Analecta
REVIEW OF FACULTY OF ENGINEERING
Analecta Technica Szegedinensia
ISSN 2064-7964

Vol. 10, No. 1 2016

DESIGN AND IMPLEMENTATION OF A DISTRIBUTED SOFTWARE ARCHITECTURE FOR DATA ACQUISTION SYSTEMS

Á. Papp
Technical Institute, Faculty of Engineering, University of Szeged, Hungary, Moszkvai krt. 5-7, 6725, Szeged, Hungary,
e-mail: papparon@mk.u-szeged.hu

ABSTRACT

This paper will demonstrate the advantages of using a distributed software design for collecting environmental data. The design part will focus on achieving modularity through message queuing technologies. The essential components of the software implementation will be detailed. Finally, the testing results will be disclosed.

Keywords: data acquisition, message queuing, ZeroMQ, modular software architecture, inter-process communication

1. INTRODUCTION

The focus will be on the design of a software structure intended to use for transferring measured data between data collection systems and data acquisition devices. The main directive was to create a highly scalable and easily customable framework, including an encryption mechanism to grant security for data transfers. After reviewing literature [1], [2], [3] and [4] the following statement is true: two common software architecture styles are in use these days, layered and modular architecture. Layered design is driven by classifying solutions based on technical function, causing layers of increasing size to build on each other – this can be modelled with an inverted pyramid. In counterpart, using modularity in software design implements the theory “divide and conquer” by breaking a problem down to smaller, manageable, and in most cases independent modules. The choice was made on modular design, since it is particularly suitable for creating scalable architectures, and achieving high-availability trough redundancy.

2. MODULAR SOFTWARE DESIGN

Remote clients can be embedded systems or computers running POSIX compliant operating systems. To extend connectivity, remote devices can be used as a gateway to other not natively supported devices implementing standard digital communication interfaces, like RS-232, RS-485, CAN bus, IO-Link and more.

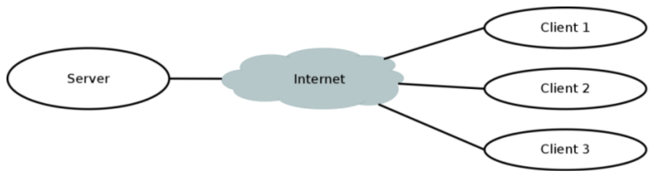


Figure 1. Main architecture topology

In the developed application the TCP layer was used as shown on Fig. 1. The Client ellipses are the remote sensor devices. The server application processes and stores the data received, Fig. 2 is intended to summarize the server-side operations. In details, the following tasks are done by the server application:

a) Listening on a TCP socket for receiving messages from the clients. The server’s socket is intended to be available from Internet side, since it is likely that the remote devices have limited network connectivity because of the use of NAT or firewall. By keeping the server’s socket open, the clients can initiate the connection, and send their requests. The server will send its reply on the established TCP connection.

b) Encryption and decryption of the messages. Since the socket is unprotected, data encryption is essential to keep low the likelihood of making successful hacker attacks. Elliptic curve cryptography was chosen as encryption method, because of its robustness and efficiency [5].
c) Running validation algorithms to avoid the further processing of invalid data segments.
d) Connection to a Data Base Management System (DBMS), to store the received measurement data. The two main concepts, SQL and NoSQL DBMS systems were compared in the context of the current application, and the decision was made on PostgreSQL, but the possibility was kept open to implement a database connection module for a desired DBMS, without the need of changing, or refactoring the non-database related modules. Supporting NoSQL based approaches can have also many advantages, especially, when handling a greater amount of remote devices, or using higher sampling frequency.
e) Notification service, to inform administrators or users about operation states, and error cases. E-mail and SMS messages are currently supported.

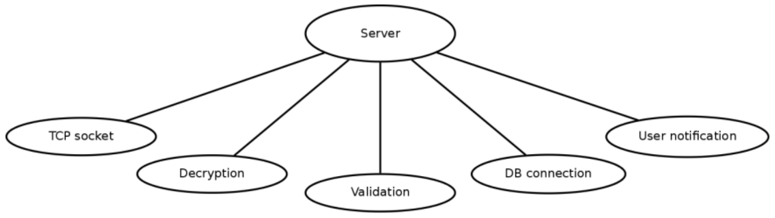


Figure 2. Server-side operations

Client side operations include the following:

a) Data acquisition. Different types of interfaces are supported, including ampere and voltage level measurements, RS-232 point-to-point communication, MODBUS protocol on RS-485/422 bus, Dallas 1-Wire bus-system and the list can be further expanded, by implementing further modules.
b) Post production creates ready-to-send messages from the acquired data.
c) Message encryption and decryption algorithms are making the client able to securely communicate to the server.
d) Connection to the TCP socket running at the server side. Sending collected, encrypted data packages, and receiving the server's commands or acknowledgements of the transmission success.

Fig. 3 is intended to visualize client-side operations.

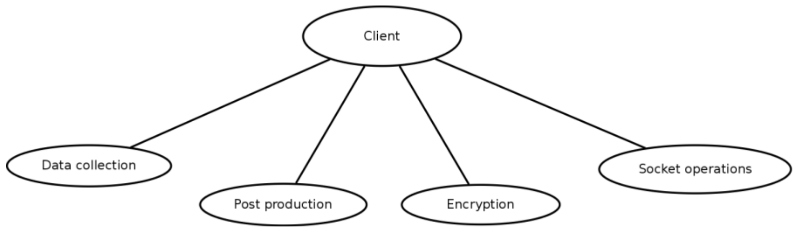


Figure 3. Client-side operations

The listed server- and client-side functions are all realized in software modules. The modules are logically on the same level; none of them can block another or use the CPU-time at the expense of other modules (unless it is desired). One of the most important reasons to choose modular architecture was to make the application's different parts work independently of the others. In case of partial hardware error this feature allows the system to remain alive, with exception of the failing components.

3. MESSAGE QUEUING

According section 1, the application describes many communication contexts. From the furthest point of view, there are two actors: the server-side and the client-side application. All the clients need to communicate with the server in a synchronized, full duplex way, to send collected data and to receive the server's acknowledgement or administrative commands.

From the application context, the actors are the modules which the application is built up. The existence of a communication channel between the actors are elementary, otherwise there would not be any interface to affect the operation of the individual modules.

The phase message queuing summaries the whole process of message transceiving: sending, receiving, storing and error handling. The message queuing solution, which have interfaces to multiple protocols or multiple transport layers is often called Message-oriented middleware, because as a router it stands in the middle and translates messages between different systems or architectures [6], [7].

The presented system is using the ZeroMQ open source software libraries to handle messaging related tasks at both contexts: between the two applications (server- and client-side), and between the independent software modules (the application components). There are two main differences between the transmission methods. The inter-application messages are transmitted on the TCP/IP network, while the inter-process messages are transmitted using a special socket type, developed to link a pair of processes. The second difference is that the inter-application messages are transmitted without encryption.

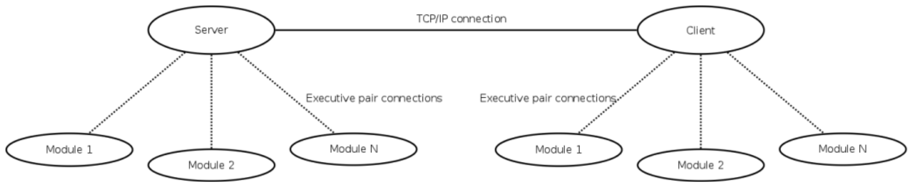


Figure 4. Inter-application and inter-process communication channels

Fig. 4 shows a simplified topology of the communication channels used by the system. The ZeroMQ libraries are designed to create message-transmitting sockets between different systems. Without using ZeroMQ libraries the programmer must handle many special cases, e.g. the fail of the physical layer, and needs to use locks, semaphores and other wait states to avoid the access of the system memory in a concurrent way [5].

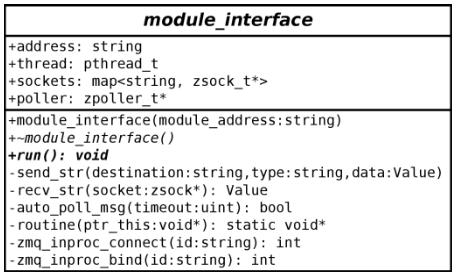
3. IMPLEMENTATION

3.1 Interface classes

Both the server- and the client-side have the same object-oriented software structure. An abstract class – called *module interface* – was defined to be the parent class of all the modules. The interface implements the functions which are necessary for the modules to have them started properly and prepares communication channels to the other modules.

The only task to do by the parent class' constructor is to create a new thread. The thread-creation process consists a few steps – after the creation of the thread itself, a pointer is set to point to the new thread, and then this pointer will be used to call the child class' run method. At this point the module is started properly as an independent thread of execution.

The modules are inheriting many tools and utilities from the interface class, nevertheless this paper will focus on the inter-process communication. UML diagrams, as a standard software-class visualizing tool will be used to represent the base abstract classes [8], [9]. Fig. 5 shows the class diagram of the module interface's C++11 implementation, used at the client-side. The analysis of Fig. 5 will not follow the same order, as shown on the diagram; instead logical order will have a preference.



```
module_interface
+address: string
+thread: pthread_t
+sockets: map<string, zsock_t*>
+poller: zpoller_t*
+module_interface(module_address:string)
+~module_interface()
+run(): void
-send_str(destination:string,type:string,data:Value)
-recv_str(socket:zsock*): Value
-auto_poll_msg(timeout:uint): bool
-routine(ptr_this:void*): static void*
-zmq_inproc_connect(id:string): int
-zmq_inproc_bind(id:string): int
```

Figure 5. Partial class diagram of module_interface (C++11)

The modules are inheriting the *address* parameter, which will be used to identify the module and to refer to a module instance from other modules. This name must be unique in the application context. The *thread* attribute is the pointer to the class itself, which gets its value after starting the new thread. The *routine* is an intermediate function – it can be called by the new thread creation function and is able to call the child's run function, after the thread creation is done. The *run* function is an abstract method, must be implemented in every module.

The *sockets* is a map object, a container for inter-process sockets. The sockets are prepared for point-to-point communication. To have a communication link between two modules, one of them need to bind and the other one needs to connect to the same channel. This function is implemented through the private member functions *zmq_inproc_bind* and *zmq_inproc_connect*. After having both end of the channel initiated, a full duplex, asynchronous communication channel is ready to use.

The *poller* is also a special ZeroMQ object, which is used to detect incoming data, when multiple sockets have been defined. When the *poller* object is properly initiated, the function *auto_poll_msg* can be used to receive data from either socket defined.

Finally, two remaining functions will be detailed. *send_str* can be used to send a message to another module. The parameters need to include the *address* of the destination module, the *type* of the message, both of them represented by a string, and the *data* itself. The data's type is Value, which is a JsonCpp object type, containing structured data in JavaScript Object Notation form. *recv_str* can be used to receive messages, but it is implicated by the function *auto_poll_msg*.

Analecta
REVIEW OF FACULTY OF ENGINEERING
Analecta Technica Szegedinensia
ISSN 2064-7964

Vol. 10, No. 1 2016

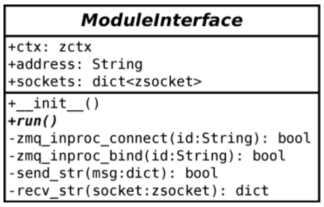

ModuleInterface
+ctx: zctx
+address: String
+sockets: dict<zsocket>
+__init__()
+run()
-zmq_inproc_connect(id:String): bool
-zmq_inproc_bind(id:String): bool
-send_str(msg:dict): bool
-recv_str(socket:zsocket): dict

Figure 6. Partial class diagram of ModuleInterface (Python)

Server-side is implemented using Python language version 2.7, Fig. 6 shows the class diagram for the interface class. The paper will focus on the creation of the new module-thread and on the communication functions – other tools and utilities will not be explained.

The *ctx* attribute is needed by the ZeroMQ Python implementation, the pyczmq package. All the software components are getting a copy of this reference.

The *address* attribute has the similar purpose as detailed at the C++ implementation; its purpose is the identification of the module instance. The *sockets* attribute is a container, a Python dict object, which contains references to ZeroMQ sockets. The *__init__* function is the constructor of the abstract class. The only task done by the constructor is to create a new thread, and call the run function of the new instance. The *run* method is abstract, must be implemented in every class inherited from the ModuleInterface.

Functions *zmq_inproc_connect* and *zmq_inproc_bind* have the same purpose as before – the creation of the communication channels between the modules.

send_str is used to send messages on the default socket, which will be explained in the next paragraph. The only parameter is a Python dictionary object, and has a *source*, a *destination*, a *type* and a *data* field. Python dictionary objects have the same structure as a JSON object, so they are easy to translate in both directions. Function *recv_str* can be used to receive messages on the socket specified as a function parameter.

3.2. Inter-process communication

As mentioned before, the modules are using full duplex, asynchronous channels to communicate. Each channel can link two modules. During the normal operation it is sometimes necessary to make a module be able to communicate with more than one other module. One way to solve this problem would be to define a channel between all the modules. Connecting all the modules to each other would result in N(N-1)/2 channels, where N is the number of the modules defined. This would be the most effective solution having regard to transfer rate, but it also makes impossible to have an insight into the communication – e.g. for administration purposes. It would be also necessary to explicitly define the socket at every transmission. Despite the higher resource requirements, a special module, the *core* was defined. The *core* module acts like a router, it is intended to solve the problem of interconnecting the modules and to make possible the inspection of the whole communication. It has a socket to all modules, initiated with the *zmq_inproc_bind* function, so that every module during its start-up can immediately connect to the *core* and send or receive messages.

When the *core* receives a message it checks the header, especially the *source* and the *destination* field. After validating the *destination* field, the core forwards the message to the destination module. At this time, the *type* field has a "request_" prefix. The destination module interchanges the *destination* and the *source* fields in the message and changes the *type*'s prefix to “reply_”. After setting the *data* field, the destination module sends the message through the *core* module back to the originating sender module.

3.3. Inter-application communication

In this paragraph the communication between the client- and server-side applications will be discussed. The communication was implemented using the ZeroMQ libraries, but this time TCP/IP was chosen for the transport layer. The connection is always initiated by a client module. The module sends a message to its *tcp_client* module, which forwards the message on its special socket to the LAN or WAN. After the *tcp_server* module at the server application receives the message, it forwards to the destination module, and the destination module processes the request. The destination module on the server side replies immediately to the client on the established TCP connection, so that the *tcp_client* module can receive the reply. The *tcp_client* module then forwards the messages to the originating module through the *core*.

Since the communication takes place on a non-secured layer, data security must be ensured by cryptographically encoding every message sent over the TCP/IP socket. Elliptic curve encryption method (ECC) was an optimal choice to use for this purpose, because it provides a cost-effective algorithm for data encryption compared to the more common RSA coding.

4. RESULTS

At first, the hardware used for the testing will be detailed. A notebook with Intel Core i5 M560 processor, 8 GB DDR3 memory and a Samsung EVO840 SSD was the first configuration. The second hardware was equipped with Intel Core i5 4210U processor, 4 GB of DDR3 RAM, and a normal SATA HDD. The two notebooks were connected directly using Gigabit Ethernet interfaces, and both PC’s were running Debian 7 operation system.

The testing method consisted running one instance of the server application and up to 50 client instances. All the clients were sending the same 1.4 Kbyte size message to the server instance with 1 second interval. The messages were containing the acquired data from 4 different probes. The stability testing of the transport layer was out of our scope.

Every message sent to the server made the following tasks to be performed at the server-side:

1. Receiving the data on the TCP/IP socket.
2. Decoding the received message.
3. Sending back a positive acknowledgement to the client using encryption.
4. Pushing the 4 item to the input buffer.
5. Pushing the 4 item to 4 different database tables.
6. Downloading data checking parameters from the database.
7. Checking values based on the downloaded policy.
8. Checking too old items in the input buffer, and cleaning unwanted items.

Test results:

a) At the first case both, the server- and the client-side application was running on the first hardware configuration. The system had a message throughput about 38 messages per second.
b) 15% increase of the throughput performance could have been reached, by running server-side and client-side applications on different hardware: client application instances were moved to the other hardware. This time the highest message throughput was 44 messages in every second.

5. SUMMARY

After analyzing many data acquisition related documents ([10], [11], [12], [13], [14]), a demand on creating a multi-purpose framework could be formulated. Therefore, from the business logic approach, the main purpose was to create a system, which can collect data from a wide range of standard interfaces, and aggregate the information into databases easily. The possibility to make fine-tuning and extension of the system simple played also an important role.

The presented solution is able to perform its tasks besides enforcing the rules of modularity. Doing modifications on behalf of optimization is likely to improve the processing speed of incoming messages. The most urgent point of improvement is to achieve greater scalability, by running the server application on more physical hardware simultaneously. Redundancy test cases can be developed thereafter. Another direction of further development is creating intervening modules to extend data acquisition ability with regulatory and controlling features. In the distant future the potential to make the code open-sourced can be seen.

REFERENCES

[1] L. Bass, P. Clements, R. Kazman, Software Architecture in Practice; Addison-Wesley, 2012, 624 p.
[2] J. Bosch, M. Gentleman, C. Hofmeister, J. Kuusela, Software Architecture: System Design, Development and Maintenance; Springer, 2013, 238 p.
[3] F. Buschmann, K. Henney, D. C. Schmidt, Pattern-Oriented Software Architecture, On Patterns and Pattern Languages; Wiley, 2007, 490 p.
[4] A. T. Stephen, The Art of Software Architecture: Design Methods and Techniques; John Wiley & Sons, 2003, 336 p.
[5] ZeroMQ: Messaging for Many Applications By Pieter Hintjens, O'Reilly Media, Final Release, 2013, 516 p.
[6] L. Han, J. Guofei, Semantic Message Oriented Middleware For Publish/Subscribe Networks, Hanover, 2004, 10 p.
[7] M. Qusay, Middleware for Communications, John Wiley & Sons, 2005, 522 p.
[8] P. Harmon, M. Watson, Understanding UML: The Developer's Guide: with a Web-based Application in Java, Morgan Kaufmann, 1998, 367 p.
[9] M. Page-Jones, L. L. Constantine, Fundamentals of Object-oriented Design in UML; Addison-Wesley Professional, 2000, 458 p.
[10] G. Martinović, J. Simon, Greenhouse Microclimatic Environment Controlled by a Mobile Measuring Station, Journal of the Royal Netherlands Society for Agricultural Sciences, 70 (1), (2014), pp. 61-70
[11] J. Simon, Optimal Microclimatic Control Strategy Using Wireless Sensor Network and Mobile Robot, Acta Agriculturae Serbica 18 (36), (2013), pp. 3-12
[12] J. Sárosi, Mérési adatok gyűjtése és tárolása, International Conference on Science and Technique in the Agri-food Business (ICoSTAF 2008), Szeged, November 5-6 2008, pp. 1-6.
[13] J. Sárosi, Elimination of the Hysteresis Effect of PAM Actuator: Modelling and Experimental Studies, Technical Gazette, 22 (6), (2015), pp. 1489-1494.
[14] J. Sárosi, Accurate Positioning of Pneumatic Artificial Muscle at Different Temperatures Using LabVIEW Based Sliding Mode Controller, 9th IEEE International Symposium on Applied Computational Intelligence and Informat-ics (SACI 2014), Timisoara, Romania, May 15-17 2014, pp. 85-89.