bsi.

...making excellence a habit.™

A cost-effective IoT system simulation for cyber-resilience testing

An examination of a real-life project

A whitepaper

Introduction

This paper depicts the quick deployment of a fully scalable, multipurpose model capable of simulating a wide range of control devices and smart meters, keeping costs to a minimum. The use case presented in this paper is based on an important and common IoT (Internet of Things) component: SCADA (Supervisory Control and Data Acquisition), but it can be fully extended to an entire IoT system.

The times when control and SCADA systems were isolated are long gone. Actually, many authors have started to refer to control systems as some kind of IoT system[1].

The majority of organizations using IoT systems don't test their control devices. System administrators are reluctant to test them in the production environment (due to the threat to system availability) and the setup of a traditional testing laboratory is something complex and expensive that usually doesn't fit in budget.

This paper describes the means to building a functional testing environment that fills that gap. Instead of setting up an entire expensive and complex structure, simulation is used in an effective way, in order to accelerate the installation process and cut down the costs dramatically.

Use cases for the simulation environment

The system described in this paper has two direct applications for industrial manufacturing players.

- **Performance testing -** Real Control devices (PLCs, RTUs, DCUs) and Smart Devices (smart meters, smart actuators, instrumentation) can be connected to the simulated environment and be operated exactly the same as in production, which provides the control engineers the capability of testing the configuration without impacting the production environment, and measure different aspects of the devices (accuracy, precision, repeatability) before "going live".
- **Resilience testing -** Once the devices have been configured and tested for good behaviour under "normal conditions", penetration tests can be conducted, to gain information on how the devices would react under "abnormal circumstances" or even under a theoretical cyberattack. This information is critical in order to build a resilient architecture around them.

1 http://internetofthingsagenda.techtarget.com/feature/Ten-tips-for-migrating-to-SCADA-systems-plus-IoT

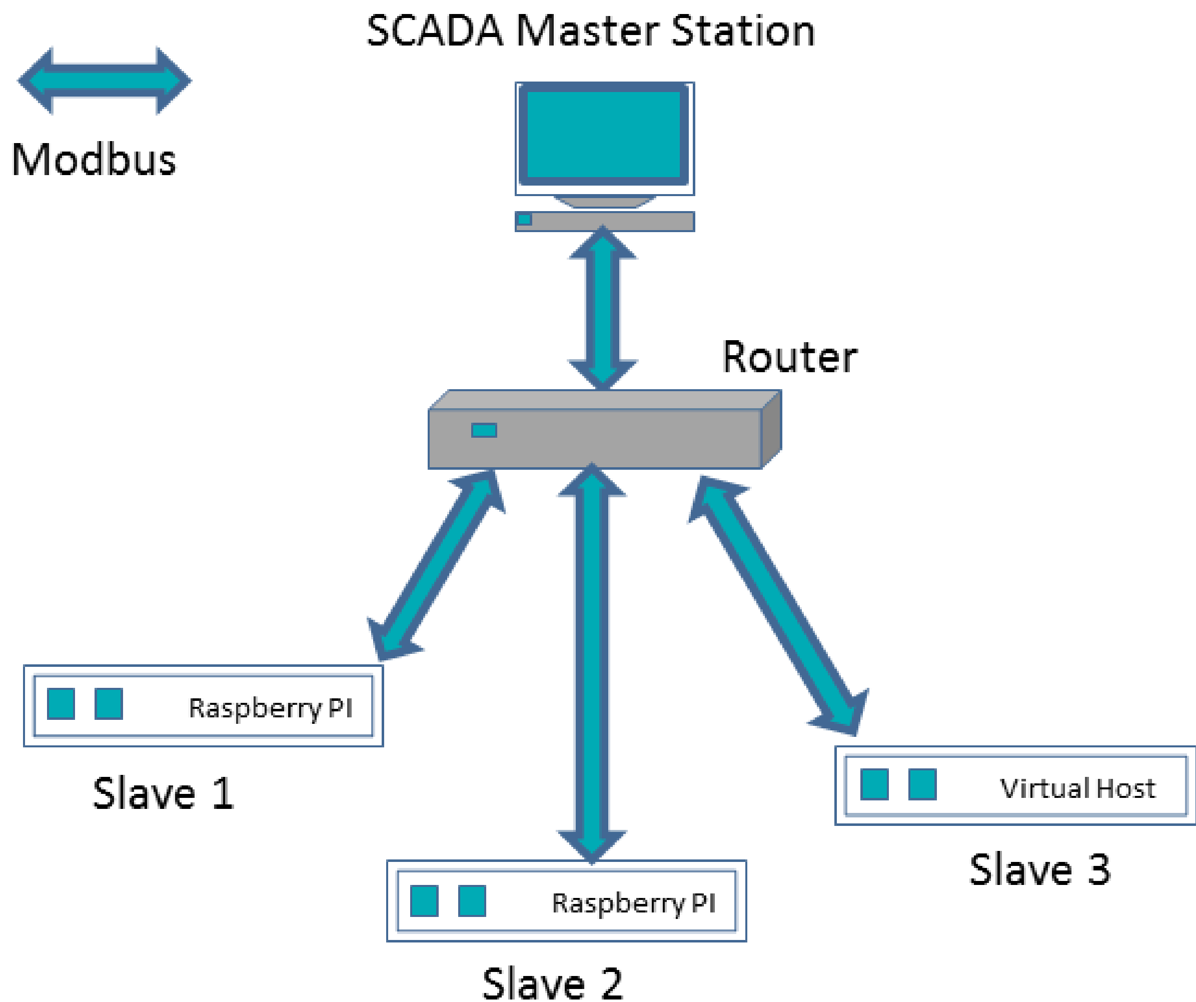
Project architecture

The project can be broken down into three major points:

- **Master station -** this represents the control station that oversees all the controlled system, with the possibility of acting in it whenever needed. For the project described in this paper, only one master station is enabled[1], although it is common in real installations to have one additional, redundant master station which would take over the system in case the main one fails
- **Slave control devices -** representing control devices acting as slaves, attending the requests of the master station. This project used Raspberry Pi with a Linux O.S. in order to simulate the slaves. The chosen architecture has three slaves (two "physical" and one "virtual", hosted in the same computer that acts as the master station) but this number can be easily escalated adding up more embedded devices.
- **Communications -** this project simulated Modbus communications over TCP between the slaves and the master station. Modbus communications are widely used in control systems all over the world, making them a suitable option for the project. Other control communication protocols are considered for future expansions of the project, as is shown overleaf.

1 The unique master station option was chosen to keep the project simple and clear. Nevertheless, the inclusion of a redundant master station is as simple as the inclusion of another pc hosting the same software

Call: +44 345 222 1711 / +353 1 210 1711 Email: cyber@bsigroup.com Visit: bsigroup.com



Master station configuration

INDIGO SCADA[1] is a free distribution SCADA software, useful for providing a quick and simple overview of a controlled process.

The use of this software was particularly appropriate for this project, so the control load could be kept to the minimum, focusing in the Modbus nodes configuration and visualization.

As can be seen in the figure overleaf, the project kept the control functions of INDIGO SCADA as simple as possible, it shows five Boolean values and five integer values for each of the slaves, each of these values, representing inputs and outputs in the controlled process[2].

The process of configuring the INDIGO SCADA master station was conducted in three simple steps:

- Protocol configuration
- Unit configuration
- HMI configuration

1 http://www.enscada.com/a7khg9/IndigoSCADA.html

2 One Boolean input could be a valve with two possible states (open/close) or a pump (on/off). The integers on the other hand can represent process values (temperature, flow, liquid level) or actuators with analogic values (degree of openness of a valve, frequency feeding an electric motor etc.)

Alarms | Status | Messages | hmi0.ui | hmi1.ui | hmi2.ui | hmi3.ui | hmi4.ui | hmi5.ui | hmi6.ui | hmi7.ui | hmi8.ui | hmi9.ui

MDB1Point01 0.00 spi	MDB1Point02 0.00 spi	MDB1Point03 0.00 spi	MDB1Point04 0.00 spi
MDB1Point05 0.00 spi	MDB1Point06 7.00 spi	MDB1Point07 7.00 spi	MDB1Point08 7.00 spi
MDB1Point09 7.00 spi	MDB1Point10 7.00 spi	MDB2Point01 0.00 spi	MDB2Point02 0.00 spi
MDB2Point03 0.00 spi	MDB2Point04 0.00 spi	MDB2Point05 0.00 spi	MDB2Point06 7.00 spi
MDB2Point07 7.00 spi	MDB2Point08 7.00 spi	MDB2Point09 7.00 spi	MDB2Point10 7.00 spi

MDB1Point05 Point Number 05

Comment	
Failure Time	2017-04-27 11:16:39.673
Last Update	2017-05-10 14:35:07.392
No. Alarms	0
No. Measures	17
No. Warnings	0

Name	spi	Minimum	Maximum	Av
VALUE	0.00	0	0	0

Name	Lower Alarm	Lower Warning	Upper Warnin
VALUE	(***)	(***)	(***)

Monitor is Running -- Active receipe: (default) -- User: (***)

Protocol configuration

The process of protocol configuration consists of mapping the required inputs and outputs with their correspondent parameters[1]. That is performed in the INDIGO SCADA via database files, which define the needed I/O, with all the required parameters. The protocol configuration is shown in the following figure:

IndigoSCADA Protocol Configurator - C:/Indigo SCADA/scada/project/modbus_database1.db

File Edit View Help

Database Structure | Browse Data | Execute SQL

Table: modbus_table

New Record | Delete Record

	modbus_function	modbus_function_w	modbus_address	offset_bit	modbus_type	ioa_control_center	iec_type_read	iec_type_write	deadband
1	1	5	0	0	VT_BOOL	1	M_SP_TB_1	C_SC_TA_1	0
2	1	5	1	0	VT_BOOL	2	M_SP_TB_1	C_SC_TA_1	0
3	1	5	2	0	VT_BOOL	3	M_SP_TB_1	C_SC_TA_1	0
4	1	5	3	0	VT_BOOL	4	M_SP_TB_1	C_SC_TA_1	0
5	1	5	4	0	VT_BOOL	5	M_SP_TB_1	C_SC_TA_1	0
6	3	16	0	0	VT_I2	6	M_ME_TE_1	C_SE_TB_1	0
7	3	16	1	0	VT_I2	7	M_ME_TE_1	C_SE_TB_1	0
8	3	16	2	0	VT_I2	8	M_ME_TE_1	C_SE_TB_1	0
9	3	16	3	0	VT_I2	9	M_ME_TE_1	C_SE_TB_1	0
10	3	16	4	0	VT_I2	10	M_ME_TE_1	C_SE_TB_1	0

< 1 - 10 of 10 > Go to: 0

Device configuration

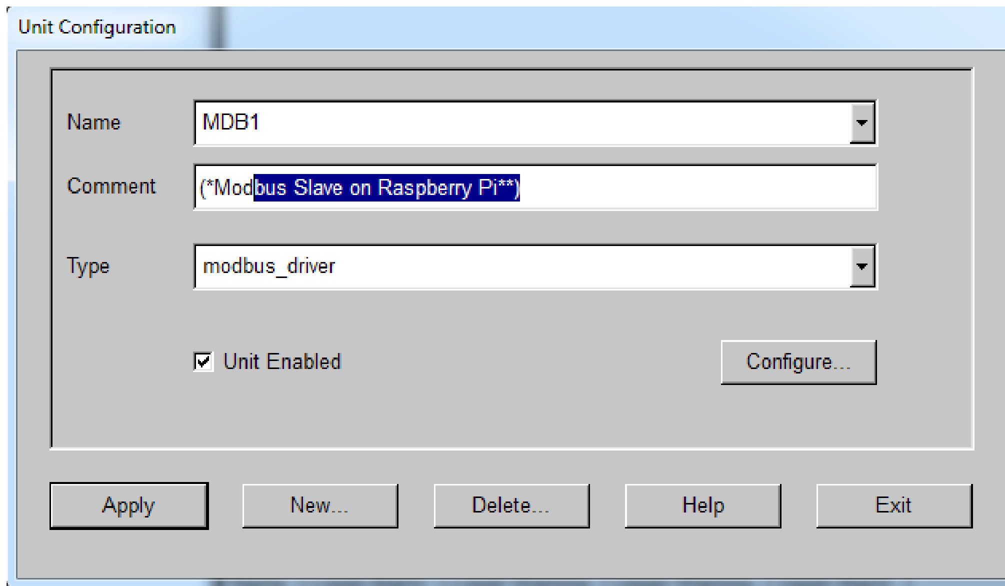
After the required inputs and outputs have been mapped during protocol configuration, the different Modbus slaves which host them need to be identified as well by the master unit.

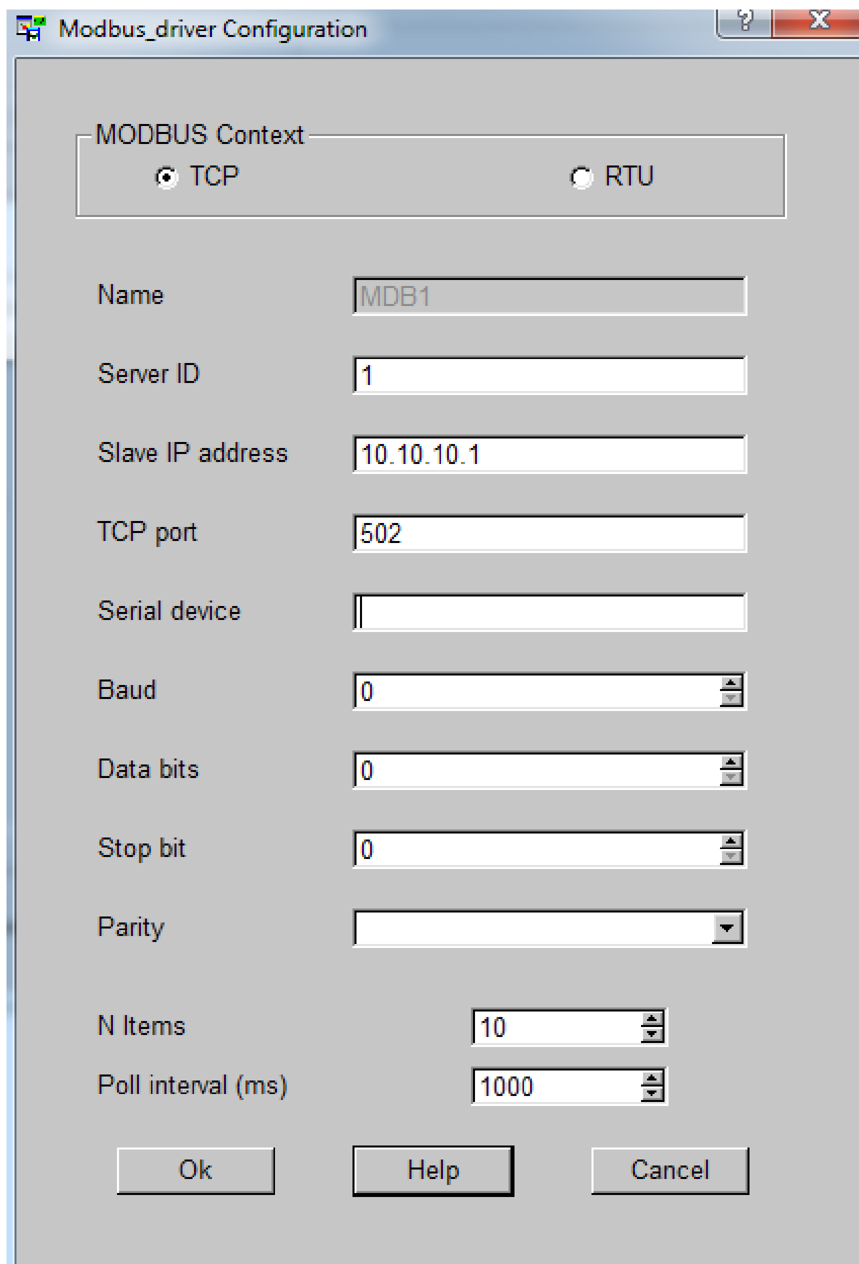
Unit configuration consists of selecting the drivers which will manage each of the Modbus slaves. These drivers need to correlate to the communication protocol of the slave they will be using (Modbus over TCP in our case).

After the driver is selected, it needs to be configured, indicating the IP address of the slave, the communication port which

1 I/O parameters could be the memory addresses, data type, or Modbus functions authorized for each of them.

will be used, and the polling interval. The unit configuration is shown in the figures below:





HMI configuration

Once the Protocol and the Modbus units are defined, only the customization of the "front-end" HMI needs to be developed.

INDIGO SCADA provides a user friendly HMI design framework, with an extensive database of grahical objects that can be used for representing the I/O of the different Modbus slaves. The HMI control panel used for this project is shown in the following figure:

- **Hard real-time operation:** This constraint leads to the weakest point when it comes to using Raspberry Pi. Raspberry Pi does not include a physical clock, so the only way of time measurement is using the software clock, which is not precise enough to be considered real-time .This issue was tackled using two approaches:
  - For applications where hard real-time is not critical, the precision of the ±150ppm may be enough.
  - For time critical applications, the hardware real-time clock DS3231[2] was added to the Raspberry Pi, achieving a precision of ±2ppm[3]. The addition and configuration of the real-time clock is detailed in Annex B.

Apart from using a Raspberry Pi for Modbus node simulation, the project also developed a virtual Modbus slave, in order to provide as many options as possible.

The virtual Modbus node was simulated in Python, hosted on the same computer as the SCADA master station. The Python code and reference to Pymodbus (library used for Modbus simulation) is described in annex D.

Slave control devices

The simulation of the Modbus slaves is the core of the project. Real control devices, such as PLCs or RTUs, have three relevant characteristics which can be used to define them for the purpose of this project:

- **Physical resilience**
- **Operational stability:** Devices need to operate constantly, (24 hours, 7 days per week), and system crashes and reboots have a high impact on the overall performance
- **Hard real-time operation:** Key requirement for process control systems

The main challenge for the project was to find a device which could operate in the same fashion, cutting down the costs

This project used multipurpose computer Raspberry Pi hosting Linux O.S to simulate the Modbus slaves. The treatment of the previously mentioned constraints was conducted as follows:

- **Physical resilience:** This constraint is not really applicable, as the project's intended use is in a laboratory, where the real devices and systems can be tested. Therefore, Raspberry Pi is valid on that front.
- **Operational stability:** Using the Linux O.S, and avoiding parallel processes running in the Raspberry Pi as much as possible, achieved this constraint completely.

Modbus slave configuration

In line with the master configuration (in INDIGO SCADA), Modbus slaves were kept as simple as possible.

The memory of each of the slaves was configured to host 4 different types of I/O:

- **Digital input:** Boolean data which can only be read
- **Coils:** Modbus denomination for digital output. Boolean data which can be read and written
- **Holding registers:** Modbus denomination for analogue output, which can be read and written
- **Input registers:** Modbus denomination for analogue input, which can only be read

The slaves implemented in the project had all four types of I/O, but in practice, only holding registers and coils are used. Using those covers the Boolean and analogue I/O spectrum, plus enabling the capability of writing and reading them.

Digital inputs and input registers would measure the same

2 https://datasheets.maximintegrated.com/en/ds/DS3231.pdf
3 The architecture represented both approaches having one raspberry operating just with the software clock, and the other one with the real-time clock attached.

type of data, but without the capability of acting on them (meaning that the master would be able to see their values, but not change them).

The quantity of the I/O in each of the slaves is limited to 5 of each of the four types. The escalation on the number of I/O is really simple and only limited by the device memory.[4] The code sample which enables the I/O in a Modbus slave can be seen in the following figure:

```
93 store = ModbusSlaveContext(
94 di = ModbusSequentialDataBlock(1, [0]*5),
95 co = ModbusSequentialDataBlock(1, [0]*5),
96 hr = ModbusSequentialDataBlock(1, [0]*5),
97 ir = ModbusSequentialDataBlock(1, [0]*5))
98 context = ModbusServerContext(slaves=store, single=True)
```

Further information on the code used in the simulated control devices can be found in Annex D.

Communications

As previously mentioned, this project chose Modbus as the communication standard for the simulations due to its extended use in industrial environments all over the world. According to Modbus:

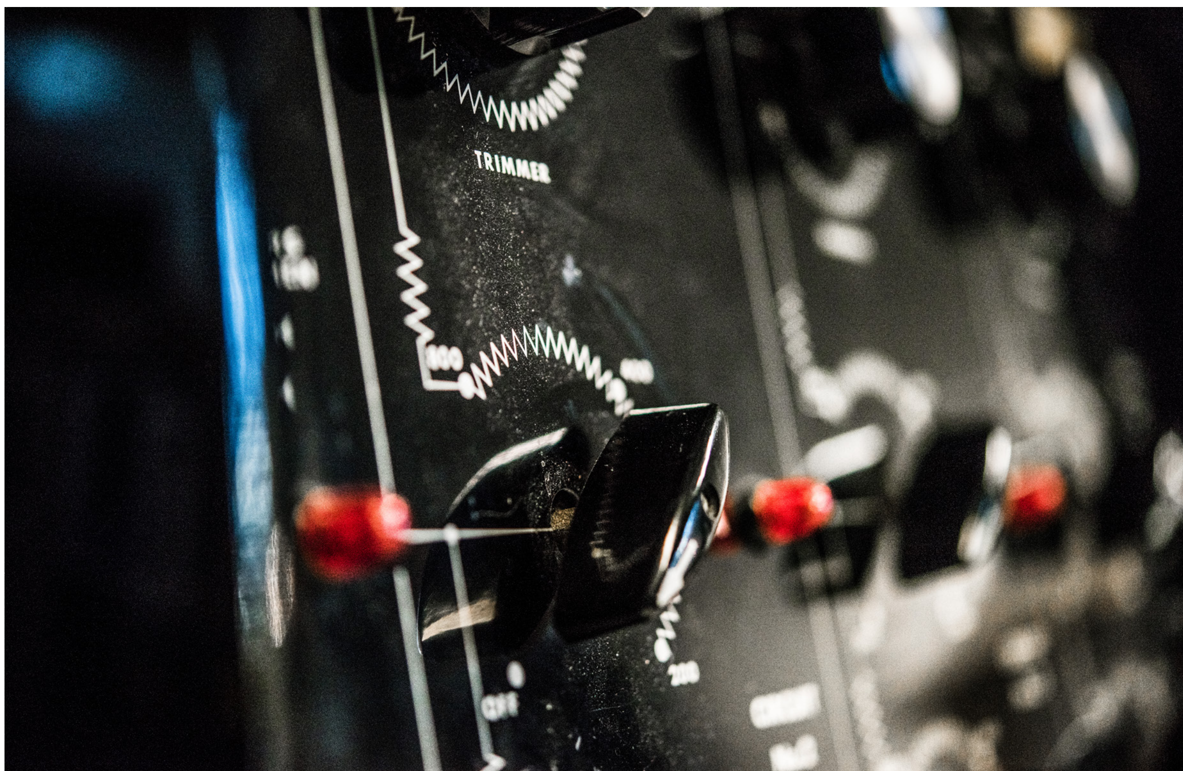
"Modbus Protocol is a messaging structure developed by Modicon in 1979. It is used to establish master-slave/client-server communication between intelligent devices. It is a de facto standard, truly open and the most widely used network protocol in the industrial manufacturing environment. It has been implemented by hundreds of vendors on thousands of different devices to transfer discrete/analog I/O and register data between control devices. It's a lingua franca or common denominator between different manufacturers. One report called it the "de facto standard in multi-vendor integration". Industry analysts have reported over 7 million Modbus nodes in North America and Europe alone."[5]

Initially, the Modbus protocol was designed in its physical layer to be conducted over serial cable. At present, LAN communications are the main stream, with Ethernet as they drive the adaptation of Modbus, which culminated with Modbus TCP/IP (the one used in this paper).

The implementation of Modbus protocol was performed in Python using Pymodbus library[6]. Further information on the Python code can be found in Annex D.

One remarkable aspect about Pymodbus is that the library actually generates real Modbus communications, making the traffic generated in this project indistinguishable from any real industrial Modbus communications. The only potential difference could be the time performance of the network, which is driven by time performance of the Raspberry Pi and the Ethernet communications.[7]

4 The Raspberry Pis used in this project use SD memory cards, and even the smallest today would be enough to host hundreds of each I/O types.
5 Definition provided by Modbus organization. http://www.modbus.org
6 https://github.com/bashwork/pymodbus
7 Time performance related to Ethernet communications was outstanding, due to the simplicity of the model (three slaves and one master). For more information about Real-time capabilities refer to Annex B



Future implementations

The project scope is not limited to Modbus communications; it was used first due to its importance and wide use. Other Industrial protocols (DNP3, Profibus, and IEC 608070) and even some complete IoT implementations (Wyiliodrin, IBM Watson, AWS IoT) are considered for future extensions of this project, as is shown in Annex A.

Annex A - Future implementation sources

This annex will expose different protocol implementations which can be used in future extensions of this project. All the tools considered here have an open license, keeping the cost to a minimum. The last of the references depicts full IoT implementations using the Raspberry Pi, listing the most relevant IoT cloud platforms.

Modbus:

Language: Python 2.7

Library source: https://pypi.python.org/pypi/pymodbus

DNP3:

Language: C/C++ (Provided the Python bindings and Java bindings)

Library source: https://github.com/automatak/dnp3

Profibus:

Language: Python 2.7

Library source: https://bues.ch/cms/automation/profibus.html

S7 PLC compatible module (AWLsim)

Language: Python 2.7

Library source: https://bues.ch/cms/automation/pilc.html

IEC 60870-5-101 /104

Language: Python 2.7

Library source: https://github.com/invveritas/PYIEC60870-104

IoT:

Cloud Platforms:

- Wyliodrin: https://www.wyliodrin.com/
- IBM Watson: https://www.ibm.com/internet-of-things/platform/watson-iot-platform/
- AWS IoT: https://aws.amazon.com/iot-platform/

Annex B - Real time operation

As previously stated in this paper, Raspberry Pi does not come with a hardware clock; it only uses the software version.

The use of software has major implications for time management:

- The device will be desynchronized when switched off
- The precision achieved by the device will be low

The first implication does not have a big impact on this project. As it is only a simulation model and therefore not used in production, its normal use will be for short periods of simulation (from hours to days). This means that in most cases, this will be acceptable to synchronize the devices each time they are powered up.

However, the impact caused by the second implication compromises the ability of the Raspberry Pi to operate in real time. The average precision of the Raspberry Pi is ±150ppm. This means approximately a precision of ±13 s/day. If the system to be simulated does not have hard real-time requirements, even this level of precision could be enough.[1] For continuous control system simulation, which usually requires hard real time in all their operations, the operation of the Raspberry Pi using only the software clock is not acceptable. This project used the hardware clock DS3231 to address this problem.

1 When simulating RTU based control systems for the power industry, the usual time constraints are soft. And a maximum delay of 13 seconds per day will be acceptable in many cases.

Call: +44 345 222 1711 / +353 1 210 1711 Email: cyber@bsigroup.com Visit: bsigroup.com

This clock is powered with a small battery (enough for years of operation), which avoids the desynchronization problem with the device when switching it off. The specified precision of the clock is ± 2ppm (less than 0.2 seconds per day) which is acceptable for most real-time control operations.

The clock is easily attached in the GPIO[2] of the Raspberry Pi, and the configuration needed for using it by the device is as follows:

- Editing /boot/config.txt file:
  - Including the following line: dtoverlay=i2c-rtc,ds3231
- Editing /lib/udev/hwclock-set file:
  - Commenting the following lines:
    - #if [-e /run/systemd/system] ; then
    - # exit 0
    - #fi

Having the Raspberry Pi time driven by DS3231 enables the capability of limiting the maximum time each operation is taking (stopping the process if it is taking more time than established). Therefore, this approach effectively enables the device operation in real-time.

Ethernet communications

In order to be able to consider the whole system 'real-time capable', the communications need to fulfil the same constraints as each of the devices (making the whole system as fast as the slowest of its members).

The project uses Ethernet as the communication conduit. Strictly speaking, it doesn't guarantee real time operations.

For the simple architecture used in this project,[3] though, the traffic in the Modbus TCP network is light enough to provide communications without significant delay.

This project's architecture is designed to facilitate more complex configuration with multiple slaves. For these cases, the use of the time sensitive Ethernet is recommended.

Time sensitive Ethernet is currently under development by an IEE task group,[4] and its functionalities and characteristics are detailed under the standard series ISO/IEC/IEEE 8802-3.[5]

For further information about time sensitive Ethernet, please refer to the following BSI white paper, which provides a general overview: Time Sensitive Ethernet whitepaper.[6]

Annex C - Elements used in the model

Hardware:

- **Raspberry Pi:** Small computer which hosts the Modbus node simulator
- **DS 3231 real-time clock:** This is used to enforce real time operation
- **LCD 20x4:** Small Liquid Cristal Display used to show the data stored in the Raspberry Pi.
- **Router:** Network element used to enable the connection between Modbus master and different slaves
- **Computer:** Hosting the SCADA software acting as Modbus master

Software:

- **Programming language:** Python 2.7 was chosen for the processes that enable the Modbus nodes, using Pymodbus library .
- **SCADA software:** The open source licensed Indigo SCADA, was used for the master station configuration.[7]

Annex D - Python code for Modbus slave node simulation

The following code represents the usage of the Pymodbus library, for further information on the library itself[8]; refer to the library documentation and tutorials.[9]

```
#!/usr/bin/env python
'''
Pymodbus Asynchronous Server Examples
--------------------------------------------------------------------------
The asynchronous server is a high performance
implementation using the twisted library as its backend.
This allows it to scale to many thousands of nodes which
can be helpful for testing monitoring software.
#---------------------------------------------------------------------------#
# import needed libraries
```

2 GPIO: General Purpose Input Output. 20 Pins included in the Raspberry Pi for general purpose functions.
3 One Master station communicating with three slaves.
4 http://www.ieee802.org/1/pages/tsn.html
5 http://shop.bsigroup.com/ProductDetail?pid=000000000030319189
6 https://www.bsigroup.com/en-GB/our-services/Cybersecurity-Information-Resilience/Resources/Whitepapers/
7 http://www.enscada.com/a7khg9/IndigoSCADA.html
8 https://github.com/bashwork/pymodbus
9 https://pymodbus.readthedocs.io/en/latest/

```
#------------------------------------------------------------------------#

from iker_async import StartTcpServer,ModbusTcpProtocol

from iker_async import StartUdpServer

from iker_async import StartSerialServer

from pymodbus.device import ModbusDeviceIdentification

from pymodbus.datastore import
ModbusSequentialDataBlock

from pymodbus.datastore import ModbusSlaveContext,
ModbusServerContext

from pymodbus.transaction import ModbusRtuFramer,
ModbusAsciiFramer

#------------------------------------------------------------------------#

# configure the service logging

#------------------------------------------------------------------------#

import logging

logging.basicConfig()

log = logging.getLogger()

log.setLevel(logging.DEBUG)

#------------------------------------------------------------------------#

# initialize your data store

#------------------------------------------------------------------------#

The datastores only respond to the addresses that they
are initialized to. Therefore, if you initialize a DataBlock to
addresses of 0x00 to 0xFF, a request to 0x100 will respond
with an invalid address exception. This is because many
devices exhibit this kind of behavior (but not all)::

#

# block = ModbusSequentialDataBlock(0x00, [0]*0xff)

#

# Continuing, you can choose to use a sequential or a
sparse DataBlock in your data context. The difference is that
the sequential has no gaps in the data while the sparse can.
Once again, there are devices that exhibit both forms of
behaviour::

#

# block = ModbusSparseDataBlock({0x00: 0, 0x05: 1})

# block = ModbusSequentialDataBlock(0x00, [0]*5)

#

#------------------------------------------------------------------------#

store = ModbusSlaveContext(

di = ModbusSequentialDataBlock(1, [1]*5),

co = ModbusSequentialDataBlock(1, [0]*5),

hr = ModbusSequentialDataBlock(1, [7]*5),

ir = ModbusSequentialDataBlock(1, [9]*5))

context = ModbusServerContext(slaves=store, single=True)


#------------------------------------------------------------------------#

# initialize the server information

#------------------------------------------------------------------------#

If you don't set this or any fields, they are defaulted to empty
strings.

#------------------------------------------------------------------------#

identity = ModbusDeviceIdentification()

identity.VendorName = 'Pymodbus'

identity.ProductCode = 'PM'

identity.VendorUrl = 'http://github.com/bashwork/
pymodbus/'

identity.ProductName = 'Pymodbus Server'

identity.ModelName = 'Pymodbus Server'

identity.MajorMinorRevision = '1.0'


#------------------------------------------------------------------------#

# run the server you want

#------------------------------------------------------------------------#

#, console=True

StartTcpServer(context, identity=identity,
address=("localhost", 502), console=False)
```


Call: +44 345 222 1711 / +353 1 210 1711 Email: cyber@bsigroup.com Visit: bsigroup.com

Cybersecurity and Information Resilience services

Our Cybersecurity and Information Resilience services enable organizations to secure information from cyber-threats, strengthening their information governance and in turn assuring resilience, mitigating risk whilst safeguarding them against vulnerabilities in their critical infrastructure.

We can help organizations solve their information challenges through a combination of:

Consulting

Cybersecurity and information resilience strategy, security testing, and specialist support

Training

Specialist training to support personal development

Research

Commercial research and horizon scanning projects

Technical solutions

Managed cloud solutions to support your organization

Our expertise is supported by:

Find out more
Call UK: +44 345 222 1711
Call IE: +353 1 210 1711
Visit: bsigroup.com