Proyecto Fin de Carrera
Ingeniería de Telecomunicación (Telemática)

Control remoto del brazo robótico RX90

Autor: Azahara Mª Porras Tejada
Tutor: Ángel Rodríguez Castaño

Dep. Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla

Sevilla, 2017

Proyecto Fin de Grado
Ingeniería de Telecomunicación

Control remoto del brazo robótico RX90

Autor:
Azahara Mª Porras Tejada

Tutor:
Ángel Rodríguez Castaño
Profesor titular

Dep. de Ingeniería de Sistemas y Automática
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2017

Proyecto Fin de Grado: Control remoto del brazo robótico RX90

Autor: Azahara Mª Porras Tejada

Tutor: Ángel Rodríguez Castaño

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Vocales:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2017

El Secretario del Tribunal

A mi familia

A mi pareja

Agradecimientos

Este trabajo no hubiera sido posible sin la orientación de mi tutor D. Ángel Rodríguez Castaño, al cual le propuse el tema y ha ido guiando los avances del mismo. Mencionar también a los profesores y alumnos del laboratorio que me han ayudado cuando surgía algún contratiempo.

Agradecer también el apoyo incondicional de mi familia y mi pareja. Sin ellos no habría soportado tantas horas de duro trabajo.

Azahara Mª Porras Tejada

Sevilla, 2017

Resumen

En la actualidad, cada vez más dispositivos se encuentran conectados a Internet. Este auge surge ante la necesidad de poder controlar a distancia objetos de la vida cotidiana, es decir, sin estar presentes físicamente en el lugar donde se encuentra el aparato. Estos avances se están introduciendo lentamente en el mundo de la industria y ésta cada vez posee más elementos que se conectan a la red para poder ser controlados y monitorizados remotamente.

Este proyecto se enfoca en ese sentido, puesto que se propone usar un típico brazo robótico conectado a un servidor para poder ser controlado remotamente.

Abstract

At present, more and more devices are connected to the Internet. This boom arises from the need to be able to control objects of daily life at the distance, this is, without the elements being physically in the place where the apparatus is located. These advances are being introduced slowly in the world of industry and each time they have more elements that connect to the network to be controlled and monitored remotely.

This project focuses on this, since it proposes to use a typical robotic arm connected to a server to be remotely controlled.

-translation by google-

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

"El esfuerzo de utilizar las máquinas para emular el pensamiento humano siempre me ha parecido bastante estúpido. Preferiría usarlas para emular algo mejor"

Edsger Dijkstra

La industria 4.0 está en pleno auge. A pesar del recelo de las empresas de introducir nuevos avances tecnológicos en su núcleo productivo, cada vez son más las que incorporan técnicas de computación distribuida y análisis de grandes cantidades de datos. Estos sistemas se usan en otros entornos para ampliar las funcionalidades de los equipos, así como aumentar el rendimiento de los mismos, haciendo su uso transparente al usuario.

En este proyecto se desarrollará una práctica de computación distribuida basada en una estructura de cliente-servidor para manejar un brazo robótico dentro de una red IP [1]. Se hará posible su control manual a través del teclado y también se podrá disponer de los datos aportados por el RX90 para conocer su estado en todo momento.

1.1 Motivación del proyecto

Actualmente la mayoría de industrias suelen estar conectadas mediante grandes buses de campo y para el control y mantenimiento de cualquier máquina, el operario debe desplazarse hasta el sitio para cambiar la configuración. El control remoto se está introduciendo lentamente en el mundo de la industria para controlar aquellas máquinas que son difícilmente accesibles por un operario, o bien para tener una visión más generalizada de la empresa. Con este fin han surgido dispositivos Scada que permiten una monitorización continua de las máquinas y permite tratar ciertos puntos de la configuración.

En este proyecto se va a avanzar más hacia un modelo de control remoto de una máquina, proponiendo encapsular los sistemas de transmisión de información tradicionales en paquetes que puedan ser enviados inalámbricamente, como puede ser un protocolo TCP. Esto nos va a aportar más flexibilidad a la hora de crear los paquetes y el canal en el que se transmita puede ser muy variado, puesto que usamos un protocolo normalizado y ampliamente tratado en redes fuera del entorno industrial. El uso del protocolo TCP nos

aportara fiabilidad en la comunicación sin tener que cambiar el lenguaje usado por las maquinas hasta ahora. Esto nos proporcionará mayor integración con las máquinas ya existentes sin necesidad de sustituirlas, puesto que solo será necesario conectarlas a un servidor para hacerlas funcionar.

1.2 Alcance y objetivos

El objetivo principal de este proyecto es establecer una comunicación Modbus TCP entre un equipo cliente y un equipo que hará de servidor para establecer la comunicación entre un operador y una maquina remota. Esta será controlada por el operario situado en la misma red IP y no será necesaria la proximidad para que la máquina pueda trabajar. Además, podremos supervisar su estado en todo momento y comprobar que realiza correctamente sus tareas.

El proyecto partía de un programa ya en funcionamiento para el RX90, en el que a través de un terminal se introducían los comandos que queríamos ejecutar en dicho brazo robótico. Dicho ejecutable se basaba en la librería libmodbus para el correcto tratamiento de los datos.

Por otro lado, se desarrolló un cliente-servidor TCP que recibía las peticiones del cliente y modificaba, en su caso, los registros que eran necesarios para su posterior lectura por el servidor.

A continuación, se pasó a la integración de ambos servicios, consiguiendo que a través del cliente anteriormente propuesto se mandaran las ordenes pertinentes y el servidor las interpretara y reenviara en lenguaje V+ al RX90. Para ello se tuvieron que realizar modificaciones como el tipo de datos que leía el servidor, así como insertar en los registros la posición correspondiente del brazo robótico. También se introdujeron mejoras como poder proporcionar la IP del cliente por teclado y la posición del robot mediante el mismo.

1.3 Estructura del documento

En esta memoria se abordará la parte conceptual del proyecto, dejando para los anexos el código empleado en su puesta en funcionamiento. Se intentará, en la medida de lo posible, justificar el texto mediante imágenes y gráficas, con el fin de aclarar el contenido.

En el primer capítulo comenzamos planteando el entorno de desarrollo y las ideas generales del proyecto. Se explica brevemente los temas a tratar y su estructura.

En el segundo capítulo veremos los conceptos principales que fundamentan la transmisión de datos desde el operario, que trabaja con el cliente TCP hasta la transmisión final al brazo robótico mediante comunicación serie.

En los siguientes capítulos se abordarán las implementaciones del cliente y el servidor con más detalle. Se justificarán sus partes y se desarrollarán las funcionalidades de cada uno de ellos.

El quinto capítulo hablaremos de las herramientas de desarrollo utilizadas. Pondremos en valor los equipos que hemos usado y explicaremos sus características técnicas básicas.

Por último, en el sexto capítulo analizaremos las conclusiones de nuestro estudio. Se plantearán también las posibles mejoras en un futuro y como estas pueden ayudar en desarrollo industrial.

Cerraremos el proyecto con los códigos añadidos en los anexos de las implementaciones del cliente y servidor.

2 PROTOCOLOS

"Los estándares son siempre obsoletos. Eso es lo que los hace estándares"

Alan Bennett

LA comunicación desde el usuario hasta el brazo robótico RX90 se ha realizado en diferentes capas, ya que cada pieza soluciona problemas diferentes. Asimismo, se han utilizado diferentes protocolos en cada una de ellas que encapsulan a los protocolos superiores, quedando una torre de protocolos especial para la comunicación en este entorno industrial.

Con el fin de poder explicar el funcionamiento completo es necesario que se conozca el trabajo que realiza cada uno de los protocolos por separado. Para establecer la estructura y orden de los datos en cada capa es indispensable remitirse a la normativa especial de cada protocolo, definidos por la autoridad correspondiente.

2.1. Comunicación por Puerto serie

La comunicación serial es un método de comunicación sencillo de transmisión de datos. se basa en el envío y recepción de la información bit a bit entre el transmisor y el receptor. Habitualmente la comunicación serial se usa para la transmisión de datos en formato ASCII.

Históricamente se ha usado este tipo de comunicación en todos los ordenadores para conectar los distintos componentes, así como para enlazar dispositivos externos al mismo. Actualmente las conexiones serie han quedado relegadas a entornos industriales y, en general, los ordenadores han dejado de incorporar este tipo de puerto. En este sentido, la conexión del RX90 con el portátil se realiza a través de un adaptador que convierte un puerto USB cualquiera a puerto serie.

El estándar más extendido para la comunicación serie viene dado por el RS-232 [2] (Recommended Standard 232) que establece el protocolo de la transmisión de datos, el cableado, las señales eléctricas y los conectores en los que debe basarse la conexión. Para que pueda establecerse la comunicación es necesario que las características de los puertos a cada extremo sean iguales. Algunas de ellas son:

- Velocidad de transmisión: Número de bits por segundo que se envían.

- Bits de datos: cantidad de bits que se transmiten en un paquete. Depende del tipo de información que se transfiere, pero debe concordar el número que se espera en la transmisión y en la recepción.

- Bits de parada: últimos bits de un paquete que indican el final del mismo. Debido a que la comunicación es asíncrona, los relojes del transmisor y el receptor no están sincronizados y dichos bits indican el fin de la transmisión de un paquete.

- Paridad: sirve para comprobar errores en la transmisión que afecten negativamente a la comunicación.

2.1.1 Ventajas e inconvenientes de Puerto serie

La principal ventaja de la comunicación serie es su simplicidad. Únicamente se usa para la comunicación punto a punto y simplemente usa pulsos de tensión en la línea para transmitir los datos.

Como inconvenientes podemos citar que las conexiones son las lentas que en otras comunicaciones punto a punto como USB, pero en nuestro caso no nos hace falta esa velocidad. Otro inconveniente es que no todos los ordenadores poseen un puerto RS-232, pero podemos solucionarlo mediante adaptadores dedicados a convertir cualquier puerto USB en un puerto serie. Por ejemplo, el adaptador que se ha utilizado durante el desarrollo del mismo que es del fabricante Magic Control Technology Corp.

2.1.2 Trama

Al ser un puerto físico, el RS-232 no posee una trama como tal. Lo que encontramos en el conector es el conjunto de pines por los que está compuesto.

Físicamente se dispone de un conector DB9 hembra en el lado del servidor y un conector DB9 macho en el lado del brazo.

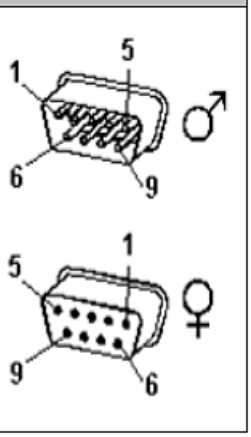
#	Pin	E/S	Función	Conector DB 9
1			Tierra de Chasis	
2	RXD	E	Recibir Datos	
3	TXD	S	Transmitir Datos	
4	DTR	S	Terminal de Datos Listo	
5	SG		Tierra de señal	
6	DSR	E	Equipo de Datos Listo	
7	RTS	S	Solicitud de Envío	
8	CTS	E	Libre para Envío	
9	RI	S	Timbre Telefónico	

Tabla 1. Pines más importantes del conector RS-232

Para realizar la comunicación se utilizan tres líneas de transmisión: una para tierra (o referencia), otra para transmitir y una tercera para recibir. Debido a que la transmisión es asíncrona, es posible enviar datos por una línea mientras se reciben datos por otra.

2.2. Comunicación Modbus

Modbus [3] es un protocolo industrial que surgió en el año 1973 ante la necesidad de establecer un intercambio de información entre autómatas programables. Anteriormente las conexiones entre los aparatos se realizaban mediante lógica cableada. La aparición de Modbus supuso un gran avance y mejora de las instalaciones ya que no era preciso modificar el cableado para cambiar el funcionamiento de un autómata. Esto permitió eliminar gran parte del cableado existente hasta entonces porque sólo bastaba la línea de comunicación entre máquinas para hacerlas funcionar.

El primer sistema que hizo uso de este sistema de comunicación fue Modicon [4] (Modular Digital Controller), inventado por la empresa Bedford Associates, y se ha convertido en un estándar de facto gracias su la publicación de forma abierta y libre de cánones en 1979.

Modbus es un protocolo a nivel de aplicación que establece un tipo de comunicación maestro-esclavo. Generalmente el maestro lo asociamos a un dispositivo interfaz humano-máquina y el esclavo lo ligamos a un dispositivo lógico programable.

2.1.3 Ventajas e inconvenientes de Modbus

La ventaja principal consiste en su capacidad de adaptabilidad a cualquier dispositivo. Todos los tipos de aparatos industriales (PLC, HMI, paneles de control, Drivers, controles de movimiento, dispositivos de entrada/salida…) son susceptibles de comunicarse mediante el uso de Modbus. Además, se puede llevar a cabo sobre cualquier tipo de red, tanto en la comunicación serie como sobre IP en redes Ethernet o Wifi.

Uno de los inconvenientes es el tamaño de la trama. La capacidad de su PDU está limitada, porque viene heredada de la primera implementación Modbus sobre comunicación serie. Este problema podemos solucionarlo realizando una asociación de comandos a números enteros, tanto en el maestro como en el esclavo, a fin de que en el mensaje únicamente tengamos que mandar un número. La asociación debe ser la misma en uno y otro extremo, ya que si no los comandos serán interpretados erróneamente.

2.1.4 Trama Modbus

Como hemos adelantado anteriormente, la trama Modbus tiene un tamaño limitado debido a la herencia dejada por el uso de este protocolo sobre líneas de comunicación serie. La capacidad máxima de estas líneas venía dada por el RS485, que viene establecido en 256 bytes.

Si calculamos a partir de esa dimensión el tamaño máximo de la PDU Modbus obtenemos que:

256 - Dirección del servidor (1 byte) - CRC (2 bytes) = 253 bytes

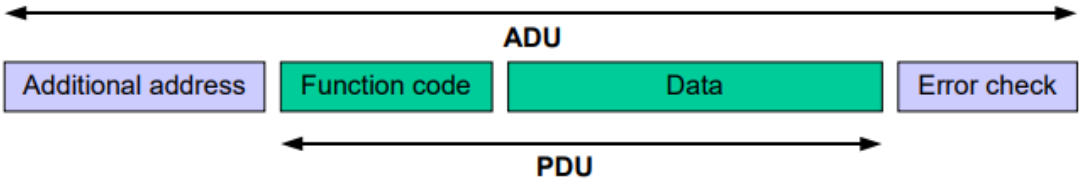


Figura 1. Trama Modbus [5]

La Unidad de datos de protocolo (PDU) está formada por un código de función y el conjunto de datos que necesita la misma. Cada código de función expresa un comportamiento predefinido entre el maestro y el esclavo y es este último el que lo implementa en base a su propia programación.

2.3. Comunicación TCP

El protocolo TCP tuvo sus orígenes en los años 70. En la Agencia de Investigación de Proyectos Avanzados de Defensa de Estados Unidos se desarrolló un protocolo de comunicaciones que permitiría asegurar la fiabilidad de la red extremo a extremo sin temor de pérdida de paquetes entre las estaciones de comunicación.

El protocolo de control de transmisión es un conjunto de reglas de la capa de transporte del modelo OSI. Está definido en el estándar RFC 793 [6] y tiene un uso muy extendido. Es utilizado frecuentemente por multitud de aplicaciones para aportar fiabilidad al canal extremo a extremo.

Las características principales de este estándar son:

- Orientado-a-conexión: se requiere el establecimiento de un vínculo entre los procesos cliente y servidor antes del intercambio de mensajes

- Punto-a-punto: la transmisión se realiza entre un equipo emisor y un receptor, siendo el resto de la red transparente a éstos.

- Flujo de bytes, fiable y ordenado: se establece un número de secuencia para que los paquetes que viajen por distintos caminos puedan ser ordenados en el destino

- Full-dúplex: se puede tener un flujo de datos bi-direccional en la misma conexión.

- Control de Flujo: conoce en todo momento el estado del buffer de recepción, debido a que se reciben asentimientos de cada paquete que ha llegado correctamente.

- Control de Congestión: el protocolo decide en cada momento si los paquetes pueden transmitirse por la red.

2.2.1 Ventajas e inconvenientes de TCP

Una de las ventajas de TCP es la aportación de fiabilidad a la conexión extremo a extremo. Esto nos puede resultar muy útil en entornos industriales donde sobrevienen multitud de ruidos electromagnéticos que pueden hacernos perder algunos paquetes. TCP se encarga de volverlos a transmitir en caso de que sea necesario.

Otra ventaja fundamental es que el tamaño de su trama puede ser variable. Aunque no tendremos problemas en nuestro caso por el reducido tamaño de la trama Modbus, esta característica es muy útil porque permite encapsular cualquier protocolo en la unidad de datos.

Por el contrario, TCP puede llegar a ser lento en entornos con ruido excesivo, ya que habrá que retransmitir muchas veces los paquetes para que lleguen a su destino o, en el peor de los casos, no se podrá establecer la comunicación entre los terminales.

2.2.2 Trama TCP

La trama TCP está compuesta por un conjunto de campos necesarios para ser interpretados por el receptor. Dentro de los datos de este protocolo insertaremos la trama Modbus descrita anteriormente con el fin de que a su llegada al destino las cabeceras sean retiradas y sólo continúe hacia la capa superior los mensajes Modbus.

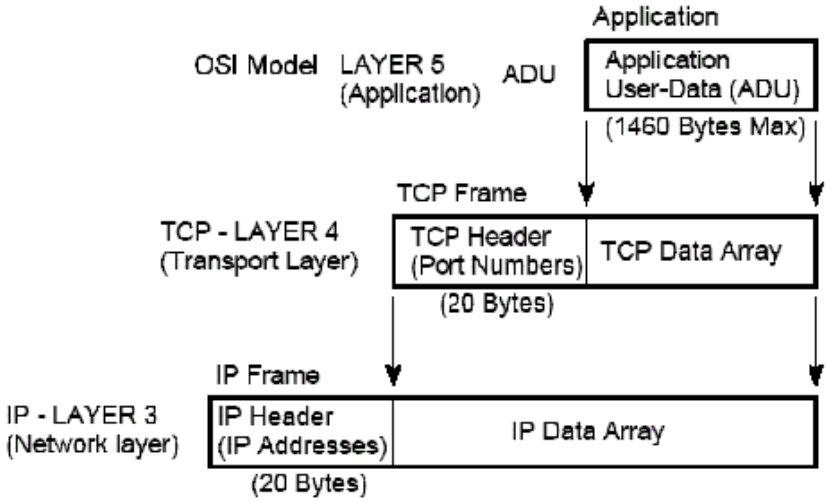


Figura 2. Encapsulación de los datos [7]

La trama TCP está formada por:

- Puerto origen: indica el número de puerto en el transmisor.

- Puerto destino: indica el número de puerto en el receptor.

- Número de secuencia: valor que sirve para ordenar los paquetes de un mismo flujo

- Número de acuse de recibo (ACK): número de secuencia del próximo paquete que espera recibir.

- Posición de los datos: indica la posición en la trama del comienzo de los datos. Puede variar debido a los campos opcionales.

- Reservado: Reservado para uso futuro

- Ventana: número de octetos de datos que el emisor está dispuesto a aceptar.

- Suma de control: complemento a uno de la trama. Sirve para comprobar que no se han producido errores.

- Puntero urgente: apunta al número de secuencia del octeto al que seguirán los datos urgentes.

- Opciones: campo variable para especificar modificaciones en la trama TCP.

- Relleno: bits para completar la trama en caso de ser menor que el tamaño mínimo, fijado en 20 octetos.

- Datos: van encapsulados los mensajes de las capas superiores, en nuestro caso Modbus.

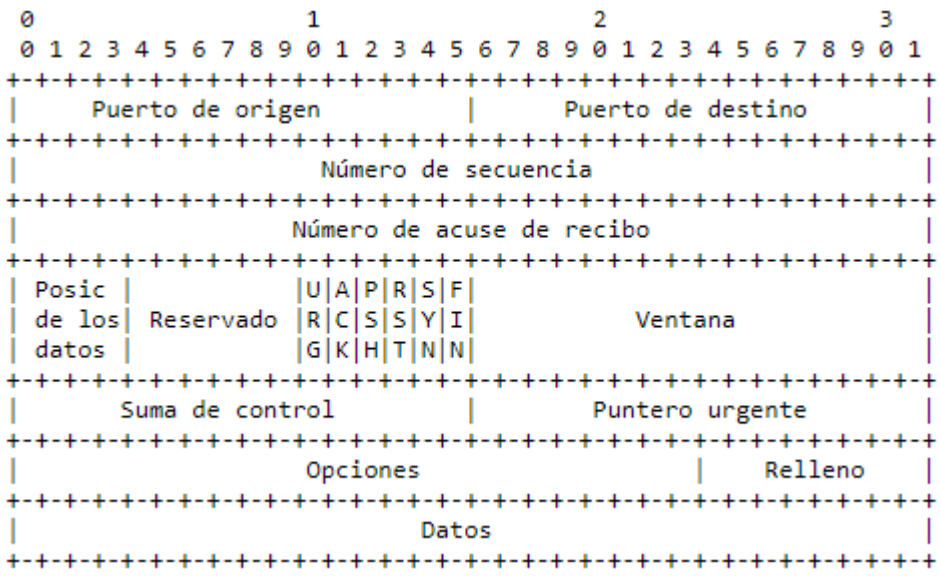


Figura 3. Formato de trama TCP (RFC 793)

3 DISEÑO DEL CLIENTE

"Creo que nuestro trabajo es ser responsables por la totalidad de la experiencia del usuario. Y si no cumple sus expectativas, es totalmente nuestra culpa, así de simple"

Steve Jobs

El objetivo que persigue la creación de un cliente es el control a cierta distancia del brazo robótico RX90. Se definirán los elementos que el operario puede manejar y a que elementos puede tener acceso para mantener protegidos ciertos datos sensibles.

Un cliente Modbus-TCP estará formado por un programa que integra ambos mundos con el fin de encapsular los datos que se quieren transmitir de manera fiable al otro extremo de la red, sin que estos sufran modificación. Es por lo que las tramas Modbus irán íntegramente encapsuladas en las tramas TCP.

3.1 Implementación del cliente

Para componer el cliente hemos tenido en cuenta las necesidades al otro extremo de la comunicación, en el servidor. Principalmente se busca que los datos lleguen íntegramente sin tener que ser modificados por el camino. Para ello, como hemos explicado anteriormente encapsularemos la trama Modbus completamente en los datos de la trama TCP.

Además, se buscará que el acceso a los comandos sea lo más rápido posible, por lo que para no tener que interpretar los caracteres que lleguen del cliente a cada orden le daremos un valor entero que es el código que usaremos para comunicar nuestras ordenes al servidor.

La primera acción que realiza el cliente al arrancar el programa es pedir nuestra dirección IP (que podrá consultarse previamente con el comando ifconfig en Linux). Si no se introduce ninguna, el programa entiende que estamos trabajando en local por lo que le asigna la dirección interna automáticamente.

Una vez este paso, iniciamos el cliente Modbus con la dirección correspondiente, el puerto y el tamaño del mapa de registros al que vamos a acceder (previamente acordado con el servidor).

Si todo funciona correctamente aparecerá un menú con las principales opciones.

3.1.1 Comandos implementados

Podremos controlar el RX90 manualmente mediante el teclado numérico, desplazando el brazo hacia cualquier posición del espacio y abrir y cerrar la pinza para coger y soltar objetos. La precisión de los movimientos viene fijada en el servidor, pero puede sernos de gran utilidad para trabajos que no estén previamente programados.

En segundo lugar, podemos mover el brazo a una posición determinada, fijada por los ángulos de rotación de cada articulación. Podemos mover el brazo a una posición predefinida o introducirla nosotros mismos.

También disponemos de una opción para mover un objeto de una posición a otra previamente establecidos. Esto se hace así para automatizar el proceso y no depender del operario para que el brazo pueda funcionar. Como esta opción usa distintos movimientos uno detrás de otro es necesario espaciarlos mediante el comando

sleep para que al servidor le dé tiempo para leer los comandos de los registros, enviarlos y que el RX90 pueda ejecutarlos sin problemas. Incluimos una opción que nos permite volver a la posición inicialmente establecida en caso de necesitarlo.

Desde el cliente podemos revisar el estado actual de los registros del sistema gracias a una opción incorporada. Al ejecutar esta opción nos mostrara una lista con el valor de la dirección de memoria accedida, así como el valor de dicho registro. Este comando es muy útil para comprobar que los comandos se van ejecutando eficazmente, y en caso de error poder comprobar donde falla.

Para controlar en todo momento la posición del brazo robótico se dispone de una opción rápida en la que nos dice únicamente la posición en la que se encuentra.

Como los registros de los que se dispone son enteros sin signo, para poder entender los ángulos negativos obtenidos al leer la posición de los registros es necesario convertir los enteros sin signo a enteros con signo, esto se realiza mediante una sencilla función de conversión. De la misma manera disponemos de una función que realiza con rapidez la impresión de la posición para poder insertarla en cualquier parte del código.

Disponemos de un fichero con las funciones de comunicación TCP-Modbus del cliente. En ellas hemos implementado el cliente Modbus con todas las funciones que este puede utilizar: conectarse y desconectarse del servidor Modbus, escribir en uno o varios registros, leer de uno o varios registros.

3.1.2 Formación completa de la trama y envío

La orden que queremos transmitir al RX90 pasa por el servidor mediante una trama Modbus encapsulada en una trama TCP, y que a su vez se encuentra introducida dentro del campo de datos del mensaje IP.

Esta estructura se forma principalmente por la función que el cliente requiere y los datos para llevarla a cabo, como puede ser la posición a la que queremos desplazar el brazo robótico, que forma la trama Modbus. Posteriormente, se añaden las cabeceras TCP, entre las que destacan el puerto origen y destino de la comunicación, estos son, el puerto TCP del cliente y el puerto TCP del servidor. Por último, conseguimos que el protocolo IP encapsule todo lo anterior en su campo de datos y añada nuevamente su cabecera. Entre los campos destacados de dicha cabecera se encuentran la IP origen y destino de dicho mensaje. El servidor se considera en este caso destino del mismo, ya que usaremos esta misiva para actualizar los datos del registro.

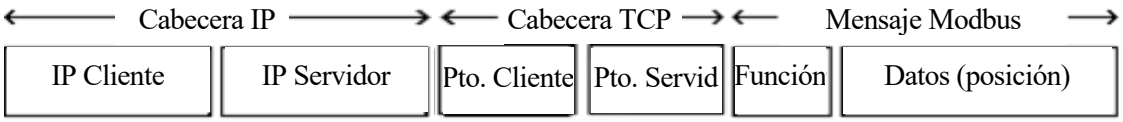


Figura 4. Trama Modbus- TCP sobre IP

4 DISEÑO DEL SERVIDOR

"Hoy en día la mayoría del software existe no para resolver un problema, sino para actuar de interfaz con otro software"

I. O. Angell

El objetivo que persigue la creación de un servidor es el de servir como punto de unión entre dos tecnologías. A un lado se encuentra la comunicación Modbus-TCP con la que se comunica con el cliente para que éste le haga llegar sus peticiones, y al otro lado se encuentra la comunicación serie con la que el servidor puede mandar los comandos recibidos al RX90.

Para apoyar la traducción de órdenes desde la comunicación Modbus-TCP realizada mediante C++ a la comunicación por puerto serie, en la que se envían con lenguaje V+, el servidor suministra una serie de registros, almacenados en memoria caché en los que guarda momentáneamente las variaciones que deben realizarse sobre el RX90.

4.1 Implementación del servidor

En la aplicación del servidor primero iniciamos el servidor TCP-Modbus con la IP y el puerto al que nos vamos a conectar. Una vez iniciado, damos los valores iniciales a los registros correspondientes. Ya estamos preparados para iniciar la conexión con el RX90, al que le pasaremos el nombre del conector a través del cual nos vamos a comunicar y la posición en la que queremos que se inicialice.

El servidor se mantiene siempre encendido a la espera de solicitudes de parte de los clientes. Cada conexión de nuevo cliente se va almacenando en el primer registro, destinado a ello, con lo que comprobaremos cuantos operarios hacen uso del brazo robótico.

Cada solicitud nueva que llega se procesa leyendo el registro de escritura del cliente, en el que indica el tipo de comando que quiere que se ejecute.

4.1.1 Comandos implementados

Para los comandos de movimiento manual, no se guarda la posición en la que queda nuestro brazo porque se entiende que son comandos de prueba y únicos, por lo que no es necesario que se guarden las posiciones exactas para volver a repetirlas. En los movimientos automáticos, el cliente escribe la nueva posición en los registros y es el servidor el encargado de leer esa posición y transformar el mensaje en comandos que el RX90 pueda entender.

Para poder llevarlo a cabo disponemos de un fichero de funciones que hemos programado, en el que implementamos las principales funciones del servidor: iniciar el servidor, espera a la conexión de cliente, recibir comandos, envío de confirmación al cliente, analizar la consulta, leer uno o varios registros, escribir uno o varios registros, enviar excepción.

En el momento de analizar la consulta comprobaremos el tipo de registro al que queremos acceder. Por defecto todos los registros son de lectura y escritura para el servidor. En el caso del cliente, puede leer todos los registros, pero definimos un rango en el que poder escribir tanto la orden enviada al servidor como las posiciones de rotación del brazo robótico. Si el cliente intenta escribir en otros registros, el servidor no se lo permitirá. Esto se realiza así para preservar la seguridad y la integridad de todos los registros críticos y no permitir que se modifiquen a libre albedrio.

4.1.2 Recepción de la trama

Para que la comunicación TCP funcione, previamente se ha tenido que establecer un vínculo entre cliente y servidor. Esto lo realiza automáticamente el protocolo TCP cuando el cliente intenta conectarse por primera vez al servidor.

Cuando el servidor recibe un mensaje, primero comprueba si la dirección IP de destino es para él. Después, desencapsula los datos de la trama IP y comprueba que es una trama TCP sin errores, calculando el complemento a uno de dicha trama y comparándola con el código de error que trae esta al final. Si todo está correcto, extraemos el mensaje Modbus que está incrustado en los datos del protocolo TCP. Por último, se copia la instrucción en el registro correspondiente, así como la nueva posición del brazo.

4.1.3 Interpretación de comandos

La traducción de lenguaje se realiza mediante los registros de los que dispone el servidor. En estos registros quedan anotadas las alteraciones que se realizan en el brazo a petición del cliente, así como las anotaciones que el servidor realiza para su correcto funcionamiento. Los registros, ordenados por el lugar que ocupan en la memoria del dispositivo, son:

Dirección	Descripción	Escrito por
0x0000	Número de conexiones de clientes	servidor
0x0001	Estado del fichero de registro (0: cerrado / 1: abierto)	servidor
0x0002	Estado del RX90	servidor
0x0003	Comando recibido por el cliente	cliente
0x0004	Posición de rotación de la base del RX90 (eje 1)	cliente
0x0005	Posición de rotación de la 1ª articulación RX90 (eje 2)	cliente
0x0006	Posición de rotación de la 2ª articulación RX90 (eje 3)	cliente
0x0007	Posición de rotación de la base de la articulación pinza del RX90 (eje 4)	cliente
0x0008	Posición de rotación de la articulación de la pinza del RX90 (eje 5)	cliente
0x0009	Posición de rotación de la base de la pinza del RX90 (eje 6)	cliente
0x000A	Estado de la pinza (0: abierta / 1: cerrada)	cliente
0x000B	Código de error	servidor

Tabla 2. Listado de registros del servidor

4.1.4 Envío por puerto serie

Para poder enviar un comando por el puerto serie es imprescindible que el cliente haya enviado un comando. El servidor por sí sólo no controlará al brazo robótico.

Para enviar una orden de ejecución al RX90, el servidor debe revisar los registros y extraer de ellos el comando que ha escrito el cliente, así como la nueva posición a la que se va a enviar. Como los registros son de tipo entero sin signo, debemos realizar una conversión forzada y convertir los valores negativos al formato entero con signo que es el que interpreta nuestro autómata.

El mensaje ahora se construye en lenguaje V+. Para ello se interpreta el comando y se escriba la posición como una cadena de caracteres precedida de su correspondiente comando en V+, que será interpretado por el RX90 cuando éste lo reciba.

5 DESARROLLO DE LA IMPLEMENTACIÓN DEL PROYECTO

"Cuando alguien diga: 'quiero un lenguaje de programación al que sólo tenga que decirle lo que quiero hacer', denle una piruleta".

Alan J. Perlis

En este capítulo vamos a describir los distintos elementos que hemos necesitado para llevar a cabo la idea del control remoto. De un lado, partíamos del integrante físico industrial, el brazo robótico RX90. Por otra parte, no habría sido posible llevarlo a cabo sin el estudio en profundidad de los lenguajes de programación V+ y C++.

El comienzo de este extenso trabajo se realizó con la prueba del funcionamiento del RX90 con un test de funcionamiento realizado por Francisco Royal. Le siguió la comprensión de los comandos más generales del lenguaje V+, así como su implementación y envío hacia el brazo. Por otra parte, se profundizó en el estudio de los servidores Modbus-TCP, comprobando cómo se realizaban las comunicaciones mediante estos protocolos y los mensajes que debían de intercambiar cliente y servidor para el establecimiento de la conexión. Por último, se procedió a la integración de los mismos y al desarrollo de las funcionalidades de ambos servicios descritas en los apartados anteriores.

5.1 Plataforma de desarrollo

El desarrollo del proyecto se ha llevado a cabo sobre sistemas Linux de 64 bits. En concreto, se ha usado la distribución de Ubuntu 16.04 [8]. Como el ordenador sobre el que se ha trabajado sólo disponía de una partición Windows, se ha trabajado sobre una máquina virtual montada con VMware 12 [9].

Se ha elegido esta distribución porque era la versión más nueva y estable cuando se comenzó el proyecto, aunque no necesariamente deben estar los equipos actualizados para poder utilizar el servicio, ya que el servicio se ejecuta sobre los terminales del cliente y servidor correspondiente.

La Universidad de Sevilla ya dispone de una red IP para los alumnos de la misma, por lo que se han utilizado las direcciones IP asignadas por la misma para la configuración de los equipos. Al utilizarse un único equipo como servidor y los demás como clientes del mismo, se obtiene una red en estrella. Esta es la configuración más adecuada en este caso, puesto que, en un supuesto entorno real, cada máquina dispondrá de su pequeño servidor al que se conectarán varios operarios que vayan revisando los aparatos.

El editor de texto utilizado en la elaboración de los ficheros ha sido Gedit, debido a su simplicidad y a la utilización de colores que éste realiza para una mejor visualización de elementos propios del lenguaje C++, como pueden ser los tipos de variables o de funciones, las estructuras de control (condiciones, bucles), etc.

El elemento elegido para ser monitorizado ha sido un brazo robótico RX90 disponible en el laboratorio de automática de la Escuela de Ingenieros de la Universidad de Sevilla. Esta máquina utiliza el lenguaje V+ para recibir órdenes desde el equipo servidor. Es un modelo parecido al que podemos encontrar en un entorno real, por lo que resulta conveniente a la hora de extrapolar los resultados del proyecto a la industria existente.

5.1.1 RX90

El brazo robótico RX90 [10] ha sido fabricado por la compañía Stäubli en el año 1996. Es un modelo ultrarrápido, fijado al suelo, con un alcance de aproximadamente 0,9 m. Dispone de una pinza con una capacidad máxima de carga de hasta 11kg, por lo que resulta adecuado para el transporte de pequeños objetos de un punto a otro dentro de su alcance.

Posee 6 ejes de rotación. Los grados de rotación de cada eje, comenzando desde el suelo hacia arriba son:

- Eje 1: 320º
- Eje 2: 275º
- Eje 3: 285º
- Eje 4: 540º
- Eje 5: 225º
- Eje 6: 540º

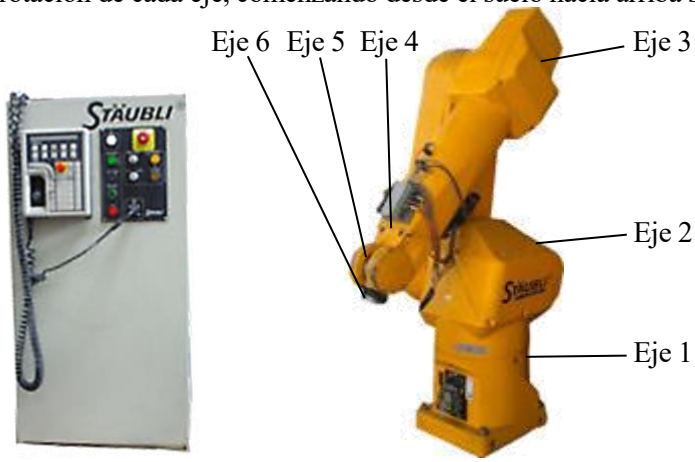


Figura 5. RX90

La comunicación con el servidor se realiza mediante puerto serie. El sistema se comunica mediante lenguaje V+, por lo que todas las instrucciones se han de traducir a dicho lenguaje para su correcta interpretación en la máquina.

5.1.2 Lenguajes de programación

Este proyecto es un buen ejercicio de integración de lenguajes, donde hemos tenido que traducir órdenes de un lenguaje a otro para establecer una comunicación extremo a extremo. Evidentemente no es posible traducir todas y cada una de las funciones que pueden implementarse. Simplemente hemos traducido aquellas que se requerían en cada momento.

La comunicación entre máquinas se realiza en los lenguajes específicos para cada cometido. Los programas en C++ ayudan a una mejor comprensión y organización de las ideas. El lenguaje V+ puede ser similar a otros lenguajes que se pueden encontrar en entornos industriales.

5.1.2.1 V+

V+ [11] [12] es un lenguaje de programación textual de medio nivel altamente estructurado. Fue creado por la firma Adept Technologies. Dicho lenguaje utiliza palabras y estructuras predefinidas para establecer acciones y la configuración del robot. Es un lenguaje basado en Pascal, que facilita la programación estructurada. Algunas características que lo definen son:

- Inteligibilidad: es fácilmente legible por cualquier persona ya que las palabras usadas adquieren significado.

- Adaptabilidad: podemos reescribir y mejorar los programas sin esfuerzo. Se pueden crear funciones que realicen una tarea concreta y repetitiva. Esto mejora la estructura del código y su reutilización.

- Fiabilidad: se trata de un lenguaje pensado para reaccionar ante imprevistos, muy útil en entornos industriales donde un fallo puede resultar crítico.

- Transportabilidad: se pueden crear programas en otros ordenadores y trasladarse luego a la máquina que va a ejecutar los comandos.

5.1.2.2 C++

C++ [13] es un lenguaje de programación de alto nivel, textual, estructurado y compilado. Fue creado por Bjarne Stroustrup en los laboratorios de At&T en 1983. Es un lenguaje derivado de C que incorpora conceptos de la programación orientada a objetos. Esta nueva percepción de la programación dividida en clases y objetos permitía al programador tener más control sobre sus funciones y poder abstraerse a la hora de flexibilizar el código.

Para poder hacer uso de los programas escritos en C++ es necesario compilar el código, es decir, traducir el código a lenguaje máquina. Existen numerosos compiladores, pero el usado en Linux es g++.

Las principales características que lo definen son:

- Lenguaje orientado a objetos: nos permite crear entidades que poseen variables y funciones propias con las que desarrollan un determinado comportamiento dentro de nuestro programa

- Estructurado: el código se puede dividir en funciones y estas a su vez en ficheros.

- Reutilización de código: las clases pueden ser instanciadas múltiples veces para la creación de objetos diferentes.

5.2 Sincronización entre máquinas

Para que el programa funcione correctamente es importante seguir un orden a la hora de iniciar nuestras herramientas, ya que si cambiamos el orden las dependencias entre ellos nos impedirán seguir el proceso. Debemos comenzar siempre iniciando el brazo robótico y una vez esté montado arrancamos nuestro servidor. Posteriormente, arrancamos el cliente.

Si el RX90 no está iniciado previamente, el servidor nos dará fallo al intentar abrir una instancia para comunicarse con él. Lo mismo ocurre al iniciar el cliente, puesto que necesita la confirmación de establecimiento de la conexión con el servidor para poder dar comienzo a la comunicación.

5.2.1 Puesta en marcha del RX90

Para hacer funcionar nuestro brazo robótico, primero debemos accionar el interruptor que da suministro eléctrico al equipo. Posteriormente, comprobamos que no está bloqueado el brazo y pulsamos el botón COMP|PWR del mando de control del RX90. Se encenderá en el panel central una luz verde correspondiente a ARM POWER ON. Presionaremos ese botón para armar nuestro brazo robótico. A partir de entonces contaremos con una luz verde en ARM POWER que nos indicará que el RX90 está listo para usarse.

En caso de ser necesario, se dispone de un pulsador rojo en el panel central para detener el funcionamiento del RX90. Volviendo a tener que realizar todos los pasos para volver a ponerlo en funcionamiento.

5.2.2 Puesta en marcha del servidor

Para iniciar el servidor Modbus-TCP previamente debemos haber instalado la librería libmodbus-3.0.6 como se ha descrito anteriormente. Una vez hecho esto, debemos mirar en el terminal del equipo servidor con ifconfig la dirección IP asociada a nuestra máquina. Debemos sustituir por esta dirección, en el caso de no ser la misma, la dirección IP que viene fijada en el fichero server.cpp. Es importante que se realicen estos pasos previos, de no ser así nuestra comunicación no se efectuará.

Una vez estén guardados los ficheros procederemos a ejecutar el comando make en el directorio donde se encuentran los archivos modificados. Esto creará un ejecutable que podrá iniciarse simplemente con escribir ./server en el terminal.

5.2.3 Puesta en marcha del cliente

Para iniciar el cliente Modbus-TCP, nuevamente hemos de instalar la librería libmodbus-3.0.6 antes de su ejecución. Además, tenemos que mirar igualmente la dirección IP asociada a nuestro equipo. Esto no sería necesario si ejecutamos el cliente en el mismo equipo del servidor. En este caso, usaremos la dirección local para establecer la comunicación con el mismo.

Una vez que conocemos nuestra IP arrancamos el cliente sencillamente escribiendo ./client en el terminal del equipo cliente. Entonces nos preguntará si queremos cambiar la dirección IP. Si nos encontramos en un equipo distinto al servidor, introduciremos la IP obtenida anteriormente y quedará iniciado nuestro cliente Modbus-TCP.

5.3 Depuración y pruebas de funcionamiento

El escenario escogido para realizar las pruebas comprende un brazo robótico conectado a un servidor por puerto serie y dos clientes conectados al servidor mediante Modbus-TCP, como se muestra en la figura:

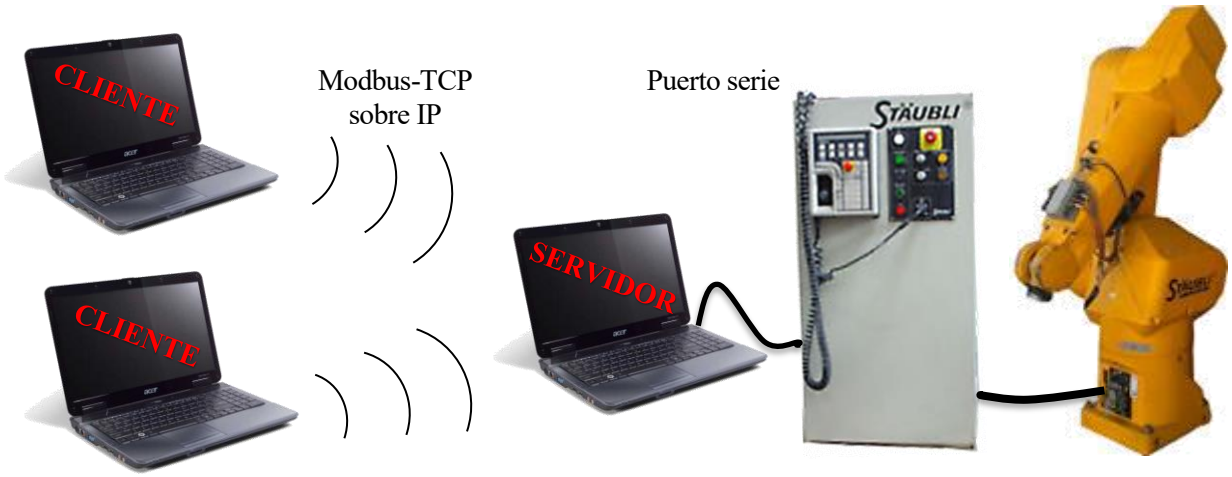


Figura 6. Escenario de pruebas

Para estar completamente en la certeza del correcto funcionamiento del conjunto primero deberemos comprobar cada comunicación por separado mediante scripts sencillos.

5.3.1 Comprobación de la comunicación serie

Para comprobar la comunicación serie con el RX90 no necesitamos la parte de comunicación Modbus. Crearemos un sencillo programa que simplemente introduciendo los movimientos (arriba, abajo, izquierda, derecha y coger con la pinza) por teclado comprobemos inmediatamente la reacción del autómata.

5.3.2 Comprobación de la comunicación Modbus-TCP

Para este apartado prescindimos de la comunicación serie y nos centramos en el intercambio de mensajes entre el cliente y servidor. Nos bastará con crear un programa en el que asignaremos un valor a cada comando, los mismos en el cliente y en el servidor. Posteriormente, enviaremos dichos comandos a través de Modbus-TCP, que completarán los registros con los valores enviados.

Podemos crear una función, además, que nos muestre el valor de los registros escritos para visualizar así que los valores enviados se corresponden efectivamente con los que se han guardado en los registros.

Otra herramienta para comprobar los campos de las cabeceras TCP es Wireshark. Wireshark es un analizador de red, que captura los paquetes de la interfaz asignada.

Podemos realizar a modo de ejemplo una captura de paquetes:

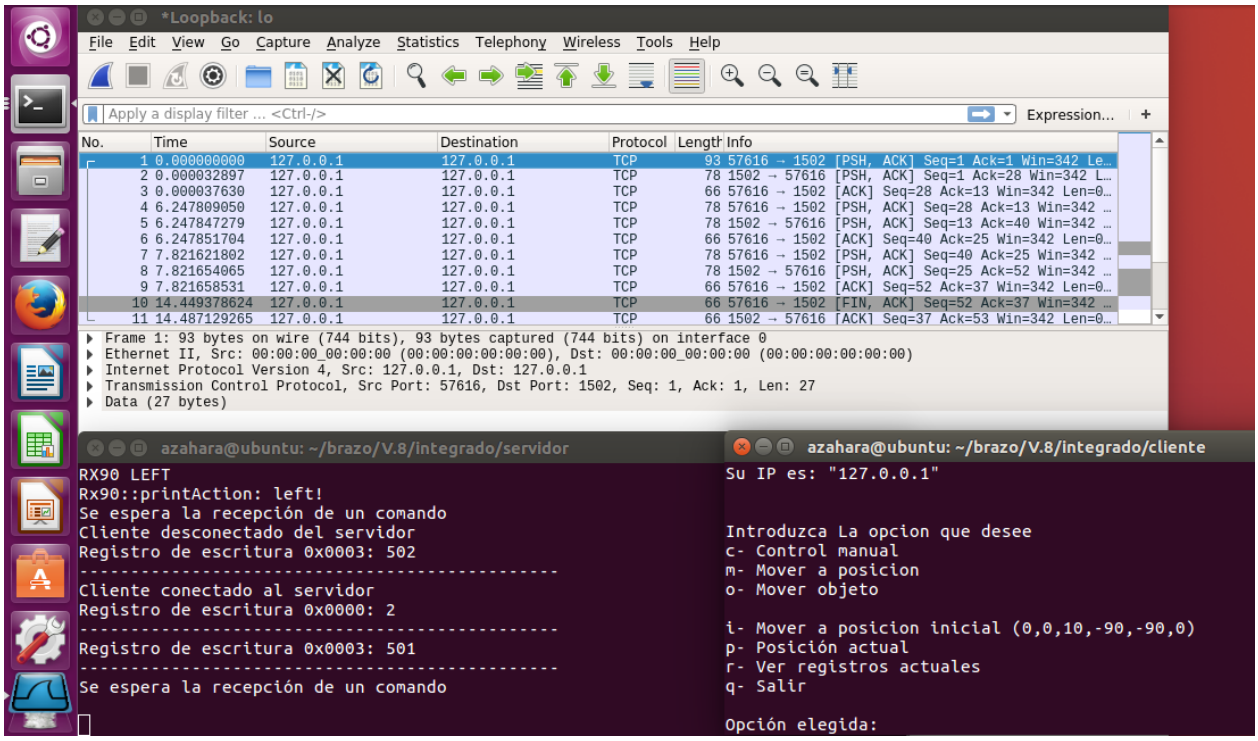


Figura 7. Captura Wireshark

Como se puede observar, los paquetes se han realizado en una prueba local. Podemos observar cómo se envía un paquete de petición desde el cliente, con el puerto TCP 57616 hacia el servidor, con el puerto TCP 1502. Si accedemos a la información adicional de dicha captura podremos observar en el apartado datos el mensaje completo enviado.

De la misma forma, podemos ver cómo el servidor envía un mensaje de asentimiento en sentido opuesto como fórmula para indicar al cliente que no se han producido errores en la transmisión. De esta forma el cliente no tiene que volver a reenviar el mismo mensaje.

5.3.3 Comprobación de la sincronización

Para este apartado deberemos fusionar los dos apartados anteriores. Se debe anidar la orden mandada por el cliente e inscrita en los registros del servidor con las órdenes que finalmente se mandan al RX90.

De la misma forma, se deberá comprobar que los valores guardados en los registros son los que efectivamente el cliente, y que a partir de éstos el servidor es capaz de extraer los datos y enlazarlos para enviarlos por puerto serie. Si todo funciona correctamente, el brazo robótico RX90, realizará las acciones que el cliente desea.

6 RESULTADOS Y CONCLUSIONES

"Hay tres maneras de adquirir sabiduría: primero, por la reflexión, que es la más noble; segundo, por imitación, que es la más sencilla; y tercero, por la experiencia, que es la más amarga".

Confucio

En este capítulo se presentarán las consecuencias de la realización del proyecto. Estudiaremos cómo el uso de servidores Modbus-TCP pueden mejorar las tareas realizadas en entornos reales dentro de la industria.

De la misma forma, analizaremos los objetivos cumplidos y cuál puede ser la línea de mejora del proyecto con el fin de adaptarlo aún más a las demandas reales.

6.1 Objetivos conseguidos

La introducción de la comunicación Modbus-TCP en áreas en las que se utilizaban el cableado de forma tradicional han visto mejorar su alcance y consolidar su uso al abaratar objetos, conectividad y almacenaje de datos de forma abierta. Sistemas de control y tele-medida tradicionalmente desarrollados en entornos cerrados y de alto precio pueden ser sustituidos por soluciones de más bajo coste y arquitectura de protocolos y representación de datos abiertos.

Con su uso se genera una interacción y recolección de datos que, tras el consiguiente análisis, puede proporcionar nuevas informaciones que permitan mejorar costes en diferentes áreas, justificando la rentabilidad y el retorno de la inversión (ROI) de forma directa.

El uso de estos principios básicos permitirá a la industria del futuro poder personalizar la fabricación sin dejar atrás la producción en masa, posibilitando también el auto diagnóstico, el auto ajuste y la auto optimización de los procesos, siendo también clave para asistir a los trabajadores en la mejora de sus condiciones laborales y en la realización de su actividad.

Con relación a los objetivos personales conseguidos, afirmo que he logrado dominar unos lenguajes, C++ y V+, hasta ahora desconocidos para mí. He sentido la satisfacción de ver cómo he manejado herramientas ajenas a los estudios realizados y cómo éstas pueden ser de utilidad en entornos reales.

6.2 Futuras líneas de trabajo

El mundo de la industria 4.0 abre un abanico inmenso de posibilidades de desarrollo de aplicaciones. Cualquier máquina puede estar conectada en la actualidad y usar el servicio cliente- servidor Modbus-TCP aquí realizado para mejorar la funcionalidad y el manejo de las mismas.

En nuestro caso, el proyecto puede avanzar en muchos sentidos, siendo muy útil la integración de un mismo servidor para el control de varios autómatas. Esto ahorraría gastos de inversión en infraestructuras y aprovecharía todos los recursos de la máquina servidor. Bastaría con aumentar el número de registros y asignar rangos determinados a cada autómata.

Otra posible mejora puede ser la introducción de ficheros de registro en formato texto, en el que se vayan grabando todos los cambios realizados en los registros del servidor.

ANEXO A: CÓDIGO IMPLEMENTADO EN EL CLIENTE

client.cpp

```
/*************************************************************************/
/*                                                                       */
/*                        CLIENTE TCP PARA RX90                          */
/*                                                                       */
/*************************************************************************/

#include <iostream>
#include <stdio.h>
#include <string.h>
#include <time.h>
#include "ClientModBUS.h"
#include "constant.h"
#include <unistd.h>

#define NUM_REGISTERS   0x32
#define COMMAND_ADDR    3
#define ADDR_INI        0

int uintToInt(uint16_t value);
void char2int(char newPosition[29],uint16_t position[6]);
void imprimePosicion(uint16_t position[6]);

int main()
{
      char direccion[16]="127.0.0.1"; // 192.168.109.148
      int puerto=1502;
      char newPosition[29]="0,0,10,-90,-90,0"; // 0,-150,10,-90,-90,0
      int numReg;

      char option='0';
      uint16_t value;
      uint16_t tab[NUM_REGISTERS];

      uint16_t position[6];

      printf("////////////////////////////////////////////////\n");
      printf("\n");
      printf("               CONTROL REMOTO RX 90               \n");
      printf("\n");
      printf("\n");
      printf("             Azahara Mª Porras Tejada             \n");
      printf("////////////////////////////////////////////////\n\n");

      // Cambiar (o no) la dirección IP del cliente
      printf("Su direccion IP es %s. ¿Desea cambiarla? [s/n] " , direccion);
      if(getc(stdin)=='s'){
            printf("Introduzca la nueva IP: " );
```

```
            scanf("%s", direccion);
      }
      printf ("Su IP es: \"%s\" \n\n",direccion);

      //Se crea la interfaz cliente con la direccion IP, en el puerto 1502, y
se indica el tamaño del mapa de registros
      ClientModBUS modbus(direccion,puerto,0x0020);
      if(!modbus.connect())
            return 0;

      /**** Opciones ****/

      while(option!='q')//||option=='0')//mientras no se pulse salir se
vuelve al menu tras realizar las distintas opciones
      {

            printf("\nIntroduzca La opcion que desee\n");

            printf("c- Control manual\n");
            printf("m- Mover a posicion\n");
            printf("o- Mover objeto\n\n");
            printf("i- Mover a posicion inicial (0,0,10,-90,-90,0)\n");

            printf("p- Posición actual\n");
            printf("r- Ver registros actuales\n"); //leer varios registros
            //printf("v- Vover a posicion "160,-150,10,-90,-90,0"\n");
            printf("q- Salir\n\n");
            printf("Opción elegida: ");
            scanf("%s",&option);

            /**** Opciones para el control manual****/
            if(option=='c'){
                  printf("CONTROL MANUAL\n");
                  while(option!='0')//mientras no se pulse salir se vuelve al
menu tras realizar las distintas opciones
                  {

                        printf("\nControl por teclado numerico:\n");
                        printf("        (8)           \n");
                        printf("         ^            \n");
                        printf("    (4)<   >(6) \n");
                        printf("         v            \n");
                        printf("        (2)           \n\n");

                        printf("5- Cerrar/Abrir pinza      \n\n");

                        printf("0- Salir\n\n");

                        printf("Opción elegida: ");
                        scanf("%s",&option);

                        // Arriba (8)
                        if(option=='8'){
                              printf("UP\n");
                              value= ROBOT_UP;

      if(modbus.write_only_register(COMMAND_ADDR,value))
                                    printf("Valor registro Escrito en %04X es:
%d\n ROBOT_UP\n",COMMAND_ADDR&0xFFFF,value);
                        }
```

```
                        // Abajo (2)
                        if(option=='2'){
                              printf("DOWN\n");
                              value= ROBOT_DOWN;

      if(modbus.write_only_register(COMMAND_ADDR,value))
                                    printf("Valor registro Escrito en %04X es:
%d\n ROBOT_DOWN\n",COMMAND_ADDR&0xFFFF,value);
                        }


                        // Izquierda (4)
                        if(option=='4'){
                              printf("LEFT\n");
                              value= ROBOT_LEFT;

      if(modbus.write_only_register(COMMAND_ADDR,value))
                                    printf("Valor registro Escrito en %04X es:
%d\n ROBOT_LEFT\n",COMMAND_ADDR&0xFFFF,value);
                        }


                        // Derecha (6)
                        if(option=='6'){
                              printf("RIGHT\n");
                              value= ROBOT_RIGHT;

      if(modbus.write_only_register(COMMAND_ADDR,value))
                                    printf("Valor registro Escrito en %04X es:
%d\n ROBOT_RIGHT\n",COMMAND_ADDR&0xFFFF,value);
                        }

                        // Delante (7)
                        if(option=='7'){
                              printf("FORWARD\n");
                              value= ROBOT_FORWARD;

      if(modbus.write_only_register(COMMAND_ADDR,value))
                                    printf("Valor registro Escrito en %04X es:
%d\n ROBOT_FORWARD\n",COMMAND_ADDR&0xFFFF,value);
                        }


                        // Detras (9)
                        if(option=='9'){
                              printf("BACKWARD\n");
                              value= ROBOT_BACKWARD;

      if(modbus.write_only_register(COMMAND_ADDR,value))
                                    printf("Valor registro Escrito en %04X es:
%d\n ROBOT_BACKWARD\n",COMMAND_ADDR&0xFFFF,value);
                        }

                        // Coger (5)
                        if(option=='5'){
/**/                          printf("PINZA\n");
                              value= ROBOT_CATCH;

      if(modbus.write_only_register(COMMAND_ADDR,value)){
                                    printf("Valor registro Escrito en %04X es:
%d\n ROBOT_CATCH\n",COMMAND_ADDR&0xFFFF,value);
```

```
      if(modbus.read_only_register(ADDR_PINZA,&value))
                                    {
                                          if(value==CLOSE_PINCERS){
                                                printf("Cerramos la pinza\n");
                                          }
                                          else{
                                                printf("Abrimos la pinza\n");
                                          }
                                    }
                              }
                        }

                  }

            // Mover a la posición indicada
            else if(option=='m'){
                  numReg=7;   //total de registros a escribir
                  getc(stdin);
                  printf("Mover a posicion %s. ¿Desea cambiarla? [s/n] " ,
newPosition);
                  if(getc(stdin)=='s'){
                        printf("Introduzca la nueva posicion: " );
                        scanf("%s", newPosition);
                        //getc(stdin);
                  }

                  char2int(newPosition,position);
                  printf("Mover a posicion: ");
                  imprimePosicion(position);

                  value= ROBOT_AUTOMOVE;
                  position[0]=value;


      if(modbus.write_many_registers(COMMAND_ADDR,numReg,position))
                  {
                        for(int i=0;i<numReg;i++)
                              printf("Registro %04X cambia a:
%d\n",(COMMAND_ADDR+i)&0xFFFF,uintToInt(position[i]));
                  }
                  else
                        printf("Error en ESCRITURA DE POSICION\n");
            }

            // Mover objeto
            else if(option=='o'){
                  //sleep(2);
                  //newPosition[29]="0,0,10,-90,-90,0"; // 0,-150,10,-90,-90,0

                  // Se mueve hasta un poco antes de la posición del objeto
                  numReg=7;   //total de registros a escribir
                  value= ROBOT_AUTOMOVE;
                  position[0]=value;
                  position[1]=position[1]-10;


      if(modbus.write_many_registers(COMMAND_ADDR,numReg,position))
                  {
```

```
                        for(int i=0;i<numReg;i++)
                              printf("Registro %04X cambia a:
%d\n",(COMMAND_ADDR+i)&0xFFFF,uintToInt(position[i]));
                  }
                  else
                        printf("Error en ESCRITURA DE POSICION\n");

                  /*****************/
                  if(modbus.read_many_registers(COMMAND_ADDR,numReg,position))
                  {
                        printf("La posición obtenida es: ");
                        imprimePosicion(position);
                  }
                  else
                        printf("Error en operacion LECTURA DE VARIOS
REGISTROS\n");
                  /*****************/

                  sleep(4);

                  // Abrimos la pinza
                  if(modbus.read_only_register(ADDR_PINZA,&value))
                  {
                        if(value==CLOSE_PINCERS){
                              printf("La pinza estaba cerrada\n");
                              value= ROBOT_CATCH;

      if(modbus.write_only_register(COMMAND_ADDR,value)){
                                    printf("Valor registro Escrito en %04X es:
%d\n ROBOT_CATCH\n",COMMAND_ADDR&0xFFFF,value);
                                    }
                        }
                        else{
                              printf("La pinza estaba abierta\n");
                        }
                  }
                  else
                        printf("Error en operacion LECTURA DE VARIOS
REGISTROS\n");
                  /*****************/

                  sleep(4);

                  // Se mueve hasta rodear el objeto
                  printf("Mover a posicion %s\n", newPosition);
                  value= ROBOT_AUTOMOVE;
                  position[0]=value;
                  position[1]=position[1]+10;


      if(modbus.write_many_registers(COMMAND_ADDR,numReg,position))
                  {
                        for(int i=0;i<numReg;i++)
                              printf("Registro %04X cambia a:
%d\n",(COMMAND_ADDR+i)&0xFFFF,uintToInt(position[i]));
                  }
                  else
                        printf("Error en ESCRITURA DE POSICION\n");

                  /*****************/
                  if(modbus.read_many_registers(COMMAND_ADDR,numReg,position))
                  {
```

```
                        printf("La posición obtenida es: ");
                        imprimePosicion(position);
                  }
                  else
                        printf("Error en operacion LECTURA DE VARIOS
REGISTROS\n");
                  /*****************/

                  sleep(4);

                  // Coge el objeto (cerramos la pinza)
                  if(modbus.read_only_register(ADDR_PINZA,&value))
                  {
                        if(value==OPEN_PINCERS){
                              printf("La pinza estaba abierta\n");
                              value= ROBOT_CATCH;

      if(modbus.write_only_register(COMMAND_ADDR,value)){
                                    printf("Valor registro Escrito en %04X es:
%d\n ROBOT_CATCH\n",COMMAND_ADDR&0xFFFF,value);
                              }
                        }
                        else{
                              printf("La pinza estaba cerrada\n");
                        }
                  }
                  else
                        printf("Error en operacion LECTURA DE VARIOS
REGISTROS\n");
                  /*****************/

                  sleep(4);

                  // Desplazamos el objeto a la nueva posicion
                  value= ROBOT_AUTOMOVE;
                  position[0]=value;
                  position[1]=position[1]+90;


      if(modbus.write_many_registers(COMMAND_ADDR,numReg,position))
                  {
                        for(int i=0;i<numReg;i++)
                              printf("Registro %04X cambia a:
%d\n",(COMMAND_ADDR+i)&0xFFFF,uintToInt(position[i]));
                  }
                  else
                        printf("Error en ESCRITURA DE POSICION\n");

                  /*****************/
                  if(modbus.read_many_registers(COMMAND_ADDR,numReg,position))
                  {
                        printf("La posición obtenida es: ");
                        imprimePosicion(position);
                  }
                  else
                        printf("Error en operacion LECTURA DE VARIOS
REGISTROS\n");
                  /*****************/

                  sleep(4);

                  // Abrimos la pinza
```

```
                  if(modbus.read_only_register(ADDR_PINZA,&value))
                  {
                        if(value==CLOSE_PINCERS){
                              printf("La pinza estaba cerrada\n");
                              value= ROBOT_CATCH;

      if(modbus.write_only_register(COMMAND_ADDR,value)){
                                    printf("Valor registro Escrito en %04X es:
%d\n ROBOT_CATCH\n",COMMAND_ADDR&0xFFFF,value);
                              }
                        }
                        else{
                              printf("La pinza estaba abierta\n");
                        }
                  }
                  else
                        printf("Error en operacion LECTURA DE VARIOS
REGISTROS\n");
                  /*****************/

                  sleep(4);

                  // Se mueve hasta un poco antes de la posición del objeto
                  value= ROBOT_AUTOMOVE;
                  position[0]=value;
                  position[1]=position[1]-10;


      if(modbus.write_many_registers(COMMAND_ADDR,numReg,position))
                  {
                        for(int i=0;i<numReg;i++)
                              printf("Registro %04X cambia a:
%d\n",(COMMAND_ADDR+i)&0xFFFF,uintToInt(position[i]));
                  }
                  else
                        printf("Error en ESCRITURA DE POSICION\n");

                  /*****************/
                  if(modbus.read_many_registers(COMMAND_ADDR,numReg,position))
                  {
                        printf("La posición obtenida es: ");
                        imprimePosicion(position);
                  }
                  else
                        printf("Error en operacion LECTURA DE VARIOS
REGISTROS\n");
                  /*****************/

            }
            // Volver a la posición inicial
            if(option=='i'){
                  value= ROBOT_AUTOMOVE;
                  position[0] = value;
                  position[1] = 0;
                  position[2] = 0;
                  position[3] = 10;
                  position[4] = -90;
                  position[5] = -90;
                  position[6] = 0;

                  printf("Mover a posicion: ");
                  imprimePosicion(position);
```

```
      if(modbus.write_many_registers(COMMAND_ADDR,numReg,position))
                  {
                        for(int i=0;i<numReg;i++)
                              printf("Registro %04X cambia a:
%d\n",(COMMAND_ADDR+i)&0xFFFF,uintToInt(position[i]));
                  }
                  else
                        printf("Error en ESCRITURA DE POSICION\n");
            }

            //Lectura de todos los registros del sistema
            else if(option=='r')
            {

                  printf("REGISTROS DEL SISTEMA\n");

                  numReg=20;//total de registros para mostrar todos por
pantall
                  if(modbus.read_many_registers(ADDR_INI,numReg,tab))
                  {
                        for(int i=0;i<numReg;i++)
                              printf("El valor del registro leido en 0x%04X
es: %d\n",(ADDR_INI+i)&0xFFFF,uintToInt(tab[i]));
                  }
                  else
                        printf("Error en operacion LECTURA DE VARIOS
REGISTROS\n");
            }

            else if(option=='p')
            {

                  printf("POSICION ACTUAL\n");

                  numReg=7;//total de registros para mostrar todos por pantall

                  if(modbus.read_many_registers(COMMAND_ADDR,numReg,position))
                  {
                        printf("La posición obtenida es: ");
                        imprimePosicion(position);
                  }
                  else
                        printf("Error en operacion LECTURA DE VARIOS
REGISTROS\n");


            }

            else if(option!='q'&& option!='0'){
                  printf("Error: Option %c doesn't exist\n",option);
            }
      }
      return 0;
}

int uintToInt(uint16_t value) {
      int entero;
      if (value>65176)
            entero=-(65536-value);
```

```
      else
            entero=value;
      return entero;
}



void char2int(char newPosition[29],uint16_t position[6]){
      char copia[29];
      char * pos;
      int i=1;
      strcpy(copia, newPosition);
      //printf ("Splitting string \"%s\" into tokens:\n",newPosition);
      pos = strtok (newPosition," ,");   // Aqui deja solo la coma
      while (pos != NULL)
      {
            //printf ("%s\n",pos);     // Aqui deberias guardar tu dato en el
array!
            position[i]=atoi(pos);
            //printf ("%d\n",position[i]);
            pos = strtok (NULL, " ,");  // Aca tambien iria solo la coma.!!
            i++;
      }
      //printf("Position:
%d,%d,%d,%d,%d,%d\n",uintToInt(position[1]),uintToInt(position[2]),uintToInt(
position[3]),uintToInt(position[4]),uintToInt(position[5]),uintToInt(position
[6]));
      strcpy(newPosition, copia);

}

void imprimePosicion(uint16_t position[6]){
      printf("%d,%d,%d,%d,%d,%d\n",uintToInt(position[1]),uintToInt(position[
2]),uintToInt(position[3]),uintToInt(position[4]),uintToInt(position[5]),uint
ToInt(position[6]));
}
```

ClientModbus.cpp

```
/*********************************************************************/
/*                                                                   */
/*                      CLIENTE MODBUS PARA RX90                     */
/*                                                                   */
/*********************************************************************/

#include "ClientModBUS.h"

/**Constructor: Recibe como argumentos la IP y el puerto en los que se va a
realizar la conexión**/
ClientModBUS::ClientModBUS(char* ip,short port,int timer)
{
   ctx = modbus_new_tcp(ip, port);
    struct timeval response_timeout;

    /* Define a new and too short timeout! */
    response_timeout.tv_sec = timer;
    response_timeout.tv_usec = 0;
    modbus_set_response_timeout(ctx, &response_timeout);
}
```

```
ClientModBUS::~ClientModBUS()
{

}
/**connect: Conecta el cliente con el servidor ModBUS

      Recibe: Nada
      Devuelve:
            true: conexión realizada correctamente
            false: Fallo al conectarse con el servidor
**/
bool ClientModBUS::connect()
{

    if (modbus_connect(ctx) == -1)
    {
        fprintf(stderr, "Connection failed: %s\n",modbus_strerror(errno));
        modbus_free(ctx);
        return false;
    }
    else
      return true;
}

/**disconnect: Realiza la desconexión con el servidor ModBUS con el servidor
ModBUS

      Recibe:Nada
      Devuelve: Nada

**/
void ClientModBUS::disconnect()
{
    modbus_close(ctx);
}

/**write_only_register: Realiza la escritura de un solo registro
      Recibe:
            addr:  Dirección del registro a escribir
            value: Valor que se le quiere asignar a dicho registro

      Devuelve:
            true:  Todo correcto
            false: No se pudo realizar la operación
**/
bool ClientModBUS::write_only_register(int addr, int value)
{
    int rc;
    rc=modbus_write_register(ctx, addr, value);
    if (rc != 1)
    {
         fprintf(stderr, "write_only_register: %s\n",modbus_strerror(errno));
       return false;
    }
    else
       return true;

}
```

```
/**write_many_registers: Realiza la escritura de varios registros
      Recibe:
            addr: Dirección del primer registro a escribir
            num_registers: Numero de registros a escribir
            value: Tabla de valores para dichos registros

      Devuelve:
            true:  Todo correcto
            false: No se pudo realizar la operación
**/
bool  ClientModBUS::write_many_registers(int addr,int num_registers,uint16_t
values[])
{
    int rc;
    rc=modbus_write_registers(ctx, addr, num_registers, values);
    if (rc != num_registers)
    {
         fprintf(stderr, "write_many_registers:
%s\n",modbus_strerror(errno));
       return false;
    }
    else
       return true;



}
/**read_only_register: Realiza la lectura de un solo registro
      Recibe:
            addr:  Dirección del registro a leer
            value: Parametro por referencia donde se guardara el dato leído

      Devuelve:
            true:  Todo correcto
            false: No se pudo realizar la operación
**/
bool ClientModBUS::read_only_register(int addr,uint16_t* value)
{
    int rc;
    rc=modbus_read_registers(ctx, addr, 1,value);
    if (rc != 1)
    {
         fprintf(stderr, "read_only_register: %s\n",modbus_strerror(errno));
       return false;
    }
    else
      return true;
}


/**read_many_registers: Realiza la lectura de varios registros
      Recibe:
            addr: Dirección del primer registro a leer
            num_registers: Numero de registros a leer
            value: Tabla donde se guardarán los valores de los registros
leídos

      Devuelve:
            true:  Todo correcto
            false: No se pudo realizar la operación
**/
```

```
bool ClientModBUS::read_many_registers(int addr,int num_registers,uint16_t
values[])
{

    int rc;
    rc=modbus_read_registers(ctx, addr,num_registers,values);
    if (rc != num_registers)
    {
         fprintf(stderr, "read_many_registers: %s\n",modbus_strerror(errno));
       return false;
    }
    else
       return true;


}
```

ClientModbus.h

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <modbus/modbus.h>
#include "constant.h"




class ClientModBUS
{
      private:
            modbus_t *ctx;

      public:
            ClientModBUS(char* ip,short port,int timeout);
            ~ClientModBUS();
            bool  connect();
            void  disconnect();
            bool  write_only_register(int addr, int value);
            bool  write_many_registers(int addr,int num_registers,uint16_t
values[]);
            bool  read_only_register(int addr,uint16_t*);
            bool  read_many_registers(int addr,int num_registers,uint16_t[]);


};
```

constant.h

```
/*********************************************************************/
/*                                                                   */
/*               CONSTANTES DEL CLIENTE DEL RX90                     */
/*                                                                   */
/*********************************************************************/
```

```
#define ADDR_RX90_STATE       0x0002 // ADDR_RX90_STATE y ADDR_CLIENT deben
coincidir al ejecutar un comando

#define ADDR_CLIENT           0x0003
// Las direcciones 4, 5, 6, 7, 8 y 9 se usan para guardar el ángulo de
rotación (posicion) de los ejes del RX90
#define ADDR_INI_POSITION     0x0004
#define ADDR_FIN_POSITION     0x0009
#define ADDR_PINZA            0x000A

// Estado pinza (ADDR_PINZA)
#define OPEN_PINCERS    0
#define CLOSE_PINCERS   1

// Tipos de registros
#define READ_ONLY  0
#define WRITE_ONLY       1
#define READ_WRITE       2
#define NOT_PERMITTED    3

// Comandos Modbus
#define READ_REGISTERS        0x03 // Valor que indica el comando para leer
registros
#define WRITE_MULTIPLE_REGISTERS 0x10 // Valor que indica el comando para
escribir varios registros
#define WRITE_ONLY_REGISTER   0x06 // Valor que indica el comando para
escribir un único registro
#define READ_WRITE_REGISTERS  0x17 // Valor que indica el comando para
leer/escribir registros
#define WRITE_MASK_REGISTER   0X16
#define READ_FIFO       0x18 // Valor que indica el comando para leer
registros de la pila

//Estado del sistema (ADDR_RX90_STATE)
#define DISCONNECT           -1
#define RX90_COMMAND         1
#define NO_COMMAND           0
#define STATE_POWERINGON     101
#define STATE_POWERINGOFF    102

//Robot command (ADDR_CLIENT)
#define COMMAND_START 501
#define COMMAND_FINISH 502

#define ROBOT_UP   908
#define ROBOT_DOWN       902
#define ROBOT_LEFT       904
#define ROBOT_RIGHT      906
#define ROBOT_BACKWARD   907
#define ROBOT_FORWARD    909
#define ROBOT_CATCH      905
#define ROBOT_DROP       0

#define ROBOT_AUTOMOVE   307

//Errores
#define ERROR_NO_ERROR   800
#define ERROR_UNKNOW     801
#define ERROR_SENSOR     802
#define ERROR_ACTUADOR   803
#define ERROR_MODBUS     804
```

Makefile

```
all: client
	rm -rf *.o

client: client.o ClientModBUS.o
	g++ client.o ClientModBUS.o  -o client -lmodbus

client.o: client.cpp
	g++ -c client.cpp

ClientModBUS.o: ClientModBUS.cpp
	g++ -c ClientModBUS.cpp

clean:
	rm -rf *o client

install-libmodbus:
	cp libmodbus-3.0.6/lib/libmodbus.* /usr/lib
```

ANEXO B: CÓDIGO IMPLEMENTADO EN EL SERVIDOR

server.cpp

```
/*********************************************************************/
/*                                                                   */
/*                      SERVIDOR TCP PARA RX90                       */
/*                                                                   */
/*********************************************************************/

#include "serverModBUS.h"
#include "constant.h"
#include "Rx90.h"

void mover (serverModBUS server, int reg, Rx90 rx90);
void pinza (serverModBUS server, int reg, Rx90 rx90);

int main()
{
      // Brazo robótico Rx90
      char dispositivo[]="/dev/ttyUSB0";
      //char posicion[]="122,-77,-43,55,45,-43";
      char posicion[]="150,-150,10,-90,-90,0";
      char ipServer[16]="192.168.109.148"; // 127.0.0.1
      int puerto=1502;
      int nClient=1;
      int reg=0; //entero en el que se guarda el tipo de comando recibido
      uint16_t addr;

      // Servidor modbus
      serverModBUS server;
      printf("Iniciando servidor...\n");
      //Se inicia el servidor
      int sinit=server.init(ipServer,puerto,NUM_REG); //ip, puerto, numero de
registros para crear

      /*
      REGISTROS DEL SISTEMA:
      0x0000: Número de conexiones de clientes                    (servidor)
      0x0001: Estado del fichero de registro (0:cerrado/1:abierto) (servidor)
      0x0002: Estado del RX90                                     (servidor)
      0x0003: Comando recibido por el cliente                     (cliente)
      0x0004: Posicion de rotacion de la base del RX90            (cliente)
      0x0005: Posicion de rotacion de la 1ª articulacion RX90     (cliente)
      0x0006: Posicion de rotacion de la 2ª articulacion RX90     (cliente)
      0x0007: Posicion de rotacion de la base de la articulación
              pinza del RX90                                      (cliente)
      0x0008: Posicion de rotacion de la articulacion de la pinza
              del RX90                                            (cliente)
      0x0009: Posicion de rotacion de la base de la pinza del RX90 (cliente)
      0x000A: Estado de la pinza (0: abierta / 1: cerrada)        (servidor)
```

```
      0x000B: Código de error
      */

      if(sinit){
            server.write_only_register(ADDR_RX90_STATE, DISCONNECT);
            server.write_only_register(ADDR_CLIENT, NO_COMMAND);
            for (addr=ADDR_INI_POSITION;addr<=ADDR_FIN_POSITION;addr++){
                  server.write_only_register(addr, NO_COMMAND);
            }
            server.write_only_register(ADDR_PINZA, OPEN_PINCERS);

            printf("Servidor iniciado corectamente\n");

      }
      else {
            printf("Error al iniciar el servidor Modbus\n");
            return 0;
      }

      Rx90 rx90(dispositivo, posicion);
      Rx90::Action action = Rx90::NONE;
      server.write_only_register(ADDR_RX90_STATE, STATE_POWERINGON);

      //Se espera que el cliente se conecte
      printf("Esperando conexión del cliente\n");
      server.waitClient();
      printf("Cliente conectado al servidor\n");
      //server.write_only_register(NCLIENT, nClient);
      server.write_only_register(ADDR_CLIENT, COMMAND_START);

      while(true)
      {
            printf("Se espera la recepción de un comando\n");
            //Se espera la recepción de un comando
            int command=server.receiveCommand();

            if (command == RX90_COMMAND){ //si se recibe comando del RX90
                  server.sendConfirmation(); // Enviamos confirmación TCP

                  reg=server.read_only_register(ADDR_CLIENT); //obtiene el
comando escrito por el cliente
                  printf("Comando recibido: %d\n",reg);
                  printf("Realizando accion...\n");

                  if (reg==ROBOT_UP) {
                        action = Rx90::UP;
                        server.write_only_register(ADDR_RX90_STATE, reg);
                        rx90.move(action);
                        printf("RX90 UP\n");
                  }
                  else if (reg==ROBOT_DOWN) {
                        action = Rx90::DOWN;
                        server.write_only_register(ADDR_RX90_STATE, reg);
                        rx90.move(action);
                        printf("RX90 DOWN\n");
                  }
                  else if (reg==ROBOT_LEFT) {
                        action = Rx90::LEFT;
                        server.write_only_register(ADDR_RX90_STATE, reg);
                        rx90.move(action);
                        printf("RX90 LEFT\n");
```

```
                  }
                  else if (reg==ROBOT_RIGHT) {
                        action = Rx90::RIGHT;
                        server.write_only_register(ADDR_RX90_STATE, reg);
                        rx90.move(action);
                        printf("RX90 RIGHT\n");
                  }
                  else if (reg==ROBOT_BACKWARD) {
                        action = Rx90::BACKWARD;
                        server.write_only_register(ADDR_RX90_STATE, reg);
                        rx90.move(action);
                        printf("RX90 BACKWARD\n");
                  }
                  else if (reg==ROBOT_FORWARD) {
                        action = Rx90::FORWARD;
                        server.write_only_register(ADDR_RX90_STATE, reg);
                        rx90.move(action);
                        printf("RX90 FORWARD\n");
                  }
                  else if (reg==ROBOT_CATCH) {
                        if
(server.read_only_register(ADDR_PINZA)==OPEN_PINCERS){
                              printf("La pinza estaba abierta:
%d\n",server.read_only_register(ADDR_PINZA));
                              reg==ROBOT_CATCH;
                              server.write_only_register(ADDR_RX90_STATE,
reg);
                              action = Rx90::CATCH;
                              rx90.move(action);
                              server.write_only_register(ADDR_PINZA,
CLOSE_PINCERS);
                              printf("Pinza cerrada:
%d\n",server.read_only_register(ADDR_PINZA));
                        }
                        else if
(server.read_only_register(ADDR_PINZA)==CLOSE_PINCERS){
                              printf("La pinza estaba cerrada:
%d\n",server.read_only_register(ADDR_PINZA));
                              reg==ROBOT_DROP;
                              server.write_only_register(ADDR_RX90_STATE,
reg);
                              action = Rx90::DROP;
                              rx90.move(action);
                              server.write_only_register(ADDR_PINZA,
OPEN_PINCERS);
                              printf("Pinza abierta:
%d\n",server.read_only_register(ADDR_PINZA));
                        }
                        else {
                              printf("No se conoce el estado de la pinza\n");
                        }
                  }
                  else if (reg==ROBOT_AUTOMOVE) {
                        server.write_only_register(ADDR_RX90_STATE,
ROBOT_AUTOMOVE);
                        vector<int>
leePos=server.read_many_registers(ADDR_INI_POSITION,6);

                        action = Rx90::AUTOMOVE;
                        rx90.automove(leePos);
                  }
                  else {
```

```
                        action = Rx90::NONE;
                  }

                  Rx90::printAction(action);

            }

            if(command==DISCONNECT){
                  printf("Cliente desconectado del servidor\n");
                  server.write_only_register(ADDR_CLIENT, COMMAND_FINISH);
                  server.waitClient();
                  printf("Cliente conectado al servidor\n");
                  nClient++;
                  server.write_only_register(NCLIENT, nClient);
                  server.write_only_register(ADDR_CLIENT, COMMAND_START);
            }


      }
      server.write_only_register(ADDR_RX90_STATE, STATE_POWERINGOFF);
      rx90.~Rx90();
      server.write_only_register(ADDR_RX90_STATE, DISCONNECT);
      server.close();

}
```

ServerModbus.cpp

```
/*************************************************************************/
/*                                                       */
/*                SERVIDOR MODBUS PARA RX90                    */
/*                                                       */
/*************************************************************************/

#include "serverModBUS.h"
#include "constant.h"


/*********************************************************/
/* Constructor:Inicializa las variables de la clase       */
/* modbus                                    */
/*                                           */
/* Recibe: Nada                                    */
/* Devuelve: Nada                            */
/*********************************************************/

serverModBUS::serverModBUS()
{
      mb_mapping=NULL;
      ctx=NULL;
      rc=-1;
}

/*************************************************************************
****/
/* init: Inicializa la tabla de registros y el contexto TCP
      */
/* del cliente                                                  */
/*                                                    */
/*  Recibe:                                                  */
/*    ipServer: IP del servidor                                  */
```

```
/*    portServer: Puerto del servidor                                   */

/*    num_holding_registers: Número de registros de lectura y escritura
      */
/*                                                                      */
/*  Devuelve: (bool)                                                    */
/*    true: Servidor iniciado correctamente                             */
/*    false: error al inicializar                                   */
/*                                                                      */
/************************************************************************
****/
bool serverModBUS::init(char* ipServer, uint16_t portServer,int
num_holding_registers)
{
    ctx = modbus_new_tcp(ipServer, portServer);
    if(ctx==NULL)
    {
        fprintf(stderr, "Failed to create ctx: %s\n",modbus_strerror(errno));
      return false;
    }

    mb_mapping = modbus_mapping_new(0,0,num_holding_registers,0);
    if (mb_mapping == NULL) {
        fprintf(stderr, "Failed to allocate the mapping: %s\n",
                modbus_strerror(errno));
        modbus_free(ctx);
        return false;
    }
    socket = modbus_tcp_listen(ctx, 10);
    return true;
}

/************************************************************************
****/
/* waitClient: Espera la conexión de un cliente                         */
/*                                                                      */

/*  Recibe: Nada                                                        */
/*  Devuelve:                                                       */

/*                                                                      */
/************************************************************************
****/
void serverModBUS::waitClient()
{
    modbus_tcp_accept(ctx, &socket);
}


/************************************************************************
****/
/* read_only_register: Realiza la lectura de un solo registro
      */
/*                                                                      */

/* Recibe:                                                              */
/*    addr: direccion del registro a leer                           */
/* Devuelve:(int)                                                       */
/*    Valor del registro leído o -1 si ha habido error                  */

/*                                                                      */
```

```
/*************************************************************************
****/
int serverModBUS::read_only_register(uint16_t addr)
{
      int value;
      if(mb_mapping!=NULL && mb_mapping->nb_registers > addr)
      {
            value=mb_mapping->tab_registers[addr];
            return value;
      }
      else
            return -1;
}


/*************************************************************************
****/
/* read_many_registes: Realiza la lectura de varios registros
      */
/*                                                              */

/* Recibe:                                                      */
/*    addr: direccion del primer registro a leer                          */
/*    num_registers: número de registros a leer                 */
/* Devuelve: vector <int>                                       */
/*    Vector con los valores de los registros o vector vacio
      */
/*    si ha habido error                                        */

/*                                                              */
/*************************************************************************
****/
vector<int> serverModBUS::read_many_registers(uint16_t addr,int
num_registers)
{
      vector <int> values;
      if(mb_mapping!=NULL && (mb_mapping->nb_registers > addr +
num_registers))
      {
            for(int i=0;i<num_registers;i++)
                values.push_back(mb_mapping->tab_registers[addr+i]);
      }
      return values;
}

/*************************************************************************
****/
/* write_only_register: Realiza la escritura de un solo registro
      */
/*                                                              */

/* Recibe:                                                      */
/*    addr: direccion del registro a escribir                             */
/*    value: valor del registro a escribir                                */
/* Devuelve:(bool)                                              */
/*    true: Operación realizada correctamente                             */
/*    false: Operación fallida                                  */

/*                                                              */
/*************************************************************************
****/
```

```
bool serverModBUS::write_only_register(uint16_t addr,int value)
{
      if(mb_mapping!=NULL && (mb_mapping->nb_registers > addr))
      {
            //printf("Valor a escribir: %d\n", value);
            printf ("Registro de escritura 0x%04X: %d\n",addr,value);
            printf ("-----------------------------------------------\n");
            mb_mapping->tab_registers[addr]=value;
            return true;
      }
      else
            return false;
}


/*************************************************************************
****/
/* write_many_register: Realiza la escritura de varios registros
      */
/*                                                              */

/* Recibe:                                                      */
/*    addr: direccion del primer registro a escribir                  */
/*    num_registers: Numeros de registros a escribir                  */
/*    values: vector con los valores de los registros a escribir      */

/*                                                              */
/* Devuelve:(bool)                                                    */
/*    true: Operación realizada correctamente                         */
/*    false: Operación fallida                                        */

/*                                                              */
/*************************************************************************
****/
bool serverModBUS::write_many_registers(uint16_t addr,int
num_registers,vector <int> values)
{
      if(num_registers<=values.size())
            return false;

      else
      {
            if(mb_mapping!=NULL && (mb_mapping->nb_registers > addr +
num_registers))
            {
                  for(int i=0;i<num_registers;i++) {
                        mb_mapping->tab_registers[addr+i]=values.at(i);
                        printf ("Registro de escritura 0x%04X:
%d\n",addr+i,values[i]);
                  }
                  printf ("-----------------------------------------------
\n");
                  return true;
            }
            else
                  return false;
      }
}


/*************************************************************************
****/
```

```
/* receive_command: Se queda esperando hasta recibir una serie de
      */
/* comandos concretos                                                  */
/*                                                                     */
/*                                                                     */

/* Recibe: Nada                                                        */

/*                                                                     */
/* Devuelve:(int)                                                      */
/*    Comandos posibles:                                               */
/*          ROBOT COMMAND               1      (Comando hacia el robot)
      */
/*          BREAKER COMMAND ACTION      2      (Comando de breaker)
      */
/*          ROBOT BREAKER COMMAND ACTION 3     (Comando de robot y breaker)
      */
/*          DISCONNECT             4  (Cliente desconectado)            */
/*                                                                     */
/*                                                                     */
/*********************************************************************
****/
int serverModBUS::receiveCommand()
{
      int type_request;
      int applyCommand=false;
      bool flagRX90=false;
      char tecla='0';

      do
      {
            rc = modbus_receive(ctx, query);
            //if(kbhit())
            //    tecla = getch();
            if (rc != -1)
            {
                  type_request=analizeRequest();
                  //printf("Comando tipo %d\n",type_request);
                  if(type_request!=READ_ONLY && type_request!=NOT_PERMITTED)
                  {
                        //printf("Direccion cliente:
0x%04X:%d\n",clientOperation.addr_write);

                        int addr;
                        addr=clientOperation.addr_write/*+i*/;
                        if(addr==ADDR_CLIENT)
                              flagRX90=true;


                        if(flagRX90)
                        {
                              flagRX90=false;
                              applyCommand=RX90_COMMAND;
                        }
                        else
                        {
                               applyCommand==NO_COMMAND;
                               sendConfirmation();

                        }
                  }
                  else
```

```
                  {
                        if(type_request==NOT_PERMITTED)
                              sendException(1);
                        else
                              sendConfirmation();
                        applyCommand == NO_COMMAND;
                  }

                  printf("Pulse 'Ctr+C' si desea salir...");

            }
            else
            {
                  applyCommand=DISCONNECT;
            }
      }while(applyCommand == NO_COMMAND && tecla!='q');
      return applyCommand;
}
/**************************************************************************
****/
/* sendConfirmation: envía una confirmación de que la petición se ha
      */
/* realizado correctamente                                          */
/*                                                          */
/*                                                          */

/* Recibe: Nada                                                     */

/* Devuelve:Nada                                            */
/*                                                          */
/**************************************************************************
****/
void serverModBUS::sendConfirmation()
{
      modbus_reply(ctx, query, rc, mb_mapping);
}

/**************************************************************************
****/
/* sendConfirmation: envía una excepción si la petición no se ha        */
/* realizado correctamente                                          */
/*                                                          */
/*                                                          */

/* Recibe: Nada                                                     */

/* Devuelve:Nada                                            */
/*                                                          */
/**************************************************************************
****/
void serverModBUS::sendException(int exception)
{
      modbus_reply_exception(ctx,query,exception);
}


/**************************************************************************
****/
/* close: Cierra el servidor modBUS y libera la tabla de registros
      */
/*                                                          */
```

```
/* Recibe: Nada                                                        */

/* Devuelve:Nada                                                   */
/*                                                                 */
/*******************************************************************
****/
void serverModBUS::close()
{
      modbus_mapping_free(mb_mapping);
      modbus_close(ctx);
      modbus_free(ctx);
}


/*******************************************************************
****/
/* analizeRequest: Analiza el tipo de petición                          */
/*                                                                 */

/* Recibe: Nada                                                        */

/* Devuelve:(int)                                                  */
/*    READ_ONLY 0 (solo lectura)                                   */
/*    WRITE_ONLY 1      (solo escritura)                           */
/*    READ_WRITE 2      (lectura y escritura)                            */
/*    NOT_PERMITTED 3   (petición rechazada)                             */
/*                                                                 */
/*******************************************************************
****/
int serverModBUS::analizeRequest()
{
      int type;
      clientOperation.function=query[7];
      switch(clientOperation.function)
      {
            case READ_REGISTERS:
                  type=READ_ONLY;
                  clientOperation.addr_read=(query[8]<<8) +query[9];
                  clientOperation.numReg_read=(query[10]<<8) + query[11];
                  break;
            case WRITE_ONLY_REGISTER:
                  type=WRITE_ONLY;
                  clientOperation.addr_write=(query[8]<<8) +query[9];
                  clientOperation.numReg_write=1;
                  break;
            case WRITE_MULTIPLE_REGISTERS:
                  type=WRITE_ONLY;
                  clientOperation.addr_write=(query[8]<<8) +query[9];
                  clientOperation.numReg_write=(query[10]<<8) + query[11];
                  break;
            case READ_WRITE_REGISTERS:
                  type=READ_WRITE;
                  clientOperation.addr_read=(query[8]<<8) +query[9];
                  clientOperation.numReg_read=(query[10]<<8) + query[11];
                  clientOperation.addr_write=(query[12]<<8) +query[13];
                  clientOperation.numReg_write=(query[14]<<8) + query[15];
                  break;
            case READ_FIFO:
                  type=READ_ONLY;
                  clientOperation.requestWriting=false;
                  clientOperation.addr_read=(query[8]<<8) +query[9];
                  clientOperation.numReg_read=(query[10]<<8) + query[11];
```

```
                  break;
            case WRITE_MASK_REGISTER:
                  type=WRITE_ONLY;
                  clientOperation.numReg_write=1;
                  clientOperation.addr_write=(query[8]<<8) +query[9];
                  clientOperation.mask_and=(query[10]<<8) +query[11];
                  clientOperation.mask_or=(query[12]<<8) + query[13];
                  break;
            default:
                  break;
      }
//todos los registros son de lectura y escritura, aqui comprueba que donde
intenta escribir es un registro de escritura
      if(type!=READ_ONLY)
      {
            for(int i=0;i < clientOperation.numReg_write;i++)
            {
                  int reg=clientOperation.addr_write+i;
                  //printf("Registro del cliente: 0x%04X\n",reg);
                  if(reg<ADDR_CLIENT &&  reg>ADDR_FIN_POSITION) //comprueba
que solo se puede escribir en los registros asignados
                  {
                        return NOT_PERMITTED;
                  }
            }
            return type;
      }
      else
            return READ_ONLY;

}
```

ServerModbus.h

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <stdlib.h>
#include <errno.h>
#include <iostream>
#include <vector>
#include <modbus/modbus.h>
#include "constant.h"




using namespace std;

//Estructura donde se guardan los datos del comando recibido
typedef struct data_modBUS
{
      uint16_t function;
      uint16_t addr_read;
      uint16_t numReg_read;
      uint16_t addr_write;
      uint16_t numReg_write;
      uint16_t mask_and;
      uint16_t mask_or;
      uint16_t requestWriting;
}Operation;
```

```
class serverModBUS
{
      private:
            modbus_t *ctx;                              //Contexto TCP del
servidor Modbus
            modbus_mapping_t *mb_mapping;               //Tabla de registros del
servidor Modbus
            uint8_t query[MODBUS_TCP_MAX_ADU_LENGTH]; //Cadena de bytes con la
peticion del cliente
                  int rc;
            Operation clientOperation;                  //Datos de la petición
recibida
            int socket;                                 //Socket por el que
escucha el servidor

      private:
            int analizeRequest();                       //Funcion que analiza la
respuesta y rellena clientOperation
      public:
            serverModBUS();                                       //Constructor
de la clase
            bool init(char* ip,uint16_t port,int num_holding_registers);
//Inicia el servidor y la tabla de registro
            void waitClient();
            int read_only_register(uint16_t addr);
      //Función para leer un solo registro
            vector<int> read_many_registers(uint16_t addr,int num_registers);
            //Función para leer varios registros
            bool write_many_registers(uint16_t addr,int num_registers,
vector<int> values); //Función para escribir varios registros
            bool write_only_register(uint16_t addr,int value); //Función para
escribir en un solo registro
            int receiveCommand();                          //Función que recibe
los comandos del cliente
            void sendConfirmation();                       //Función que envía la
confirmación de una petición
            void sendException(int exception);          //Función que envía
una excepción a una petición
            void close();

};
```

RX90.cpp

```
//
// C++ Implementation: Rx90
//
// Description:
//
//
// <franroyal@yahoo.es>
//
//

#include "Rx90.h"
#include <sstream>
#include <cmath>
#include <stdexcept>
```

```
using namespace LibSerial;

// Inicio del programa
Rx90::Rx90(const std::string& serialPort, const std::string& originPoint) {

	init(serialPort, originPoint);
	x = 0.0;
	y = 0.0;
	z = 0.0;
}

Rx90::~Rx90() {
	close();
}

// Definimos las variables
void Rx90::init(const std::string& serialPort, const std::string&
originPoint) {

	serial.Open( serialPort.c_str() );
	serial.SetBaudRate( SerialStreamBuf::BAUD_9600 );
	serial.SetParity( SerialStreamBuf::PARITY_NONE );
	serial.SetCharSize( SerialStreamBuf::CHAR_SIZE_8  );
	serial.SetFlowControl( SerialStreamBuf::FLOW_CONTROL_NONE );
	serial.SetNumOfStopBits( 1 );
	serial.unsetf( std::ios_base::skipws ) ;

	// Set origin precision point
	std::stringstream command_origin;
	command_origin << "DO SET #ORIGIN=#PPOINT(" << originPoint.c_str() <<
")";
	sendCommand(command_origin.str());

	sendCommand("SPEED 30");
	sendCommand("DO ABOVE");
	sendCommand("DO MOVE #ORIGIN");
	sendCommand("HERE ORIGIN", true);
	sendCommand("DO OPENI");
	sendCommand("DO ENABLE CP");
}

void Rx90::close() {
	serial.Close();
}

void Rx90::sendCommand(const std::string& command, bool waitQuestionMark) {

	if(serial.IsOpen()) {
		serial << command << END;

		if(waitQuestionMark) {
			char qm;
			do { serial >> qm; } while(qm != '?');
			serial << END;
		}

		char r;
		do {
			serial >> r; std::cout << r;
		} while (r != '.');
```

```
	}
	else {
		std::cout << "Serial port not opened" << std::endl;
	}
}

void Rx90::panic() {
	sendCommand("PANIC");
}

void Rx90::move(const Action& action) {

	switch(action) {
		case NONE:
			break;
		case UP:
			z += DELTA_VH;
			break;
		case DOWN:
			z -= DELTA_VH;
			break;
		case RIGHT:
			x -= DELTA_VH;
			break;
		case LEFT:
			x += DELTA_VH;
			break;
		case BACKWARD:
			y += DELTA_VH;
			break;
		case FORWARD:
			y -= DELTA_VH;
			break;
		case CATCH:
			catchIt();
			break;
		case DROP:
			dropIt();
			break;
		default:
			;
	}

	// send the command
	send(x, y, z);
}

void Rx90::send(double x, double y, double z){
	std::stringstream position;
	position << "DO SET P" << "=SHIFT(ORIGIN BY " << (int)x << "," <<
(int)y << "," << (int)z << ")";
	std::string command = position.str();
	sendCommand(command);
	sendCommand("DO MOVE P");
}

void Rx90::catchIt() {
		sendCommand("DO CLOSEI");
}

void Rx90::dropIt() {
		sendCommand("DO OPENI");
```

```
}

void Rx90::automove(vector<int> leePos) {

	ostringstream pos;
	pos << '"'<<uintToInt(leePos[0])<<','<<uintToInt(leePos[1])<<','<<
uintToInt(leePos[2])<<','<<uintToInt(leePos[3])<<','<<uintToInt(leePos[4])<<'
,'<<uintToInt(leePos[5])<<'"';
	printf("RX90 MOVES TO ");
	cout << pos.str();
	std::cout << '\n';

	//char cadenaPosicion[]="160,-150,10,-90,-90,0";
	std::stringstream posicion;
	posicion << "DO SET P=#PPOINT(" << pos.str() << ")";
//?????????????????
	//sendCommand(posicion.str());
	sendCommand("DO MOVE P");
}

void Rx90::printAction(const Action& action) {

	std::cout << "Rx90::printAction: ";
	switch(action) {
		case NONE:
			std::cout << "none!";
			break;
		case UP:
			std::cout << "up!";
			break;
		case DOWN:
			std::cout << "down!";
			break;
		case RIGHT:
			std::cout << "right!";
			break;
		case LEFT:
			std::cout << "left!";
			break;
		/*case UP_RIGHT:
			std::cout << "up-right!";
			break;
		case UP_LEFT:
			std::cout << "up-left!";
			break;
		case DOWN_LEFT:
			std::cout << "down-left!";
			break;
		case DOWN_RIGHT:
			std::cout << "down-right!";
			break;*/
		case BACKWARD:
			std::cout << "backward!";
			break;
		case FORWARD:
			std::cout << "forward!";
			break;
		case CATCH:
			std::cout << "catch!";
			break;
		case DROP:
			std::cout << "drop!";
```

```
                  break;
            case AUTOMOVE:
                  std::cout << "automove!";
                  break;
            default:
                  std::cout << "unexpected!";
      }
      std::cout << std::endl;
}

int Rx90::uintToInt(uint16_t value) {
      int entero;
      if (value>65176)
            entero=-(65536-value);
      else
            entero=value;
      return entero;
}
```

RX90.h

```
//
// C++ Interface: Rx90
//
// Description:
//
//
// <franroyal@yahoo.es>
//
//

#ifndef RX90_H
#define RX90_H

#include <SerialStream.h>
#include <iostream>
#include <vector>

#include <stdint.h>

#define DELTA_VH 50
#define END "\r\n"

using namespace std;

class Rx90 {
public:
      Rx90(const std::string& serialPort, const std::string& originPoint);
      ~Rx90();

      enum Action { NONE, UP, DOWN, LEFT, RIGHT, UP_LEFT, UP_RIGHT,
DOWN_LEFT, DOWN_RIGHT, BACKWARD, FORWARD, CATCH, DROP, AUTOMOVE };
      vector<int> leePos;
      static void printAction(const Action& action);
      void move(const Action& action);
      void automove(vector<int> leePos);
      void panic();

private:
      void init(const std::string& serialPort, const std::string&
originPoint);
```

```
      void close();
      void catchIt();
      void dropIt();
      void send(double x, double y, double z);
      void sendCommand(const std::string& command, bool waitQuestionMark =
false);
      int uintToInt(uint16_t value);
      LibSerial::SerialStream serial;
      double x, y, z;
      uint16_t value;
};

#endif  // RX90_H
```

Constant.h

```
/*****************************************************************************/
/*                                                                           */
/*               CONSTANTES DEL SERVIDOR DEL RX90                            */
/*                                                                           */
/*****************************************************************************/

#define NUM_REG   20

/*

      REGISTROS DEL SISTEMA:                                          (Usado por)

      0x0000: Número de conexiones de clientes                         (servidor)
      0x0001: Estado del fichero de registro (0:cerrado/1:abierto) (servidor)
      0x0002: Estado del RX90                                          (servidor)
      0x0003: Comando recibido por el cliente                          (cliente)
      0x0004: Posicion de rotacion de la base del RX90                 (cliente)
      0x0005: Posicion de rotacion de la 1ª articulacion RX90          (cliente)
      0x0006: Posicion de rotacion de la 2ª articulacion RX90          (cliente)
      0x0007: Posicion de rotacion de la base de la articulación
              pinza del RX90                                           (cliente)
      0x0008: Posicion de rotacion de la articulacion de la pinza
              del RX90                                                 (cliente)
      0x0009: Posicion de rotacion de la base de la pinza del RX90 (cliente)
      0x000A: Estado de la pinza (0: abierta / 1: cerrada)             (servidor)
      0x000B: Código de error                                          (servidor)
      */

#define NCLIENT               0x0000
#define F_REGISTRO            0x0001
#define ADDR_RX90_STATE       0x0002 // ADDR_RX90_STATE y ADDR_CLIENT deben
coincidir al ejecutar un comando
#define ADDR_CLIENT           0x0003
// Las direcciones 4, 5, 6, 7, 8 y 9 se usan para guardar el ángulo de
rotación (posicion) de los ejes del RX90
#define ADDR_INI_POSITION     0x0004
#define ADDR_FIN_POSITION     0x0009
#define ADDR_PINZA            0x000A

// Estado pinza (ADDR_PINZA)
#define OPEN_PINCERS    0
#define CLOSE_PINCERS   1
```

```
// Tipos de registros
#define READ_ONLY  0
#define WRITE_ONLY       1
#define READ_WRITE       2
#define NOT_PERMITTED    3

// Comandos Modbus
#define READ_REGISTERS       0x03 // Valor que indica el comando para leer
registros
#define WRITE_MULTIPLE_REGISTERS 0x10 // Valor que indica el comando para
escribir varios registros
#define WRITE_ONLY_REGISTER    0x06 // Valor que indica el comando para
escribir un único registro
#define READ_WRITE_REGISTERS   0x17 // Valor que indica el comando para
leer/escribir registros
#define WRITE_MASK_REGISTER    0X16
#define READ_FIFO        0x18 // Valor que indica el comando para leer
registros de la pila

//Estado del sistema (ADDR_RX90_STATE)
#define DISCONNECT            -1
#define RX90_COMMAND          1
#define NO_COMMAND            0
#define STATE_POWERINGON      101
#define STATE_POWERINGOFF     102

//Robot command (ADDR_CLIENT)
#define COMMAND_START 501
#define COMMAND_FINISH 502

#define ROBOT_UP   908
#define ROBOT_DOWN       902
#define ROBOT_LEFT       904
#define ROBOT_RIGHT      906
#define ROBOT_BACKWARD   907
#define ROBOT_FORWARD    909
#define ROBOT_CATCH      905
#define ROBOT_DROP       0

#define ROBOT_AUTOMOVE   307

//Errores
#define ERROR_NO_ERROR   800
#define ERROR_UNKNOW     801
#define ERROR_SENSOR     802
#define ERROR_ACTUADOR   803
#define ERROR_MODBUS     804
```

Makefile

```
all: server Rx90.o
	rm -rf *.o

server: server.cpp serverModBUS.o Rx90.o
	g++ -o $@ -g -I/usr/local/include/ $< serverModBUS.o Rx90.o -
L/usr/local/lib/ -lserial -lpthread -lmodbus

#server.o: server.cpp
#	g++ -c server.cpp
#clear:
#	rm -rf *.o

serverModBUS.o: serverModBUS.cpp
	g++ -c serverModBUS.cpp

Rx90.o: Rx90.cpp Rx90.h
	g++ -c -g -I/usr/local/include/ $< -lpthread

clean:
	rm -rf *o server

install-libmodbus:
	cp libmodbus-3.0.6/lib/libmodbus.* /usr/lib
```

REFERENCIAS

[1] IETF, RFC 791: Internet Protocol, Marina del Rey, California, 1981.

[2] National Instruments, EIA/TIA RS-232C Standard, 2005.

[3] National Instruments, "Información Detallada sobre el Protocolo Modbus," 16 Octubre 2014. [Online]. Available: http://www.ni.com/white-paper/52134/es/. [Accessed 12 Julio 2017].

[4] MODICON, Inc., Industrial Automation Systems, Modicon Modbus Protocol Reference Guide, Massachusetts, 1996.

[5] Modbus-IDA, Modbus Application Protocol Specification, 2006.

[6] IETF, RFC 793: Transmission Control Protocol, Marina del Rey, California, 1981.

[7] Simply Modbus, "Modbus TCP/IP," 2017. [Online]. Available: http://www.simplymodbus.ca/TCP.htm. [Accessed 10 Julio 2017].

[8] Ubuntu, "Ubuntu 16.04.2 LTS (Xenial Xerus)," 15 Febrero 2017. [Online]. Available: http://releases.ubuntu.com/16.04/. [Accessed 26 Febrero 2017].

[9] VMware, "Descargar VMware Workstation Player," 4 Julio 2016. [Online]. Available: https://www.vmware.com/es/products/player/playerpro-evaluation.html. [Accessed 20 Enero 2017].

[10] Stäubli Faverges, Robot Familia RX Serie 90 - CS7, 1996.

[11] Stäubli Faverges, Lenguaje V+, 1997.

[12] Adept, "V+ Language Reference Guide, v17.x," 1994-2014. [Online]. Available: http://www1.adept.com/main/KE/DATA/V%20Plus/V%20Language%20Reference/vlangTOC.html. [Accessed 10 Junio 2017].

[13] J. Liberty and R. Cadenhead, Programación C++, Madrid: Anaya Multimedia, 2011.