría del software y protocolos que se usan en el sector; las limitaciones técnicas que se encuentran al trabajar con dispositivos empotrados que pueden estar cerca de su límite de obsolescencia, son habituales las restricciones al disponer de memoria o en el tiempo de respuesta de los dispositivos y la débil adhesión a los estándares que prevalece entre los distintos competidores del sector que se encuentran sin incentivos para dar soporte a funcionalidades que ellos mismos no utilizan, dificultando mucho la compatibilidad.

Estas problemáticas dificultan mucho el trabajo de los vendedores de agregadores de sistemas en la industria. Entre ellas se puede contar a la empresa para la que se afrontan los desarrollos de este trabajo. Se trata de una

compañía mediana que comercializa sistemas de alarmas y control para el sector naval. El sistema que vende la empresa es el encargado de aglutinar la información de todos los otros sistemas que operan en el navío y gestionar el estado de una serie de parámetros emitiendo alarmas en casos de mal funcionamiento para alertar al personal de abordo.

La dificultad surge al ser el software de la empresa el encargado de interactuar y recibir datos de todos los otros equipos a bordo. De esta forma, es solo en este software en el que se ven reflejados los datos del navío y por tanto es el único sitio en el que se pueden detectar y depurar muchos errores. De esta forma es que la empresa asume unilateralmente los costes que se derivan de esta incompatibilidad.

Esta dinámica, lejos de desaparecer, se ha intensificado en los últimos años debido al cambio de enfoque en la interacción entre los sistemas que componen el navío. Anteriormente, la mayoría del equipamiento a bordo no contaba con sistemas de información propios, la interacción por tanto se reducía a la lectura y escritura de entradas y salidas físicas. No obstante, en los últimos años hemos visto la inclusión en cada vez más subsistemas de unidades de control. El modelo de interacción por tanto pasa cada vez más por la comunicación con otros equipos externos.

Se puede ver que la empresa tiene un interés marcado por simplificar la forma en la que su software interactúa con el resto de sistemas del barco. De esta forma podrían aislar la complejidad derivada de la heterogeneidad de las comunicaciones de sus aplicaciones. Este cambio resulta clave ya que abre las puertas a sustituir todo el conjunto de tecnologías usado, dejando atrás la adaptabilidad de lenguajes de bajo nivel a cambio de ganar la comodidad y velocidad de desarrollo que ofrecen soluciones de más alto nivel.

Otro interés de la compañía derivado de la problemática expuesta es reducir la incertidumbre a la hora de comunicarse con dispositivos ajenos, más aún cuando esta incertidumbre puede suponer la realización de asistencias que suelen resultar muy costosas para la empresa. Para este fin, resulta clave contar con un buen software de simulación que no se limite a lo prescrito por los distintos

estándares de comunicación, sino que se pueda adaptar al funcionamiento particular de los sistemas encontrados en el día a día de los despliegues a los que se enfrenta la empresa.

Aunque el problema que se plantea resulta común a toda una serie de protocolos de comunicación con dispositivos industriales que se usan en el sector naval (CANJ1939, NMEA2k, NMEA0183, OPC-UA, etc.) el protocolo para el que, por su ubicuidad, resulta más importante atajar el problema es Modbus.

1.2 Objetivos

Como se ha visto en el apartado anterior la empresa hacia la que orientamos nuestros desarrollos tiene un problema claro: La intercomunicación de su sistema con equipos Modbus es muy compleja y poco fiable. La solución que planteamos a dicho problema es triple:

Primero, se busca implementar una librería de Modbus que aúne la flexibilidad y simplicidad necesaria para llevar a cabo los proyectos de la empresa. Un elemento clave en la motivación y los requisitos para emprender este desarrollo es la facilidad de modificación y mantenimiento por parte del personal de la empresa, es por eso que se ha decidido llamarla Tweakable Modbus. A partir de esta librería se desarrollarán los otros dos objetivos.

Segundo, se quiere crear una solución de software que actúe de intermediaria entre los dispositivos Modbus y el software de la compañía. Como requisitos se plantean una configuración simple y reproducible y la exposición de la información en una forma que resulte cómoda y familiar a desarrolladores de alto nivel: una API REST. Por sacar la información del bus se ha decidido llamar a este producto Ultrabus.

Por último, contaremos con un simulador de esclavo de Modbus configurable y reproducible. El propósito de esta herramienta será su integración

con los procedimientos de testing de la empresa que se encuentran en proceso de gradual automatización. Por similitud a su hermano se va a llamar Ultraslave.

De esta forma podremos mejorar la manera de comunicarse de todo tipo de aplicaciones, incluyendo el software de alarmas de nuestra empresa con dispositivos industriales basados en Modbus. Al mismo tiempo se puede facilitar los despliegues de la compañía habiendo podido realizar pruebas simuladas antes de instalar nuestros equipos en el navío.

Como objetivos transversales más allá del desarrollo, la empresa está realizando una apuesta por la investigación de nuevas tecnologías que puedan sustituir al conjunto de tecnologías que usa actualmente. Al tratarse este de un desarrollo que goza de una gran independencia del resto del sistema, se cuenta con gran libertad para elegir las herramientas para resolver el problema.

Para superar los inconvenientes planteados por los lenguajes de bajo nivel utilizados hasta el momento, se ha decidido utilizar lenguajes que garanticen la seguridad de memoria descargando esta tarea del programador. Además, teniendo en cuenta el carácter centrado en comunicaciones de los productos a desarrollar, se va a apostar por la implementación de tecnologías asíncronas simplificando la estructura del código y mejorando el rendimiento al acabar con la espera activa.

Estos cambios podrían tener un impacto muy importante sobre todos los productos de la empresa y, por tanto, las experiencias adquiridas en estos desarrollos son de suma importancia.

1.3 Relevancia

La mayoría de los problemas que se pretenden resolver no son específicos del sector naval, por lo que esperamos que las soluciones implementadas puedan servir también para proyectos industriales de todo tipo.

En este sentido, la utilización de las herramientas reduciría notablemente los costes para una empresa que buscara integrar sus sistemas para el análisis de datos a través de técnicas de Big Data o la elaboración de Dashboards u otros mecanismos de Busisness Intelligence. Esta reducción de costes es aún más clave si se tiene en cuenta la gran cantidad de empresas de tamaño mediano y pequeño que componen el tejido industrial de nuestro país.

Del mismo modo, al construir estas alternativas de software se puede, en muchos casos, estar alargando la vida útil de mucho equipamiento industrial. De esta forma, al mismo tiempo que se reducen los costes para las empresas se consigue reducir la producción de bienes de equipo y, por tanto, el consumo de recursos y la emisión de contaminantes. Esta solución tiene potencial para impulsar la economía circular.

Por otra parte, se puede señalar que a pesar de los esfuerzos de la Modbus Organization con el lanzamiento de Modbus Secure, la mayoría de dispositivos que usan Modbus no pueden ser securizados sin realizar cambios sustanciales en su programación. Ofrecer una alternativa segura por diseño, fácil y barata de implementar puede permitir a muchas organizaciones mejorar su seguridad manteniendo el hardware adquirido. Esto resulta crítico en un entorno como el industrial en el que la acción de agentes hostiles puede traducirse en costosas pérdidas.

1.4 Motivación personal

A nivel particular, el proyecto me parece muy interesante ya que me permite desarrollar por mí mismo todos los roles que debe de asumir un Ingeniero Informático cuando se plantea llevar a cabo un proyecto. Acostumbrado a trabajar en equipos más grandes, este grado de autonomía me parece muy importante para poder desarrollar todas las competencias que se esperan de un graduado.

Por otra parte, el hecho de que este problema sea genuino, es decir, el haberlo encontrado y diagnosticado en el mundo real y el haber tenido que bregar de cerca con sus consecuencias, me motiva para superarlo. También considero que este desarrollo puede jugar un papel clave en el devenir histórico de los sistemas de la empresa y que las prácticas y decisiones que se tomen aquí podrían tener consecuencias en años venideros.

Por último, tengo una especial fascinación por alargar la vida de piezas de tecnología que se podrían descartar como obsoletas. Hay algo emocionante en recortar y recortar de una aplicación para que pueda caber en un microcontrolador obsoleto o en alargar la vida de un equipo que no soporta ningún protocolo de este siglo. Además, creo que en este interés puede haber algo útil para una sociedad en la que la obsolescencia programada se ha convertido en un factor más de nuestras vidas.

1.5 Limitaciones del proyecto

Es importante entender que al menos de momento las herramientas que se proponen son herramientas de desarrollo que no buscan ser usadas por un público generalista. Sin embargo, a partir de ellas se podrían desarrollar sin duda soluciones con una GUI que fueran más usables en un contexto interactivo. Este desarrollo escapa de los objetivos de este documento.

Por otra parte, también hay que señalar que *Tweakable Modbus,* y el resto de programas por consecuencia, no soportan la totalidad del estándar de Modbus. Esto se debe a que al plantear las prioridades del desarrollo nos hemos orientado a las prácticas más comunes en la industria y no a las prescripciones del estándar. Hay parte de lo no implementado que podría llegar a resultar útil, pero no es el objetivo de la herramienta seguir al pie de la letra los cánones del estándar.

Capítulo 2
Estado del Arte

El propósito de este capítulo es reflejar el estudio de las tecnologías utilizadas que ha sido necesario realizar para llevar a cabo el proyecto. En este caso se ha planteado su división en dos bloques principales: el primero estará dedicado a detallar el funcionamiento, problemáticas y estado del ecosistema de herramientas de Modbus; el segundo estará dedicado a las herramientas que se han elegido para enfrentar el proyecto: se van a analizar las ventajas e inconvenientes del lenguaje de programación Rust, el runtime asíncrono Tokio, el gestor de bases de datos SQLite, las API REST y el formato de texto JSON. Este análisis estará siempre orientado a justificar los procesos de decisión que han llevado a la selección de cada una de estas herramientas.

2.1.- Modbus

Modbus es un protocolo de comunicaciones de capa de aplicación orientado a la comunicación con equipos industriales. Con un modelo de comunicaciones basado en la arquitectura cliente/servidor, habitualmente referidos por la comunidad de usuarios como maestro/esclavo, Modbus se ha convertido en un estándar de facto para la comunicación de dispositivos industriales.

A pesar de ser un protocolo de capa de aplicación, Modbus no es completamente agnóstico a las tecnologías de comunicación que se usan bajo él, y es por esto por lo que existen diversas versiones del estándar orientadas a distintos protocolos de capas inferiores. Entre ellos, merece la pena destacar a Modbus TCP, que se ha convertido en los últimos años en el estándar de facto, y a Modbus RTU, que se comunica sobre protocolos serie como RS-232 y RS-485 y aún cuenta con cierto grado de implementación en la industria, sobre todo en equipos pequeños o en el fin de su vida útil. En los últimos años, la Modbus Organization ha llamado a dejar de lanzar nuevos dispositivos basados en Modbus RTU, pero sigue ofreciendo documentación para dispositivos retrocompatibles. También se debe llamar la atención sobre la existencia de Modbus RTUOverTCP, una variante que utiliza la estructura de Modbus RTU a pesar de canalizarse a través de conexiones TCP/IP.

A pesar de que existen numerosas alternativas a Modbus más modernas y potentes para la comunicación con dispositivos industriales, como OPC-UA, es la facilidad de implementación e interoperatividad la que mantiene popular al protocolo. Para entender estas características, es esencial entender la simplicidad del funcionamiento de Modbus.

2.1.1 Funcionamiento

El mensaje básico de comunicación en Modbus es la PDU (Protocol Data Unit), esta puede a su vez estar inmersa dentro de una ADU (Application Data Unit) si resulta necesario para los protocolos de comunicación de capas anteriores. La estructura de la ADU varía en función de estos protocolos.

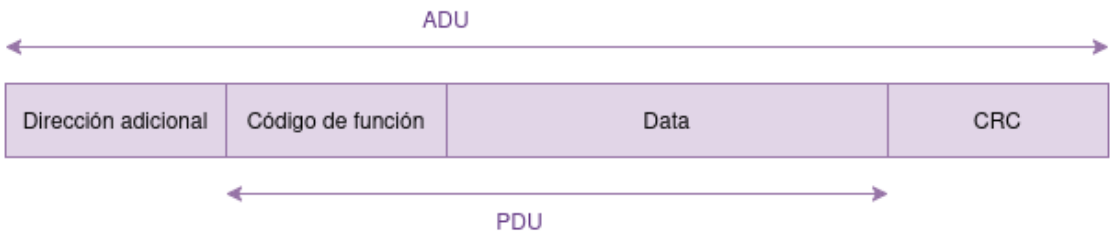

Figura 1: Descripción de la estructura de una ADU Modbus, en concreto Modbus RTU

Como se puede apreciar en la imagen, el primer elemento de la PDU es el código de función o function code. Este código describe las distintas operaciones que se pueden realizar en Modbus y condiciona la estructura del resto de la trama.

Así, el esquema de una conversación sigue el siguiente modelo: primero, el maestro realiza una petición, el esclavo evalúa la petición y, en caso de poder realizarla, envía su respuesta con el mismo código de función que le indicó el maestro. Si la petición no se puede realizar por ser errónea o no respetar alguna de las restricciones configuradas por el esclavo, en su lugar se responde con el mismo código de función del maestro más 0x80 y un código de excepción descriptivo del problema encontrado.

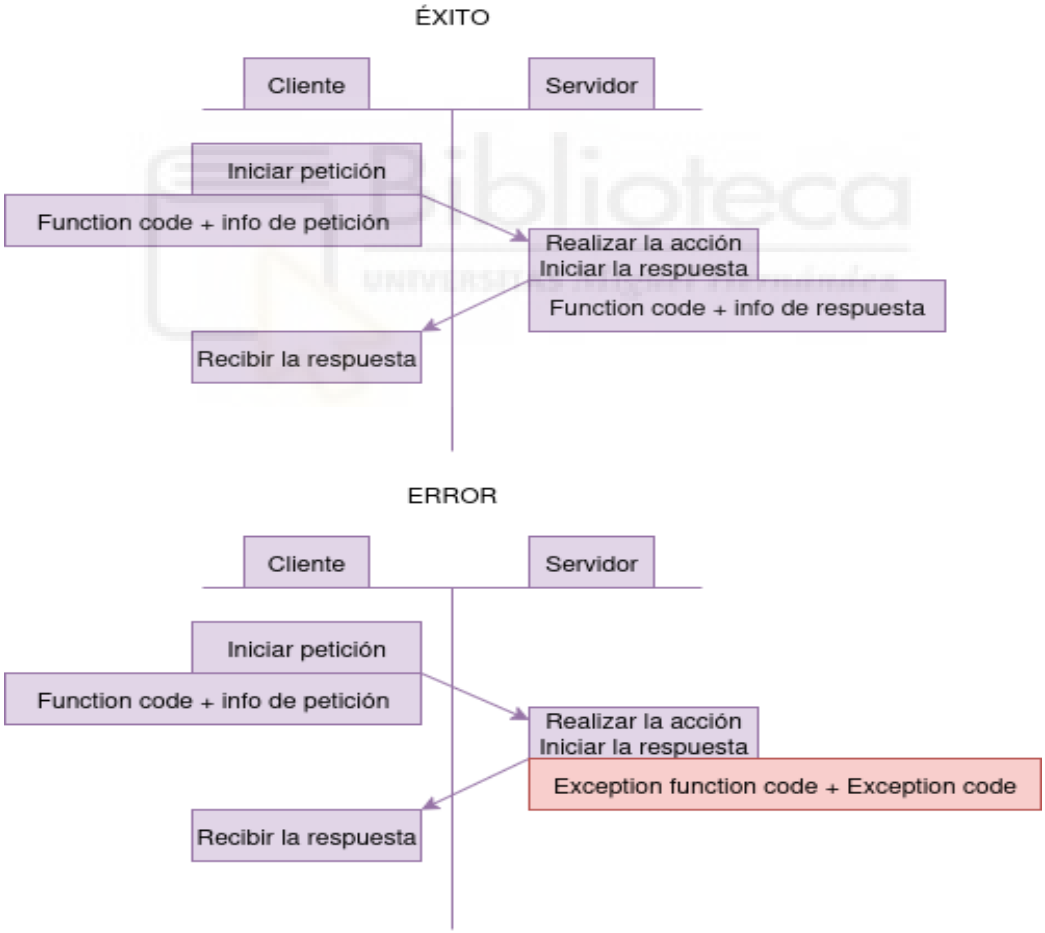

Figura 2: Esquema de conversación de protocolo Modbus

El estándar de Modbus define 21 códigos de función distintos y se reserva la utilización de otros 14, pero permite a los fabricantes implementar sus propias funcionalidades utilizando el resto de los códigos. La siguiente tabla describe los códigos de función públicos descritos por el estándar, como se puede apreciar en los últimos códigos descritos algunas funciones cuentas además con un segundo subcódigo.

Código de función	Función
1	Read coils
2	Read Discrete Inputs
3	Read Holding Registers
4	Read Input Register
5	Write Single Coil
6	Write Single Register
7	Read Exception Status
8	Diagnostic
11	Get Comm Event Counter (Serial Only)
12	Get Comm Event Log (Serial Only)
15	Write Multiple Coils
16	Write Multiple Registers
17	Report Server ID (Serial Line Only)
20	Read File Record
21	Write File Record
22	Mask Write Register
23	Read/Write Multiple Registers
24	Read FIFO Queue
43	Encapsulated Interface Transport
43/13	CANopen General Reference Request and Response PDU
43/14	Read Device Identification

Tabla 1: Enumeración de los códigos de función públicos de Modbus

Escapa de los objetivos de este documento explicar en detalle el funcionamiento de todos y cada uno de los códigos de función descritos por el

estándar. Sin embargo, sí se considera que la explicación de las operaciones más comunes en el protocolo puede resultar útil para comprender mejor la problemática y las soluciones implementadas.

Para poder comprender las operaciones, es necesario primero exponer la manera en la que Modbus organiza su modelo de datos. El estándar describe 4 tablas de datos que se distinguen por el tipo de datos que contienen y los permisos de lectura y escritura que ofrecen al maestro.

De esta forma, con un tipo de datos binario, existen los Coils y los Input Status, y como palabras de 16 bits, los Holding Registers y los Input Registers. A su vez, los Coils y los Holding Registers pueden ser leídos y escritos por el maestro mientras que los Input Status y los Input Registers tienen permisos de solo lectura. Esta información se encuentra resumida en la tabla 2.

Tabla	Tipo de datos	Permisos
Input Status (o Discrete Inputs)	Booleanos	Solo lectura
Coils	Booleanos	Lectura y escritura
Input Registers	Enteros de 16 bits	Solo lectura
Holding Registers	Enteros de 16 bits	Lectura y escritura

Tabla 2: Descripción del modelo de datos de Modbus

El estándar no define cómo estos tipos se deben almacenar en memoria. En la mayoría de sistemas, cada tabla contiene valores independientes de las otras tablas, pero se permite que distintas tablas referencien la misma memoria permitiendo así su tratamiento usando distintos códigos de función.

Los primeros códigos de función a estudiar son los de lectura, es decir, Read Input Status, Read Coils, Read Input Registers y Read Multiple Holding Registers. Todos estos mensajes comparten la misma estructura de trama. Se indica la dirección por la que empezar a leer y la cantidad de valores a leer.

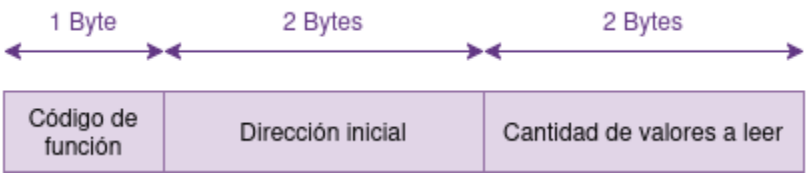

Figura 3: Estructura de las peticiones de lectura de Modbus

Las respuestas también son idénticas salvo por el hecho de estar compuestas por valores de 1 bit o de 16 bits.

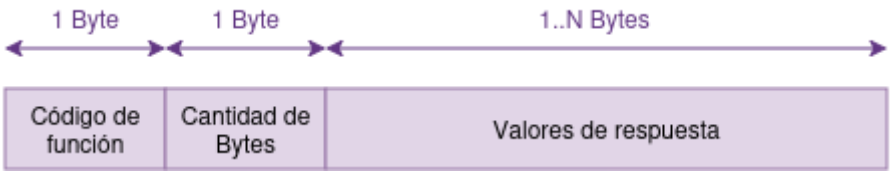

Figura 4: Estructura de las respuestas de lectura de Modbus

Para la escritura, el estándar ofrece más opciones. Por una parte, se definen las operaciones de escritura de un solo registro, Write Single Coil y Write Single Register. Las tramas de estas instrucciones incluyen la dirección sobre la que operar y el valor para asignar a esta dirección. Al usar Write Single Coil en concreto, se debe tener en cuenta que los valores del bit no se reflejan con un cero o un uno como en las operaciones anteriores. Para el caso de esta instrucción, el cero se refleja como 0x0000 y el uno como 0xFF00.

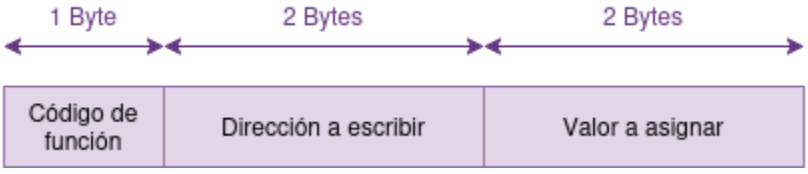

Figura 5: Estructura de las peticiones de escritura simple de Modbus

La estructura de la respuesta es idéntica a la pregunta, pero el valor devuelto es posterior a realizar la actualización del valor, confirmando así la operación.

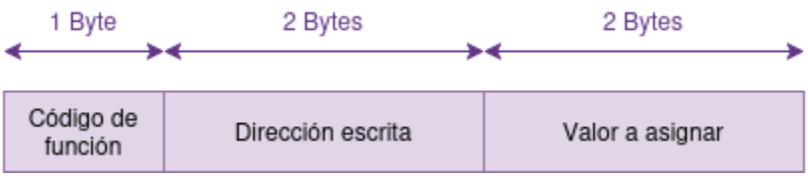

Figura 6: Estructura de las respuestas de escritura simple de Modbus

Modbus también cuenta con operaciones que permiten la escritura múltiple: Write Multiple Coils y Write Multiple Holding Registers. La estructura de las tramas indica la dirección inicial, la cantidad de valores a escribir y la cantidad de bytes que estos ocupan en la trama. Al igual que con las lecturas, la estructura de estas operaciones varía en función de la longitud de los tipos de datos.

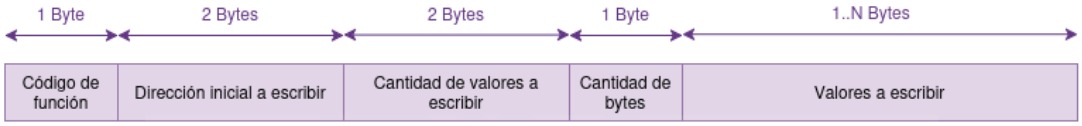

Figura 7: Estructura de las peticiones de escritura múltiple de Modbus

La respuesta indica la dirección inicial y la cantidad de valores escritos. En este caso no existe una confirmación de los valores después de realizar la transacción.

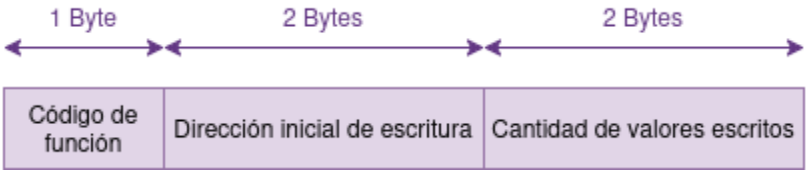

Figura 8: Estructura de las respuestas de escritura múltiple de Modbus

Otra operación interesante es Read Write Multiple Registers, que permite combinar una lectura y una escritura en una sola transacción. La estructura de la trama es el resultado de la combinación de una trama de lectura múltiple y una trama de escritura múltiple.

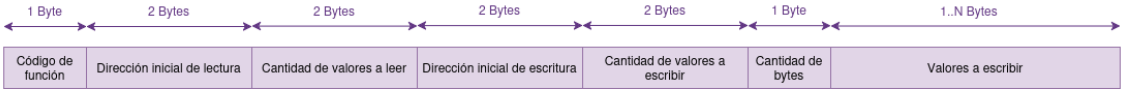

Figura 9: Estructura de las peticiones de escritura/lectura múltiple de Modbus

La respuesta es idéntica a la de una escritura simple: no se recibe ninguna confirmación del proceso de escritura.

Por último, también se considera interesante analizar la operación Mask Write Register, que permite realizar las operaciones lógicas bit a bit OR y AND sobre los contenidos actuales del registro.

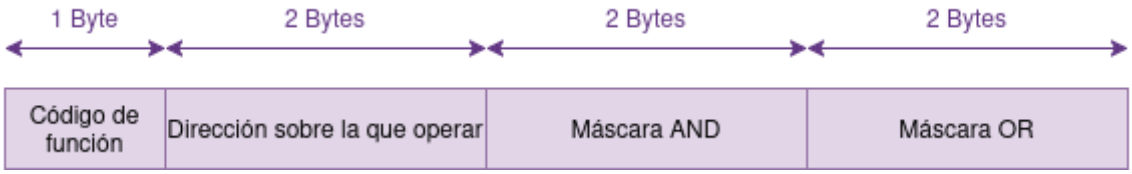

Figura 10: Estructura de las peticiones de escritura con máscara de Modbus

La respuesta tiene la siguiente estructura:

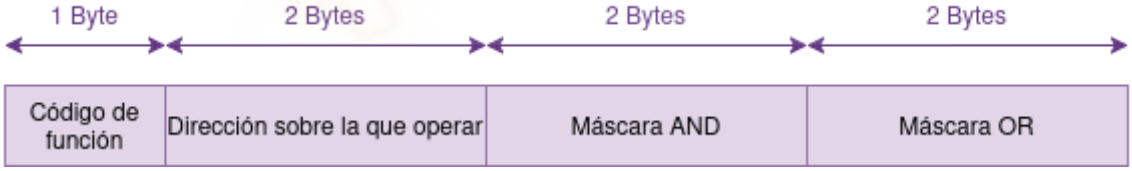

Figura 11: Estructura de las respuestas de escritura con máscara de Modbus

Habiendo estudiado el funcionamiento de las operaciones más comunes en el protocolo, volvamos ahora nuestra atención a las especificidades de las implementaciones de Modbus RTU y Modbus TCP.

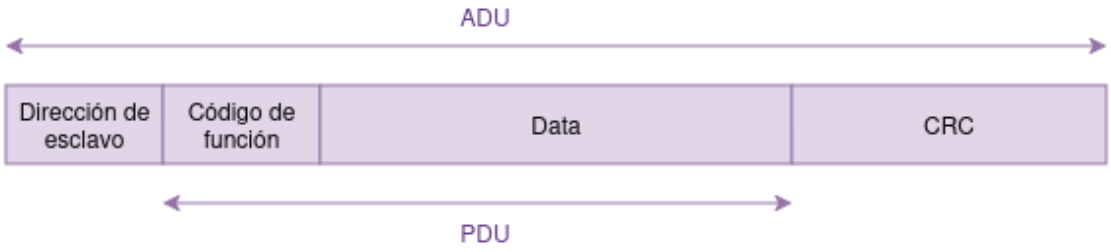

Figura 12: Estructura de la ADU de Modbus RTU

Como se puede observar en la figura 12 en la estructura de la trama RTU, se observa que la ADU está compuesta simplemente por un identificador de esclavo, usado para especificar que esclavo debe contestar en casos en los que haya más de un dispositivo conectado al bus, un código CRC de 16 bits y la PDU. Este código se utiliza como un mecanismo de detección de errores, ya que los protocolos sobre los que trabaja Modbus RTU no garantizan la entrega correcta de la información.

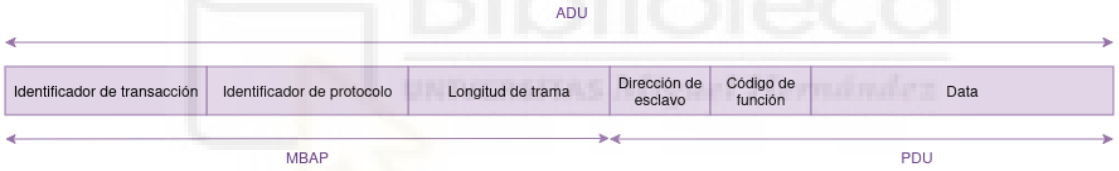

Figura 13: Estructura de la ADU de Modbus TCP

Por contraste, al analizar la trama de Modbus TCP en la figura 13, se aprecia que la ADU está compuesta de más campos. Estos campos también pueden ser referidos como MBAP o Modbus Application Protocol Header. La información que se encuentra es el ID de transacción, el ID de protocolo y el ID de unidad. Cada uno de estos campos tiene implicaciones a explicar.

Es necesario que la trama cuente con un ID de transacción, ya que Modbus TCP, a diferencia de Modbus RTU, sí soporta el procesamiento simultáneo de varias transacciones. Con este ID, el maestro y el esclavo pueden ponerse de acuerdo para saber qué respuesta corresponde a cada pregunta.

El campo de ID de protocolo abre la opción a integrar otros protocolos propietarios sobre Modbus. Por defecto, cuando el protocolo de comunicación es Modbus, su valor es cero.

Por último, el ID de unidad permite definir distintos esclavos en una comunicación TCP. Esto no parece necesario a priori, ya que TCP ya es de por sí un protocolo direccional, pero está orientado al funcionamiento de gateways entre Modbus TCP y RTU. De esta forma, un solo dispositivo con una única dirección IP puede brindar comunicación con una multitud de esclavos RTU.

2.1.2 Problemáticas derivadas

A partir de las experiencias de uso concreto que se han tenido con Modbus, hemos podido observar algunas problemáticas que bien fruto de limitaciones del estándar o bien resultado de implementaciones particulares, se deben tener en cuenta a la hora de plantear el desarrollo.

- Soporte para operaciones simultáneas:

De acuerdo con la guía de implementación de Modbus TCP, cada esclavo debe de especificar la cantidad de operaciones simultáneas que soporta hasta un máximo de 16. En la práctica, la mayoría de dispositivos de Modbus TCP solo soportan una comunicación simultánea. Esto, sumado a unos altos tiempos de respuesta, puede suponer un cuello de botella importante en el intercambio de datos con el esclavo, reduciendo el rendimiento de la comunicación.

A pesar de todo, sí existen dispositivos que soporten más de una comunicación simultánea, pero las librerías y herramientas de simulación existentes no permiten configurar la cantidad de transacciones simultáneas. Sin herramientas para testear y explotar esta capacidad, el software de la empresa se ve obligado a trabajar con una sola transacción a la vez. Nuestra solución debería permitir configurar este parámetro al usuario.

- Limitaciones de tipos de datos

Como se ha expuesto con anterioridad Modbus solo soporta dos tipos de datos: valores booleanos o registros de 16 bits. Existe una necesidad de la

industria de transferir información usando otros tipos de datos pero Modbus no ofrece una forma estandarizada de representar esta información.

En la mayoría de los casos los tipos de datos, que suelen ser más grandes que los que Modbus soporta, se forman por composición uniendo los valores de varios registros. La problemática al trabajar de esta manera es que el orden en el que se transfieren los bytes de estos tipos de datos no está estandarizado de ninguna manera. Nuestra solución debe de soportar la transmisión de estos tipos de datos que funcionan por "encima" del protocolo y dejar suficiente configurabilidad como para adaptarse a la manera de transmitirlos de distintos dispositivos.

- Tamaños máximos de petición no estandarizados

La cantidad de valores a leer o escribir en una petición está teóricamente limitada por el tamaño de los campos Byte Count o Quantity of Coils/Registers. Sin embargo, los límites de las peticiones que pueden gestionar los equipos reales tienden a ser muy inferiores. Además, estos límites pueden ser distintos entre tablas, siendo común permitir más lecturas de valores de 1 bit que lecturas de valores de 16 bits.

La problemática se manifiesta similar al anterior: las librerías existentes no permiten especificar el máximo de valores en una sola solicitud, y por tanto se suele elegir una opción excesivamente conservadora. De nuevo, este parámetro debería poder ser configurado por el usuario.

- Consultas con "huecos"

De forma similar al punto anterior al intentar agrupar el máximo de lecturas posibles en una sola transacción, muchas veces aparece el problema de los "huecos". Al leer las direcciones 4 y 6 pero no la 5, surge la obligación de dividir la petición en dos.

Por este motivo, algunos equipos soportan la lectura de direcciones vacías con un valor por defecto, pero este no es el caso para todos los equipos, ya que de hecho algunos equipos solo permiten la lectura de una cantidad

determinada de valores en blanco. Este es otro campo que debe de ser configurable en nuestra solución.

- Seguridad

Modbus TCP no implementa mecanismos de seguridad. Esta problemática es conocida, y desde la Modbus Organization se ha apostado por el desarrollo de Modbus Secure un nuevo estándar que implementa Modbus TCP usando TLS.

Si bien esta solución puede resultar muy útil para equipos nuevos, los equipos ya existentes, en muchos casos, no pueden ser actualizados a esta versión. Así surge la necesidad de aislar estos equipos de redes más amplias en las que puedan verse expuestos a agentes hostiles.

De acuerdo con el buscador de OSINT Shodan, el 29 de octubre de 2025 había 729.359 dispositivos con el puerto 502, estándar en Modbus TCP, expuesto a internet. Es difícil saber cuántos de estos dispositivos realmente están utilizando Modbus, pero la carencia de una solución segura para exponer estos equipos a internet supone un gran riesgo, más en el clima actual de tensiones internacionales.

2.1.3 Alternativas existentes

Teniendo en cuenta los objetivos, se procede a analizar los productos existentes dentro de dos nichos del ecosistema de Modbus: por una parte, las API de comunicación con equipos de Modbus, y, por otra parte, los simuladores de esclavo.

- API de comunicación de Modbus

La alternativa que mejor encaja en este nicho es la API de Modbus de Unserver. Esta solución permite monitorizar varios esclavos de Modbus de forma simultánea y consultar los datos de forma sencilla a través de una API REST. Estos datos, a su vez, se almacenan de forma agrupada en tramos de distintas duraciones para facilitar su análisis.

El principal problema de esta alternativa es su modelo de pago, que incrementa el coste de forma pronunciada en función de la cantidad de esclavos y valores a monitorizar. Además, se trata de una aplicación desarrollada usando .Net y que solo es compatible con el sistema operativo Windows. Por último, la página web de la empresa que la comercializaba ha dejado de estar disponible imposibilitando su adquisición y uso.

Se han contemplado también otras soluciones que, a pesar de no encajar exactamente en el nicho definido, pueden llegar a desempeñar las mismas funciones. Es el caso de sistemas de conectividad que están destinados a agregar datos obtenidos a través de distintos protocolos industriales de comunicación como KEPSERVER.

A pesar de estar principalmente orientado hacia el protocolo OPC-UA, KEPSERVER soporta comunicaciones con Modbus, y podría servir para monitorizar y comunicarse con dispositivos Modbus. Además, ofrece más formas de acceder a la información que el otro competidor, con soporte para protocolos como MQTT, REST, SNMP y OBDC.

El principal inconveniente de esta solución es la naturaleza propietaria del software y el modelo de precios que ofrecen, más orientado a cerrar ofertas concretas para una empresa que ofrecer un precio estándar por licencia.

- Simuladores de Modbus

La solución más popular en la industria para la simulación de esclavos de Modbus es Modbus Slave. La aplicación es compatible con Modbus RTU y Modbus TCP y cuenta con una interfaz intuitiva. Además, es bastante eficiente permitiendo la simulación de hasta 100 esclavos de Modbus en un solo equipo. Al llevar tanto tiempo siendo la opción de esclavo más popular, cuenta con una fiabilidad probada.

También cuenta con la capacidad de integrar un lenguaje de scripting para modificar los valores expuestos por Modbus. El lenguaje que soporta es Visual Basic Script, que en la actualidad está obsoleto y ya no cuenta con soporte.

Sin embargo, es una aplicación de Windows y, por tanto, no es compatible con otros sistemas operativos. También hay que tener en cuenta que no se puede ejecutar de forma “headless”, sin interfaz, y que por tanto no puede ser fácilmente integrada con tests automáticos.

A todo esto, se debe añadir su naturaleza propietaria, que fuerza a pagar una licencia por cada equipo en el que se use. Además, los ficheros de guardado que genera siguen un formato propietario que dificulta su generación programática.

La mayoría de las limitaciones que presenta Modbus Slave son superadas por la siguiente herramienta que se presenta. Modbus Tools es una suite de herramientas de código abierto que cuenta con un simulador de maestro y un simulador de esclavo.

Están implementados como herramientas usando el framework Qt, y tienen soporte para MacOS, Windows y Linux. También cuentan con un lenguaje de scripting, pero en este caso se trata de Python. Por otra parte, sus ficheros de guardado utilizan XML y son fáciles de interpretar.

A pesar de todo, cuenta con ciertas limitaciones como no poder configurar la dirección de origen de las tablas. Algunos sistemas empiezan a contar en cero y otros empiezan en uno. Además, también es incapaz de ejecutarse en un entorno “headless”.

2.2 Herramientas utilizadas

La selección de las tecnologías para el proyecto se ha hecho de acuerdo con los intereses estratégicos de la empresa a largo plazo. En este punto se explicarán las potencialidades y limitaciones de Rust, Tokio, Sqlite, REST y JSON.

2.2.1 Rust

Rust es un lenguaje de programación lanzado por la Mozilla Foundation en 2010. Está orientado a ser una alternativa a C y C++ ofreciendo a los desarrolladores el mismo control de bajo nivel, al mismo tiempo que se simplifican algunos aspectos complejos como la gestión de memoria y la concurrencia. En este sentido podemos enumerar algunas ventajas:

- Rendimiento

Una de las principales ventajas de Rust es compartida con sus dos competidores. Al tratarse de un lenguaje de programación compilado, sin recolector de basura y que exprime todo el potencial del backend de compilación LLVM, Rust presenta unos muy buenos rendimientos comparables con los de C y C++. En la figura 14 se puede observar la comparativa generada por el proyecto speed comparison calculada utilizando la fórmula de Leibniz para calcular π.

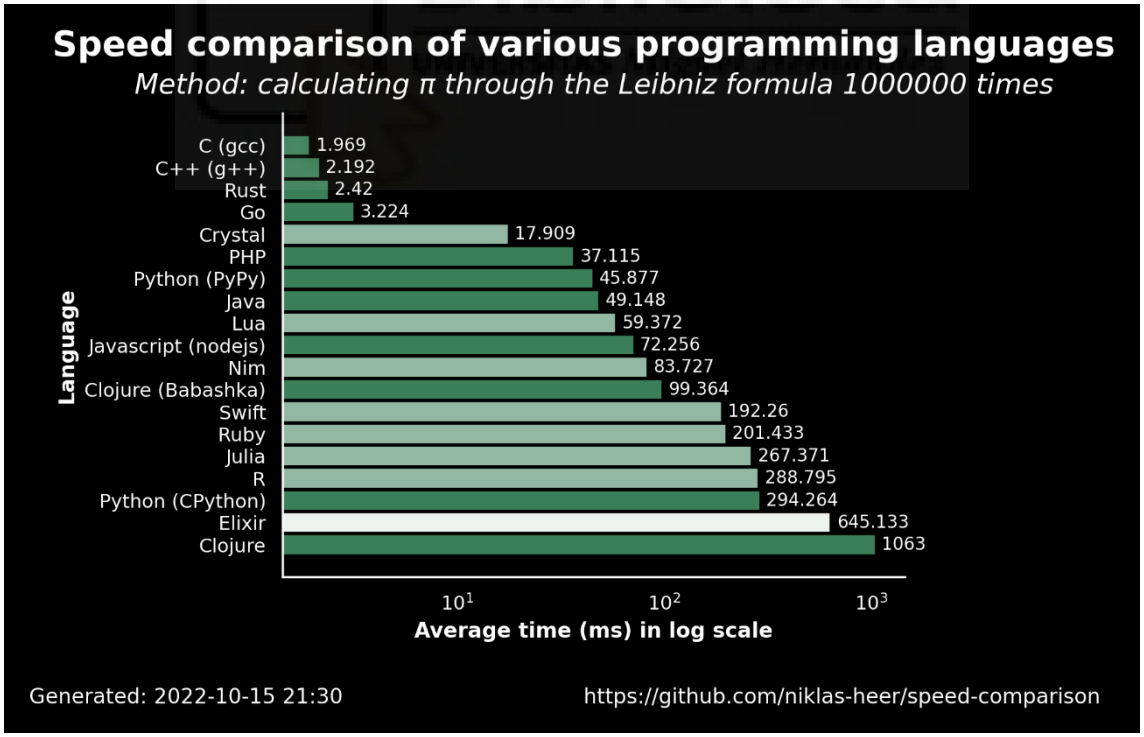

Figura 14: Comparativa de velocidad de lenguajes de programación de acuerdo con el proyecto speed comparison

- Seguridad de memoria

Sin embargo, esta eficiencia viene acompañada de un sistema de gestión de memoria que es capaz de garantizar la seguridad de memoria sin necesidad de un recolector de basura. Este avance es clave ya que protege a los desarrolladores de toda una serie de vulnerabilidades que se derivan de la gestión incorrecta de memoria.

De acuerdo con un estudio interno realizado por Google para el proyecto Chromium, cerca del 70% de los bugs con altos impactos de seguridad fueron provocados por problemas de seguridad de memoria. Se puede apreciar el origen de los bugs severos reportados en el proyecto en la figura 15.

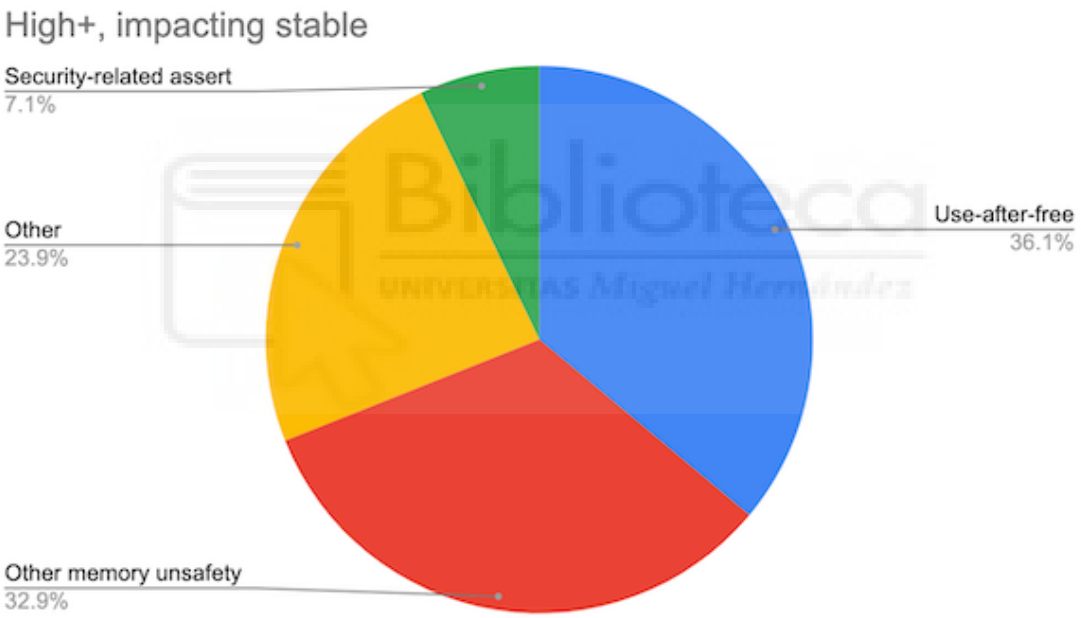

Figura 15: Origen de los bugs de severidad alta detectados en la rama estable del proyecto Chromium.

La mitigación de estos riesgos se vuelve aún más crítica si consideramos las especificidades de los entornos en los que se despliega el software de la empresa. Se trata de software que se usa para monitorizar el estado de sistemas críticos, que se despliega en circunstancias en las que actualizar o dar soporte puede ser imposible durante meses, por encontrarse en alta mar, y que está en

ejecución de forma ininterrumpida. En este contexto, un error tan simple como una pequeña fuga de memoria puede poner en riesgo la operativa de todo el navío.

- Concurrencia segura

Otra fuente común de errores en la programación de bajo nivel son las condiciones de carrera. La seguridad de memoria le permite al compilador de Rust evaluar el código para detectar muchos problemas de concurrencia en tiempo de compilación. De esta forma se pueden detectar y evitar muchos errores que por su naturaleza concurrente serían muy difíciles de depurar en fases posteriores del desarrollo.

La librería estándar de Rust también implementa toda una serie de tipos que facilitan el diseño de procesos concurrentes como mpsc, (multiple producers single consumer) un canal que habilita una comunicación entre distintos hilos, o Arc, un puntero con contador de referencias que permite el acceso seguro desde distintos hilos.

- Herramientas y ecosistema

A diferencia de otros lenguajes de programación, Rust cuenta con su propio sistema de compilación y gestión de paquetes estándar: Cargo. Cargo simplifica enormemente la construcción de proyectos de Rust superando alternativas más complejas como Make y CMake. A su vez, esta facilidad para integrar librerías en proyectos se traduce en un ecosistema de librerías muy amplio.

Así, se encuentran en Rust librerías y frameworks que brindan mucha más funcionalidad y ergonomía de la que se podría esperar para un lenguaje de programación de sistemas. Existen opciones populares para la implementación de servidores web como Actix Web o Axum; interacción con bases de datos como Diesel o Rusqlite; distintos runtimes asíncronos como Tokio o Smol; e incluso algunos frameworks que aprovechan la potencia de WASM para implementar soluciones de front-end web en Rust como Yew.

Sin embargo, Rust también cuenta con una serie de inconvenientes a tener en cuenta antes de evaluar su adopción.

- Tiempos de compilación

Debido a todas las comprobaciones adicionales que realiza el compilador de Rust, los tiempos de compilación son mucho más lentos que los de C y C++. Esto es sobre todo un problema al plantear proyectos grandes.

- Curva de aprendizaje

La manera inusual de gestionar la memoria en Rust puede suponer un reto para programadores que no estén familiarizados con el lenguaje. Este hecho puede suponer una traba para equipos que hayan decidido empezar a utilizar Rust en su trabajo.

Otro punto de complejidad para nuevos desarrolladores puede ser la forma de gestionar el asincronismo en Rust. La implementación del modelo async/await basada en traits a pesar de resultar potente, puede ser bastante confusa.

Sin embargo, la comunidad de Rust cuenta con numerosos recursos formativos para facilitar esta transición. Entre ellos debemos destacar el Rust Book.

2.2.2 Tokio

Tokio es el runtime asíncrono para Rust más popular y está diseñado para la gestión de tareas de entrada y salida, concurrencia y programación basada en eventos. Además, ofrece una reimplementación de la librería estándar de Rust, que permite realizar operaciones de forma asíncrona.

Es importante detenerse en este punto para explicar para qué es un runtime asíncrono y en qué se diferencia con otros modelos de paralelismo, como el de OpenMP. Tanto los runtimes asíncronos como OpenMP sirven para manejar múltiples tareas, pero están diseñados para problemas distintos. Un runtime asíncrono está optimizado para operaciones de entrada/salida que

esperan, como lecturas de archivos o comunicaciones en red; de esta forma permite que muchas tareas cooperativas se ejecuten en un solo hilo sin bloquearse, cediendo el control explícitamente cuando esperan. En cambio, OpenMP facilita la ejecución paralela de código pesado para la CPU, como cálculos numéricos o procesamiento de datos masivos, mediante múltiples hilos del sistema operativo. Mientras los runtimes asíncronos maximizan la eficiencia en programas que pasan mucho tiempo esperando, OpenMP maximiza la velocidad de cómputo distribuyendo trabajo pesado entre núcleos, y no gestiona operaciones de entrada/salida.

Se exponen ahora las ventajas de Tokio:

- Ecosistema

Tokio cuenta con un gran ecosistema de librerías que facilitan el desarrollo de aplicaciones basadas en comunicaciones. Dentro del ecosistema de Tokio destacan MIO, el gestor de entrada/salida, Tracing, un framework de recolección de logs, o Tower, un framework para generar middlewares reutilizables.

- Velocidad

Comparado con otras opciones de runtime asíncronos que suelen utilizar lenguajes interpretados de alto nivel como Javascript o Python, Tokio, al utilizar un lenguaje compilado como Rust, es significativamente más rápido.

- Flexibilidad

Tokio permite definir la política de división de trabajo del runtime para adaptarse a todo tipo de necesidades. Del mismo modo, permite definir la cantidad de hilos sobre las que se dividirán las tareas de la aplicación.

La desventaja principal de usar Tokio, por otra parte, es que su propio despliegue consume una cantidad importante de recursos del sistema. Esta característica no lo hace muy apropiado para trabajar en entornos empotrados como un microcontrolador. En estos casos es mejor utilizar su competidor Smol, que escala mejor para tareas de este tamaño.

2.2.3 SQLite

SQLite es un sistema de gestión de bases de datos diseñado para ser embebido en aplicaciones. A diferencia de otras bases de datos, SQLite se distribuye como una librería en C y no tiene dependencias externas. Algunas ventajas de esta solución son:

- Tamaño reducido

La inclusión de SQLite en un ejecutable añade menos de 1 MB de tamaño. Esto lo hace muy adecuado para sistemas empotrados en los que no se cuenta con mucho almacenamiento.

- Portable

Al tratarse de una librería escrita en C sin dependencias, son muchas las plataformas que soportan SQLite. Al elegir este motor de bases de datos, no hacemos asunciones sobre las plataformas en las que se ejecutará nuestra aplicación.

- SQL Completo

A pesar de su reducido tamaño, SQLite soporta todo el estándar de SQL, permitiendo el desarrollo de un código de interacción con bases de datos portable y reutilizable en otras plataformas.

Sin embargo, SQLite también cuenta con algunas desventajas que no lo hacen apropiado para todos los casos de uso:

- Ausencia de replicación nativa

Al no tratarse de una base de datos con una estructura cliente/servidor, no existen mecanismos para sincronizar una base de datos entre varias instancias. Esto puede resultar muy problemático para el almacenamiento de datos críticos, ya que supone que la pérdida de un solo equipo conllevaría la pérdida de todos los datos.

- Limitaciones de tamaño

SQLite no está pensado para gestionar bases de datos de gran tamaño. A pesar de que las restricciones absolutas para el tamaño de una base de datos son bastante altas (140 TB), el rendimiento de la base de datos disminuye significativamente a medida que estas crecen. Una solución mejor para bases de datos de tamaños tan elevados sería una base de datos con una arquitectura cliente/servidor como PostgresSQL, MySQL o MariaDB.

2.2.4 API REST

REST es un conjunto de líneas de diseño para la implementación de API. No se trata de un estándar cerrado, sino más bien de una serie de axiomas orientados a diseñar API. A continuación, se enumeran los principios de diseño de REST:

Primero, para ser considerada REST, debe contar con una arquitectura cliente/servidor con comunicación a través de HTTP. También se debe de contar con recursos, estos son los elementos básicos que el cliente solicita al servidor.

Segundo, la comunicación entre el cliente y el servidor debe ser stateless. Es decir, cada solicitud debe de ser independiente de las anteriores y tratada de forma distinta. De esta forma no es necesario almacenar información de sesión, y se pueden agrupar peticiones de distintos usuarios.

Tercero, los datos deben de poder almacenarse en caché optimizando la comunicación entre el cliente y el servidor. Este punto, junto al anterior, permite optimizar significativamente el funcionamiento de la API. Poniendo el supuesto de que mil usuarios pregunten por el mismo recurso, solo la primera solicitud se procesará de forma plena y el resto recibirán la repuesta desde el caché de la API.

Cuarto, la API debe de seguir una interfaz uniforme, es decir, los recursos deben de tener una única identificación posible. Del mismo modo, se debe garantizar la unidad entre la forma de presentar datos al cliente y la forma en la que el cliente puede modificar estos datos. En este sentido, la representación que se envía al cliente debe de ser suficiente para editar los datos. Los mensajes

que se envíen al cliente deberán ser autodescriptivos en lo que refiere a la manera de procesar la información. Por último, los recursos deberán contener enlaces a otros recursos asociados para permitir la navegación inmediata entre recursos.

Quinto, la API debe de estar planteada de tal forma que se soporte una jerarquía en capas. El cliente no debe de tener en cuenta la estructura interna de servidores que sostienen la API. En este sentido, no se deben hacer asunciones que impidan la implementación de Middlewares, como balanceadores de carga o autentificadores de usuario.

Sexto, la API debe soportar la solicitud de código por parte del cliente. Este código está orientado a su ejecución en el cliente, podemos pensar en scripts de Javascript, y permite desplazar parte de la funcionalidad del servidor al cliente.

Se pasa ahora a analizar las ventajas de REST:

- Independencia de lenguajes y tecnologías

Al transmitir información usando el formato JSON, son muchos los lenguajes y frameworks que pueden interactuar con REST. Esto supone una gran ventaja comparativa con respecto a SOAP o gRPC, que cuentan con formatos más cerrados para la transferencia de información. Al utilizar esta tecnología no se deben hacer asunciones sobre las aplicaciones que interactuarán con nuestro software en el futuro.

- Simplicidad y familiaridad

De entre las alternativas para el diseño de API, REST es por mucho la más popular. Por tanto, REST es la tecnología con la que los desarrolladores están más acostumbrados a interactuar, reduciendo así la necesidad de documentación. De la misma manera, al ser una tecnología tan popular, son numerosas las librerías y herramientas que facilitan la implementación de este tipo de API en infinidad de lenguajes.

- Cacheabilidad

Al unificar la información en los recursos, estos sirven como unidades mínimas de consulta. En este sentido resulta más fácil cachear las solicitudes de datos, ya que no existen tantas formas de solicitar la misma información. Esto no resulta muy relevante para nuestro caso de uso, ya que al consultar información en tiempo real no podemos cachear las respuestas.

A pesar de las ventajas descritas, REST también cuenta con desventajas:

- Consultas fijas

A diferencia de otras tecnologías como GraphQL, no se puede especificar una consulta que devuelva solo ciertos campos, lo cual nos lleva a devolver demasiada información en muchos casos. Este es el inconveniente de agrupar la información en recursos, a pesar de ganar en cacheabiliad, se pierde flexibilidad.

- Exceso de peticiones

Al no poder agrupar la información a voluntad del cliente, es probable que sean necesarias más peticiones para obtener la información necesaria con REST que con otras tecnologías de API. Este inconveniente es el opuesto al anterior, pero también surge de la problemática de organizar la información en recursos.

2.2.5 JSON

JSON es un formato de texto ligero orientado al intercambio de datos. A pesar de que originalmente fue diseñado para ser utilizado dentro del lenguaje de programación JavaScript en los últimos años han surgido herramientas para interactuar con el formato en gran cantidad de lenguajes de programación entre ellos Rust.

JSON está constituido por dos estructuras básicas. Por una parte, existen los objetos o diccionarios que están compuestos por conjuntos de parejas clave-

valor. Su estructura resulta similar a los HashMap en Rust o los diccionarios de Python.

Por otra parte, tenemos las listas que contienen una sucesión de valores en un orden determinado. Se puede pensar en la estructura std::vector de C++ o en el Vec de Rust.

Cada una de estas estructuras puede a su vez contener una estructura dentro de sí o valores de los tipos básicos soportados. Estos tipos son las string, cadenas de caracteres, números, booleanos y el tipo null, que se usa para connotar la ausencia de un valor.

Entre las ventajas de JSON se pueden enumerar:

- Facilidad de lectura

Resulta muy sencillo comprender las estructuras de JSON por parte de lectores humanos. Este factor resulta clave porque facilita la creación y comprensión de textos en este formato por parte del usuario. De esta forma también se facilita la creación de documentación, ya que hasta cierto punto el formato es autodocumentante.

- Amplitud de herramientas

La mayoría de los lenguajes de programación contemporáneos cuentan con una amplitud de herramientas que permiten la generación, lectura y modificación de datos en formato JSON. En el caso concreto de Rust existe la librería Serde que permite la serialización y deserialización de las estructuras del lenguaje a JSON de forma automática sin necesidad de escribir código para ello.

- Popularidad

Al ser tan ampliamente utilizado la mayoría de los desarrolladores están familiarizados con el formato. Si se utiliza JSON para definir la configuración de los productos que atañen a este trabajo esta familiaridad existente reducirá la curva de aprendizaje a la hora de utilizar el software.

Sin embargo, JSON cuenta con algunas desventajas:

- Complejidad de procesamiento

Al contar con una estructura más compleja que otras alternativas el procesamiento del formato es más pesado y puede resultar más lento.

- Uso de memoria

Al tratarse de un formato basado en texto la representación de los tipos no siempre es ideal y eso puede resultar en utilización innecesaria de memoria.

Capítulo 3
Propuesta

El objetivo de este punto es describir la especificación y el diseño de Tweakable Modbus, la librería de Modbus configurable; Ultrabus, nuestra API de comunicación con Modbus y Ultraslave nuestro simulador de esclavo configurable y reproducible.

3.1 Tweakable Modbus

Como se ha explorado en capítulos anteriores la empresa considerada enfrentaba numerosas problemáticas derivadas de la complejidad de comunicarse con dispositivos muy diversos a través de Modbus. Esta problemática impide confiar en librerías externas de Modbus y, por tanto, ha impulsado este trabajo de desarrollar implementaciones propias tanto de esclavo como de maestro en Modbus.

Sin embargo, hasta la fecha estas implementaciones han estado siempre integradas dentro del código de las distintas aplicaciones de la empresa y se han visto muy influenciadas por las especificidades de cada caso de uso. Esto a su vez ha generado problemas de mantenibilidad ya que las correcciones de los errores que se han ido detectando no se han ido corrigiendo a lo largo de las distintas bases de código.

Ha emergido de esta manera la necesidad de generar una única librería de comunicaciones con Modbus que abstraiga todo el comportamiento relacionado con el protocolo de las distintas aplicaciones. Para poder acomodar las necesidades dispares de los distintos softwares de la empresa se ha intentado maximizar la configurabilidad de cada uno de los aspectos de la comunicación. Este es el rol que Tweakable Modbus debe suplir.

3.1.1 Requisitos

Los requisitos que se plantean para el diseño de Tweakable Modbus son:

1. Soporte para subprotocolos de Modbus: inicialmente la librería solo contará con soporte para Modbus TCP pero no se descarta que un desarrollo posterior añada compatibilidad con Modbus RTU o Modbus RTUOverTCP. La interfaz de la librería debe de acomodar comunicaciones con dispositivos Modbus independientemente del subprotocolo.

2. Soporte para roles de Modbus: la librería debe de contar con soporte para trabajar tanto como maestro como esclavo.

3. Trabajar con categorías nativas de Modbus: La librería no debe de soportar tipos de datos o categorías que trabajen por encima de Modbus. Las adaptaciones que se hagan por encima del protocolo son responsabilidad de las bases de código que consumen la librería. Al mismo tiempo la librería debe encaminarse para ofrecer la máxima libertad para trabajar con esos valores superiores como se necesite.

4. Rendimiento: las aplicaciones de la empresa se ejecutan en hardware muy diverso y tienen un modelo de concurrencia basado en la existencia de muchos hilos simultáneos para gestionar distintas comunicaciones. En este contexto no solo es crucial que la librería procese preguntas y respuestas de forma rápida, sino que además debe de evitar a toda costa la espera activa.

5. Soporte para explotar más de una transferencia simultánea: Como se vio en el capítulo 2, la mayoría de dispositivos que soportan Modbus en el

mercado no implementan más de una transferencia simultánea, en consecuencia, la mayoría de herramientas de desarrollo y simulación tampoco soportan esta opción. Por las ventajas de rendimiento que puede ofrecer se ha decido darle soporte a esta opción. Cada dispositivo soporta una cantidad distinta de operaciones simultáneas por lo que este parámetro debe de ser configurable.

6. Interfaz sencilla: La interfaz de programación debe de estar estructurada de tal manera que puede ser usada por un programador que no esté familiarizado con los funcionamientos internos de Modbus.

7. Control de grano fino sobre las operaciones bloqueantes: Al soportar más de una petición simultánea la librería debe habilitar mecanismos para configurar las peticiones a realizar y posteriormente dar la orden de realizarlas todas a la vez o en grupos más pequeños.

8. Trazabilidad: A través de la integración con sistemas de logging la librería debe de aportar información de cada una de las operaciones clave en el procedimiento de la comunicación de cara a depurar posibles errores. De la misma manera estos logs también deberán de estar orientados a diagnosticar fallos en la otra parte de la comunicación.

3.1.2 Diseño

El fragmento de código 1 muestra las estructuras expuestas por la librería de cara a los desarrolladores/usuarios. De acuerdo con lo expuesto en puntos anteriores la librería se ha implementado en el lenguaje Rust. En el primer fragmento de código, líneas de la 1 a la 6, se puede ver la descripción de la estructura interna de la librería. Para utilizar una estructura modular que facilite la reutibilizalidad, extensibilidad y mantenibilidad se ha dividido el código en los módulos codec, common, communication, master, messages y slave.

```rust
mod codec;
mod common;
mod communication;
mod master;
mod messages;
mod slave;

pub use master::ModbusMasterConnection;
pub use master::ModbusMasterConnectionParams;

pub use slave::ModbusSlaveConnection;
pub use slave::ModbusSlaveConnectionParameters;
pub use slave::ModbusCallBack;

pub use common::ModbusDataType;
pub use common::ModbusResult;
pub use common::ModbusTable;
pub use common::ModbusAddress;
pub use messages::ExceptionCode;
```

Fragmento de código 1: Fichero lib.rs de Tweakable Modbus, describe la interfaz de la librería

A continuación, se van a enumerar las responsabilidades del código de cada módulo:

Codec: Contiene la lógica para la serialización y deserialización de las tramas de Modbus para las distintas implementaciones soportadas. Con serialización y deserialización se hace referencia a las funciones para transformar las estructuras de código de la librería en las tramas del protocolo Modbus expresadas como cadenas de bytes. En este caso solo están implementadas las funciones asociadas con ModbusTCP pero la estructura está diseñada para acomodar también ModbusRTU y ModbusRTUOverTCP. Las especificidades de cada una de estas implementaciones del protocolo Modbus han sido expuestas en el capítulo 2.

Common: Contiene la definición de tipos y estructuras que describen categorías básicas de Modbus y se utilizan a lo largo de la base de código. Además, algunas de estas estructuras se exponen para poder ser utilizadas por el código del usuario.

Communication: Está compuesto por la lógica de los niveles inferiores del modelo de comunicación. El objetivo de este componente es ofrecer una interfaz abstracta unificada para poder unir el funcionamiento de las distintas implementaciones del protocolo Modbus. De esta forma permite trabajar con una

conexión TCP para ModbusTCP o en el futuro, con una conexión serie RS-232 o RS-485 para ModbusRTU.

Master: Contiene el código necesario para gestionar una comunicación Master de Modbus utilizando la librería. La forma de utilizar las estructuras expuestas por este componente se explicará más adelante.

Messages: Contiene las estructuras utilizadas para representar a lo interno de la librería el comportamiento de los distintos tipos de tramas de Modbus.

Slave: Contiene el código necesario para gestionar una comunicación Slave de Modbus utilizando la librería. La forma de utilizar las estructuras expuestas por este componente también se explicará más adelante.

Además de la estructura interna del código cabe también llamar la atención sobre las distintas estructuras expuestas al usuario. Se describirán a continuación los tipos auxiliares que permiten al usuario utilizar las estructuras de maestro y esclavo para establecer sendas comunicaciones.

ModbusDataType es un enumerado de Rust que representa la información de un solo valor de Modbus. Se ha decidido utilizar el enumerado para representar este concepto ya que en Rust las distintas variantes de un enumerado pueden contener miembros distintos con tipos diferentes. Al utilizar esta capacidad del lenguaje se pueden definir dos variantes que reflejan y contienen los dos tipos básicos de Modbus: el booleano y el registro de 16 bits.

Por otra parte, el tipo ModbusResult es utilizado por la librería para representar los distintos resultados que se pueden dar al manipular el valor de una dirección utilizando el protocolo Modbus. Cuenta con tres variantes que también contienen miembros distintos entre sí: la variante de resultado de lectura que contiene un ModbusDataType con el valor leído, la variante de error que contiene un código de excepción y la variante de confirmación de escritura. Este tipo es utilizado por la librería para responder a las peticiones de modo maestro que realiza al usuario y al mismo tiempo puede ser utilizado por el usuario para concretar la respuesta a una petición cuando se trabaja desde el rol del esclavo.

ModbusTable es un enumerado simple que permite distinguir las cuatro tablas que define el estándar de Modbus: Coils, InputStatus, HoldingRegisters y InputRegisters. Como ya se explicó en el capítulo 2 estas son las tablas utilizadas en el protocolo Modbus para direccionar los distintos campos.

ModbusAddress es una estructura de Rust que contiene toda la información necesaria para direccionar un campo en el protocolo Modbus. La información contenida en la estructura es: el ID del esclavo, la tabla de Modbus del campo y su dirección representada como un entero de 16 bits.

Por último, ExceptionCode es un enumerado simple que representa los distintos tipos de excepciones descritos por el estándar de Modbus, expuestos en el capítulo 2. Utilizando este tipo, el usuario puede especificar el tipo de excepción que quiere lanzar ante la manipulación errónea de un valor en el modo esclavo.

Tras la explicación de los tipos auxiliares se explica a continuación el comportamiento de la función de maestro de Modbus expuesta por la librería. Por tanto, se debe explorar el struct ModbusMasterConnection.

Esta estructura, o struct, representa una comunicación de Modbus con el rol de maestro. Como se expuso en el capítulo 2, a través de este tipo se permite al usuario establecer una comunicación a través de Modbus con otro sistema que soporte este protocolo. Gracias a la adhesión al estándar de Modbus este otro sistema puede ser tanto una solución de software, como por ejemplo un simulador, como una solución que integre tanto hardware como firmware, por ejemplo, un PLC de uso industrial.

Aunque la comunicación establecida es contra un único sistema, la librería está preparada para soportar más de un identificador de esclavo. Esto puede parecer un tanto innecesario, pero de esta forma se acomoda el funcionamiento de dispositivos como los gateways de ModbusTCP/RTU que permiten multiplexar la conexión con distintos dispositivos ModbusRTU a través de una única conexión TCP/IP.

Una vez definida la comunicación, se permite al usuario cargar sobre ella las peticiones que quiere realizar al esclavo Modbus. En esta fase se pueden definir tantas peticiones como sean necesarias, pero estas aún no se efectuarán. Será cuando se utilice el método de query o su variante query_with_params cuando se realicen todas las peticiones definidas en bloque.

De esta forma la librería cumple con los requisitos definidos de control de grano fino y capacidad para explotar más de una transferencia simultánea. Al permitir al usuario definir una serie de peticiones y decidir el momento preciso en el que todas estas se deben efectuar damos la libertad para exprimir al máximo las capacidades de comunicación de cada dispositivo Modbus.

Sin embargo, la simultaneidad de las peticiones está sometida a otras restricciones. El usuario puede indicar la cantidad máxima de peticiones que desea que se pregunten de forma simultánea, acomodando así el funcionamiento de otros muchos dispositivos que no permiten más de una petición a la vez.

Así, si por ejemplo, se han definido tres peticiones a realizar sobre una comunicación que solo soporta la gestión de dos transacciones simultáneas, el comportamiento será el siguiente: al utilizar la función de query se lanzarán tantas peticiones como permita la configuración, en este caso dos peticiones: a partir de ese momento la librería pasará a esperar una respuesta del esclavo o que se agote el tiempo máximo de respuesta; una vez acabado el ciclo de las dos primeras peticiones se procederá a realizar la última petición y a esperar su respuesta.

Se puede ilustrar de forma más clara el funcionamiento de esta función con un ejemplo en código. A continuación, en el fragmento de código 2, se representa un ejemplo de interacción por parte de código usuario con la funcionalidad de maestro de Tweakable-Modbus.

```rust
async fn modbus_master() {
    let slave_address = SocketAddr::from_str("127.0.0.1:502").unwrap();

    //Definimos la comunicación ModbusMasterConnection aportando el socket del esclavo
    let mut master_connection = ModbusMasterConnection::new_tcp(slave_address);

    //Añadimos nuestra primera petición
    let result = master_connection.add_read_coils_query(1, 0, 1);

    if let Result::Err(err) = result {
        eprintln!(
            "Ha habido un error definiendo una petición: {}",
            err.to_string()
        );
        return;
    }

    //Añadimos nuestra segunda petición
    let result = master_connection.add_write_multiple_holding_registers_query(
        8,
        25,
        vec![1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    );

    if let Result::Err(err) = result {
        eprintln!(
            "Ha habido otro error definiendo una petición: {}",
            err.to_string()
        );
        return;
    }

    //Realizamos las querys!
    let query_result = master_connection.query().await;

    if let Result::Err(err) = query_result {
        eprintln!(
            "Ha habido un error al realizar las peticiones: {}",
            err.to_string()
        );
        return;
    }

    //Como sabemos que no ha sido un error desencapsulamos los resultados
    let correct_results = query_result.unwrap();

    //Gestionamos los resultados
    for (address, result) in correct_results {
        print!(
            "En la dirección {} de la tabla {:?} del esclavo {}...",
            address.address, address.table, address.slave_id
        );
        match result {
            ModbusResult::Error(exception_code) => {
                println!("ha habido un error con el código : {:?}", exception_code);
            }
            ModbusResult::ReadResult(value) => {
                println!("se ha leído el valor {:?}", value)
            }
            ModbusResult::WriteConfirmation => {
                println!("se ha escrito correctamente");
            }
        }
    }
}
```

Fragmento de código 2: Ejemplo de utilización de las funciones de maestro de Tweakable Modbus

Como se puede apreciar, lo primero a hacer para usar la librería es definir la dirección del socket contra el que estableceremos la comunicación. En rdyr caso se ha definido la IP de loopback del equipo y el puerto 502, por lo que para que el ejemplo funcione correctamente se debería estar ejecutando un esclavo de Modbus sobre ese puerto.

De acuerdo con los requisitos establecidos, las únicas comunicaciones soportadas por el momento son las de ModbusTCP. Sin embargo, gracias a la estructura de la API de la librería añadir nuevos protocolos como ModbusRTU o MobusRTUOverTCP sería tan sencillo como añadir nuevos métodos para instanciar estos tipos de comunicaciones, el resto de la interfaz podría continuarse utilizando del mismo modo.

Una vez declarada la dirección del socket ésta se puede utilizar para instancia un struct ModbusMasterConnection contra dicha dirección. Como se ha explicado con anterioridad este struct contiene los miembros y métodos necesarios para representar una comunicación de Modbus en la que se ejerce el rol de maestro.

Tras haber definido la comunicación se empieza a definir las distintas peticiones a realizar. La primera petición añadida consiste en una lectura de la primera dirección de la tabla de Coils del esclavo con el ID 1.

Para definir las peticiones se hace uso de las funciones definidas en la API con este fin. Para este caso concreto utilizaremos la función add_read_coils_query a la que se le indicará por parámetros que se quiere leer del esclavo 1, la dirección 0 y que se quiere leer un único valor.

Tras añadir la nueva query se debe de comprobar que se ha agregado de forma correcta gestionando el caso del error. Cabe tener en cuenta que estos errores no son derivados del envío y recepción de la trama, que aún no ha sucedido. Su función pasa por permitir al usuario gestionar errores debidos a la introducción de parámetros incorrectos en la query.

La siguiente petición a realizar es una escritura de 10 Holding Registers en el esclavo 8. En este caso además de indicar la ID del esclavo y la dirección

también se ha de añadir los valores a escribir. Para transmitir esta información, se utiliza la estructura de datos Vec de la librería estándar de Rust. Al igual que antes se debe de gestionar el error.

Cabe señalar que en el ejemplo no se enumeran todos los métodos desarrollados para definir peticiones y que como se puede asumir contamos con métodos para realizar escrituras y lecturas sobre todas las tablas que permiten estas operaciones. A su vez contamos con otro método para definir peticiones con el function code ReadWriteMultipleRegisters que se explicó en el capítulo 2.

Una vez se han añadido todas las peticiones que se desea realizar sobre el esclavo se procede a llamar al método query. Esta función asíncrona desata los procedimientos necesarios para llevar a cabo las transacciones. En este caso se ha optado por utilizar la configuración por defecto para realizar las peticiones, si se hubiera querido configurar algún aspecto concreto de la comunicación se debería de haber utilizado el método query_with_params.

El comportamiento de este método es análogo al anterior, pero permite pasar como argumento el struct ModbusMasterConnectionParams. Este struct contiene la información necesaria para permitir al usuario detallar algunos aspectos concretos del funcionamiento de la comunicación. En concreto los parámetros que se pueden especificar son el tiempo máximo de respuesta y la cantidad máxima de peticiones simultáneas.

Cuando acabe la comunicación y prosiga el flujo del código la función devolverá los resultados. Antes de poder procesarlos se debe gestionar la posibilidad de que se haya dada algún error en la comunicación.

Una vez desencapsulados, los resultados son expresados como un HashMap que relaciona direcciones de Modbus, expresadas como ModbusAddress, con resultados de Modbus, expresados como ModbusResult. Para acabar con el ejemplo se procesan en un bucle uno a uno los resultados y se muestra por consola la información obtenida de cada dirección bien sea el valor leído, la confirmación de la escritura correcta de un valor o un código de excepción que informa de alguna problemática con la operación.

Tras la explicación de las estructuras de la librería que permiten definir una comunicación Modbus de maestro se expone el funcionamiento de los tipos asociados con el rol de esclavo en Modbus.

Al igual que con la comunicación de maestro contamos con el struct ModbusSlaveConnection para representar un servidor de Modbus. Para definir un esclavo se debe aportar la dirección en la que escuchará el esclavo Modbus y una estructura que implemente el trait ModbusCallback.

Este trait está diseñado para permitir al usuario definir el comportamiento que se quiere desencadenar cuando el protocolo intente leer o escribir sobre una dirección. Con este fin se debe implementar los dos métodos que se heredan de la interfaz: on_read y on_write. En el fragmento de código 3 se puede observar su declaración.

```
1. #[async_trait::async_trait]
2. pub trait ModbusCallBack: Send + Sync {
3.      async fn on_read(&self, addr: ModbusAddress) -> Result<ModbusDataType,
        ExceptionCode>;
4.      async fn on_write(&self, addr: ModbusAddress, value: ModbusDataType) -> Result<(),
        ExceptionCode>;
5. }
```

Fragmento de código 3: Declaración del trait ModbusCallBack

Como se puede apreciar en el fragmento de código anterior la cabecera de las funciones abstractas define la estructura necesaria para gestionar la lectura y escritura sobre una sola dirección. De esta forma el cálculo de los valores que corresponden a cada dirección se debe realizar de forma independiente del contexto de la petición. Desde el punto de vista del código del usuario es indiferente si la petición ha sido múltiple o de un solo campo o si se ha realizado a través de instrucciones de lectura y escritura simples o con operaciones combinadas como ReadWriteMultipleRegisters.

De esta forma se encamina al usuario a implementar soluciones que aporten consistencia a los valores expuestos en las tablas y que no condicionen el comportamiento de estos a la forma en la que se realizan las peticiones.

Además, se consigue reducir considerablemente la cantidad de código que debe de escribir el usuario para interactuar con las funciones de esclavo de la librería.

La definición del struct ModbusSlaveConnection no implica la inicialización del servidor. Para empezar a escuchar peticiones en la dirección de socket designada se debe de utilizar el método serve.

Para exponer en más detalle el funcionamiento de esta funcionalidad se va a ilustrar a través de un ejemplo. Se va a implementar un servidor tipo “echo” que responda a todas las peticiones de lectura que se realicen sobre la tabla InputRegisters con un valor equivalente a su dirección; así la dirección uno contendrá el valor uno, la dirección dos el valor dos y así sucesivamente. No se ve necesario implementar la escritura para un ejemplo tan sencillo, por lo que se responderá a cualquier petición de escritura con una confirmación, pero no se procesarán los valores aportados de ninguna forma. Para implementar este comportamiento se ha desarrollado el fragmento de código 4:

```
struct EchoModbusCallback {}

#[async_trait::async_trait]
impl ModbusCallBack for EchoModbusCallback {
    async fn on_read(&self, addr: ModbusAddress) -> Result<ModbusDataType, ExceptionCode> {
        if addr.table == ModbusTable::InputRegisters {
            return Ok(ModbusDataType::Register(addr.address));
        } else {
            return Err(ExceptionCode::IllegalFunction);
        }
    }

    async fn on_write(
        &self,
        addr: ModbusAddress,
        value: ModbusDataType,
    ) -> Result<(), ExceptionCode> {
        return Ok(());
    }
}

async fn modbus_slave() {
    let slave_address = SocketAddr::from_str("127.0.0.1:502").unwrap();

    let callback = EchoModbusCallback {};
    let mut slave_connection = ModbusSlaveConnection::new_tcp(slave_address, Box::new(callback));

    slave_connection.serve().await.unwrap();
}

```

Fragmento de código 4: Ejemplo de utilización de las funciones de esclavo de Tweakable Modbus

Como se puede apreciar lo primero que se hace es definir el struct EchoModbusCallback este struct es el que utilizaremos para implementar el trait ModbusCallback y definir el comportamiento del esclavo. En este caso no ha sido necesario definir ningún miembro para el struct.

Una vez definido el struct se implementará para él el trait ModbusCallback. Al tratarse de un trait compuesto por funciones asíncronas se ha decidido utilizar la librería async_trait para simplificar las declaraciones de las funciones. El trait define dos funciones a implementar.

En la función on_read se recibe el valor de la dirección por la que se está preguntando. En este caso se comprueba que sea una petición hacia la tabla InputRegisters y de ser así se devuelve el valor de la dirección. En caso de tratarse de una petición sobre otra tabla simplemente se devuelve el exception code que indica que esta operación no está permitida. Para la función on_write simplemente se devuelve un resultado de Ok independientemente de los parámetros aportados.

Una vez se ha definido el trait en la función del ejemplo se define la dirección sobre la que debe de escuchar el esclavo. A diferencia de con el rol de maestro la dirección que se defina aquí no representa al dispositivo contra el que se va a abrir una comunicación si no que representa la dirección propia sobre la que el equipo va a empezar a escuchar peticiones.

Acto seguido, se instancia el struct EchoModbusCallback y con este junto a la dirección del esclavo se define una ModbusSlaveConnection.

Una vez definida la comunicación lo único que debe hacerse es utilizar la función serve para empezar a gestionar peticiones. Al igual que con el rol de maestro existe una variante de serve llamada serve_with_params que permite al usuario definir algunos de los parámetros de la comunicación.

Esta función toma como argumento el struct ModbusSlaveConnectionParams que permite especificar los esclavos sobre los que el servidor debe permitir operar, la whitelist de direcciones IP que se deben

permitir y el tiempo durante el cual se debe de mantener viva la conexión TCP tras realizar una transacción.

Para concluir la explicación sobre el ejemplo cabe destacar que la función serve no acaba hasta que surge un error y que por tanto monopoliza la ejecución de la tarea sobre la que se ejecuta. Si se quiere que el esclavo funcione en paralelo con otras funcionalidades de código dentro de la misma aplicación se debe de llamar a la función creando una nueva tarea como en el fragmento de código 5.

```
1.        tokio::spawn(async move{ slave_connection.serve().await.unwrap() });
```

Fragmento de código 5: Utilización del código de ejemplo de las funciones de esclavo en una nueva tarea

3.2 Ultrabus

El objetivo de Ultrabus es aislar el resto del software de la empresa de las complejidades específicas de la comunicación con dispositivos Modbus. Para este propósito, se plantea una aplicación que funcione como un gateway exponiendo la información de los dispositivos de Modbus a través de una API REST.

3.2.1 Requisitos

Los requisitos establecidos para la fase de diseño para Ultrabus son:

1. Exponer valores simples y agregados: La aplicación debe de mostrar el estado en tiempo real de los valores monitorizados. Al mismo tiempo, debe permitir la consulta de datos agregados en grupos de distintos períodos temporales dando la opción de filtrar entre dos momentos.

2. Configurabilidad: Las comunicaciones, esclavos y valores a monitorizar deben de ser configurables por el usuario. De la misma manera para cada una de estas categorías se debe de poder especificar parámetros que permitan maximizar la compatibilidad con todo tipo de dispositivos Modbus.

3. Reproducibilidad: Todo el funcionamiento de la aplicación debe de estar definido por la configuración. No se debe permitir modificar el funcionamiento de la aplicación sin cambiar la configuración. Toda la configuración debe de estar contenida en un solo fichero.

4. Soporte para monitorizar más de una comunicación: El programa debe soportar la comunicación con más de un dispositivo de Modbus y debe de mostrar la información de todas las comunicaciones a través de una única interfaz.

5. Capacidad para trabajar con tipos de datos por encima del estándar de Modbus: A pesar de que el estándar de Modbus solo soporta trabajar con valores booleanos y con registros de 16 bits la mayoría de dispositivos exponen tipos de datos distintos a través de la combinación o reinterpretación de los contenidos de uno o más registros. Ultrabus debe de soportar los tipos de datos: booleano, byte, enteros de 16, 32 y 64 bits con y sin signo y números de coma flotante de precisión simple y doble. Al no estar estos tipos de datos dentro del estándar de Modbus también se deberá contar con flexibilidad a la hora de interpretar los bits de los valores nativos para formar estos nuevos valores. Tanto la endianness, como la longitud del valor y el desplazamiento inicial del valor deben de ser configurables.

5. Minimización de las peticiones: Se debe de minimizar el efecto de la monitorización sobre los equipos. En este sentido, se deben agrupar todas las peticiones posibles. La frecuencia con la que se solicita cada dato también debe de ser configurable para este fin.

6. Capacidad para trabajar con almacenamiento limitado: La aplicación debe poder desplegarse en dispositivos con almacenamiento limitado. Con este fin debe existir un procedimiento para la eliminación de valores y agregaciones viejos. Se priorizará la eliminación de valores simples y agregaciones de períodos cortos ya que al ser estas más numerosas ocupan más espacio. De esta manera se podrá configurar la cantidad de valores y agregaciones de cada período a conservar en disco.

7. Agnosticidad de las agregaciones: Resulta imposible saber a priori el tratamiento de datos que se realizará con las agregaciones. Es por lo que se considera interesante almacenar más de una métrica para dar más opciones a la hora de realizar un análisis de datos. De cada agregación se almacenará la media, mediana, moda, máximo, mínimo y la cantidad de muestras sobre las que se ha construido.

3.2.2 Diseño

En la figura 16 se puede observar un esquema de los distintos subsistemas de la aplicación.

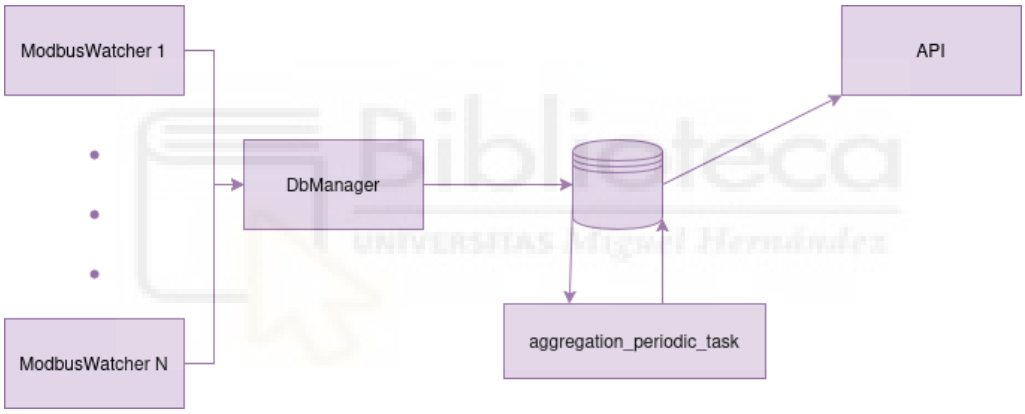

Figura 16: Estructura interna de Ultrabus

Se observa que la aplicación cuenta con un ModbusWatcher para gestionar cada comunicación Modbus descrita en el fichero de comunicación. Cada ModbusWatcher es responsable de construir las peticiones óptimas, realizar las preguntas al esclavo y gestionar los resultados para extraer los valores a monitorizar.

Estos valores son transferidos al DbManager que los inserta en la base de datos. De esta forma este componente multiplexa el acceso a la base de datos para todos los ModbusWatcher evitando interbloqueos. La comunicación entre los distintos ModbusWatcher y el DbManager utiliza el tipo mpsc de la librería de

Tokio. Este tipo nos habilita un canal Multiple Producer Single Consumer que permite pasar mensajes entre distintas tareas de Tokio de forma no bloqueante.

Esto resulta necesario debido a las limitaciones que tiene SQLite para gestionar varias sesiones al mismo tiempo. Junto a esta medida también se usa la librería R2D2 de Rust que permite regular el acceso a la base datos entre distintas tareas de forma bloqueante. De esta forma se pueden realizar las escrituras y lecturas sobre la base de datos desde distintas tareas sin riesgo a caer en condiciones de carrera.

Sobre la base de datos también opera la aggretation_periodic_task que tiene dos objetivos: por una parte, ha de construir las agregaciones a partir de los valores simples leyendo de la base de datos todos los valores correspondientes a un período y realizando las operaciones necesarias para finalmente insertar los valores de la agregación de nuevo en la base de datos; por otra parte, también se encargará de eliminar los registros que excedan la capacidad especificada en la configuración.

Por último, la información de la aplicación será consultable a través de una API REST que realizará consultas de lectura sobre la base de datos para obtener la información solicitada. Esta API será la encargada tanto de proveer el valor en tiempo real, como las agregaciones y además permitirá consultar la configuración de los valores.

- Estructura de la API:

En este apartado se expondrá el diseño de la estructura de la interfaz de programación de Ultrabus:

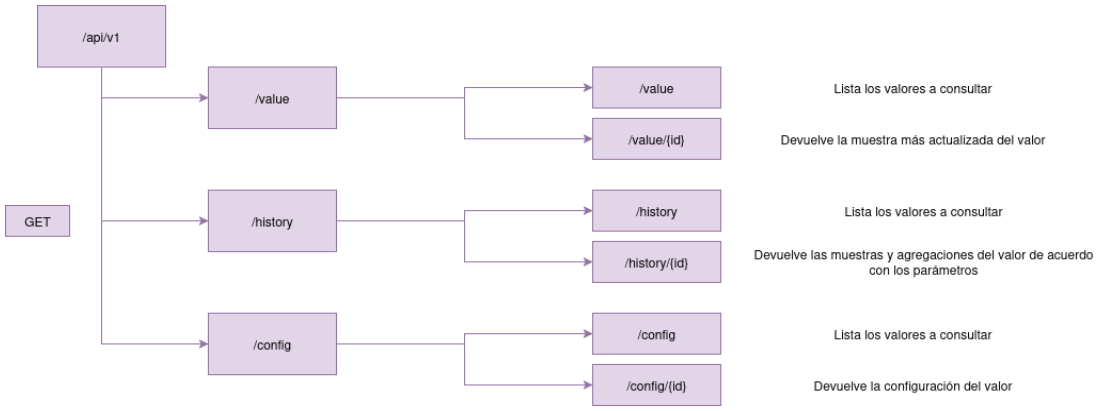

Figura 17: Estructura de la API de Ultrabus

Como se puede observar en la figura 17, la API cuenta con 3 endpoints diferenciados:

El endpoint /value permite consultar la última muestra de cada valor configurado. Para ello se debe aportar el ID del valor como un argumento en el path. La estructura JSON que devuelve está recogida en el fragmento de código 6:

```
1. {
2.     "value_id": "Holding_de_prueba",
3.     "value": {
4.         "Integer": 0
5.     },
6.     "secs_since_epoch": 1756738699
7. }
```

Fragmento de código 6: Respuesta del endpoint /value de Ultrabus

Se indica el identificador del valor, el valor en sí con su tipo y el momento en el que se recibió el valor en formato POSIX.

El endpoint /history permite consultar todos los valores y agregaciones que se ajusten a los parámetros indicados en la petición. Los parámetros permitidos son:

- start_date: Marca el timestamp inicial de la petición. Se devolverán todos los valores posteriores a esta fecha. Se expresa en formato POSIX.

- end_date: Marca el tiemestamp final de la petición. Se devolverán todos los valores anteriores a esta fecha. Se expresa en formato POSIX.

- min_group: Marca el tipo de agregación mínimo incluido en la petición. Se devolverán todas las agrupaciones y valores que pertenezcan a un grupo igual o superior. Los valores permitidos son NoGrouping, Minute, Hour y Day.

- max_group: Marca el tipo de agregación máximo incluido en la petición. Se devolverán todas las agrupaciones y valores que pertenezcan a un grupo igual o inferior. Los valores permitidos son NoGrouping, Minute, Hour y Day.

La estructura de la respuesta se encuentra en el fragmento de código 7:

```
[
      {
        "aggregation_info": {
            "value_id": "Holding_de_prueba",
            "period": "Minute",
            "start_time": {
                "secs_since_epoch": 1756322497,
                "nanos_since_epoch": 0
            },
            "finish_time": {
                "secs_since_epoch": 1756322557,
                "nanos_since_epoch": 0
            },
            "average": {
                "Integer": 20
            },
            "median": {
                "Integer": 20
            },
            "moda": {
                "Integer": 20
            },
            "min": {
                "Integer": 20
            },
            "max": {
                "Integer": 20
            },
            "ammount": 609
        }
    },
    {
        "value_info": {
            "value_id": "Holding_de_prueba",
            "value": {
                "Integer": 20
            },
            "secs_since_epoch": 1756321314
        }
    },
]
```

Fragmento de código 7: Respuesta del endpoint /history de Ultrabus

La respuesta es una lista de los elementos que cumplen las condiciones indicadas en la petición. Se aprecian elementos de dos tipos, por una parte, hay valores simples que presentan una estructura similar al endpoint anterior y por otra parte están las agregaciones que contienen la fecha de inicio y fin, el identificador del valor al que hacen referencia, el período de la agregación y el valor de la media, la mediana, el mínimo, el máximo, la moda y la cantidad de elementos que forman la agregación.

Por último, el endpoint /config permite consultar la información de un valor especificado como parámetro en el path. Devuelve la misma estructura que se haya reflejado en el fichero de configuración.

```
{
    "id": "Holding_de_prueba",
    "starting_address": 10000,
    "table": "HoldingRegisters",
    "starting_bit": 0,
    "bit_length": 16,
    "data_type": "UnsignedInteger16",
    "byte_swap": false,
    "word_swap": false,
    "double_word_swap": false,
    "poll_time": "100ms",
    "max_polls_to_keep": 2592000,
    "max_minute_aggregations_to_keep": 1814400,
    "max_hour_aggregations_to_keep": 8760,
    "max_day_aggregations_to_keep": null
}
```

Fragmento de código 8: Respuesta del endpoint /config de Ultrabus

El funcionamiento de todos estos campos se explicará con posterioridad en el apartado de descripción de la configuración de Ultrabus.

Al realizar peticiones a los endpoints sin especificar una id se brindará una lista de los valores sobre los que se puede realizar dicha operación. Este comportamiento es idéntico para los 3 endpoints.

- Diseño de la base de datos

Para almacenar la información de las muestras y las relaciones se ha dispuesto una base de datos usando SQLite. El modelo entidad relación de la base de datos es el siguiente:

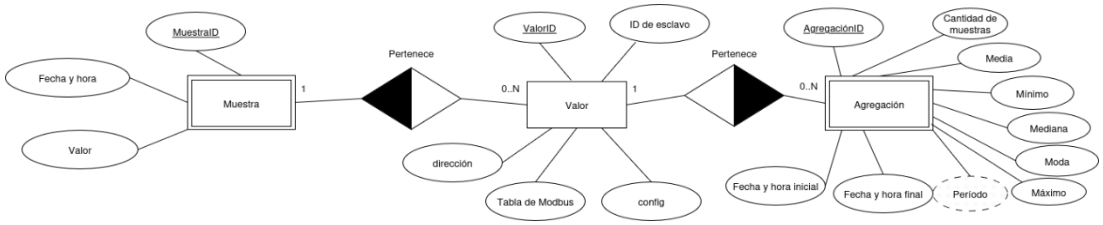

Figura 18: Modelo entidad relación de la base de datos de Ultrabus

Se detalla también la información de los contenidos de cada table en la base de datos.

Tabla Valor		
Nombre	Tipo	Descripción
ValorID	Cadena	(PK) Identifica el valor, se corresponde con el ID para consultar en la API
ID de esclavo	Entero	ID del esclavo de Modbus del que se extrae el valor
Dirección	Entero	Dirección de Modbus de la que se extrae el valor
Tabla de Modbus	Enumerado	Especifica la tabla de Modbus de la que se extrae el valor
Config	JSON	Configuración del valor del canal

Tabla 3: Contenidos de la tabla Valor

Tabla Muestra		
Nombre	Tipo	Descripción
Muestra ID	Entero	(PK) Identifica la muestra
ValorID	Cadena	(FK) Especifica a que valor corresponde la muestra
Fecha y hora	Fecha	Momento en el que se recibió la muestra
Muestra	Blob	Valor de la muestra capturada como una cadena de bytes.

Tabla 4: Contenidos de la tabla Muestra

Tabla Agregación		
Nombre	Tipo	Descripción
Agregación ID	Entero	(PK) Identifica la agregación
ValorID	Cadena	(FK) Especifica a que valor corresponde la agregación
Fecha inicio	Fecha	Momento en el que se recibió la muestra que formó la agregación
Fecha fin	Fecha	Momento en el que se recibió la última muestra que formó la agregación
Período	Enumerado	Representa la duración del período de la agregación. Los valores válidos son minuto, hora y día.
Media	Blob	Valor de la media guardado como una cadena de bytes
Mediana	Blob	Valor de la mediana guardado como una cadena de bytes
Moda	Blob	Valor de la moda guardado como una cadena de bytes
Mínimo	Blob	Valor del mínimo guardado como una cadena de bytes
Máximo	Blob	Valor del máximo guardado como una cadena de bytes
Cantidad	Entero	Cantidad de muestras que forman la agregación

Tabla 5: Contenidos de la tabla Agregación

Se pueden apreciar una serie de modificaciones entre el modelo entidad relación planteado inicialmente y la estructura final de las tablas que se han construido. Se puede observar que las tablas Agregación y Muestra no se han

construido como entidades débiles, es decir, la clave foránea de la tabla Valor no se ha incluido en su clave primaria. Se ha tomado esta decisión para mejorar el rendimiento de la base de datos simplificando la estructura de estas tablas.

También es interesante llamar la atención sobre el tipo de datos utilizado para almacenar los valores tanto de las muestras como de las agregaciones. Se ha elegido el tipo Blob ya que en la misma tabla deben de coexistir tipos de datos distintos que no se pueden conocer a priori.

- Configuración Ultrabus

La configuración se almacena en un único fichero JSON que debe de ser generado por el usuario. Como se ha expuesto en el capítulo dos se ha elegido el formato de texto JSON por su simplicidad y popularidad.

La estructura básica de un fichero de configuración se refleja en el fragmento 9:

```
[
    {
        "ip": "127.0.0.1",
        "port": 8080,
        "slaves": [
            {
                "id": 1,
                "values": [
                    {
                        "id": "Holding_de_prueba",
                        "starting_address": 10000,
                        "bit_length": 16,
                        "data_type": "UnsignedInteger16",
                        "table": "HoldingRegisters",
                        "poll_time": "100ms"
                    }
                ]
            }
        ]
    }
]
```

Fragmento de código 9: Ejemplo de fichero de configuración de Ultrabus.

Se puede apreciar que el fichero de configuración está compuesto de una serie de arrays de JSON anidadas. Así en la lista de nivel superior se definen las distintas conexiones que debe de gestionar Ultrabus. Cada una cuenta con una serie de parámetros asociados entre ellos la lista de esclavos a monitorizar. Cada uno de los esclavos definidos en esta lista cuenta sus propios parámetros para

adaptarse al funcionamiento concreto del dispositivo y a su vez cuenta con una lista de los valores a monitorizar en el esclavo. Los valores, por último, cuentan tanto con parámetros para definir como se deben de consultar y almacenar como con parámetros que rigen la interpretación y formato de los datos. Estos últimos parámetros son compartidos con Ultraslave.

- Parámetros

A continuación, se presenta la enumeración de los parámetros de configuración que soporta el software junto con la función que lleva a cabo cada uno.

Parámetros de conexión (Ultrabus)		
Nombre	Función	Notas
ip	Dirección IP del esclavo Modbus TCP	Por defecto: 127.0.0.1
port	Puerto TCP del esclavo Modbus TCP	Por defecto: 502
max_simultaneous_connections	Cantidad máxima de conexiones que abrir contra el esclavo Modbus TCP	Por defecto: 1
max_response_time	Tiempo máximo que esperar para la respuesta a una pregunta	Por defecto: 1s

Tabla 6: Parámetros de configuración de conexión en Ultrabus

Parámetros de esclavo (Ultrabus)		
Nombre	Función	Notas
id	Identificador del esclavo Modbus	Por defecto: 1, debe de ser un valor entre 1 y 255
max_register_ammount	Cantidad máxima de valores sobre los que operar en una sola petición	Por defecto: 255
max_gap_size_in_query	Tamaño máximo de los “huecos” permitidos en una petición	Por defecto: 0

Tabla 7: Parámetros de configuración de esclavo en Ultrabus

Parámetros de valor (Ultrabus)		
Nombre	Función	Notas
id	Cadena de texto de identificación del valor	Marca la ruta por la que será accesible el valor en la API. Este campo es obligatorio
starting_address	Primera dirección que forma parte del valor	Este campo es obligatorio
table	Tabla de Modbus en la que se encuentra el valor	Los valores válidos son: DiscreteInput (Input Status), Coils, InputRegisters, HoldingRegisters. Este campo es obligatorio
poll_time	Frecuencia de consulta del valor	Este campo es obligatorio.
max_polls_to_keep	Máximo de valores simples que guardar en la base de datos	Si asignamos null al valor no habrá límite al guardado de muestras. Por defecto: 2592000

max_minute_aggregations_to_keep	Máximo de agregaciones de un minuto a guardar en la base de datos	Si asignamos null al valor no habrá límite al guardado de agregaciones de un minuto. Por defecto: 1814400, tres semanas.
max_hour_aggregations_to_keep	Máximo de agregaciones de una hora a guardar en la base de datos	Si asignamos null al valor no habrá límite al guardado de agregaciones por hora. Por defecto: 8760, un año
max_day_aggregations_to_keep	Máximo de agregaciones de un día a guardar en la base de datos	Si asignamos null al valor no habrá límite al guardado de agregaciones por día. Por defecto: null

Tabla 8: Parámetros de configuración de valor en Ultrabus

Parámetros de formato de valor (Ultrabus y Ultraslave)		
Nombre	Función	Notas
starting_bit	Primer bit que forma parte del valor	Debe de ser un valor entre uno y 16. Por defecto: 1
bit_length	Longitud del valor en bits	Este campo es obligatorio
data_type	Tipo de datos del valor	Este campo es obligatorio. Los valores permitidos son Boolean, Byte, UnsignedInteger16, SignedInteger16, UnsignedInteger32, SignedInteger32, UnsignedInteger64, SignedInteger64, Float y Double
byte_swap	Permite intercambiar los bytes pares por los impares	Por defecto: false

word_swap	Permite intercambiar los grupos de 2 bytes pares por los impares	Por defecto: false
double_word_swap	Permite intercambiar los grupos de 2 bytes pares por los impares	Por defecto: false

Tabla 9: Parámetros de formato de valor en Ultrabus y Ultraslave

3.2.3 Caso de uso

Con el objetivo de mostrar con más detalle el funcionamiento de la aplicación se plantea el siguiente caso de uso:

Se considera un dispositivo Modbus que expone los parámetros de funcionamiento de un grupo electrógeno en el contexto de la planta eléctrica de un buque de tamaño mediano. Se indica tanto la dirección IP y el puerto TCP del dispositivo, 192.168.0.43:502, como el ID de esclavo Modbus del dispositivo, 3.

De los campos expuestos por el dispositivo se necesita monitorizar el valor de las revoluciones por minuto del motor, el voltaje proporcionado por el generador y si el grupo se encuentra acoplado al resto de la instalación eléctrica de la embarcación.

La información de las revoluciones por minuto se expone como un entero sin signo de 16 bits, el voltaje se representa como un número de coma flotante de precisión simple y el estado de acople del grupo se representa como un booleano. A su vez el valor de las revoluciones por minuto y el voltaje es de solo lectura, pero el valor del estado de acople permite también la escritura. Además, se indica que para poder realizar las auditorías del navío se debe almacenar el voltaje medio del grupo cada hora durante al menos un año. Al consultar la documentación del dispositivo se encuentra la siguiente tabla referente a los campos Modbus disponibles.

Campo	Formato	Tabla	Dirección
Revoluciones por minuto	Entero sin signo de 16 bits	InputRegisters	200
Voltaje	Número de coma flotante de precisión simple	InputRegisters	140
¿Acoplado?	Booleano	Coils	6

Tabla 10: Valores expuestos por el dispositivo de control de grupo electrógeno

En la tabla se indica la información necesaria para interpretar y monitorizar los valores. Por una parte, se aporta la información sobre cómo se representa el valor en los registros de Modbus, es decir el formato, y por otra parte se representa la información necesaria para el direccionamiento de los distintos campos, es decir la tabla y la dirección.

Para resolver el problema propuesto se desplegará una instancia de Ultrabus en un dispositivo conectado a la misma red IP que el equipo a monitorizar. Para desplegar el software será necesario descargar el binario precompilado de la aplicación y crear un fichero de configuración que se adapte al caso de uso. Como se ha visto en el punto de diseño este fichero es el encargado de designar los campos a monitorizar y los parámetros de la comunicación utilizando los parámetros que se han expuesto en dicho punto. Con la información aportada se ha compuesto el siguiente fichero de configuración recogido en el fragmento de código 10:

```
[
    {
        "ip": "192.168.0.43",
        "port": 502,
        "slaves": [
            {
                "id": 3,
                "values": [
                    {
                        "id": "revoluciones_por_minuto",
                        "starting_address": 200,
                        "table": "InputRegisters",
                        "bit_length": 16,
                        "data_type": "UnsignedInteger16",
                        "poll_time": "500ms"
                    },
                    {
                        "id": "voltaje",
                        "starting_address": 140,
                        "table": "InputRegisters",
                        "bit_length": 32,
```

```
                        "data_type": "Float",
                        "poll_time": "100ms",
                        "max_hour_aggregations_to_keep": 8760
                    },
                    {
                        "id": "acoplado",
                        "starting_address": 6,
                        "table": "Coils",
                        "bit_length": 1,
                        "data_type": "Boolean",
                        "poll_time": "1s"
                    }
                ]
            }
        ]
    }
]
```

Fragmento de código 10: Configuración de Ultrabus para monitorizar los valores expuestos por el grupo electrógeno

Al analizar el fichero de configuración se aprecia que primero se ha configurado la dirección y el puerto del dispositivo. Dentro de la conexión con el dispositivo se ha definido un esclavo con el ID 3 que contiene los campos del grupo electrógeno.

Para cada valor se ha definido un ID, la primera dirección que compone el campo, la tabla Modbus en la que se encuentra, su longitud en bits, el tipo que se debe usar para interpretarlo y la frecuencia con la que consultar el valor. Para decidir la frecuencia con la que consultar los valores se ha consultado de nuevo la documentación del dispositivo para averiguar la frecuencia con la que se reporta el valor de los sensores.

Como se puede apreciar en el fragmento para el valor del voltaje se ha añadido un parámetro adicional llamado max_hour_aggregations_to_keep. Como se ha visto en el punto anterior este parámetro marca la cantidad máxima de agregaciones de una hora de duración a almacenar en la base de datos. De acuerdo con lo solicitado se ha puesto este parámetro a 8760 que son las horas que tiene un año.

Se debe tener en cuenta que en el caso del campo de voltaje Ultrabus no preguntará solo por la dirección 140, sino que también consultará el estado del siguiente registro. Esto se debe a que la longitud del valor se ha configurado en

32 bits y como se expuso en el capítulo 2 la longitud de un InputRegister es de 16 bits.

Si se laza el programa con la configuración aportada la monitorización del dispositivo comenzará. A partir de este punto se podrá empezar a realizar consultas a través de la API para consultar el estado de los distintos valores. La estructura de la API de Ultrabus ya ha sido expuesta en el punto anterior.

Si por ejemplo, se quiere consultar si el grupo electrógeno se encuentra acoplado actualmente se realizaría una petición HTTP con el método GET al siguiente endpoint.

http://{ip}:{puerto_de_api}/api/v1/value/acoplado

Si en cambio se quiere consultar el valor de las medias horarias de voltaje entre dos momentos dados del tiempo se podría realizar la siguiente petición GET a la API:

http://{ip}:{puerto_de_api}/api/v1/history/start_date={timestamp_inicial}&end_date={timestamp_final}&max_group=Hour&min_group=Hour

3.3 Ultraslave

De la misma manera que la necesidad de aislar las comunicaciones con dispositivos Modbus motivó el desarrollo de Ultrabus, la necesidad de poder testear distintas configuraciones de Modbus y simular de forma sencilla situaciones límite del protocolo de comunicación ha sido la causa para desarrollar Ultraslave.

Ultraslave es un esclavo configurable y reproducible de Modbus TCP orientado a ejecutarse de forma “headless”, es decir, sin interfaz. Además de exponer una comunicación de esclavo de Modbus que permite la consulta de valores, Ultraslave también expone una API al usuario que permite consultar y modificar el estado de la aplicación.

3.3.1 Requisitos

Los requisitos establecidos en la fase de diseño son los siguientes:

1. Simulación de esclavo: Ultraslave debe de permitir la configuración y simulación de esclavos de Modbus TCP. La aplicación a su vez expondrá el contenido del esclavo a través de un puerto definido por el usuario para poder ser consultado.

2. Configurabilidad: Tanto los valores del esclavo como los parámetros de la comunicación con él deben de ser configurables. Esta configurabilidad debe de estar orientada hacia la posibilidad de simular el comportamiento de equipos que no se adhieren completamente al estándar.

3. Reproducibilidad: Al igual que con Ultrabus, todo el comportamiento de la aplicación debe de emerger del fichero de configuración. No se debe de poder modificar el funcionamiento de la aplicación sin modificar la configuración. Toda la configuración debe de estar contenida en un solo fichero.

4. Soporte para peticiones de más de un maestro de forma simultánea: Ultraslave debe de permitir gestionar más de una comunicación simultánea. De la misma manera usando los mecanismos de Modbus TCP también se debe soportar la gestión de más de una transacción simultánea para el mismo maestro.

5. Capacidad para trabajar con datos por encima del estándar: Al igual que con Ultrabus se debe de soportar la exposición por Modbus de tipos de datos distintos de los soportados de forma nativa por el protocolo. Además, se debe de maximizar la configuración en la representación de los valores para adaptarse a la simulación de distintos dispositivos.

6. Capacidad de ejecutarse sin interfaz: Ultraslave debe de poder ejecutarse sin interfaz para poder ser integrado con procesos de testing automatizado.

3.3.2 Diseño

En la figura 19 podemos ver los distintos componentes implicados en el funcionamiento de Ultraslave y las relaciones que existen entre ellos.

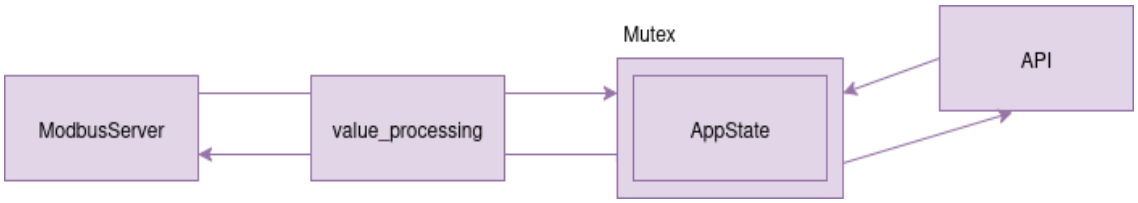

Figura 19: Estructura interna de Ultraslave

El estado de cada uno de los valores se almacena en la estructura AppState y es sobre esta estructura sobre la que realizan accesos tanto la API como el servidor de esclavo de Modbus. Para gestionar la concurrencia de estos accesos se hace uso de un mutex, en concreto, se usa tokyo::sync::mutex el tipo que nos ofrece la librería estándar de Tokio para gestionar el acceso concurrente entre distintas tareas.

Al plantear el diseño surgió una duda sobre el formato que se debería utilizar para almacenar en memoria el estado de los valores. Por una parte el ModbusServer necesita tener la información relativa a los valores de cada registro de Modbus, pero por otra parte desde la API sen debe poder consultar el estado real de los valores independientemente de su representación en el ModbusServer.

La solución que se ha decidido implementar es almacenar los valores tal como se representan en la API. Cuando el ModbusServer procesa una petición de lectura no accede a regiones de memoria ya inicializadas con los valores de las tablas de Modbus sí no que en ese momento se calcula el valor correspondiente a cada registro. Esta solución consume un poco más de

recursos, pero garantiza un menor retardo en la propagación de los cambios al eliminar la necesidad de generar una representación intermedia de datos.

De la misma manera cuando se realiza una petición de escritura los valores asignados a los registros que forman parte de nuestro tipo de dato complejo son utilizados para recalcular su valor. De esta manera se garantiza la atomicidad del cambio de valor evitando situaciones en las que se quiera consultar un valor durante un cambio llevando a un estado en el que algunos registros ya han cambiado, pero otros aún no.

- Estructura de la API

En la figura 20 se puede ver reflejada la estructura de la API de Ultraslave:

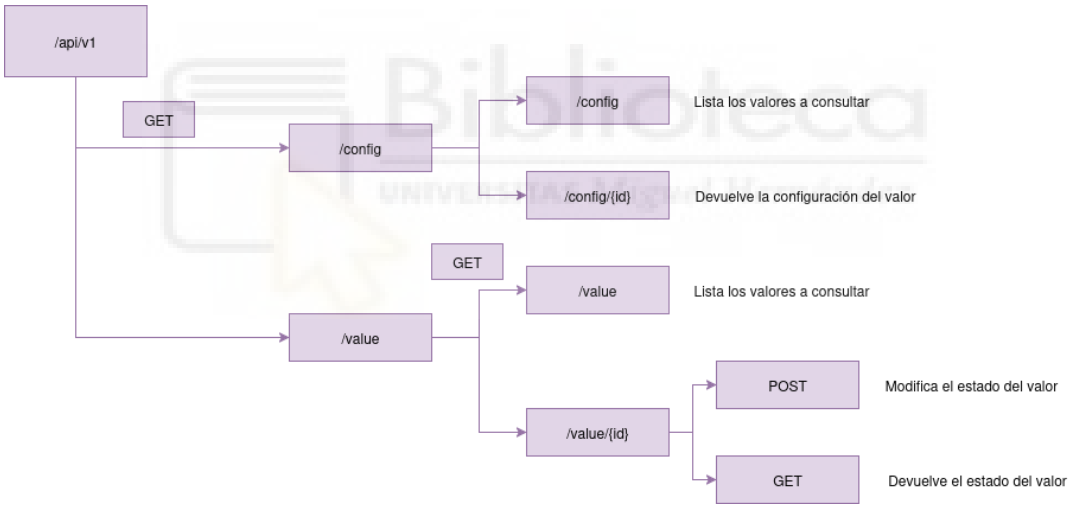

Figura 20: Estructura de la API de Ultraslave

El endpoint de /config permite al usuario consultar el estado de la configuración de cada valor. Si se realiza una consulta sin indicar ningún identificador en la ruta nos devuelve una lista JSON con los valores que se pueden consultar.

Al consultar la configuración de un valor en concreto devuelve la siguiente estructura JSON que se puede observar en el fragmento 11:

```
{
    "id": "prueba",
    "starting_address": 0,
    "table": "HoldingRegisters",
    "starting_bit": 0,
    "bit_length": 32,
    "data_type": "UnsignedInteger32",
    "byte_swap": false,
    "word_swap": false,
    "double_word_swap": false,
    "default_value": {
        "Integer": 6666
    }
}
```

Fragmento de código 11: Respuesta del endpoint /config de Ultraslave

Se puede apreciar que los campos de formateo y direccionamiento del valor son comunes con respecto a Ultrabus pero además existe el campo default_value. Se explicará su contenido en el apartado de configuración más adelante.

El endpoint /value es el que permite al usuario monitorizar y modificar el estado de los distintos valores expuestos por el esclavo. De acuerdo con los principios REST se utilizará la operación GET para obtener el valor actual y la operación POST para modificar el valor.

Al consultar el estado del valor devolverá la siguiente estructura JSON:

```
{
    "Integer": 6666
}
```

Fragmento de código 12: Respuesta del endpoint /value de Ultraslave

Se puede apreciar que los contenidos de la respuesta son mínimos especificando solo el tipo de datos y el valor a asignar. Esta misma estructura es la que se debe utilizar para modificar el estado del valor. Para este fin se puede utilizar una petición POST contra el mismo endpoint en cuyo contenido se encuentre la estructura con el tipo y valor deseado. Se debe tener en cuenta que solo se podrán asignar valores con tipos compatibles a los que están definidos en la configuración de Ultraslave.

Al igual que con Ultrabus toda la configuración de la aplicación está incluida en un solo fichero JSON. A continuación, se presenta un ejemplo del fichero de configuración en el fragmento de código 13:

```
[
    {
        "port": 5007,
        "slaves": [
            {
                "id": 1,
                "values": [
                    {
                        "id": "prueba",
                        "table": "HoldingRegisters",
                        "starting_address": 0,
                        "bit_length": 32,
                        "default_value": {
                            "Integer": 6666
                        },
                        "data_type": "UnsignedInteger32"
                    }
                ]
            }
        ]
    }
]
```

Fragmento de código 13: Ejemplo de configuración de Ultraslave

Como se puede apreciar la raíz del fichero JSON es una lista de configuración de las distintas conexiones de esclavo. Dentro de cada uno de estos elementos además de la configuración propia de la comunicación se cuenta con una lista de esclavos. Para cada uno de los esclavos a su vez existe una lista de valores a exponer en la comunicación. De esta manera el funcionamiento de los ficheros de configuración de Ultrabus y Ultraslave resulta bastante similar.

- Parámetros

Se expondrá a continuación el funcionamiento de cada uno de los parámetros de Ultraslave. Los parámetros de formateo de valor son comunes con Ultrabus, descritos anteriormente.

Parámetros de conexión (Ultraslave)		
Nombre	Función	Notas
port	Puerto TCP del esclavo Modbus TCP	Por defecto: 502
connection_time_to_live	Tiempo que mantener viva la conexión TCP con el maestro	Por defecto: 3s

Tabla 11: Parámetros de configuración de conexión en Ultraslave

Parámetros de esclavo (Ultraslave)		
Nombre	Función	Notas
id	Identificador del esclavo de Modbus	Por defecto: 1
max_coils	Cantidad máxima de valores que forman parte de la tabla Coils	Por defecto: 65535
max_discrete_inputs	Cantidad máxima de valores que forman parte de la tabla Discrete Inputs	Por defecto: 65535
max_holding_registers	Cantidad máxima de valores que forman parte de la tabla Holding Registers	Por defecto: 65535
max_input_registers	Cantidad máxima de valores que forman parte de la tabla Input Registers	Por defecto: 65535

Tabla 12: Parámetros de configuración de esclavo en Ultraslave

Parámetros de valor (Ultraslave)		
Nombre	Función	Notas
id	Cadena de texto de identificación del valor	Marca la ruta por la que será accesible el valor en la API. Este campo es obligatorio
starting_address	Primera dirección que forma parte del valor	Este campo es obligatorio
default_value	Valor por defecto al iniciar la aplicación	Para especificar el valor por defecto del formato se debe de utilizar la misma estructura que para cambiarlo desde la API. Por ejemplo: "default_value": { "Integer": 6666 }

Tabla 13: Parámetros de configuración de valor en Ultraslave

3.3.3 Caso de Uso

Para ejemplificar mejor el funcionamiento de la aplicación se va a exponer un caso de uso práctico del software en el contexto de trabajo habitual de la empresa.

En la especificación del proyecto del sistema de alarmas y control de un navío se específica la necesidad de monitorizar los sensores de nivel de los distintos tanques del navío. Los tanques que se deben monitorizar son el tanque de aguas sucias, el tanque de combustible y el tanque de agua dulce. Estos sistemas actúan como esclavos Modbus y exponen diversos datos para ser consultados. Se encuentran conectados a un único gateway de Modbus RTU/TCP que expone la información a través del puerto TCP 502. Antes de realizar la instalación se quiere asegurar la plena compatibilidad de las soluciones de software de la empresa con los equipos en cuestión.

Con este fin, se decide utilizar Ultraslave para simular el funcionamiento de todos los esclavos y garantizar la compatibilidad. Para ello se investiga la

documentación de los equipos a simular y se encuentra que cada uno de ellos expone el nivel de los tanques en forma de porcentaje en la tabla de Input Registers, la información del tanque de combustible y del tanque de agua dulce, que cuentan con una precisión superior, se representan como números de coma flotante mientras que la información del tanque de aguas sucias, que cuenta con un sensor menos preciso, se representa como un entero de 16 bits. Además, se expone como un Input Status si el tanque está siendo rellenado en este momento o no. La documentación proporciona la siguiente tabla:

Campo	Formato	ID de esclavo	Tabla	Dirección
Porcentaje de lleno del tanque de aguas sucias	Entero sin signo de 16 bits	1	Input Registers	300
¿Se está llenando el tanque de aguas sucias?	Booleano	1	Input Status	450
Porcentaje de lleno del tanque de combustible	Número de coma flotante de precisión simple	3	Input Registers	300
¿Se está llenando el tanque de combustible?	Booleano	3	Input Status	450
Porcentaje de lleno del tanque de agua dulce	Número de como flotante de precisión simple	8	Input Registers	300
¿Se está llenado el tanque de agua dulce?	Booleano	8	Input Status	450

Tabla 14: Parámetros aportados por los sistemas de gestión de tanques

Como se ha expuesto en el punto anterior todo el comportamiento de Ultraslave viene definido por el fichero de configuración especificado por el

usuario. Para el caso planteado se ha desarrollado el siguiente fichero de configuración:

```
[
    {
        "port": 502,
        "slaves": [
            {
                "id": 1,
                "values": [
                    {
                        "id": "porcentaje_lleno_aguas_sucias",
                        "table": "InputRegisters",
                        "starting_address": 300,
                        "bit_length": 16,
                        "default_value": {
                            "Integer": 0
                        },
                        "data_type": "UnsignedInteger16"
                    },
                    {
                        "id": "llenando_aguas_sucias",
                        "table": "DiscreteInputs",
                        "starting_address": 450,
                        "bit_length": 1,
                        "default_value": {
                            "Boolean": false
                        }
                    }
                ]
            },
            {
                "id": 3,
                "values": [
                    {
                        "id": "porcentaje_lleno_combustible",
                        "table": "InputRegisters",
                        "starting_address": 300,
                        "bit_length": 32,
                        "default_value": {
                            "Integer": 0
                        },
                        "data_type": "Float"
                    },
                    {
                        "id": "llenando_combustible",
                        "table": "DiscreteInputs",
                        "starting_address": 450,
                        "bit_length": 1,
                        "default_value": {
                            "Boolean": false
                        }
                    }
                ]
            },
            {
                "id": 8,
                "values": [
                    {
                        "id": "porcentaje_lleno_agua_dulce",
                        "table": "InputRegisters",
                        "starting_address": 300,
                        "bit_length": 32,
                        "default_value": {
                            "Integer": 0
                        },
                        "data_type": "Float"
                    },
```

```
                    {
                        "id": "llenando_agua_dulce",
                        "table": "DiscreteInputs",
                        "starting_address": 450,
                        "bit_length": 1,
                        "default_value": {
                            "Boolean": false
                        }
                    }
                ]
            }
        ]
    }
]
```

Fragmento de código 14: Configuración de Ultraslave para la monitorización del estado de los tanques

Como se puede apreciar en la raíz del fichero de configuración se listan las distintas conexiones a simular. En el caso a tratar se simula la conexión del dispositivo gateway TCP/RTU único que permite comunicar con los distintos esclavos RTU. Se especifica que esta conexión se encuentra disponible a través del puerto 502.

Dentro de la conexión se definen distintos esclavos que se corresponden con los distintos dispositivos que monitorizan el estado de los tanques: el dispositivo 1 gestiona el tanque de aguas sucias, el dispositivo 3 el tanque de combustible y el dispositivo 8 el tanque de agua dulce.

A su vez, cada dispositivo contiene una lista de valores a monitorizar. Como se puede apreciar estos valores se corresponden con los especificados en la tabla del fabricante. De cada valor se aportan los parámetros de formato comunes a Ultrabus y Ultraslave además del valor por defecto que tomará el campo al iniciar el simulador.

Al iniciar el software estos valores se expondrán a través de una conexión con el rol de esclavo Modbus en el puerto designado además de a través de una API expuesta en un puerto que el usuario puede configurar por parámetro. Se pasa a enumerar las acciones que se abren al usuario para interactuar con los campos a través de la API.

Como se expuso en el punto anterior la API cuenta con dos endpoints: config y value. Desde config se puede consultar la configuración de un valor en

concreto mientras que a través de value se puede consultar y modificar los valores de los distintos campos.

Un ejemplo de una petición de consulta de configuración, en este caso sobre el campo porcentaje_lleno_aguas_sucias sería la siguiente:

GET http://{ip}:{puerto}/api/v1/config/porcentaje_lleno_aguas_sucias

Y la respuesta brindada por el programa sería la siguiente:

```
{
    "id": "porcentaje_lleno_aguas_sucias",
    "starting_address": 300,
    "table": "InputRegisters",
    "starting_bit": 0,
    "bit_length": 16,
    "data_type": "UnsignedInteger16",
    "byte_swap": false,
    "word_swap": false,
    "double_word_swap": false,
    "default_value": {
        "Integer": 0
    }
}
```

Fragmento de código 15: Ejemplo de respuesta del endpoint /config de Ultrabus

Como se puede ver se indica la información correspondiente al campo por el que se ha preguntado. Cabe anotar que se devuelven más campos que los configurados inicialmente, esto se debe a que a estos campos se le han asignado valores por defecto al no haber sido rellenados en la configuración.

Si en cambio se realiza la siguiente petición

GET http://{ip}:{puerto}/api/v1/config/porcentaje_lleno_aguas_sucias

Se contestará con la siguiente estructura que contiene el valor actual del campo

```
{
        "Integer": 0
}
```

Fragmento de código 16: Ejemplo de respuesta del endpoint /value de Ultraslave

Si sobre el mismo endpoint se realiza una petición utilizando el método POST a la que se adjunta la misma de estructura de JSON se podrá asignar un nuevo valor al campo.

Capítulo 4 Conclusiones y trabajo futuro

4.1.- CONCLUSIONES

En este punto se expondrán los resultados de este trabajo en relación con los objetivos marcados inicialmente en el capítulo 1. No solo se analizará el desempeño en los objetivos principales de desarrollo (Tweakable Modbus, Ultrabus y Ultraslave) si no que prestará especial atención a los objetivos transversales del proyecto entendiendo este como un prototipo funcional para la renovación del conjunto de herramientas de software empleadas por la empresa considerada.

Los objetivos planteados inicialmente para el desarrollo de Tweakable Modbus han sido alcanzados satisfactoriamente. En particular, se ha conseguido construir una herramienta lo suficientemente versátil para poder adaptarse a todos los casos de uso que se dan en la empresa y lo suficientemente simple como para poder ser expandida y adaptada para el uso general en ámbitos industriales.

Los objetivos marcados para el desarrollo de Ultrabus también se han alcanzado. Cabe destacar que la aplicación ha sido desplegada en procesos de testing y depuración dentro del departamento de I + D de la empresa. Estas

primeras experiencias en un entorno real han revelado que la facilidad de configuración junto con la reproducibilidad hace de la aplicación una herramienta muy útil no solo a la hora de diseñar soluciones permanentes para proyectos sino también para realizar pruebas intensivas de funcionamiento.

Lo mismo se puede decir de Ultraslave que también ha experimentado cierta adopción limitada en el testing en el departamento de I + D. En este caso se debe destacar la capacidad del software de simular equipos con condiciones adversas lo cual ha permitido gestionar errores que hasta ahora no podían ser simulados.

En cuanto a los objetivos secundarios del desarrollo se valorarán por separado los relativos a la adopción de lenguajes con seguridad de memoria y por otra parte los respectivos a la utilización de tecnologías asíncronas.

En lo referente a los primeros, el desarrollo de este proyecto ha demostrado la idoneidad de Rust para el tipo de desarrollos que emprende la empresa. Sin embargo, la alta curva de aprendizaje y la alta especialización del equipo de desarrolladores en C++ han resultado en que aún no haya empezado el proceso de adopción de este lenguaje. En la actualidad se están valorando opciones para realizar una prueba más inmersa en los ritmos de trabajo de la empresa con otra pieza de software que cuente con una gran independencia del resto de sistemas desarrollados, permitiendo que el desarrollo en Rust no condicione al resto del equipo.

En cuanto a los segundos, sí que se puede hablar de un gran éxito. La utilización de tecnologías asíncronas se ha adoptado enérgicamente en varios de los desarrollos emprendidos y ha brindado muy buenos resultados a la hora de mejorar el rendimiento de distintas aplicaciones desarrolladas por la empresa. Al no haber adoptado Rust se ha decidido utilizar el framework de C++ Asio que ofrece una funcionalidad similar a pesar de no contar con la misma ergonomía.

4.2.- POSIBLES DESARROLLOS FUTUROS

En este último punto se expondrán las líneas de trabajo que se han abierto tras finalizar el desarrollo del proyecto:

Una mejora que se considera que tendría un gran impacto sería dar soporte a Modbus RTU y Modbus RTUOverTCP. Como se explicó en el capítulo 2 Modbus no es completamente agnóstico al conjunto de comunicaciones que se usan bajo él y en la experiencia real se han encontrado muchos dispositivos con los que no se ha podido interactuar ya que estaban limitados a una de estas variantes del protocolo. De la misma manera, como se explicó en el capítulo 3, Tweakable Modbus fue diseñado para poder soportar estos protocolos en un futuro y tanto Ultrabus como Ultraslave podrían adaptarse sin modificaciones sustantivas.

Otro desarrollo que se debería considerar es la posibilidad de añadir interfaces gráficas de usuario a Ultrabus y Ultraslave. Esta mejora haría las aplicaciones aptas para muchos más casos de uso orientados a personal menos especializado en tecnologías de la información. Sin embargo, estos desarrollos no deben comprometer el carácter headless de las aplicaciones permitiendo que estas sigan siendo desplegadas sin interfaz.

Por último, el modelo de Ultrabus para consultar datos a través de API REST ha demostrado ser muy útil para la extracción de datos de equipos industriales. Se considera que desarrollos similares para otros protocolos del ámbito industrial como CAN J1939, NMEA2k, Profibus etc. podrían resultar igualmente útiles. Con estos desarrollos y aprovechando el soporte de REST para las jerarquías en capas se podría implementar una única API unificada que muestre información de dispositivos con distintos protocolos de forma conjunta.

Capítulo 5
Bibliografía

[1] Índice de especificaciones de la Modbus Foundation

https://www.modbus.org/modbus-specifications

[2] Especificación del protocolo de aplicación Modbus

https://assets.noviams.com/novi-file-uploads/modbus/pdfs-and-documents/Modbus_Application_Protocol_V1_1b3.pdf

[3] Guía de implementación de mensajes de Modbus

https://assets.noviams.com/novi-file-uploads/modbus/pdfs-and-documents/Modbus_Messaging_Implementation_Guide_V1_0b.pdf

[4] Evolucionando a Modbus seguro, Incibe

https://www.incibe.es/incibe-cert/blog/evolucionando-modbus-seguro

[5] Búsqueda en Shodan, dispositivos con el puerto TCP 502 expuesto a internet a 29 de octubre de 2025

https://www.shodan.io/search?query=port%3A502

[6] Modbus Slave

https://www.modbustools.com/modbus_slave.html

[7] Modbus Tools

https://github.com/serhmarch/ModbusTools

[8] Kepserver

https://www.kepserverexopc.com/

[9] The Rust Book

Steve Klabnik, Carol Nichols

No Starch Press (2023)/ Octubre de 2025

[9] Speed comparison, Niklas Heer

https://github.com/niklas-heer/speed-comparison

[10] Memory safety, The Chromium Projects

https://www.chromium.org/Home/chromium-security/memory-safety/

[11] Tokio

https://tokio.rs/tokio/tutorial

[12] Tokio Internals

https://cafbit.com/post/tokio_internals/

[13] Tamaño en memoria de Sqlite

https://www.sqlite.org/footprint.html

[14] What is a REST API? Redhat

https://www.redhat.com/es/topics/api/what-is-a-rest-api

[15] REST APIs, IBM

https://www.ibm.com/es-es/think/topics/rest-apis

[16] Especificación de JSON

https://www.json.org/json-es.html