Informática Industrial Curso 2020-2021

Práctica 2

Enunciado

Programar una aplicación en C++ que implemente un servidor Modbus, inicialmente un Modbus/RTU y posteriormente un Modbus/TCP.

Para ello se diseñarán las siguientes clases:

Mensaje

Representa un mensaje Modbus como una secuencia de bytes (`uint8_t`). Tendrá, al menos, los siguientes métodos:

- `Mensaje()` constructor por defecto crea un mensaje vacío.
- `Mensaje(std::vector<uint8_t> bytes)` constructor que crea un mensaje con los bytes del vector pasado.
- `std::string toString()` devuelve representación del mensaje indicando el número de bytes y el valor de sus bytes en hexadecimal. Por ejemplo si el mensaje contiene los bytes 199, 106 y 112 el string devuelto debe ser '`[(3) c7 6a 70 ]`'
- `unsigned size()` devuelve el número de bytes que tienen el mensaje en ese momento.
- `uint8_t getByteAt(unsigned ind)` devuelve el byte que se encuentra en la posición `ind` del mensaje.
- `void setByteAt(unsigned ind, uint8_t dato)` coloca el byte `dato` en la posición `ind`.
- `void pushByte_back(uint8_t dato)` añade un doble byte `dato` al final del mensaje.
- `bool crcOK()` indica si el CRC del mensaje es correcto
- `void aniadeCRC()` añade el CRC al mensaje
- `uint16_t getWordAt(unsigned ind)` devuelve el doble byte situado a partir de la posición `ind` del mensaje.
- `void setWordAt(unsigned ind, uint16_t dato)` coloca el doble byte `dato` a partir de la posición `ind`.
- `void pushWord_back(uint16_t dato)` añade un doble byte `dato` al final del mensaje.
- sobrecargar el operador `<<` para que se vuelque por un `std::ostream` lo mismo que `toString()`.

En todos los métodos que reciban un índice sobre el mensaje, si éste es incorrecto se deberá lanzar una excepción `std::out_of_range`.

Para el manejo de los dobles bytes se usará la ordenación big-endian utilizada en el protocolo Modbus.

ModbusRTU

Representa al servidor Modbus/RTU. Esta clase tendrá, al menos, el siguientes métodos:

- `ModbusRTU(uint8_t id)` constructor donde se especifica el identificador del dispositivo.
- `Mensaje peticion(Mensaje recibido)` que recibe como parámetro un mensaje de petición Modbus/RTU y devuelve la respuesta correspondiente.

El dispositivo completo ha de tener definido:

- Las primeros 20 salidas digitales con un valor inicial de 0 en las posiciones pares y de 1 las posiciones impares.
- Las primeras 20 entradas digitales:
  - Las 15 primeras tendrán un 0 si el valor del registro analógico de entrada correspondiente (a partir del 30100) es par, y un 1 en caso contrario.
  - Los restantes deberán estar inicialmente a 0.
- Los primeros 10 registros analógicos de salida, con un valor inicial de 0, 4, 8, 12, ...
- Los registros analógicos de entrada del 30100 al 30119, que deberán contener en cada momento:
  - Los 3 primeros: el contador de peticiones recibidas, el numero de bytes recibidos (excluyendo el CRC) , el número de bytes enviados (excluyendo CRC).
  - Los 6 siguientes: el año, el mes, el día, la hora, el minuto y el segundo actuales, respectivamente.
  - Los 4 siguientes: el UID, el GID , el PID , el PPID del proceso.
  - El resto deben estar inicialmente a 0 los pares y a 1111 los impares.

Parte 1

Realizar la clases `Mensaje` y todos sus métodos.

Parte 2

Implementar la funcionalidad de los registros (digitales y analógicos) de salida (funciones: 03, 01, 05, 15, 06 y 16). Se variará el contenido de los mismos como consecuencia de las peticiones de escritura recibidas.

Parte 3

Implementar la respuesta de error adecuada cuando se reciba paquete:

- ID que no corresponde: no devolver nada.
- CRC incorrecto: no devolver nada.
- Solicite función no implementada: devolver error 01
- Solicite registro fuera de rango: devolver error 02
- Paquete tenga datos inválidos o tamaño incorrecto: devolver error 03.

Parte 4

Implementar la funcionalidad de los registros, digitales y analógicos, de entrada (funciones 02 y 04), generando los errores correspondientes.

Parte 5

Crear la clase `ModbusTCP`. Esta clase tendrá, al menos, el siguientes métodos:

- `ModbusTCP(uint16_t puerto, uint8_t unitId)` donde se indica el puerto TCP donde debe escuchar y el identificador de la unidad `ModbusRTU` conectada.
- `void atiende(unsigned numClientes)` abre la conexión TCP , acepta conexiones de los clientes redirigiendo las peticiones a la unidad `ModbusRTU` y devolviendo la respuesta al cliente. Debe controlar los errores y las desconexiones de los clientes. Cuando haya atendido a `numClientes` clientes debe cerrar el socket y terminar. En caso de error en la comunicación TCP deberán lanzar una excepción del tipo `std::runtime_error` con un mensaje adecuado.

Parte 6

Crear la clase `ModbusTCP2`. Esta clase tendrá, al menos, el siguientes métodos:

- `ModbusTCP2(uint16_t puerto, uint8_t uId1, uint8_t uId2)` donde se indica el puerto TCP donde debe escuchar, el identificador de una primera y una segunda unidad `ModbusRTU` conectadas.
- `void atiende(unsigned numClientes)` abre la conexión TCP , acepta conexiones de los clientes redirigiendo las peticiones a la unidad `ModbusRTU` correspondiente según su Id y devolviendo la respuesta al cliente. Debe controlar los errores y las desconexiones de los clientes. Cuando haya atendido a `numClientes` clientes debe cerrar el socket y terminar. En caso de error en la comunicación TCP deberán lanzar una excepción del tipo `std::runtime_error` con un mensaje adecuado.

Parte 7

Crear la clase `ModbusTCP2Multiple` que tenga la misma funcionalidad que `ModbusTCP2` pero que permita el acceso concurrente de hasta `numClientes` clientes simultáneos. Se deberá garantizar el acceso en exclusión mutua a las unidades. Hacer que las unidades tarden 2 segundos en responder a una petición, para poder comprobar que se respeta la exclusión mutua.

Parte 8

Crear la clase `ModbusUDP2` que tenga la misma funcionalidad que `ModbusTCP2` pero que utilice el protocolo de red UDP/IP en lugar del TCP/IP.

Guía de estilo

Al escribir el código fuente se debe de respetar las siguientes reglas:

- Indicar autor y fecha en las primeras línea de los fichero fuente.
- Usar dos espacios para cada nivel de sangrado, utilizando caracteres espacio.
- En cada línea no deberá haber más de una instrucción; aunque una instrucción puede ocupar

varias líneas. En ese caso las líneas de continuación deberán tener dos niveles de sagrado más que la inicial (4 espacios más).
- No utilizar `using namespace std` sino especificar el espacio de nombre cuando sea necesario (`std::cout`, `std::endl`, `std::string`, `std::vector`, etc.)
- Dejar un espacio antes y después de cada operador, en especial de los de asignación (`=`, `+=`, `-=`, …), comparación (`>`, `<`, `==`, …) y los de envío y recepción de `stream` (`<<` y `>>`).
- Las líneas deberán tener, preferiblemente, menos de 72 caracteres y nunca superar los 80.
- Utilizar mensajes de depuración allí donde sea conveniente. Se utilizará la salida de error (`std::cerr`) para mostrar dichos mensajes.
- Declarar las variables justo antes de ser utilizadas y, si es necesario, hacer la declaración y la inicialización en la misma instrucción.
- Se usarán dos ficheros para cada clase: en `Clase.hpp` estará la declaración, y en `Clase.cpp` la definición de TODOS los métodos (no usar métodos *inline*).
- Los atributos de una clase, o bien tendrán un identificador que comience por el carácter barra baja (`_atributo`), o bien se accederán siempre a través del puntero `this` (`this->atributo`).

Guía de diseño

- Si la información se puede representar de varias maneras equivalentes, almacenar en el objeto **sólo una** de las representaciones.
- Optimizar el código, es decir, tratar de utilizar los métodos ya definidos para que no aparezca código que realice la misma funcionalidad en dos sitios distintos.
- Poner atributo `const` a los métodos que no modifican el objeto implicado.