Enabling IoT connectivity for Modbus networks by using IoT edge gateways

Ghenadie COROTINSCHI
Ştefan cel Mare University of Suceava
13, Universitătii Street, RO-720229 Suceava
corotinschi_ghenadie@yahoo.com

Vasile Gheorghiţă GĂITAN
Ştefan cel Mare University of Suceava
13, Universitătii Street, RO-720229 Suceava
gaitan@eed.usv.ro

***Abstract*—This paper aims at presenting the implementation of a functional model of IoT gateway. The Gateway aims at extending the connectivity of Modbus devices networks to IoT by performing local data processing. Within the implementation process, we've used modern development technologies, such as .NET Core, and a microservice-based application architecture. The implementation is based on a bus data scanning component, using pre-processed commands for read and write actions. Scanned data is stored locally, and when there is a change in the values, it is transmitted to all interested clients via an MQTT broker. A case study was conducted to select the best broker. This study aims at highlighting the performance of three brokers (ActiveMQ, Mosquitto and HiveMQ) by transmitting a continuous data flow. The IoT gateway was implemented using Raspberry PI devices.**

Keywords—IoT; Edge Computing; Modbus gateway;

I. INTRODUCTION

The Modbus protocol is one of the oldest communication standards. This was developed by Modicon for use in PLC (programmable logic computers) communication. Communication between devices is done using serial lines (RS232 or RS485) and has a master-slave architecture. The network may include only one Master device and up to 247 Slave devices. The Master device has the ability to write information on Slave devices.

In case of both industrial and home automation applications, it is often necessary to use the various data acquisition and handling devices from different hardware manufacturers. Due to simplicity and the low number of resources used, Modbus is an optimal solution for the interaction between the various components of an automation system.

Device interaction within a Modbus network is based on the existence of a Master device. In SCADA (Supervisory control and data acquisition) applications, the data from devices is acquired by reading coils and registers. Data reading is performed periodically by the master device. The frequency of registry values change depends on the type of data stored. Depending on the type of device, some measurements may vary from one read to another. In the case of thermostat devices, measurements such as the status of orders (heating/cooling), the status of the operating mode (summer/winter) vary rarely, while in the case of measurements such as temperature or humidity, their value changes periodically. Regular scanning of these data by SCADA applications involves the execution of periodic, often redundant tasks, leading to the consumption of processing resources and the loading of the network traffic with unnecessary information.

A solution would be to decentralize these tasks towards low processing power local devices having the role to perform some of the local data query and processing actions, and when the acquired data undergoes changes, these would be sent to the SCADA server.

The Modbus protocol is based on a pooling refreshing data process. Many of the current communications protocols have the ability to communicate using the publish-subscribe technique. Many of the existing automation applications use the Modbus protocol, and replacing these devices with other high performance devices to meet the connectivity requirements of IoT applications would be expensive.

The paper is organized as follows: Section II includes the motivation for the research work described in this paper. In section III, we present some of the developing efforts made in developing gateways that want to expand the current industrial networks to the IoT universe. The proposed solution for local data processing and extending the connectivity for the industrial networks is presented in section IV. A case study regarding the performance of the three MQTT Brokers(ActiveMQ, HiveMQ and Mosquitto) was described in section V. In section VI we present the conclusions and the future development directions.

II. MOTIVATION OF THE WORK

The universe of IoT devices is constantly expanding. Gartner Research [1], estimates that around 11.19 billion IoT devices will be connected to the Internet in 2018, that will be more than 33,6% than 2017 and 75.45% more than 2016. This trend will continue arriving at 20.41 billion in 2020.

According to the latest report published by Gartner Research on SaaS(Software as a Service) in October 2017[2], the growth of the SaaS market is about 28% in 2018. Modern industrial devices implement communication protocols that allow cloud connectivity, which implies a quicker and efficient

978-1-5386-1495-2/18/$31.00 ©2018 IEEE

data analysis of the data collected from industrial networks. In case of the older industrial networks exchanging the device with new modern devices is not a solution, mainly because of the high costs.

The current paper aims to study the probability of implementing an IoT gateway that should extend the connectivity on the devices that don't support cloud connectivity. Also, some algorithms of local data processing were implemented to filter relevant data and decreasing network overload, helping in the implementation of the edge computing concept.

III. RELATED WORK

Modern software technologies are evolving quickly mainly of the significant technological of the hardware devices from the last decade. From the technological point of view, we are facing the 4th technological wave called also Industry 4.0. First industrial wave was based on the advancing of the sectors like mechanization, water power, and steam power. The second one was focused on improving the mass production processes, assembly lines and electricity. The 3rd one was based on the evolution of the computer and automation processes. The last wave of the technological evolution, Industry 4.0, defines the actual trend of automation and data exchange between technological processes and includes cyber-physical systems, cognitive computing and Internet of Things.

The evolution of the modern technologies leads to an increased efficiency of all industrial domains. Most of the industrial domains are facing major changes due to the integration of the cloud connectivity systems. By using cloud connectivity solutions, we can improve the performances and reduce the maintenance cost of automation systems. Exposing these systems to the cloud platforms can lead to series of challenges regarding the security of the data and technological processes. In [3] authors are conducting a study over the development of an "Intrusion Detection Systems(IDS)", based on the hardware data package parser for 4 industrial communication protocols(Profinet, OPC UA, Modbus and S7comm). Other efforts on integrating industrial networks into IoT universe were done by the authors [4], they developed an industrial gateway for the S7 and Modbus TCP protocols. Authors from [5], researched the possibility of implementing a smart switch that aims to solve the problem of the devices with the same SlaveID between the Modbus networks, by implementing a software defined network architectures.

IV. PROPOSED SOLUTION

The solution proposed in this paper aims at extending the capacity of processing the data acquired from Modbus devices, but also at extending these networks towards the IoT universe. The purpose of the application is to help implement the FogComputing concept by processing data at the Cloud base. To this purpose, the following modules have been implemented within the application: *Generic object compiler*, *Refreshing data module, Cloud connectivity module* and *Database export utility*. Both the Data Scanner Module and the Cloud Connectivity Module run on a RaspberryPI device. Fig. 1 shows the general architecture of the application by which we wish to extend the connectivity of the modules from Modbus networks. The IoT edge module includes a database with the settings necessary for extending the Modbus network connectivity, and the modules required for connectivity with other IoT clients.

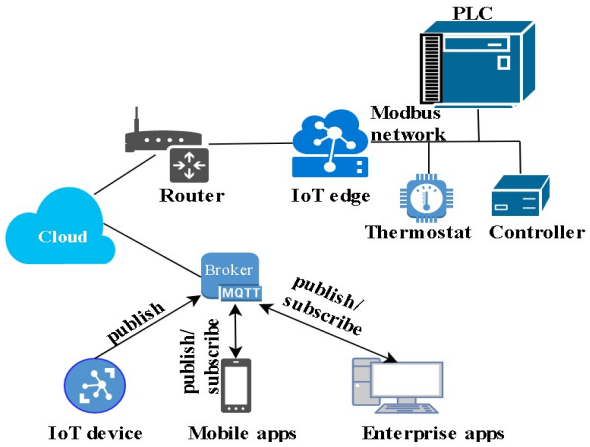

Fig. 1. Application overview

A. *Generic object compiler*

For an efficient management of reading requests, it is necessary to define the types of modules used by the application. For each type of Modbus device, there are defined registers to be read and the type of data stored by these registers. Each Modbus readable measurement is configured by the following tuple *("ItemName", "DeviceID", "ReadAddress", "WriteAddress", "StartAddress", "StartBit", "Length", "DataType").*

When the database is to be exported in order to be used by the scanning module, the items are processed and an internal map of the memory areas that must be read by the device is generated. This preprocessing will generate the following tables: *ReadRequest, ReadAnswer, WriteRequest, WriteAnswer, Objects* and *Devices*. The logic of generating a database is to be able to have locally available all the data needed to scan the devices. SQLite is a process library that implements a serverless, zero-configured and self-contained library. Serverless architecture makes it possible to implement drivers on various systems, such as Windows, Linux or Android. According to [6], SQLite, with all options enabled, can be included in a memory of up to 500KiB.

The database generation algorithm involves creating a unified way of generating requests for reading data and writing values on the bus. Based on the *ResponseLength* field, the scanning algorithm will know the length of the byte string it needs to receive as a response. Fig. 2 shows the database diagram and the relationships between the tables.

The *ReadRequestFormat* field is a string and contains a JSON based on which the data read request from the bus can be generated. As a practical example, to read 3 registers from the 0 address, the 03 function will be used. The format of the data to be transmitted on the bus is the following „0x01 0x03 0x00 0x00 0x00 0x03 0x05 0xCB".

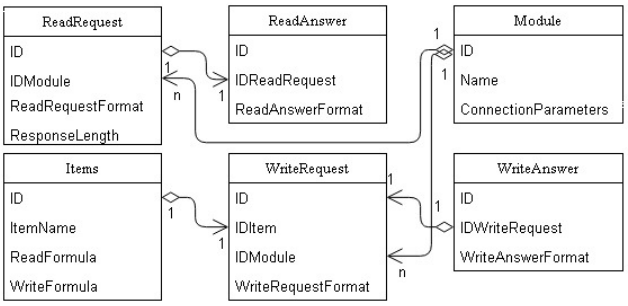

Fig. 2. Database structure

Fig. 3 shows a way of defining a read command using generic objects. This type of object definition is also used for the *ReadAnswerFormat, WriteRequestFormat, WriteAnswerFormat* fields in the *ReadAnswer, WriteRequest,* and *WriteAnswer* tables.

```
{"Obj1":
        {"StartByte":0,"StartPosition":0,"Length":8,
        "DataType":"Byte","ID_Object": -1,"Value":"0x01"},
"Obj2":
        {"StartByte":1,"StartPosition":0,"Length":8,
        "DataType":" Byte","ID_Object": -1,"Value":"0x03"},
        ....
"Obj8":
        {"StartByte":7,"StartPosition":0,"Length":8,
        "DataType":" Byte","ID_Object": -1,"Value":"0xCB"}
}
```

Fig. 3. Data format of the ReadRequestFormat field

Based on the records in the *ReadAnswer* table, we will be able to parse the received byte string and populate the objects with values. The algorithm allows us to generate requests for other types of communication protocols by parsing the *ReadRequestFormat* field. The *StartBit* field indicates the position within the array of bytes to be transmitted on the serial. *StartPosition* indicates the position within the byte selected by *StartByte*.

The start address and type of each object are entered into the Modbus object configuration application. These objects are not always at consecutive addresses, the SQLite database generation algorithm is allowing the generation of data read requests for all consecutive addresses.

B. Refreshing data module

Data scanning is performed based on the records in the *ReadRequest* and *ReadAnswer* tables. The bus data scanning algorithm loads all of the records in the *ReadRequest* table. After the *ReadRequestFormat* field is parsed, an array of bytes will be generated to be sent to the bus, and then the precompiled commands will be sequentially sent to the communication channel identified by the *IDModule* field. The *ResponseLength* field in each record in the *ReadRequest* table is filled in, so that the scanning algorithm sends a byte string and waits for a *ResponseLength* byte length response.

When receiving data, the scanning algorithm receives a byte array on the basis of which it will populate each item in the database. The *ID_Object* field will be filled in with a record ID from the *Items* table, when the *ReadAnswerFormat* field, which is in the JSON format, like the one shown in Fig. 2, is parsed. Each JSON object will describe an acquired measurement that is found in the byte array received from the bus. Fig. 4 illustrates the way to decode the response received and to populate the items with values.

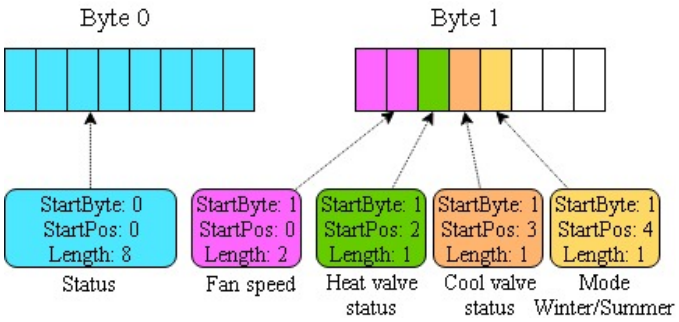

Fig. 4. Decoding of the received data based on ReadAnswerFormat

After parsing the received response, the values acquired for each item go through an additional scaling filter. If the object in the *Items* table has the *ReadFormula* field filled in, then the object will be scaled according to the filled in formula. For example, some manufacturers of thermostats indicate that each unit in the setpoint register corresponds to a value of 0.2°C, so that in order to read correctly the value of the data acquired, this value must be multiplied by 0.2. In this case, the formula for scaling the value upon reading will be „*SET_VALUE *0.2*”, where SET_VALUE is the value acquired from the bus, and the formula for writing on the bus is „*SET_VALUE/0.2*” where SET_VALUE is the value sent for writing. The algorithm looks for the SET_VALUE string and replaces it with the read value, then assesses the result of the resulting formula, and the value of the object is updated in the memory, which is then reported to the cloud connectivity module.

C. Cloud connectivity module

For data connectivity with the IoT universe, we've looked for an optimal solution for connecting the device on which the data refresh module is running. The communication solution must be easy to configure, must use scarce resources, be reliable, efficient, must have QoS (Quality of Services) capabilities and allow clients to be notified through the Publish-Subscribe technique.

Among the protocols analyzed for IoT, the following came into prominence: AMQP - is a queuing system that is mainly used to communicate between servers (S2S); XMPP - is based on XML telegram exchange and is suitable for server-to-server communication (S2S) and MQTT which is a communication protocol (lightweight message transmission protocol) used for devices with limited resources. For our solution we have chosen the MQTT broker. Its choice was due to its capabilities to interact with resource-constrained devices.

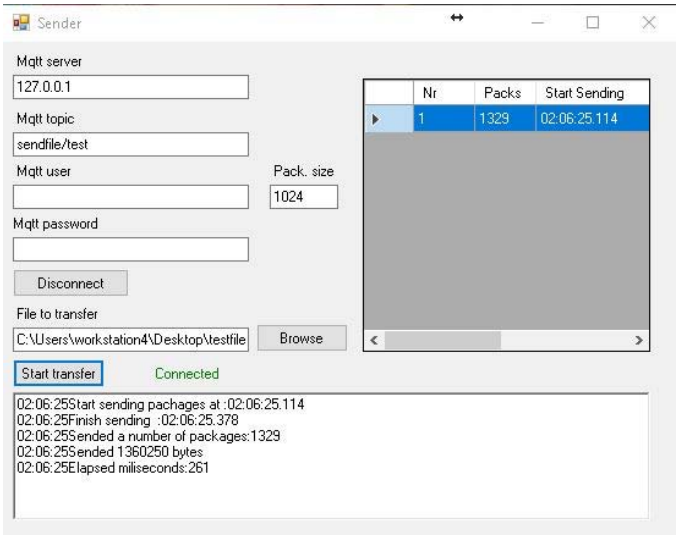

Fig. 5. MQTT Broker Sender client

MQTT is a technology that is based on the existence of a server (broker). There are numerous broker implementations on the market, for various platforms and operating systems. In order to be able to choose the best broker, several tests have been performed to show the performance of each broker. The proposed solution involves the transmission of the SQLite database to the clients, and for this reason a test application, Fig. 5, for transmitting and receiving data via MQTT, has been developed. A Windows 10 operating system, CPU: Intel Core I7, RAM: 32 GB was used for testing.

D. Database export utility

In order to optimize the data scanning application, a local database with the settings of each module type, and of the read/write requests is required. Using a file type database, brings benefits due to portability. This type of database can easily be exported from one client to another.

After generating objects through *Generic Utility Compiler*, read and write objects and requests are created and stored into the database. The export of data to the clients is performed by dividing the database file into packages equal to the configurable size. Each package has a {*FileTimestamp, PackageIndex, NrOfPackages*} header followed by the content of the database and a checksum for verifying the integrity of the data.

MQTT Clients connect to a broker, and then the subscription topics takes place. Each client subscribes to the following topics: *RPIReadValue, RPIWriteValue*, and *RPIDbUpdate*. The *RPIReadValue* topic is used to query the status of a particular item. *RPIWriteValue* is used to change the status of an item. The *RPIDbUpdate* topic is used to initiate a database update procedure. The Database Generation and Export Utility will send the built messages to all connected clients. The messages will be received and stores locally by the client. The *NrOfPackages* parameter indicates the total number of packages to be received. Based on the *PackageIndex* parameter, the exact position where the data will be filled in the rebuilt file is known. When all packages are received, the application restores the database file, after which the device scan procedure is restarted.

V. MQTT BROKERS CASE STUDY

Several broker implementations, including Mosquitto [7],ActiveMQ[8] and HiveMQ [9], have been considered during the testing process. The analysis of the MQTT Broker Performance was based on sending a data file by dividing it into data packages and rebuilding the file at its destination. For an overall analysis, there were performed tests involving 3 different sizes of the data packages that are transmitted, namely 256, 512 and 1024 bytes. Each broker has been configured to allow an unlimited number of inflight transactions. The data send and receive procedure was run 20 times for each broker, and for each of the three sizes of the transmitted data package. Each package was transmitted using QoS2 (Exactly once). The following time intervals were monitored:

- *Total time (TT)* - the amount of time elapsed from the start of the transmission by the Sender client until the Receiver client receives the last package.
- *Time for the arrival of the first package(TAFP)* - the amount of time elapsed since the first package is transmitted by the Sender client until the Receiver client receives the first package.
- *Broker sending time(BST)* – the amount of time between the arrival of the first and the last package.

TABLE I. BROKERS RECEIVE TIME

Average Time (ms)	Mosquitto (Bytes)			ActiveMQ (bytes)			HiveMQ (bytes)		
	256	*512*	*1024*	*256*	*512*	*1024*	*256*	*512*	*1024*
TT	3637	1123	401	3400	919	350	10749	4587	2090
TAFP	426	220	126	1187	341	288	1835	268	475
BST	3210	903	273	2211	578	62	8915	4321	1615

TABLE I. , shows the average of the data acquired for each broker. Test data can be found at [10].

Fig. 6 shows the total reception times of a 1.3MB file, in 1024 byte packages, resulting in a number of 1329 packages. Given that the differences between Mosquitto, ActiveMQ and HiveMQ are quite large, we performed a logarithmic scaling of the acquired data, for a clearer analysis of the data received using the ActiveMQ and Mosquitto brokers.

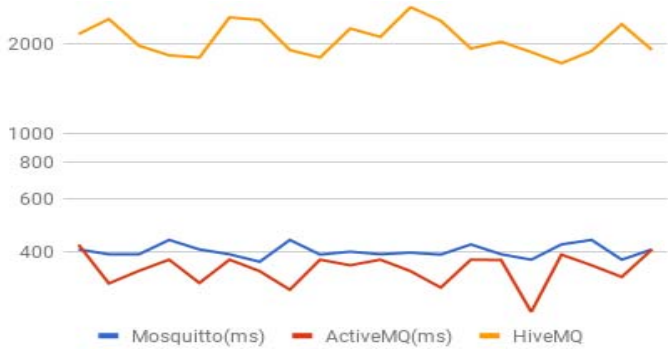

Fig. 6. Test results for TT duration – packages of 1024 bytes

The analysis of Fig. 6 shows that the TT duration of the HiveMQ broker is quite long as compared to the other 2 brokers. However, the analysis of the data provided in Fig. 7 highlights the fact that the TAFP duration of the HiveMQ broker can be compared with the one of the other 2 brokers, but it does not have a constant feature.

Another interesting aspect in analyzing the test data using Mosquitto and ActiveMQ brokers, indicates that although Mosquitto has a shorter transmission time for the first message, having two times lower values, the total transmission time of all packages is longer. Considering this data, we conclude that it is more advantageous to transmit data using Mosquitto when we have a small flow of packages per client and it takes a short time to send the packages, while ActiveMQ is more efficient when we have to deal with a large number of sent data packages per client. ActiveMQ includes a more effective management of each client's message queues.

To implement this project, we used the Mosquitto Broker. For the architecture proposed in this paper, a longer time is acceptable for the database update procedure on the RaspberryPI device, because this operation will be performed upon the installation of the system, and, with few exceptions, when modifying the number and type of Modbus devices. Using Mosquitto, we have obtained shorter times in the data refresh procedure, largely also due to the local data preprocessing algorithms, the remote data transmission taking place only if they have undergone changes.

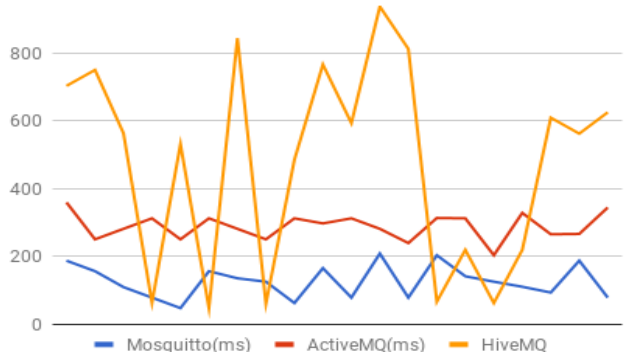

Fig. 7. Test results for TAFP duration – packages of 1024 bytes

VI. CONCLUSIONS

This paper presents an architectural model of an IoT gateway. Its role is to perform local data processing and send remotely the changes of the scanned objects. The Modbus protocol is based on the existence of a Master device that interrogates the status of each item. The role of the master device has been taken over by the RaspberryPI device, which runs a software application that has a microservice-based architecture. For a more effective network traffic, a pre-processed commands database was used. By using pre-processed commands, the reduction of the used bandwidth is avoided and the network traffic is alleviated. The compatibility of the proposed solution with cloud services for the storage and analysis of data, such as Amazon Web Services IoT [11]or Azure IoT [12], is ensured by using a publish-subscribe messaging architecture.

The possibility to implement other communication protocols is ensured by using a generic way of defining objects. After processing the *ReadRequest* tables, a byte string to be transmitted on the bus shall be obtained and the results will be interpreted and the objects will be populated with values by using the *ReadAnswer* table. A case study was conducted using 3 MQTT brokers (ActiveMQ, Mosquitto and HiveMQ), in order to identify an optimal broker.

As further research, it is also desirable to implement other communication protocols using also other data transmission environments, such as Wi-Fi or ZigBee.

ACKNOWLEDGMENT

This project benefited from technical support through the „Operational Program for Competitiveness 2014-2020, Axis 1", Contract no. 39-347, dated 9/27/2016 „BRAIN-IN - Intelligent Automation Systems for Building Management, Production and Industrial Automation".

REFERENCES

[1] Forecast Analysis: Internet of Things — Endpoints, Worldwide, 2017 Update, https://www.gartner.com/doc/3841268/forecast-analysis-internet-things-

[2] Market Share Analysis: Enterprise Application Software as a Service, Worldwide, 2016, https://www.gartner.com/doc/3810899/market-share-analysis-enterprise-application

[3] S. Bhardwaj, P. Larbig, R. Khondoker and K. Bayarou, "Survey of domain specific languages to build packet parsers for industrial protocols," 2017 20th International Conference of Computer and Information Technology (ICCIT), Dhaka, 2017, pp. 1-6

[4] M. Hemmatpour, M. Ghazivakili, B. Montrucchio and M. Rebaudengo, "DIIG: A Distributed Industrial IoT Gateway," 2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC), Turin, 2017, pp. 755-759.

[5] J. C. Chiu, A. T. Liu and C. C. Liao, "Design the DNS-Like Smart Switch for Heterogeneous Network Base on SDN Architecture," 2016 International Computer Symposium (ICS), Chiayi, 2016, pp. 187-191

[6] SQLite secifications , https://www.sqlite.org

[7] Mosquitto broker, https://mosquitto.org/

[8] ActiveMQ broker, http://activemq.apache.org/

[9] HiveMQ broker, https://www.hivemq.com/

[10] Brokers testing results , https://docs.google.com/spreadsheets/d/1mt5a3i6m-EZwqvIQ4EOaS6gM3fq-uNvczATaQM-jtAE/edit?usp=sharing

[11] Amazon Web Services IoT , https://aws.amazon.com/iot/

[12] Azure IoT hub, https://azure.microsoft.com/en-us/services/iot-hub/