ISSN 2079-3316 PROGRAM SYSTEMS: THEORY AND APPLICATIONS no.4(35), 2017, pp. 285–303

Y. V. Shevchuk, E. V. Shevchuk, A. Y. Ponomarev, I. A. Vogt,
A. V. Elistratov, A. Y. Vakhrin, R. E. Yarovicyn

Etherbox: a protocol for modular sensor networks

Abstract. Etherbox is an application level protocol for sensor networks that achieves the flexibility needed for modular sensor networks by representing all commands and data transferred in the form of virtual machine bytecode. We introduce the Etherbox protocol and compare it with MQTT-SN and CoAP, then consider the software architecture of sensor nodes and the controlling computer.

Key words and phrases: sensor network, IoT, Etherbox, MQTT, MQTT-SN, CoAP.

Introduction

We treat a sensor network based on TCP/IP as shown on Fig. 1. The network is an interconnected system of *sensor nodes* with a number of sensors and actuators connected to every node. The system gives Internet users access to sensor data: raw or pre-processed, live or archived. The users can also remotely control the actuators, manually or automatically.

The sensor nodes use protocols based on TCP/IP and so could in principle be connected to the Internet directly — by a 1 . . . 3 level gateway, with no application level protocol change. In today's practice though it is common to use an application level gateway which uses one application level protocol to talk to sensor nodes and another application level protocol to talk to the Internet. Here are the reasons for protocol separation:

- sensor nodes working in pulse mode (activity—sleeping) cannot present themselves as servers on the Internet. They can send data in client mode to a server on their own schedule. The application level gateway could act as a dual server, for sensor nodes and for Internet users;
- low sensor network bandwidth requires terse protocols;
- low sensor network bandwidth makes the network vulnerable not only to targetted DoS attacks, but even to ordinary network noise;

DOI: 10.25209/2079-3316-2017-8-4-285-303

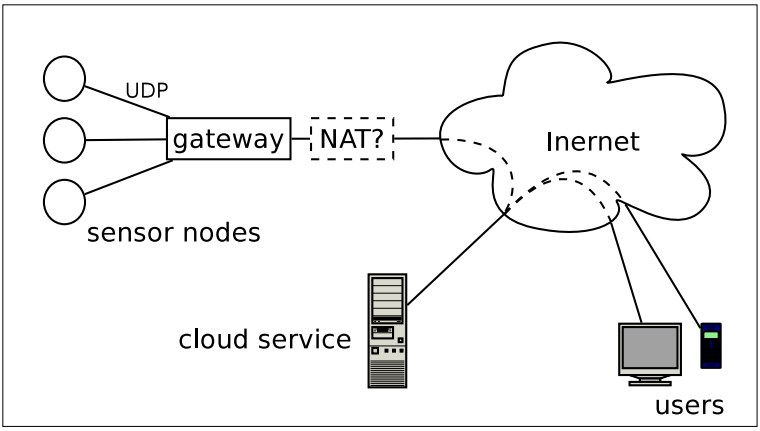


FIGURE 1. Sensor network accessible via the Internet

- mismatch of Internet protocol versions: sensor networks utilize IPv6 for its wider address space and autoconfiguration, while today's Internet access is most often IPv4 — and with NAT [16], not unlikely;
- UDP-based protocols are most suitable for sensor nodes, but NAT support for them may be poor (short UDP flow lifetime in busy NATs);
- TCP-based protocols work well with NAT and are generally good for Internet access to services, but are not so good for sensor networks (see section 2, page 288);
- to hook sensor networks to existing SNMP-based monitoring systems one needs protocol separation and a gateway as SNMP is anything but a terse protocol. People do work on gatewaying between SNMP and sensor network protocols [22].

That said, using IP-based protocols in sensor networks still makes sense, as it facilitates mixing several network technologies in one sensor network.

The Etherbox protocol is a sensor network protocol for use between sensor nodes and a gateway. We consider it in comparison it with two well-known protocols targetted to the same niche, MQTT-SN [1] and CoAP [3]. Then we consider the software architecture of sensor nodes and of the gateway. When talking of Etherbox protocol we call the gateway a *controlling station of the sensor network* as it not only servers

as a gateway between two distinct application layer protocols, but also *organizes* [8] the sensor network[1].

1. Modular sensor nodes

The Etherbox protocol is primarily intended for sensor networks with modular nodes [9]. Modular sensor nodes are assembled from a limited set of ready modules, making a node with the number and types of interfaces fitting the node's duty. The modules are interconnected with I^2C [7] and RS-485 [12] extension buses[2]. The modules fall into two classes: *peripheral* and *base* modules.

Peripheral modules feature one or more interfaces for sensors or actuators. A sensor node can contain one to several dozens[3] peripheral modules. Peripheral module software polls sensors or sets actuator control signals, optionally corrects sensor and actuator data based on calibration parameters, and acts as a slave on an extension bus. This functionality is usually implemented with a low-end microcontroller: 8-bit CPU, 128 byte data RAM, 8 Kbyte instruction flash.

Base modules have one or more sensor network interfaces (Ethernet, IEEE 802.15.4, Bluetooth LE, WiFi, ...). There is exactly one[4] base module per sensor node. Base module software implements network stack, acts as a master on extension buses of the node and its duty is to pass data between peripheral modules and the sensor network, both ways. Base modules employ miscrocontrollers with 32-bit CPU, 32 Kbyte data RAM, 128 Kbyte instruction flash, or larger.

Coalescing modules using a bus and dynamic bus address assignment implemented in software makes the process of sensor node assembly free of conflicts[5], even when installing multiple modules of the same type. This way, sensor node assembly or adding more modules presents no problems hardware-wise. But once the module is assembled, one

[1]The controlling station functionality can optionally be implemented in the cloud. The gateway functions are then reduced to tunneling packets between the sensor network and the controlling station.

[2]Other bus types can also be used, for example CANBus of Bluetooth LE.

[3]The limit on the number of peripheral modules is set by bus circuitry and bus address length. As an example, modern RS-485 transceivers allow for up to 256 devices per bus segment.

[4]It is also possible to use more than one base module per sensor node to improve reliability; at any moment exactly one of the modules controls the extension bus, others are in hot reserve.

[5]to compare with, in modular Arduino systems one has to carefully coordinate I/O signals used by the modules ("shields") being installed [13].

has to support all the modules installed, taking into account the types of sensors or actuators connected to them. In particular, for every sensor we need a polling scenario which is determined not only by the sensor type, but also by its purpose in the specific system.

As the experience shows, having to pre-configure every sensor node to its specific functions in the system before installation greatly complicates the installation process. It is much more convenient for a human worker to install modules with universal software and connect sensors to them in the order he finds the most convenient on the ground. The worker will have to document the resulting configuration: serial numbers of the modules installed and which sensor or actuator is connected to which module. Most often a smartphone photo or video can go for documentation. Later the documentation can be used to configure the node's software remotely.

There is an approach to remote sensor node configuration [10] that employs *proglets* — small programs in the form of bytecode of a specialized virtual machine Etherbox32vm. The Etherbox protocol is a means of delivering proglets from the controlling station to sensor nodes, as well as delivering the results of proglet execution from sensor nodes back to the controlling station.

2. The Etherbox protocol

Interaction with a sensor node using Etherbox protocol vaguely resembles interaction with a remote host using the remote shell protocol (rsh [11]). In remote shell mode every data portion sent by the user contains one or more commands that are executed by a shell interpreter on the remote host. Execution results are returned to the user. Commands can include iteration operators to create permanently functioning scenarios at the remote host. Etherbox protocol implements the same idea adapted to sensor networks and modular structure of sensor node.

Like the other two sensor network protocols we consider (MQTT-SN and CoAP[6].), Etherbox is implemented on top of UDP protocol [5]. The choice is motivated as follows:

- UDP is much easier to implement in constrained devices than TCP;
- TCP is strictly point-to-point, while UDP can be used with multicast addressing to save sensor network traffic;

[6]CoAP is UDP-based, but currently a TCP-based version is also under development [23], motivated by use cases where UDP transmission is hindered by firewalls or NAT

- TCP guarantees to deliver all data sent by restransmitting lost packets, but in sensor networks to deliver fresh data quickly is sometimes more important than to have no data loss;
- when sensor nodes work in pulse mode (activity—sleeping) the network RTT varies drastically — on the order of seconds, which can cause needless packet retransmissions when using TCP [24].

As sensor nodes are constrained devices, the commands should be in a form that is easy to interpret on the node. When using the remote shell protocol, the commands are text strings which require parsing before interpretation. With Etherbox protocol, the commands come in the form of small programs in Etherbox32vm [10] bytecode (proglets), which can be interpreted with no preparation.

Proglet execution results are returned to the controlling station in the form of memory dump of the proglet after execution. Returning result in this form has the benefit of reducing sensor node efforts needed to build the result to the minimum. To reduce the traffic volume the proglet can choose to send not the whole memory image but only a fragment containing the data that really need to be sent.

2.1. Messaging patterns

The natural messaging pattern for the Etherbox protocol is "request-reply", where a sensor node is a server, the controlling station is a client. The request packet contains a proglet that is executed by Etherbox32vm virtual machine on the sensor node. When the proglet finishes execution, normally of abnormally, the memory dump of the whole proglet is returned to the controlling station in a reply packet. Using this messaging pattern one can, for example, retrive data from one or more sensors connected to the sensor node.

There are commands in Etherbox32vm that can be used by a proglet to use other messaging patterns.

The "request—no reply" messaging pattern is implemented with `bif` command which suppresses the automatic reply upon proglet termination. This messaging pattern is useful for multicast requests that are processed by many nodes simultaneously: it avoids the chorus of replies which can easily cause network overload. If, however, the replies are desired, one can still use the "request-reply" pattern but add an `mdelay` command with random argument to spread the replies in time and thus avoid the overload[7].

[7] CoAP has a similar random delay facility at the protocol level: Leisure period [3].

Table 1. Request packet structure

Size (bytes)	Name	Description
20/40	iphdr	IPv4/IPv6 header
8	udphdr	UDP header
16	hmac	message integrity check
4	sec	timestamp (seconds)
3	usec	timestamp (microseconds)
1	handle	reply handler index
0 . . . 1428	proglet	Etherbox32vm bytecode

Table 2. Reply packet structure

Size (bytes)	Name	Description
20/40	iphdr	IPv4/IPv6 header
8	udphdr	UDP header
8	eui	unique sensor node ID (EUI-64)
16	hmac	message integrity check
4	sec	timestamp (seconds)
3	usec	timestamp (microseconds)
1	handle	reply handler index
1	exitcode	proglet termination status
2	lastaddr	proglet termination address
1	pad	padding – reserved field
0 . . . 1416	image	proglet memory dump (whole or fragment)

The "request—multiple replies" messaging pattern is implemented with `send` command which triggers reply before proglet termination. It the proglet is large, the `send3` command can be used to send a fragment of the proglet's memory instead of the whole. The "request—multiple replies" messaging pattern is similar to PUBLISH message in MQTT protocol and also has a direct counterpart in CoAP: the *observe* [4] option which tells the node to repeat the reply as soon as the requested parameter value is changed. Etherbox protocol can do more in this respect: the notion of "parameter value change" can be defined algorithmically in the proglet. For example, we can consider it a state change if a temperature changed by 10 ADC units at least, and not before 10 seconds have passed after the last state change.

2.2. Packet structure

The structure of Etherbox protocol packets is shown in Tables 1, 2.

The `hmac` field exists to check packet integrity and authenticity [14].

Fields `sec` and `usec` are a timestamp (seconds since UN*X epoch) of when the packet was created. One purpose of the timestamp is to counter replay attacks: a sensor node will reject the packet unless the following condition is true:

$$(sec_i, usec_i) > (sec_{i-1}, usec_{i-1}), \tag{1}$$

where i — ordinal number of packets received from the same source. If packets happen to be reordered in transfer (UDP gives no guarantees as to preserving packet order), this condition will cause packet drop. We do nothing to prevent the packet drop; the packet may be resent by an application level retry logic.

The second purpose of the timestamp is to synchronize node's clock to wall clock. The algorithm used for synchronization is very basic and unlike NTP [15] does not take into account packet propagation delay and its variation, but guarantees monotonicity of the node's clock. Monotonicity is important as the clock is used to fill `sec, usec` fields in reply packets which will in turn undergo the condition 1 (page 291) check when the reply is received by the controlling station.

The `handle` field chains request and reply packets so the controlling station can pick the correct reply handler in case there is more than one proglet active on the sensor node. If we were using the "remote shell" protocol, to run more than one concurrent processes on the node we would have to open a separate connection for every process, or use a single connection and add information for demultiplexing the process output at application level. For modular sensor nodes running several resident proglets per node is a usual practice (section 3.4) so it makes sense to add the information for demultiplexing (the reply handler index) to application layer protocol header.

The `proglet` field contains ready to run proglet bytecode, including instructions and initialized data.

The `eui` field is a globally unique identification of the node (EUI-64 [20]) which makes it possible for controlling station to identify the sender node even if the source IP address is link-local [17], or obtained with DHCP [18], or changed by NAT [16].

The `exitcode` field shows if the proglet has finished with an error. Zero code tells the reply has been sent upon successful termination of the proglet, or before proglet termination with a `send` command. Non-zero code points at abnormal proglet termination, and in this case the `lastaddr` field contains the address of the virtual machine instruction that caused the error.

The `image` field is the proglet memory dump, whole or a fragment. The schema of the memory dump is known to the reply handler at the controlling station which is chosen by the `handle` field.

2.3. Initial configration of sensor nodes

Initial configuration is what happens after a sensor node is powered on, involving network connection setup and entering the routine data exchange state. Initial configuration procedure is often left beyond protocol specifications. We will consider the procedure for a network using the Etherbox protocol, and also for MQTT-SN [1] and CoAP [3] protocols, as far as we can see it from published specifications.

With MQTT-SN protocol, a node is logically a client of a remote MQTT broker; after initialization it spends time by publishing and receiving messages associated with *topics* known to other broker's clients. The node begins by determining the address of the MQTT-SN gateway. The address may be pre-configured in non-volatile memory of the node, or discovered dynamically with `ADVERTISE`, `SEARCHGW`, and `GWINFO` messages. Then the node uses `REGISTER` messages to register names of the topics it is going to publish, and `SUBSCRIBE` messages to subscribe to the topics it wishes to receive. Finally, the node enters the routine mode where it regularly publishes data with `PUBLISH` messages and handles `PUBLISH` messages relayed by the broker from other clients. The mapping of topic names to sensors and actuators connected to the node, as well as the publication schedule need to be pre-configured in non-volatile memory of the node.

With CoAP protocol a node can ether be a server, passively waiting for connections from the gateway or other nodes, or a client, originating messages to the gateway or other nodes. The protocol includes discovery messages that can be sent to multicast address to discover the neighboring nodes and resources provided by them using the standard URI `coap://host/.well-known/core`. A server CoAP node, once started, is ready to handle requests from anyone knowing its resource URI. A client CoAP node can send requests to the gateway or neighboring nodes by pre-configured addresses and scenarios, or use the discovery procedure and then send requests to discovered neighbors for discovered resources, but still by pre-configured scenario.

With Etherbox protocol a node after power-up acts as a server, waiting requests from the controlling station — ready to receive packets with proglets and execute them. The node is unaware about the sensors and actuators connected to itself, although it can obtain the list of peripheral

modules connected by scanning the extensions buses. The node can obtain an IP address with DHCP, or start with only a link-local address [17], [21], but will always join the `etherbox-all` multicast group.

The routine cycle of the controlling station includes a regular (though slow, say once a minute) transmission of a proglet with the only `poll` command to the `etherbox-all` multicast group. On the `poll` command all configured nodes execute a command chain configured before with `savepoll` command. The chain can be different for every node and is formed according to the node's role in the system. But unconfigured nodes have no `savepoll` chain and so the `poll` proglet finishes abnormally with `exitcode` set to `NEEDCONF`. This way the controlling station learns of a node that needs configuration.

The configuration procedure consists of sending one or more proglets to the node. In the simplest case the only proglet is sent, with `setkey` command that establishes common access key for this subnetwork and a `savepoll` command with empty command chain. A node configured this way will continue working as a server: wait for requests (e.g. "read all sensors") and reply to them. The node will send no packets of its own accord.

More complex configurations use proglets with "request—multiple reply" messaging pattern. The proglets stay resident on the node, replying periodically or when certain conditions are met. The replies typically contain results of sensor acquisition. This way one can implement the "publisher" part in the "publish-subscribe" model. As to the "subscriber" part, it can be implemented with help from the controlling station: it makes a subscription at the broker and then when the data arrive if forwards them to appropriate sensor nodes with Etherbox protocol-controlling station acting as a client, sensor nodes as servers. There are more details in the section 3.1.

Thus the distinctive feature of the networks that employ the Etherbox protocol is keeping nodes' configuration outside of the nodes — on the controlling station. The controlling station keeps the list of all nodes that can appear in the network (EUI-64 identifier and access key), as well as lists of peripheral modules installed in each node: EUI-64 identifier, module type, types af sensors and actuators connected to the module, location of the sensors and their semantics in the whole system. This information, called "the network description", originates from human workers who install the sensor nodes and is maintained at the controlling station by the network administrator.

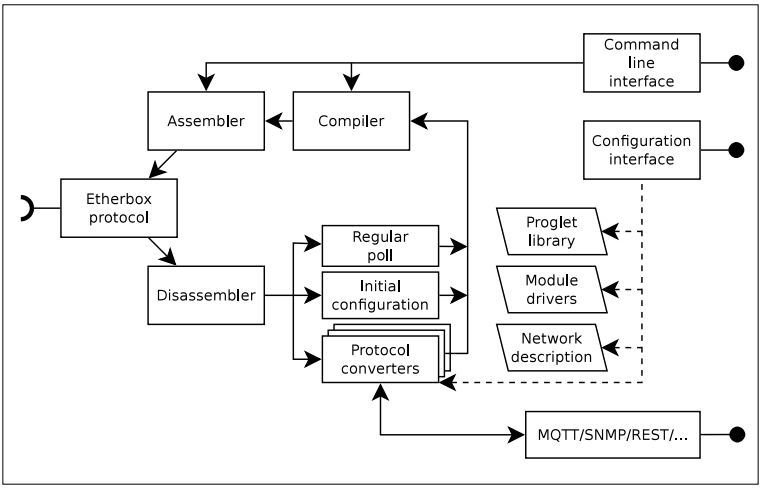


FIGURE 2. Controlling station software structure

3. Software architecture

3.1. Controlling station software

Controlling station talks to sensor nodes by Etherbox protocol while providing access to the sensor network with one or more standard protocols.

The controlling station software structure is shown at Fig. 2.

The chain *Compiler* — *Assembler* — *Etherbox protocol* implements compiling proglets from the source with ad hoc arguments and sending them in Etherbox protocol packets to sensor nodes or multicast groups. Etherbox protocol packets arriving from the sensor network are handled by the *Etherbox protocol* software module which checks the packet authenticity, extracts the proglet memory dump from the `image` field of the reply and hands it on to the *Disassembler*. The disassembler converts the dump to the text form convenient for subsequent handling.

Then the reply handler is chosen using the `handle` field of the reply packet header. As a rule, the handler is in the same software module as the code that formed the request the reply is for. Two of these software modules — *Regular poll* and *Initial configuration* are permanent parts of the controlling station software, while *Protocol converters* are added using the *Configuration interface*.

As the reply handler is bundled with the request builder, the reply structure is known to the reply handler. There is no need for any tags in the reply data or a separate schema. The software module author develops both the proglet that forms the reply and the reply handler. The schema implicitly presents in the algorithms of the proglet and the reply handler.

The *Regular poll* module sends proglets with the only `poll` command to multicast address `etherbox-all` at a configured interval. The module can also send the proglets individually to unicast addresses of the nodes that are outside the multicast area. The `poll` command makes each receiving node to execute a command chain the node received diring initial configuration.

The *Initial configuration* module works with nodes that were recently powered up and are not yet configured. The module uses the node's data from the *Network description* and configures the node as described in the section 2.3, sendind the node one or more proglets. The proglets are taken from the *Proglet library*. The *Protocol converter* modules take part in the initial configuration, adding to the list of initial configuration the proglets they need at the node.

The *Protocol converter* modules provide data transfer from the sensor network to users and back from users to sensor nodes using standard protocols. As an example, we will consider the MQTT protocol [**2**] which uses the "publish-subscribe" data exchange model.

Etherbox—MQTT converters act as reply handles, handling disassembled Etherbox protocol packets. They map the sensor coordinates (peripheral module ID, module channel name) to the logical MQTT topic space and pass the data to the MQTT protocol for publishing.

MQTT—Etherbox converters start by sending subscription requests to the MQTT broker, forming the list of topics based on the *netwosk description*. Published data received from the MQTT broker are demultiplexed by the topic name and go to the converter responsible for the topic. The data arrival is considered as a command to one of sensor nodes — for example, "turn on a relay". The convertor maps the topic name to the sensor coordinates (node address, peripheral module ID, peripheral module interface name), picks a fitting proglet from the *Proglet library* and passes it on to the *Compiler*. The data received by subscription appear as constants in proglet compile time. Alternatively, the proglet can be pre-compiled, then the data be patched in the binary image of the proglet. Finally, the binary proglet image is sent with Etherbox protocol to unicast or multicast address.

The *peripheral module drivers* are software components that provide methods for encoding commands and decoding status blocks of peripheral modules. In proglet's memory image sensor data usually appear in the "raw" form — within a status block read from the peripheral module. Peripheral module drivers know the status block layout and can pick the desired data and convert them from ADC units to physical values.

Traditionally in the field of operating system the word "driver" stands for a low-level software component that interacts with a device using device-specific operations and provides an unified interface to the device for the rest of the system. In our case interacting with the device is a proglet that runs on the base module of the sensor node, while the driver resides at the controlling station and cannot interact with the device immediately.

There are two use cases for peripheral module drivers:

(*1*) the proglet acts as a dumb proxy not knowing what kind of operation it translates to the peripheral module and unable to analyze the operation result. The proglet runs the `i2c` command prepared by controlling station once and returns the result to the controlling station for analysis. This variant is only suitable for the "request—reply" messaging pattern;

(*2*) the proglet contains `i2c` commands prepared by the controlling station along with the code capable to pick relevant data fields from status blocks read from peripheral modules. To generate such code the compiler need to know the structure of peripheral module commands and status blocks. The structure information is supplied by the periperal module driver. This variant is suitable for proglets with the "request—multiple replies" messaging pattern.

The *configuration interface* is used to transfer the *network description*, the *proglet library*, and the *protocol converters* to the controlling station. The configuration interface is implemented as a remote shell protocol server which makes it possible to change the configuration interactively as well as automate the process.

The *command line interface* is an auxiliary facility that allows a human operator to make an ad hoc request to any sensor node in the terms of Etherbox32mv assembly language or in the higher level Etherbox2 language, receive replies from nodes and monitor the system transaction log in real time.

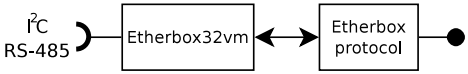


FIGURE 3. Structure of base module software

3.2. Embedded software of peripheral modules

The peripheral module software structure varies with module type and is not of interest for this article. Only one component is always present: the extension bus slave functionality (I^2C, RS-485, ...). Peripheral modules receive commands and send data only when asked by the master, the base module.

3.3. Embedded software of base modules

Base module software structure is shown at Fig. 3. It is extremely simple: in networks based on Etherbox protocol the complexity balance intentionally shifted away from sensor nodes to the controlling station.

The embedded software is universal. Ths only function it implements is proglet reception, execution and sending back replies. The Etherbox32vm virtual machine implements non-preemptive multiprogramming for proglets: several proglets can present on a node simultaneously, with control passed from one proglet to another only when the proglet executes a command that suspends its execution, for example `mdelay`. The maximum number of proglets per node is limited by the base module's memory and is about 20 for base modules with 64KB RAM.

The `Etherbox protocol` software module handles the packets arriving from the sensor networks. It checks the request integrity and authenticity, extracts the executable proglet image and passes it to *Etherbox32vm* virtial machine for execution. While executing, the proglet can execute peripheral module access commands, either read or write, in the role of the extention bus master. The proglet can also send its memory image, whole or a fragment, as a reply to the controlling station.

3.4. Loadable base module software

The loadable base module software is what facilitates sensor node configuration to its specific purpose in the system. It consists of one or more proglets delivered to the node with Etherbox protocol.

The number of resident proglets for every node is decided upon by the controlling station. In theory one can implement polling all sensors connected to the node in one large residnet proglet. But implementing

it his way it far from convenient: one would have to create a relatively big proglet individually for every sensor node, and handling the big reply at the controlling station will not be easy as well. Besides, implementing different polling intervals for different sensors with this approach is also painful.

It is more convenient to use a separate proglet for every peripheral module installed in the sensor node. The proglet library at the controlling station should contain at least one proglet for every peripheral module type used in the system. When compiled, the proglets are parametrized with information from the network description. In particular, the perpheral module identifier (EUI-64) is passed as a parameter. If there are several modules of the same type in a node, the same proglet can be compiled several times with different parameters.

This way, the loadable base module software architecture is in parallel to the modular sensor node construction.

The proglets that carry actuator control commands deserve an additional consideration. Most peripheral modules have a few input and output channels for sensor and actuator connection, so when a proglet with actuator control command arrives to a node, a resident proglet for the same peripheral module is very likely already there. Operations with most periperal module types are atomic — if so, the proglet with the actuator command can execute it without any coordination with the resident proglet and return the result to the controlling station (the "request—reply" messaging pattern).

For some peripheral module types though, not all operations are atomic. As an example, consider the BB-RS232 module that implements the RS-232 asynchronous serial interface. An RS-232 interaction session involves several operations over the I^2Cextension bus:

(*1*) write a data block for transmission to RS-232 interface;

(*2*) read BB-RS232 module status repeatedly, waiting for the RS-232 device to respond;

(*3*) read the RS-232 response from the module's FIFO buffer.

In the absence of coordination between proglets that try to address the same peripheral module, at overlap can occur which will result in accessing the device that is busy with another command, which will garble result of one or both operations.

The coordination problem can be solved as follows. At the initial configuration step, a resident proglet is loaded to the node that provides an RS-232 access API to other proglets in the form of *public function* [10]. At the API entry an interlock is implemented: the proglet that enters the API while another API operation is in progress will encounter an "interface busy" flag set and will cycle on `mdelay` command until the interface is accessible again.

3.5. Message delivery guarantees

When using the Etherbox protocol in pair with MQTT or other protocols based on TCP there is a mismatch in delivery reliability. Even at MQTT QoS=0 (best effort, no delivery guarantee) the delivery is reliable thanks to underlying TCP. As to the Etherbox protocol, it is based on unreliable UDP transport and packet loss in sensor network is not addressed at the protocol level.

The following two approaches are applicable to packets directed from the controlling station towards sensor nodes:

(*1*) do not try to mitigate packet loss between the controlling station and sensor nodes. Achieve reliability using MQTT retransmission facilities at QoS levels 1 or 2;

(*2*) implement Etherbox packet retransmission in the protocol converter. Keep `PUBLISH` messages in the converter and repeat delivery attempts until all target nodes acknowledge delivery.

The following two approaches are applicable to packets directed from sensor nodes to the controlling station:

(*1*) do not try to mitigate packet loss between sensor nodes and the controlling station, live with the losses in the hopes that the next data portion will be more lucky;

(*2*) implement Etherbox packet acknowledgement in the protocol converter. Make proglets repeat delivery attempts until the delivery is acknowledged by the converter.

The choice is left to implementors of protocol converters and proglets accompanying them. All variants can exist in the system, each sensor or actuator using the variant that suits it best.

Conclusion

Etherbox is an application level protocol for sensor networks that passes data between sensor nodes and a central host in the form of small programs in the bytecode of a specialized virtual machine Etherbox32vm (*proglets*). This approach achieves great flexibility in sensor network control while keeping the nodes' firmware comparatively simple and universal. These properties are especially important when sensor nodes are modular: the nodes can be assembled at the time of installation without the need for pre-configuring each node to its function in the network. The universal firmware provides the node with network connection, while node's configuration to its function is done later remotely. Once a node is configured, it can work either in passive mode, replying to queries, or in active mode, sending data on schedule or when certain conditions are met. Both modes can be used together to implement the "publish-subscribe" model.

Universality and simplification of sensor nodes' firmware is achieved at the cost of more complex software at the *controlling station* of the sensor network — a computer that posesses full information about all network nodes, coordinates the network operation and works as an application level gateway between the sensor network and the Internet. This complication is justified by the fact that the controlling station is a full-fledged computer with virtual memory, rich programming tools and mains-powered, while sensor nodes are small devices, constrained memory-wise, CPU-wise, power-wise, and with ascetical programming environment.

References

[1] MQTT For Sensor Networks (MQTT-SN) Protocol Specification Version 1.2., 2013, URL: http://mqtt.org/MQTT-S_spec_v1.2.pdf ↑ 286,292

[2] *MQTT 3.1.1 specification*, OASIS, 2015, URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html ↑ 295

[3] Z. Shelby, K. Hartke, C. Bormann. *The Constrained Application Protocol (CoAP)*, RFC7252, RFC Editor, 2014, URL: http://www.rfc-editor.org/rfc/rfc7252.txt ↑ 286,289,292

[4] K. Hartke. *RFC7641: Observing Resources in the Constrained Application Protocol (CoAP)*, RFC Editor, 2015, URL: http://www.rfc-editor.org/rfc/rfc7641.txt ↑ 290

[5] J. Postel. *User Datagram Protocol*, RFC768, RFC Editor, 1980, URL: http://www.rfc-editor.org/rfc/rfc768.txt ↑ 288

[6] J. Postel. *Transmission Control Protocol*, RFC793, RFC Editor, 1981, URL: http://www.rfc-editor.org/rfc/rfc793.txt ↑

[7] *UM10204. I^2C-bus specification and user manual*, NXP Semicondictor, 2014, 64 p., URL: http://www.nxp.com/documents/user_manual/UM10204.pdf ↑ 287

[8] M. D. Nedev, Yu. V. Shevchuk. "Sensory network with organization from outside", *Proceedings of the Third Russian conference ¡¡Technical and programming facilities of control, monitoring and measurement systems¿¿*, UKI'12 (Moscow, April 16–19 2012), Institute of Control Sciences of RAS, M., 2012, ISBN: 978-5-91450-100-3 (in Russian). ↑ 287

[9] S. M. Abramov, Yu. V. Shevchuk, A. Yu. Ponomarev, S. M. Ponomareva, E. V. Shevchuk. "Sensor network with module architecture", *Program systems: theory and applications*, **6**:4(27) (2015), pp. 197–208 (in Russian), URL: http://psta.psiras.ru/read/psta2015_4_197-208.pdf ↑ 287

[10] Yu.V. Shevchuk, A.Yu.Shevchuk. "Etherbox32vm virtual machine", *Program systems: theory and applications*, **7**:4(31) (2016), pp. 119–143 (in Russian), URL: http://psta.psiras.ru/read/psta2016_4_119-143.pdf ↑ 288,289,299

[11] rsh - remote shell. Linux man page, URL: https://linux.die.net/man/1/rsh ↑ 288

[12] *Electrical Characteristics of Generators and Receivers for Use in Balanced Digital Multipoint Systems*, TIA/EIA Interim Standard, TIA/EIA-485-A, Telecommunications Industry Association, Arlington, VA, USA, 1998. ↑ 287

[13] *Arduino: Shield Pin Usage*, URL: http://playground.arduino.cc/Main/ShieldPinUsage ↑ 287

[14] JH. Song, R. Poovendran, J. Lee, T. Iwata. *The AES-CMAC Algorithm*, RFC4493, RFC Editor, 2006, URL: http://www.rfc-editor.org/rfc/rfc4493.txt ↑ 291

[15] D. Mills, J. Martin, J. Burbank, W. Kasch. *Network Time Protocol Version 4: Protocol and Algorithms Specification*, RFC5905, RFC Editor, 2010, URL: http://www.rfc-editor.org/rfc/rfc5905.txt ↑ 291

[16] P. Srisuresh, M. Holdrege. *IP Network Address Translator (NAT) Terminology and Considerations*, RFC2663, RFC Editor, 1999, URL: http://www.rfc-editor.org/rfc/rfc2663.txt ↑ 286,291

[17] S. Cheshire, B. Aboba, E. Guttman. *Dynamic Configuration of IPv4 Link-Local Addresses*, RFC3927, RFC Editor, 2005, URL: http://www.rfc-editor.org/rfc/rfc3927.txt ↑ 291,293

[18] R. Droms. *Dynamic Host Configuration Protocol*, RFC2131, RFC Editor, 1997, URL: http://www.rfc-editor.org/rfc/rfc2131.txt ↑ 291

[19] *SNMP RFCs*, URL: http://www.snmp.com/protocol/snmp_rfcs.shtml ↑

[20] *Guidelines for Use of Extended Unique Identifier (EUI), Organizationally Unique Identifier (OUI), and Company ID (CID)*, IEEE, 2017, URL: http://standards.ieee.org/develop/regauth/tut/eui.pdf ↑ 291

[21] S. Thomson, T. Narten, T. Jinmei. *IPv6 Stateless Address Autoconfiguration*, RFC4862, RFC Editor, 2007, URL: http://www.rfc-editor.org/rfc/rfc4862.txt ↑ 293

[22] H. Lindholm-Ventola, B. Silverajan. *CoAP-SNMP Interworking in IoT Scenarios*, Tampere University of Technology. Department of Pervasive Computing, 2014, ISBN: 978-952-15-3219-1. ↑ 286

[23] C. Bormann, S. Lemay, H. Tschofenig, K. Hartke, B. Silverajan, B. Raymor, Ed.. *CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets*, IETF, 2017, URL: https://tools.ietf.org/html/draft-tschofenig-core-coap-tcp-tls ↑ 288

[24] M. Scharf, M. Necker, B. Gloss. "The Sensitivity of TCP to Sudden Delay Variations in Mobile Networks", Networking 2004, Lecture Notes in Computer Science, vol. **3042**, eds. N. Mitrou, K. Kontovasilis, G.N. Rouskas, I. Iliadis, L. Merakos, Springer, Berlin–Heidelberg, 2004, pp. 76–87. ↑ 289

Submitted by *prof. Sergej Znamenskij*

Sample citation of this publication:

Yury Shevchuk, Elena Shevchuk, Alexander Ponomarev et all. "Etherbox: a protocol for modular sensor networks", *Program systems: Theory and applications*, 2017, **8**:4(35), pp. 285–303.

URL: http://psta.psiras.ru/read/psta2017_4_285-303.pdf

The same article in Russian: DOI 10.25209/2079-3316-2017-8-4-263-283

About the authors:

Yury Vladimirovich Shevchuk

Head of telecommunication laboratory Multiprocessor Systems of Ailamazyan Program Systems Institute of Russian Academy of Sciences. Ph.D. Interest areas: system programming, digital electronics, computer networks, sensor networks, wide area monitoring and control, distributed programming

e-mail: sizif@botik.ru

Elena Vasilievna Shevchuk

Chief scientist of Research Center for Multiprocessor Systems of Ailamazyan Program Systems Institute of Russian Academy of Sciences. Interest areas: distributed computing, sensor networks, wide area monitoring and control

e-mail: shev@shev.botik.ru

Alexander Yurievich Ponomarev

Leading engineer of Research Center for Multiprocessor Systems of Ailamazyan Program Systems Institute of Russian Academy of Sciences. Interest areas: digital and analog electronics, sensor networks, switched-mode power conversion

e-mail: harry@opus.botik.ru

Igor Anatolievich Vogt

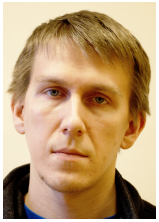
Chief scientist of Research Center for Multiprocessor Systems of Ailamazyan Program Systems Institute of Russian Academy of Sciences. Interest areas: sensor networks, ambient intelligence, distributed computing, control systems, metrology

e-mail: vogt@vgt.botik.ru

Alexey Viktorovich Elistratov

Research engineer of Research Center for Multiprocessor Systems of Ailamazyan Program Systems Institute of Russian Academy of Sciences. Interest areas: digital and analog electronics, sensor networks, computer aided design

e-mail: concept@pereslavl.ru



Andrey Yurievich Vakhrin

Research engineer of Research Center for Multiprocessor Systems of Ailamazyan Program Systems Institute of Russian Academy of Sciences. Interest areas: digital and analog electronics, sensor networks, electric power monitoring

e-mail: dispells@pereslavl.ru

Roman Evgenievich Yarovicyn

Research engineer of Research Center for Multiprocessor Systems of Ailamazyan Program Systems Institute of Russian Academy of Sciences. Interest areas: digital and analog electronics, sensor networks, electric power monitoring

e-mail: develop@pereslavl.ru