A Metaprotocol-Based Internet of Things Architecture

L. Milić\*, L. Jelenković\*\*, and I. Magdalenić\*

\* Faculty of Organization and Informatics, University of Zagreb, Croatia

\*\* Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia

luka.milic@foi.hr, leonardo.jelenkovic@fer.hr, ivan.magdalenic@foi.hr[[1]](#footnote-1)

*Abstract* – In this paper we propose a metaprotocol-based architecture for Internet of Things. The architecture is not layered, as common architectures are. Instead, all nodes can be regarded as smart nodes, utilizing roles that are assigned to them. A node can be a simple sensor node, or more complex, as, e.g., a gateway or a data storing and processing node. A single node may even have several roles. Roles can be created by custom programs or defined through rules, as in the presented prototype implementation. Regardless of the roles a node has, communication with other nodes in this architecture is based on the same metaprotocol. The metaprotocol is based on SQL operations, which allows maximal flexibility when designing a specific system. On the other hand, the proposed message encoding and reliance on the underlying transfer protocol allow the creation of very short and simple messages, enabling that even the simplest of devices may be included in such a system. Comparison with similar architectures shows expected advantages of the proposed architecture when the simplest nodes are used and also when an arbitrary network configuration is required, different from the standard layered thing-gateway-server configuration.

Keywords – Internet of Things, architecture, metaprotocol, databases

# Introduction

Internet of Things (IoT) is a name for a concept initiated by an idea by which all things from which man can have use should communicate, deduce, and decide without human interference, to free man from everything that internetworked digitalized things can do faster, easier, and better instead of him [[[2]](#endnote-1)]. The field which the notion of Internet of Things encompasses is today being intensively evolved, even though already a multitude of commercial environments exists for its implementation [[[3]](#endnote-2)]. Beginnings of the development of Internet of Things go back to the 90s, and by that name to year 1999 [[[4]](#endnote-3)].

Today, in the field of Internet of Things, thousands of articles are being published, which proves the current intensity of its development, spreading of application, and existence of a plethora of still unsolved problems [[[5]](#endnote-4)]. One of the problems of the current research is that a big majority of research is oriented on a multitude of specific applications [[[6]](#endnote-5)], which is sometimes called “connecting toothbrushes to the Internet” or “developing Intranets of Things instead of Internet of Things” in literature [[[7]](#endnote-6)]. The next problem is that often the research is not at all concerned with security but leaving security for later [[[8]](#endnote-7)], as a problem that will be easily solved later. Also, a problem is a big heterogeneity of Internet of Things due to a multitude (hundreds) of existing platforms which are not designed for interconnectivity, but as separate ecosystems [[[9]](#endnote-8)].

Challenges, which, thus, still stand before researchers, include creating an architecture flexible for any application in the field of Internet of Things, and not just a narrow group of specific problems. Also, embedding security mechanisms which will ensure all requirements, especially the requirement of privacy, must be an integral part of the architecture, and not an “afterthought”. As a pillar of privacy today, a technology of end-to-end network encryption is current, which ensures that nobody who only transfers data cannot in any way touch that data.

One of maybe the biggest problems of existing architectures is the complexity of building and using them. The fact that needs to be addressed is that most people do not know how to build computer systems, and those who do know have diverse levels of knowledge. Complexity can repel many users, which maybe because of it do not even see possible benefit of applying an Internet of Things system in their domain. The initial expectations about a huge use of Internet of Things are maybe not yet realized exactly because of this problem.

In this paper a new IoT architecture is presented which is based on three principles. The first one is disregarding the difference between nodes which represent things, gateways, and application nodes so that all nodes are observed simply as smart nodes. The second foundation is describing a metaprotocol for data transfer between nodes. That metaprotocol can make use of any datalink-layer, network-layer or transport-layer protocol, and can even be used with only a physical layer. The basis of that metaprotocol are database operations from the language SQL, i.e. SQL SELECT command and its data response. The third foundation is usage of a specially designed configuration and a rule system which enable implementation of different nodes, from simple to advanced, creating Internet of Things systems of different complexity.

In this architectural description, the term “metaprotocol” is used. In overall it is the closest to application protocols in design, but since it encompassed multiple layers and its form is very flexible and dependent on the underlying protocol, we chose to use the term “metaprotocol”. We do not propose a new underlying protocol, since there are already many IoT protocols on different layers and creating another one like them seemed to be unnecessary. The proposed metaprotocol can reuse many elements from the underlying protocol and only include elements that are missing from that underlying protocol. Therefore, it can be implemented over the simplest of underlying protocols (e.g. serial connections) or over sophisticated ones (e.g. HTTPS). It is this possibility to use it over very different networks and protocols that distinguishes the proposed metaprotocol from other protocols and attempts.

As an attempt of solving a part of problems Internet of Things has, a new architecture is proposed which is based on flexibility and simplicity of implementation. Those features are accomplished by flexibility of any of its nodes. With implicit node behavior proposed by this architecture, most simple nodes can be created. More complex nodes need only adding some rules to adapt their behavior accordingly to their role in the system. Examples of such systems and rules are shown in this paper. The authors believe that, as in shown examples in this paper, the proposed architecture will enable assignment of different operations to the node, from collecting data from sensors, processing data, storing data, to forwarding messages to other nodes, responses to requests of other nodes and such, all of that depending mainly only on node settings. Controlling the node only through settings should significantly simplify building and using a system. We solved security aspects by using existing security mechanisms of underlying protocols when a protocol has such mechanisms, or by providing methods to encrypt/decrypt and sign/verify data when such mechanisms are not provided by the underlying protocol.

The most important part of the proposed IoT architecture is the attempt to make the node model as general as possible. The authors are unaware of a similar model that can represent all node categories, able to implement any kind of IoT systems. Together with the metaprotocol unifying all layers of IoT, this is what makes the proposed model of IoT unique.

The remainder of the article is organized as follows. In Section II, related works are elaborated. In Section III, the model of the system is shown. In Section IV, the system is showcased on chosen scenarios. In Section V, benefits of using the proposed system are discussed, as well as some disadvantages. Finally, in Section VI, the conclusion is given.

# Related Works

Inside the Internet of Things, mainly a few protocols are used. On the datalink layer, most frequently Bluetooth is being mentioned, and its version for resource-constrained systems called Bluetooth Low Energy (BLE), which solves its problems with energy consumption, master-slave architecture, and necessary processor power. The protocol BLE overshadowed other protocols of that layer [[[10]](#endnote-9)], like Wi-Fi and its rework called IEEE 802.11ah, which is concerned with energy consumption. The protocol IEEE 802.15.6 also is not widely applied, and neither is WiMAX, even though those protocols are sometimes mentioned as appropriate for IoT [9]. Some of the protocols which met a somewhat larger audience are, e.g., LoRaWAN and SigFox [9].

The datalink-layer protocol IEEE 802.15.4 (154 for short) [[[11]](#endnote-10)] is applied as a basis for ZigBee [[[12]](#endnote-11)], a popular solution for local networks, and for Thread protocol stack [[[13]](#endnote-12)]. Thread is also based on the protocol 6LoWPAN [[[14]](#endnote-13)] for encapsulating the protocol IPv6. Another popular protocol stack of this capacity is Z-Wave [[[15]](#endnote-14)], which is based on proprietary protocols. ZigBee and Z-Wave are designed only for communication in a local IoT network. Thread also assumes existence of a router, but considering 6LoWPAN, that router only needs to do protocol transformation, and it does not need to have a controlling role. On the network and transport communication layers in a global network, common protocols are used (IP, IPv6, TCP, UDP) considering a feasible alternative does not exist.

On the application layer, usually CoAP (adapted from HTTP) [[[16]](#endnote-15)], XMPP [[[17]](#endnote-16)], MQTT [[[18]](#endnote-17)], AMQP, RESTFUL are used [[[19]](#endnote-18)]. All of them use the publish-subscribe model, i.e. all create client-server architectures, and as such they do not provide the necessary flexibility for implementing a more general architecture.

Screening of the literature indicates that usage of specialized languages represents an issue in IoT implementation, and a need arises for a common well-known language. Therefore, we use SQL [[[20]](#endnote-19)] in the proposed metaprotocol. However, SQL usually is not used directly in protocols for IoT. The most similar existing technology is SPARQL [[[21]](#endnote-20)], which works with RDF data for semantic Web. SQL is however far more powerful than SPARQL and supports more diverse data.

In [[[22]](#endnote-21)] devices can be communicated with via REST over multiple datalink-layer protocols. They can receive specific data and be asked for specific data. The user needs to learn the rule language which is compiled into Java programming language. In [[[23]](#endnote-22)] rules are written in Lua scripting language using a semantic engine API. The engine can communicate with devices using its protocols and make conclusions based on ontologies. The rules are edited using web configuration.

In [[[24]](#endnote-23)] users can create smart objects by choosing their shape, their parts and their behaviors using a graphical interface. Behaviors are composed of rules, controlling parts of the objects having some ready functionalities, and compiled into Java. Rules are of type “when <some reading> do <some action>”. In [[[25]](#endnote-24)] rules are created using a graphical interface for composing provided services. The services are based on data collected from things, data sent to things and metadata inferred about the things. The things are also abstracted through a representation layer.

In [[[26]](#endnote-25)] there is a system which uses HTTP or a HTTP gateway to communicate with things using REST, thereby creating what is often called Web of Things on the local network. Rules are written in JavaScript and the description of their triggers is written in JSON format. They can refer to either things’ data or existing data, and they generate events for an activity-performing module, which can also be real-time monitored. In [[[27]](#endnote-26)] there is a system for graphically creating and modifying rules and it is implemented in JavaScript and jQuery technology. The system works with a rule engine capable of receiving events, memorizing data, and performing actions. The engine communicates with smart things using multiple protocols.

In [[[28]](#endnote-27)] triggers of different levels of abstraction can be defined, with the help of contextual information, over an ontologically based representation of devices classified by categories, and their services classified by their capabilities. The system is used to automatically create an appropriate graphical interface in one of the authors’ projects and to suggest rules to the user in another project. In [[[29]](#endnote-28)] existing rules from a given system are analyzed and described using formal models. Then, using the mapping learned there, when a new device is plugged in the system, it can analyze its functionalities to automatically create rules, and recommend them to the user.

In [[[30]](#endnote-29)] gateways store timestamped data from devices. Devices are abstracted using their so-called “avatars” which are described with ontologies. Higher layers can send queries to the gateways using the gateways’ query language and the gateways can respond. In [[[31]](#endnote-30)] Event-Condition-Action rules are created using a graphical editor which coincides with system’s rule language, which is then compiled into another rule language. The cloud on which the system is installed receives data from things using the local networks. Then it can push actions to an application that connects to the cloud.

There are some common unwanted features pertaining to solutions presented in this chapter, which are not present in proposed architecture. These will be further elaborated in Section V, after the architecture is presented. They are mainly related to flexibility and simplicity, as mentioned before. Of course, not all solutions have same problems, but most of them are rather common – appearing in all of them.



Fig. 1. Example of implementation of the architecture

# System Model

The system model of the proposed architecture is defined by its messages and its nodes’ activities. Several message types are defined, mostly based on the SQL SELECT command and its data response, each type for some operation in the model of IoT systems. The main purpose of messages is to allow communication between nodes that produce data (usually called “things”), nodes that process and store data (“servers”), and nodes that request such data for some purpose (“clients” or “users”). Furthermore, the proposed model incorporates the usual publish/subscribe mechanism.

There is no fixed architecture type in the proposed model, since each node can assume the role that was defined for it – role of a thing, server, client, gateway, or a combination of roles. It is expected that the standard architectural model (thing-gateway-server-client) is usually realized in the implementation, but the proposed model allows other models as well. The node role can be defined with programs it runs. In addition to message formats, there is an attempt to define a generic node model with behavior controlled through node settings, and defined through SQL capabilities.

An example of implementation of the architecture is shown on Fig. 1. This model shows all traditional roles (things, gateways, cloud, user), but there are no problems with gateways taking a part of cloud’s activities, or things communicating directly with the cloud. More generally, there are no barriers to anyone taking any part of any role, and no barriers to anyone communicating with anyone in any direction. This is made possible through a universal data model and through the system designer’s knowledge of who needs to communicate with whom. Naturally, not all devices can be planned, but knowing their roles enables correct network configuration.

The proposed message formats and types with their purpose are described next, following with the node model and node activities.

## Messages

The proposed metaprotocol does not define its (underlying) protocol. Its messages can be transferred by any available underlying protocol. However, facilities provided by the used underlying protocol can be utilized to simplify the message, omitting already present elements.

Message types (described later), message elements and its encapsulation into the underlying protocol are shown on Fig. 2. A single message can contain the following elements, in this order:

* HD – message header (1 byte)
* I D – message identification number (1 byte)
* LEN – message length (2 byte)
* DST – destination node id (8 byte)
* SRC – source node id (8 byte)
* PL – message payload (varying length)
* CRC – message checksum (4 byte).

Not all elements are required in messages. For example, if the source or destination node id can be extracted from the underlying layer (e.g., the protocol carries EUI-64), those elements can be omitted in the message. In some short messages, all or almost all elements can be omitted.

The header consists of bits I, L, D, S, R, K, C, A that define which elements are present in the message (bits I, L, D, S, R for ID, LEN, DST, SRC, CRC) and optional operations required from the message receiver, such as returning an acknowledgement message (bit K), securing the message for confidentiality (bit C) and/or authenticity (bit A).



Fig. 2. Protocol format and encapsulation

The message ID together with SRC and DST forms the message identification and is used for confirming the message has arrived, notifying error in the message, notifying unsupported operation, and recognizing duplicates, e.g. if the same message is sent through multiple routes.

In the following protocol description, mostly just the message payload will be described.

### SQL encoding

To make this protocol as appropriate to resource-constrained devices as possible, much effort has been put into making it extremely lightweight. Therefore, the messages are designed to be as short as possible, often using optional encoding of many message parts and optional drops of many message fields. Also, the message processing is simplified, using the ability to almost always discern a message type from the first character and simply not supporting many operations in the simplest nodes.

The message payload formats are inspired by SQL commands, and are further encoded, for smaller message size, with an 8-bit binary value from the ASCII 128-255 range for the most used SQL SELECT keywords. E.g., the SQL query “SELECT a, b, c FROM table WHERE a > b AND b > c ORDER BY c;” will be encoded as “X1a,b,cX2tableX3a>bX4b>cX5X6c” where “Xi” are the appropriate hexadecimal code values for the replaced keywords. With such encoding, the message payload size is significantly reduced.

Theoretically, any SQL SELECT can be a part of the message. However, since this protocol was designed for IoT environments, some of the SQL syntax was simplified to facilitate the mostly used operations. Additionally, some default table and column names in some message types are used if they are not given directly in the SQL query.

The default table name is constructed from the source or destination node identification numbers prefixed by the character “t”. Whether the source or destination id is used depends on which node provides the data. When sending data, the source id is used. When requesting data, the destination id is used. E.g., when a node with “source id” = ”0x1111111111111111” sends data, the default table name on the destination node (who will store data) is “t1111111111111111”. Similarly, when a node requests data from another node with “destination id” = ”0x2222222222222222” then the default table name is “t2222222222222222”.

The default column name is “d” (as “default”), when otherwise is not provided, and “d1”, “d2”, “d3” … when there is more than one column.

Tables, that contain data received from other nodes, besides columns given in the message, must also contain a default column “t” – a timestamp of the message (when it was received) – which also has an SQL INDEX attached to it.

Different messages, depending on what they carry, can be classified into several message types. The main message types by functionality are:

* DATA
* SELECT
* SELECT\_SUBSCRIBE
* UNSUBSCRIBE

Other message types are: UNSUBSCRIBE\_ALL, QUICK, HELLO, ERROR, ACKNOWLEDGMENT and UNSUPPORTED.

### Message type DATA

A DATA message contains some data (readings, status, reply to data request, reply to subscription etc.). A DATA message can contain a single named value in form:

name=value

or multiple named values:

nam1,nam2,…,namN=val1,val2,…,valN

or even multiple values for same names, i.e. a table:

nam1,nam2=val11,val21;val12,val22

When multiple values for same names are provided, rows of values are delimited with a semicolon (;). In the last example, there are two values for each name (“nam1”: {val11, val12}, “nam2”: {val21, val22}).

A node which received the DATA message must first expand it into an SQL INSERT query, and then execute it, if the data needs to be stored. The table name is a default one (“t” + “source id”).

DATA messages are sent from nodes that generate data (sensors) or as a response to a SELECT query.

### Message type SELECT

Retrieving data from some node is requested with a SELECT message. The SQL SELECT query can be a simple one, like “SELECT col1, col2 FROM table;” or more complex with aliases, a WHERE, and other clauses and conditions. When data is requested from a sensor node, the query can be even simpler, e.g. “SELECT val1, val2;”, or even “X1\*”.

The result of running the SQL query is obviously some data that can be packed into a DATA message and sent back to the requester.

### Message type SELECT\_SUBSCRIBE

SELECT\_SUBSCRIBE is intended for a publish/subscribe service, where service provider, upon receiving some data from its sources, sends appropriate data to its subscribers.

The message SELECT\_SUBSCRIBE is an extended SELECT message, with addition of “SUBSCRIBE id” at the very end. A node that provides such a service must save this SQL request as a “subscribe request”, identified with the given “subscribe id” and “client id”. When a SELECT\_SUBSCRIBE is received, the SELECT query it contains must be executed. Moreover, when new data is inserted into tables used in the SELECT\_SUBSCRIBE message, the SELECT query should be executed again on new data and the set difference in result, if any, sent back to client.

### Message type UNSUBSCRIBE

An UNSUBSCRIBE message is intended for canceling subscription. The message contains only the keyword “UNSUBSCRIBE” and a subscription id. The receiving node should delete such subscription in its tables.

### Other message types

An UNSUBSCRIBE\_ALL message is the same as the UNSUBSCRIBE message, except that it has the keyword ALL as the subscribe id. The receiving node should delete all subscriptions from the sending node in its tables.

A simpler DATA message, called QUICK, carries only one value without defining a name (using default “d”). Such message is intended for simple things where only very short messages are allowed. Usually all other message elements (besides header) can be omitted in such a message.

If a message payload (PL) is 0-byte or 1-byte long, we are talking about HELLO and QUICK messages, respectfully. These simplest messages are deliberately designed as such, to be as shortest as possible. However, since the message header can also be dropped in such messages, along with other fields, these messages can also be 0-byte and 1-byte long in total, respectfully. Here some trouble arises decoding whether a 1-byte long message should be decoded as a HELLO message with header (HD) or a QUICK message without one. Therefore, we define another rule for message decoding: the HELLO message must not be 1-byte long in total but at least have a zeroed header, because it is expected that the latter (QUICK) will be used more often without HD.

A HELLO message does not carry any payload and can serve as notification to the receiving nodes of sender node status (“I am online”), if there are no useful messages to be sent before it. Such a message is intended for nodes that have just come online or are available for message receipt (and maybe will not stand online much longer).

An ACKNOWLEDGMENT message serves to notify the sender of a previous message that its message was received and understood. The message payload contains “K<1-byte message id>”, identifying message for which this is the acknowledgement. This message is usually sent when the K-bit was set in a previously received message.

An ERROR message is sent when there is an error in the received message. The message is formed as “E<1-byte message id>”, identifying the troubled message with given id. An optional error description message may follow.

An UNSUPPORTED message is sent when there is a processing error on the received message, caused by an unsupported operation in the destination node. The message is formed as “N<1-byte message id>” (no U not to be confused with an UNSUBSCRIBE message) identifying the troubled message with the given id. An optional description message string may follow.

## Node Behavior

A node in the proposed system model can be anything. The node role defines its behavior. For very simple things, this behavior can be described just with “collect the sensor value and send it”. On the other hand, some node in the cloud can collect data, process it, combine results from different sources, and control complex operations. Such extreme node cases require custom programs (the first one a very simple program, while the second one a collection of multiple programs and services).

Other nodes’ behavior (between the simplest and very complex ones) can be modeled on a few operations they perform individually (periodically or on sensor input) or as a reaction to a received message. Such nodes will be analyzed and modeled in this subsection.

The node that controls some devices (actuators) or collects readings from a connected sensor can be modeled as an activity “each X time units do THIS” or “on event Y do THIS”. Operation “THIS” may include reading the sensor, sending a command to the attached actuator, storing the obtained value locally, sending some data to a remote node… Those operations may be broken down to simpler actions, which can be described with few parameters. Such breakdown is used in the proposed node model.

Another part of node’s behavior are the actions pertaining to message exchange. What to do when a message is received? Should the node store it, forward it, discard it or take some local action on it (as “read sensor”, “send command to device”, etc.)? Sometimes some additional action may be required when sending a message, such as “store a copy”, “send to a backup node”, “update something”, etc. Previously described behaviors can be modeled with “triggers” that are evaluated on every received or sent message.

The proposed node model includes both operations: time/local-event triggered and message exchange triggered.

## Node Model

The node model, based on a prototype implementation, is shown on Fig. 3. The base components are a main program and underlying protocol modules, UPMs. UPMs can encompass any underlying protocols. E.g., on Fig. 3, there are 4 UPMs, one for TCP, one for UDP, one for BLE, and one for 154. These UPMs enable usage of the proposed metaprotocol within other protocols, using encapsulation, decapsulation, extracting and injecting information from and into headers between the metaprotocol and the underlying protocol. The number of UPMs in use can vary, depending on node configuration. UPMs, directly or through other layers (protocols), use node’s network adapter(s), for transmission of the data to other nodes.

When using the full node model from Fig. 3 (as in the prototype implementation), the main program is connected with a database, which contains rules, node users, stored messages, subscription triggers, node configuration, etc. A database can be used directly or through external (C) database functions. That should be sufficient for general node operations, using sensors, receiving messages, processing messages, sending, or forwarding messages.



Fig. 3. Node model

If a node should provide direct interaction with users, then an additional web server should be included, as shown on Fig. 3. Through its web interface, users can get required information, send commands to nodes, and similar operations. In the prototype implementation, users can use the web interface to access raw messages, to inject messages or send them manually, or to manipulate rules and configuration.

For simpler nodes without a web interface, if the default behavior is not enough, the configuration must be set manually (imported into the database). If a node is even simpler, it will not use a database at all, and everything shall be programmed into the main program.

Although the prototype implementation contains all elements from Fig. 3, its modularity permits inclusion of only some required elements, enabling support for different nodes. Exempts are the simplest nodes which will have a slightly different main program, without database interaction, but still using required UPM(s), even though (just for now) UPMs are not separated yet.

## Security actions

The security in the context of this metaprotocol consists of confidentiality and authenticity. The confidentiality is realized by encryption, so that insight into message contents is blocked to those nodes which should not have it, while the authenticity is realized by message signing. If the communication occurs in a closed system, in which all nodes are trusted, the security is not obligatory, and a message can be sent unencrypted and without a signature. That especially pertains to things in a local network which is possibly physically separated from everything else. However, as soon as the message crosses such borders and it is transferred to a destination by an unsecure network, it is necessary to ensure at least some minimal security requirements, e.g., with message signing, preventing injections of fake messages.

For manually securing messages, the usual today’s cryptography algorithms are being used. A digital envelope of the original message has the following format in the payload:

* character @
* two-byte length of encrypted message
* message encrypted symmetrically
* two-byte length of encrypted key
* key encrypted asymmetrically
* initialization vector

The key for symmetric encryption and the initialization vector are generated randomly. The public key of the destination node is used for asymmetric encryption.

A digitally signed message has the following format (payload):

* original message payload
* character #
* hash of the message, encrypted asymmetrically

Characters @ and # cannot be interpreted as nothing else in SQL except parts of security. A digitally enveloped and signed message has the following combination in the payload:

* character @
* envelope remainder (as described before)
* character #
* signature remainder (as described before)

When handling messages in the rules, flags ENCRYPTED and SIGNED can be used, together with special commands ENCRYPT, DECRYPT, SIGN, and VERIFY, which will use the defined formats for envelopes and signatures.

The proposed system enables usage of the underlying-protocol security services when they are available, but also provides mechanisms when they are not, or when they are not enough. A node which sends a message defines necessary security components by setting flags C for confidentiality and A for authenticity in the message header. Other nodes that take part in message transfer must oblige in the message-defined security requirements or not forward the message if they cannot.

The metaprotocol can be used in a heterogeneous network, starting from a local network of things toward a smarter node (e.g., a gateway). The smarter node can then forward that message over the public Internet using its transfer protocols to the next node that uses the proposed metaprotocol, and so on, until the message reaches its destination. The destination can be a smarter node, or a thing on some local network. Security requirements set in the message header must be obliged on every message hop, or message must be dropped and not forwarded. Exceptions can be made by explicit node configuration (drop all, forward all, and similar).

Considering very likely processing limitations of some IoT things in the context of using cryptographic algorithms, the metaprotocol enables that such things request security even if they cannot provide it. For example, that security can be meant for after the message exits the local network.

A node can decide on what will it do with some message based on implicit rules (presented later in this section) or additional rules specially defined in that node. E.g., in the additional rules it can be checked where does the message come from and based on that, decided what to do next. After considering the implicit rules which originate from basic ideas of the proposed architecture, common additional rules will also be considered.

Simpler nodes (i.e. things) can be configured that they do not utilize security, and they simply trust everything, and forward all received messages further without changes, and in that way help that some message from a more remote thing comes to a smarter node (e.g., a gateway). The smarter node can analyze the message from the security aspect and appropriately secure it when forwarding. For that they can use the underlying-layer services or metaprotocol’s abilities or both.

The implicit security check rules are presented with an algorithm shown on Listing 1, using Python-like pseudocode. Argument msg is the message that is transported, iSRC (immediate source) is the underlying-protocol address of the previous node in transmission and iDST (immediate destination) is the underlying-protocol address of the following node, and uproto is the underlying protocol.

Listing 1. Implicit security checks before sending a message

|  |
| --- |
| check\_sending (msg, DST, iDST, uproto):  if configuration.trust\_everyone or  (not msg.C and not msg.A) or  msg.manually\_secured():  send msg to iDST  else if uproto.can\_secure(iDST, msg):  send msg securely using uproto's  security mechanisms to iDST  else:  drop msg |

A simple node can be configured to forward any message it receives. A smarter node should check flags A and C in the message header and, if they are not set, forward the message. If the flags are set, but the message is already protected by the metaprotocol’s security mechanisms, the message should also be forwarded. Otherwise, the message will be dropped unless the underlying transfer protocol can provide the necessary security mechanisms.

Security checks should similarly be performed upon message reception. The algorithm for implicit security checks upon message reception is presented on Listing 2.

Listing 2. Implicit security checks upon message reception

|  |
| --- |
| check\_receiving(msg, SRC, iSRC, uproto):  if configuration.trust\_everyone or  (not msg.C and not msg.A) or  msg.manually\_secured() or  uproto.was\_secured(iSRC, msg):  accept msg for processing  else:  drop msg |

The second part of the security mechanisms are explicit message rules which should usually activate when a message is crossing network borders.

E.g., let us say a gateway is placed between the trusted local network and the untrusted global network. One rule in the gateway could state that if a message without C or A is received from the local network and it is headed to the global network, it should be dropped. A similar rule can be made for received messages from the global network. Another rule could force encryption and signing of a received unprotected message from the local network which has flags C and A set. A complementary rule will decrypt and verify a message received from the global network before forwarding it to a local network node.

Trusted and untrusted networks can, e.g., be discerned by protocol – a local protocol can be trusted (e.g., BLE), and global ones untrusted (e.g., TCP). Other criteria can also be used to determine network boundaries, from security perspective, like the underlying-protocol addresses.

## Message routing

The message ID is randomly generated first time when a message is created for a specific destination node. The next message from the same node for the same destination will have that ID incremented in modulo 256 arithmetic. This mechanism provides ability to detect duplicate messages. Duplicates are expected in a local network when using broadcast to transmit a message from the starting node to the destination node, using other nodes as carriers. However, using ID, SRC and DST as a full message identifier, and assuming that single node will not generate a lot of messages for the same destination in a short period of time, duplicate messages can be easily detected – in the short period for which the identifiers are stored.

IoT systems can consist of nodes that are not always operating. In such a network, a message sent from one node to another may pass different paths in different times (or circumstances). When sending or forwarding a message, a node may have a several alternative routes for the next node. A single next node can be chosen, or a message can be sent to all alternatives. To optimize network usage, a smarter node should track reachability of destination node over each route, and dynamically update this information for future use.

The simplest nodes are not supposed to have complex routing and can simply use broadcast on datalink layer or on the metaprotocol layer (set DST in the message to the broadcast address). When such a node receives a broadcast message, it will both process it as if it were meant for it, and if needed, also forward it using broadcasting with all its protocols. If broadcast is used, some security features must be turned off – the message elements must be visible to all to enable forwarding. However, the message payload might still be encrypted and signed for destination node, but this would require not-so-trivial message rules.

# Example Scenarios

The following examples show some characteristic system behaviors in typical usage scenarios.

## Example 1. Basic operations

An example system shown on Fig. 4 consists of four nodes. Node-1 and Node-2 are simple things that communicate using BLE with Node-3. Node-3 and Node-4 communicate using TCP/IP stack. Node-1 represents a pressure sensor that sends its readings once each hour to Node-3. Similarly, Node-2 represents a temperature sensor that also sends its readings once each hour to Node-3. Node-4 represents a client node that requests data from Node-3 about readings from sensor nodes, Node-1 and Node-2.



Fig. 4. A simple IoT system

In the presented scenario, the first message (1) is sent from Node-1 with its readings. Node-3 receives this message and stores it. Next, Node-2 sends its readings (2). Such messages will be repeatedly sent (with new readings), once per hour. Sometime later, Node-4 requests readings for a specific period (3) and then receives the answer from Node-3 (4). The description of previous messages follows.

Node-1 sends a DATA message (message 1 from Fig. 4) using all metaprotocol fields, as shown on Table I. Since all fields are present in this message, there is no need for using BLE capabilities, and therefore for this message BLE is not analyzed. The purpose of the Table I is to showcase the proposed message format in its most complete form.

TABLE I.

Elements of message 1

|  |  |
| --- | --- |
| **Element** | **Value** |
| Header (HD) | 0b11111000 |
| Message id (ID) | 0x48 |
| Length (LEN) | 0x0009 |
| Destination node ID (DST) | 0x333333FFFE333333 |
| Source node ID (SRC) | 0x111111FFFE111111 |
| Payload (PL) | “pres=1034” |
| Cyclic redundancy check (CRC) | (4 bytes) |

The header describes message elements. In this example (header=0b11111000), the message contains all elements, even though some of them are clearly unnecessary.

Upon message creation, the message id (0x48) is generated randomly, the source and destination identifiers are usually generated from Extended Unique Identifiers (EUIs extracted from BLE addresses), the message length represents only the payload length and the CRC is calculated over the entire message. For simplicity, the source and destination identifiers in this example are set to simple values. In this example, the message payload is a string “pres=1034” (as in “pressure”), which is interpreted as a DATA message.

Upon reception of the message 1, Node-3 creates a table “t111111fffe111111” (character “t” extended with the source id) with a default column “t” (TIMESTAMP WITHOUT TIME ZONE) and one column from the DATA message – column “pres” with the type determined by the provided value, i.e. NUMERIC(4, 0).

If this were not the first message from Node-1, such table would already exist on Node-3. If the table existed, but without the column “pres”, such column would be added when this message is received. If the column “pres” existed but with a less precise type, it would be extended.

TABLE II.

BLE packet for message 2

|  |  |
| --- | --- |
| **Element** | **Value** |
| Preamble | (1 byte) |
| Access Address | (4 bytes) |
| Header | (2 bytes) |
| AdvA | 0x222222222222 |
| AdvData | 0x14 |
| Cyclic redundancy check (CRC) | (3 bytes) |

The received message is, therefore, expanded to {pres=1034,t=LOCALTIMESTAMP}, and inserted into the table “t111111fffe111111”.

In this example message, all protocol fields may not be needed since most of them could be deduced from the BLE protocol, as shown next for the message 2.

Node-2 sends its reading with a simplest message type, QUICK, which has only one byte, no other message elements or even a header, only payload. Since BLE is used, the packet that carries this message consists of fields shown on Table II.

Since the message length (calculated from the packet elements) is one byte, Node-3 will process the message as QUICK type and, therefore, generate other message elements by extracting information from the BLE packet, or calculate them, or use default values. In this example, the message header is deduced to zero (0x00), the message payload length to one, the source id to sensor’s EUI-64 (0x222222fffe222222 expanded from AdvA), the destination id to Gateway's EUI-64 since broadcast is implied. The CRC is calculated over the 1-byte message (and still checked), and its first byte is further used ad-hoc as the message id. Then, the payload is expanded from AdvData to “d=20” (0x14=20) and, as with the message from Node-1, the table “t222222fffe222222” is created with “d” and “t” columns only, and then the row {20, LOCALTIMESTAMP} is inserted.

In further examples and messages, the focus will be only on their payload. Other required fields are provided in the message or generated from the packet, but without detailed explanation.

The next event from the described scenario includes Node-4, which asks for sensor readings on 2021-01-01 from 10h to 14h. The message it sends to Node-3 is of type SELECT and contains the following SQL-standard query in its payload:

SELECT ALL t1.pres AS pres, t2.d AS temp FROM t111111fffe111111 AS t1 INNER JOIN t222222fffe222222 AS t2 ON CAST(t1.t AS DATE) = CAST(t2.t AS DATE) AND EXTRACT(HOUR FROM t1.t) = EXTRACT(HOUR FROM t2.t) WHERE CAST(t1.t AS DATE) = DATE '2021-01-01' AND EXTRACT(HOUR FROM t1.t) BETWEEN 10 AND 14 ORDER BY t1.t ASC; (311 bytes uncoded).

Using the proposed coding of SQL keywords, the message is coded as:

\xDF\x85t1.pres\x87pres,t2.d\x87temp\xADt\x11\x11\x11\xFF\xFE\x11\x11\x11\x87t1\xB4\xB8t\x22\x22\x22\xFF\xFE\x22\x22\x22\x87t2\xCAST(t1.t\x87\x9B)=CAST(t2.t\x87\x9B)\x84EXTRACT(\xB3\xADt1.t)=EXTRACT(\xB3\xADt2.t)\xFCCAST(t1.t\x87\x9B)=\x9B\'2021-01-01\'\x84EXTRACT(\xB3\xADt1.t)\x8C10\x8414\xD2\x8Ft1.t\x88

(164 bytes encoded, functions are not considered keywords).

Node-3 decodes this message, executes the selected query and responds with the result in a DATA message, e.g., with a payload:

pres,temp=1030,19;1031,20;1032,20;1031,21;1030,21 (one line).

## Example 2. Operation subscribe

In the second example, the same system is used as in the previous one, with additional functionality: Node-4 requires notification when the temperature sensor sends a value greater than 20 °C. Node-4 will, therefore, send the following subscribe request to Node-3 (message SELECT\_SUBSCRIBE):

SELECT ALL t1.d FROM t222222fffe222222 AS t1 WHERE t1.d > 20 SUBSCRIBE 1;

The SELECT extension proposed in this protocol is an addition of SUBSCRIBE <sub\_id> clause at the end of the query. The number <sub\_id> serves as a subscription identifier for this client (Node-4) and this query within Node-3.

In the subscription feature implementation, SQL triggers could be used (as in prototype implementation). Whenever data is inserted in any table mentioned in the subscription query, the query must be executed again, and if there is a set difference (A\B) from previous result, the difference is sent to subscriber.

In this example, Node-4 demands of Node-3 to execute the given query whenever data is received from the temperature sensor, i.e. whenever data is inserted into the table t222222fffe222222. For example, if Node-2 sends a temperature 23, Node-3 will insert a value of 23 into the device’s table and then, on a trigger, run the subscription query, and if it produces some new result, send it within a DATA message to Node-4 (message d=23 or 0x17).

When Node-4 wants to cancel its subscription, it must send an UNSUBSCRIBE message to Node-3:

UNSUBSCRIBE 1;

To cancel all subscriptions on a node, an UNSUBSCRIBE\_ALL message can be used. Nodes should send an unsubscription message when they are powering down for longer.

If the connection with the gateway is not reliable, the bit K in flags should be raised, requiring acknowledge response from the gateway, i.e. an ACKNOWLEDGMENT message in the form:

K\x<hex ID> (2 bytes),

where <hex ID> is the message id taken from, or calculated, from the message a node responds to.

## Example 3. Distributed system

Nodes for the next example are shown on Fig. 5. It simulates a hierarchical distributed system with Node-3 on the top and nodes Node-1 and Node-5 on the bottom of the hierarchy. Node-1 represents a sensor, Node-2 an IoT gateway near the sensor location, while Node-3 is remote server which is the default forwarding point for Node-2. Node-5 represents an IoT node at another remote location where configuration for Node-1 (and possibly others) is defined. Node-5 is not reachable directly, but instead Node-4 serves as its IoT gateway. Node-4, similarly to Node-2, uses Node-3 as the default forwarding node.



Fig. 5. Distributed IoT system example

In this example, it is assumed some simpler underlying protocol is used between Node-1 and Node-2, and between Node-4 and Node-5 (e.g. BLE), while Node-3 is connected with Node-2 and Node-4 through some Internet protocol (e.g. TCP, UDP, HTTPS, ...).

Nodes which are higher in hierarchy might not know the status and addresses of lower nodes without explicit announcement from the lower nodes to the higher ones. For this announcement (awareness), any message sent from the lower nodes will do. However, if there is no information to pass, a simple HELLO message can be used, e.g. when such nodes are started. A hello message is an empty message (without payload, zero-byte message) or two-byte message when some bit is set in the header (e.g., when asking for acknowledgment).

In the scenario that is presented in this example, a HELLO message is sent upon nodes’ startup in the following order: Node-2 to Node-3 (message 1 from Fig. 5), Node-1 to Node-2 (2), Node-2 to Node-3 (3 – forwarded message 2), Node-4 to Node-3 (4), Node-5 to Node-4 (5), and Node-4 to Node-3 (6 – forwarded message 5).

Furthermore, it is assumed that, in this scenario, Node-1 must periodically send its readings to Node-2. However, firstly Node-1 must retrieve the period in minutes from Node-5. Therefore, the next message (7) is sent from Node-1 to Node-2 (as its default gateway) but meant for Node-5. Node-2 will forward this message to its default gateway Node-3 (8). Node-3 will in its data structures find that Node-5 is reachable through Node-4 and will forward the message to it (9), and finally Node-4 will forward the message to Node-5 (10). The message Node-1 sent is a SELECT message:

SELECT ALL period FROM configs WHERE node\_id = 0x111111FFFE111111;

On Node-5, table configs (a manually created table) should have a row per sensor node, with the node’s configuration parameters. Node-5 replies with a DATA message with the payload:

period=20

for Node-1 as its destination (in the DST field of the message) but the message is sent (e.g., over BLE) to Node-4 (11), and then forwarded to Node-3 (12), Node-2 (13) and finally to Node-1 (14).

Node-1 can now start sending sensor readings every 20 minutes (message 15 from Fig. 5, Node-1 knows the time unit). Node-1 can use a DATA message (e.g. “reading=123”) or, if a result can be fitted into one byte, a QUICK message (e.g., 0x7B meaning “d=123”).

## Example 4. Security and privacy

The next example demonstrates using security of the underlying transport protocol where it is available, but also usage of additional security mechanisms used by the proposed metaprotocol when the underlying transport protocol is not enough.

For this purpose, the same nodes will be used as in Example 3 from Fig. 5. In this example, secure communication between Node-1 and Node-5 is requested, without possibility that other parties that may be included in the message transfer can view the message content (except for nodes which are security enablers). E.g., since Node-2 communicates with Node-3 using TCP/IP, there are usually a lot of nodes (routers) between them that carry their packets. None of them should be able to see the message content.

If all underlying transport protocols (between Node-1 and Node-2, between Node-2 and Node-3, etc.) support sufficiently secure mechanisms, then using those mechanisms is required through the flag C in the message header, but that is only one part of the security architecture. Let us demonstrate security mechanisms with two examples.

In these examples let us assume that Node-1 from Fig. 5 must securely send message (e.g. with payload “data=56”) to Node-5. Node-1 and/or Node-5 might have cryptographic abilities or not, or defer using them to conserve power.

In the first example let us assume that both nodes, Node-1 and Node-5, can secure messages through the metaprotocol, as presented in the subsection *III.D.* In this example, only nodes Node-1 and Node-5 can see the message payload in its original (decrypted) form. Nodes Node-2, Node-3 and Node-4 should use a secure transport mechanism (e.g. TLS, DTLS) when transporting the message since flags C and/or A are set in the message header, but they can’t decipher and/or verify the payload (unless they have appropriate keys for Node-1 and Node-5).

In the second example, Node-1 cannot use security mechanisms (e.g. its a very simple node without required capabilities). In this case it is up to Node-2 to know that, and (as a trusted node/gateway) provide the requested security. Node-2 must have explicit rules which will trigger when a message arrives from Node-1 and has flags C/A set. Also, complementary rules must exist when a message arrives for Node-1 (rules are described in the subsection *IV.E*). Node-2 acts in the name of Node-1 and therefore must have its credentials (e.g. have keys for Node-1 and the destination node). Node-5 can be capable of handling security by itself or, if not, Node-4 must have similar rules as Node-2 to handle security for Node-5. The presented example assumes that in the local network (up to the gateway, Node-2 and Node-4 in this example) security is achieved differently (e.g. by physically unreachability or by lower level protocols) and that only transmission over public network is untrustworthy.

When a node with security capabilities receives a message, it can accept the message for further processing or drop the message, if it does not have expected security protection, with implicit rules, as described in the subsection *III.D*, or handle it with additional explicit rules (which can accept or reject messages). It is up to the sender to detect that its message is not received (e.g. because the acknowledgement message is not sent back to him), and determine why – is it just a connection issue or a security issue. In the proposed architecture, we did not model sending error messages on such occasions, but that, if required, can be achieved through more complex rules.

## Example 5. Simplified application design with “protocol middleware”

**

Fig. 6. Architecture implementation guidelines

The next example presents a probable expansion of the described communication protocol with appropriate middleware which implements common operations on messages. Like the TCP and UDP protocols provided by the network layer in operating systems, the proposed middleware can provide operations for message manipulation (creation, verification, data extraction, sending, receiving, rule processing, ...).

In memory-constrained devices, such middleware will contain only a subset of operations that are required on such devices (e.g. no encryption support, only basic message types, etc.).

On devices with enough memory, more operations can be incorporated into the appropriate middleware. E.g., a simple device, that reads its sensor and sends its readings to the default gateway every hour, or on explicit request, could be programmed with the following pseudocode:

program

register\_event(ON\_MESSAGE, on\_message)

loop

default\_send(read\_sensor())

delay 1 hour

on\_message(message)

default\_send(read\_sensor())

Operations register\_event and default\_send should be provided by the middleware.

To simplify device application design even further, a general service can be created that supplements the proposed middleware. That service should be able, for most use cases, perform operations required by the device without the need for an additional application. All required behaviors (for most use cases) can be defined through device configuration, using *rules*. Basic rules can be classified into periodic rules, on-message-reception rules, on-message-delivery rules, etc. Basic set of actions could include reading a sensor into some file or table, writing a value to some peripheral (e.g. actuator), sending data to the default or a specific node, actions (activation and deactivation of other rules) on a sensor event, on message reception, and similar. Rules could be defined through editing the rule table in SQL or with a graphical interface, like in the prototype implementation. A textual representation of the interface is used in the following descriptions since its more descriptive (simpler).

A device like the previous one, but implemented with the described middleware, could have the following configuration:

RULE 1:

type = PERIODIC

period = 3600

action:

READ\_SENSOR (TEMP\_25, TABLE\_0)

SEND\_DATA (DEFAULT\_GW, TABLE\_0)

RULE 2:

type = ON\_MESSAGE

FILTER (MESSAGE.type == MSG\_TYPE\_SELECT)

action:

READ\_SENSOR (TEMP\_25, TABLE\_0)

SEND\_DATA (MESSAGE.from, TABLE\_0)

RULE 3:

type = ON\_MESSAGE

FILTER (MESSAGE.type == MSG\_TYPE\_HELLO)

FILTER (MESSAGE.flags & MSG\_FLG\_ACK != 0)

action:

SEND\_DATA (MESSAGE.from, EMPTY\_MESSAGE)

The middleware for advanced nodes (e.g., gateways) could have more abilities for filtering and forwarding messages, using databases with information about sensor nodes, which nodes to forward messages from and where to forward them. The following example configuration can be used in such a node.

RULE 1:

type = ON\_MESSAGE

FILTER (SQL\_CHECK\_FORWARD\_FROM (MESSAGE.from))

FILTER (SQL\_CHECK\_FORWARD\_TO (MESSAGE.dest))

action:

RUN\_SQL ("SELECT real\_dest FROM table\_fwd

WHERE from\_id =" + MESSAGE.from, RESULT\_0)

FORWARD\_MESSAGE (MESSAGE, RESULT\_0)

## Architecture implementation guide

A guide to implementation of the proposed architecture is shown on Fig. 6. with a UML activity diagram.

If a new system is to be created, first some overall system architecture should be defined. Otherwise, existing architecture can be used, possibly with some refinement. Next is the process of adding or updating the elements. For simpler nodes (things), simple programs can be used. Advanced nodes should use some middleware (e.g., like the one presented in this paper and implemented in the prototype) which can be configured with implicit and explicit rules. The process is repeated until a satisfactory system is created. The same implementation model can be used on already existing systems, when adding nodes to them or changing some node behavior. Change in one node should not require changes in many other nodes, or any, since the implicit metaprotocol behavior can be enough in most cases. It is analogous to the Internet: when a new computer connects it should get its IP address, no other changes are required on other computers and equipment, which are already on the Internet.

# Discussion

The proposed architecture is mostly based on a rule system, many of them implicit, which define the basic architecture behavior, with ability to add custom rules. However, there are several differences with similar systems, cited in Section II.

Some of those systems [23,30] are designed only for a specific cause (home automation, athletic training). Many of them require the user to learn a special language to be able to use them (partially [21,22], completely [25,29]). All of those systems (except [22] to some extent) can be used only in local networks. Next, all of them require an additional abstraction layer between the devices and the system, a layer which converts data to messages and vice versa. All the cited systems (except [21] to some extent) have strictly separated node categories (e.g. sensors, routers, servers). None of the previous systems enable a dynamic role (and complexity) change of a node by changing only its settings. Insertion of a simplest node, and then gradually making it more complex, is not possible in either of those systems.

The proposed architecture is, on the other hand, not used only for a specific cause but is built for general purpose IoT systems. Since SQL is used for message composition, there is no need to learn an additional programming language. In further development, an interface can be created that automates rule generation, using only data operations, understandable to the common user. Messages in the proposed metaprotocol are crossing over different networks and protocols unchanged, and there is no need to create an additional layer or converter for that.

Changing node settings (through the database or a user interface) for a dynamic change of the node’s role can be simple. A simple node may get additional roles in such a way or can be relieved of some operations. In this way, through some changes in some nodes, the architecture of the whole system may be dynamically adapted for possibly updated system requirements.

Today there are many commercial solutions for Internet of Things, like Microsoft Azure [[[32]](#endnote-31)], Amazon Web Services [[[33]](#endnote-32)], and IBM IoT Platform [[[34]](#endnote-33)]. These complete full-stack cloud-based application sets enable advanced analyses of collected data. Their architecture is strictly layered, with hierarchical node organization, services in cloud being at the top of hierarchy. Many systems can be built with such an architecture, but not all. Their flexibility is limited, and we do not consider that architecture to be general enough. Nevertheless, it is possible to combine such a system with a system that uses the proposed architecture. E.g., some commercial platform can be used for advanced data processing, for user interfaces, and similar parts of the wider IoT ecosystem. The proposed architecture could interoperate with many other architectures if an additional module that converts protocols (e.g., to/from MQTT) is provided.

TABLE III.

Operations related to addition of simple node in existing IoT system

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Architecture** | **Thing level** | **Gateway level** | **Server level** | **User level** |
| Common 4‑layer architecture | - operation of the thing  - configuration for communication with remote node | *- optional configuration of the gateway for message processing and forwarding* | - inserting added node parameters to the server system  *- optional higher-level logic* | - inserting settings for communication and message processing for added node on user application |
| Proposed architecture | - operation of the node with thing role  *- optional configuration for communication with remote node if implicit settings are not enough* | *- optional configuration of the node with gateway role for message processing and forwarding if implicit settings are not enough* | *- optional configuration for added node in the node with server role if implicit settings are not enough*  *- optional higher-level logic* | *- optional insertion of added node settings for communication and message processing on node with user role if implicit settings are not enough* |

For many users, using a commercial solution could be too costly and/or too complex to implement. The proposed architecture does not require dedicated (web) servers, database servers, or similar costly elements. The control can be embedded within things or simpler controlling nodes. Although an ordinary user cannot create such systems by himself, after an engineer designs its network and sets it up, the user can later expand this system with additional things and operations. Or the user can do it himself by copying an existing architecture made for similar systems and adapting it for his needs.

The proposed architecture does not include advanced analysis features or advanced graphical representations, present in most commercial solutions (although a simple web interface is included in the prototype implementation). Some such modules could be created and added into the proposed architecture as a part of some advanced node, but it is not included in this initial architecture since the emphasis was on simpler nodes. Also, the prototype implementation [[[35]](#endnote-34)] is currently not for much more than testing, and it needs to be further developed and tested, also simplified for deployment and usage.

## Use cases

Possible scenarios where proposed architecture should have a clear advantage over other architectures include systems where continued evolution and adaptation is mandatory. Since users change their expectation of the system as they continually learn more about its possibilities (or what other users do in similar systems), more and more systems are expected to require continuous evolution.

An illustration of using the proposed architecture for creating a new system follows. The starting point is a gateway node which will, for starters, collect all data from the local network. Adding a thing (a node) in such systems could be as simple as programming it to send its reading within the simplest of messages (e.g., just one byte of information, encapsulated in the local datalink-layer protocol). The gateway could initially process the received message and possibly generate some action (defined through the rule system supported by its middleware). When (if) later another node is added that can do advanced processing or store messages, the gateway should be updated by adding rules that forward some (or all) messages to that advanced node (e.g., some server in the cloud). Things in the local network do not need to be aware of any network beyond the gateway, and they just transmit its data, or process and act upon received messages from the gateway, or other local things. Changing a node complexity (e.g., the node can send more data, or request data) does not necessary require changes in any node, except perhaps a server that needs to respond to such a message (which should be done by some rule).

After prototyping the local network, further optimizations are supported. E.g., messages can be shortened if energy efficiency is needed, or security can be added if the network is being deployed outside of the current controlled environment, or rules can be added which filter received messages, or nodes which require data can be added, or users can be added through a node with a web interface. The local network can further evolve with additional nodes, changes in configuration and connectivity, subnetworks, changing node roles, and similar.

For the next step, a connection with a remote node might become needed if the local network should be a part of a larger system. Such connectivity could be implemented through local changes in some node (e.g., the gateway) mostly only through rules. More complex network changes might require new SQL queries (in databases, messages). A connection to other systems does not require changes in most nodes in the local network. Furthermore, the evolution of the local network could be continuous and unhindered by changes in connectivity to other networks (and changes within them).

Using other architectures for these scenarios would mean juggling with much more technologies, i.e. if evolution would even be supported, also if their complexity would be appropriate for resource-constrained devices, also if a single architecture could even be used, also if the different data models on different levels could be unified in a satisfactory way, and finally if there is a will to completely replace the software when a role is changed. Of course, if the required architecture does not need evolution and it fits some other architecture type, using it would probably be easier than using the architecture proposed in this paper, since that architecture would be optimized for such use cases.

TABLE IV.

Protocol overhead comparison

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Payload/**  **protocol** | **1 B**  **rarely** | **20 B**  **rarely** | **100 B**  **rarely** | **100 B**  **often** |
| MQTT | 64 | 83 | 163 | 123 |
| Metaprotocol with minimal header | 1 | 21 | 101 | 101 |
| Metaprotocol with full header | 21 | 40 | 120 | 120 |

Today an accent is put on quickly prototyping networks (pervasive IoT networking) not by creating complex systems from the go, but starting with systems as small as possible, as per the current needs of the user, and then optimizing or expanding the system or making it more complex and secure. Whatever the change may be, when using the proposed architecture, it is perceived as a normal increment without a significant system-wide change.

Adding a thing to an existing system may require additional changes in most IoT architectures. In the traditional architecture, all levels are strictly separated, and they have different interfaces between one another. In the proposed architecture there are no strict layers, and a node can have more than one role, and there is only one interface. Changes in IoT system elements when adding a new thing to it are illustrated in Table III.

## Communication overhead

To compare communication overhead of the proposed architecture using the defined metaprotocol with usual architectures, we use a comparison with the MQTT protocol. Similar discussion is also valid for CoAP and similar protocols.

The simplest node that we use here in this analysis occasionally needs to send a single value to the “server” node, where the application logic is implemented.

When MQTT is used, a node must have software support for it. Since MQTT is mostly implemented over TCP/IP, support for that stack must also be included in the node software (of course with lower datalink-layer protocol). That will significantly increase hardware demands for a node. In comparison, if the proposed architecture is used, a simple node could use a datalink-layer protocol (e.g., BLE) as its underlying protocol to send its data to the next node using the proposed metaprotocol. A very simple thing can have only a small subset of the proposed metaprotocol included, e.g., only a method to create and send a message.

To begin communication using MQTT, a client needs to send a CONNECT message, which will be replied by a CONNACK message. Then a PUBLISH message will be sent which will carry useful data. If the node does not keep the connection alive (the node rarely sends messages), a final DISCONNECT message must be sent. Otherwise, next useful data is sent with another PUBLISH message.

The CONNECT message is at least 31 bytes long on the application layer (2 bytes for Fixed Header + 11 bytes for minimal Variable Header + 18 bytes for EUI-64 encoded as “ClientID” using “UTF-8 Encoded String”). The CONNACK message is at least 5 bytes long on the same layer (2 bytes for Fixed Header + 3 bytes for minimal Variable Header). The PUBLISH message is at least 24 bytes long (2 bytes for Fixed Header + 21 bytes for minimal Variable Header including “<EUI-64>.d” encoded as “Topic Name” using “UTF-8 Encoded String” + 1 byte for Payload including the data). The DISCONNECT message is at least 4 bytes long (2 bytes for Fixed Header + 2 bytes for minimal Variable Header). Therefore, for all those messages, 31 + 5 + 24 + 4 bytes of traffic are required (plus payload size). Just the PUBLISH message has an overhead of 23 bytes.

In comparison, when the proposed architecture and the metaprotocol are used, since all message elements, beside the header can be omitted, the total overhead can be as small as that of a header. Even the header can be omitted in one-byte messages. When all message elements are used (header, id, length, sender and destination id of sender, CRC), the total overhead is 20 bytes.

The communication-overhead comparison is illustrated in Table IV. It can be seen that the metaprotocol is optimized for shorter messages and simpler nodes because there its advantage over MQTT is the greatest.

## Implementation notes

The prototype middleware has been implemented and hosted on Github [34]. Programming languages C++14 and PHP7 are used, with PostgreSQL for the database and Apache Web Server for web configuration. The implementation currently supports only Linux.

The middleware is implemented as follows: each UPM has a sending thread and a receiving thread. The sending thread has a message queue from which it reads, and the receiving thread writes messages into the main thread’s message queue. Also, actions can come from either database triggers or web configuration, and these are also queued so only one thing can be processed in the main thread simultaneously. The structure of the middleware can be seen on Fig. 3. Threads and message queues are not shown for simplicity.

Currently some parts of the described architecture are not yet implemented. Those include parts of duplicate detection, updating example scenarios to the described specification, parts of broadcast handling, security for some UPMs, and others. The list of implemented and unimplemented features can be found in project’s repository [34].

# Conclusion

In this paper, a new IoT architecture is proposed. The primary design goal was simplicity of implementation for simpler nodes who can communicate with a local or remote node as simply as possible, unburdened with communication protocols and network configuration. Nodes that help in this communication (using the proposed metaprotocol and architecture) can do that with simple (implicit) or advanced (custom) rules, which transform, forward, store messages, or do some other operation on them, depending on the message content (e.g., sender id, destination id).

The proposed message format is mostly based on SQL, enabling simple and advanced information transfer and command exchange. Simple messages are data messages or simple queries – commands requesting new readings or other operation from things. Advanced SQL messages (queries) may contain a custom designed query that produces the required result on the destination node and requests it back to sender. Such message formats enable creation of various systems, from standard layered architectures (thing-gateway-server-client) to custom systems with distributed nodes and roles.

Usage of the proposed architecture is demonstrated on several examples, both simple and more complex. Analysis of the proposed architecture and comparison with similar systems shows some advantages concerning simplicity of use and flexibility in system implementation.

A prototype middleware for proposed architecture is implemented and used to test its features from various perspectives, from usage in datastore and processing nodes, which can use its full potential, to simple nodes, which will need only a small fraction of its potential.

The proposed architecture does not exclude others, it can be used with other (commercial) solutions, thanks to its flexible table-based data model.

##### References

1. This work has been fully supported by the Croatian Science Foundation under the project IP-2019-04-4864. [↑](#footnote-ref-1)
2. [] L. Atzori, A. Iera, and G. Morabito, “The internet of things: A survey”, *Comput. netw.*, vol. 54, no. 15, pp. 2787-2805, 2010. [↑](#endnote-ref-1)
3. [] J. Lin, W. Yu, and N. Zhang, “A Survey on Internet of Things: Architecture, Enabling Technologies, Security and Privacy, and Applications”, *IEEE Internet of Things J.*, vol. 4, no. 5, pp. 1125-1142, 2017. [↑](#endnote-ref-2)
4. [] K. Ashton, “That 'internet of things' Thing”, *RFID j.*, vol. 22, no. 7, pp 97-114, 2009. [↑](#endnote-ref-3)
5. [] B. L. R. Stojkoska and K. V. Trivodaliev, “A review of Internet of Things for smart home: Challenges and solutions”, *J. Cleaner Prod.*, vol. 140,, no. 1, pp. 1454-1464, 2017. [↑](#endnote-ref-4)
6. [] M. Zorzi, A. Gluhak, and S. Lange, “From today's INTRAnet of things to a future INTERnet of things: a wireless- and mobility-related view”, *IEEE Wireless Commun.*, vol. 17, no. 6, pp. 44-51, 2010. [↑](#endnote-ref-5)
7. [] P. C. Mugauri, K. Aravind, and A. Desmukh, “A Survey on Applications of Internet of Things in Healthcare Domain”, *Res. J. Pharmacy and Technol.*, vol. 11, no. 1, pp. 93-96, 2018. [↑](#endnote-ref-6)
8. [] Y. Yang, L. Wu, and G. Yin, “A Survey on Security and Privacy Issues in Internet-of-Things”, *IEEE Internet of Things J.*, vol. 4, no. 5, pp. 1250-1258, 2017. [↑](#endnote-ref-7)
9. [] T. Qiu, N. Chen, and K. Li, “How Can Heterogeneous Internet of Things Build Our Future: A Survey”, *IEEE Comm. Surveys & Tutorials*, vol. 20, no. 3, pp. 2011-2027, 2018. [↑](#endnote-ref-8)
10. [] L. Oliveira, J. J. Rodrigues, and S. A. Kozlov, “MAC layer protocols for internet of things: A survey”, *Future Internet*, vol. 11, no. 1, pp. 16-57, 2019. [↑](#endnote-ref-9)
11. [] IEEE, “IEEE Standard for Low-Rate Wireless Networks”, IEEE Std 802.15.4-2015, 2015. [↑](#endnote-ref-10)
12. [] ZigBee Alliance, “ZigBee Specification”, ZigBee Document 05-3474-21, 2015. [↑](#endnote-ref-11)
13. [] Thread Group, “Thread 1.1 Specification”, Thread, 2016. [↑](#endnote-ref-12)
14. [] IETF, “Transmission of IPv6 Packets over IEEE 802.15.4 Networks”, RFC 4944, 2007. [↑](#endnote-ref-13)
15. [] Silicon Labs, “Z-Wave Specifications”, Silicon Labs DKD13867, 2020. [↑](#endnote-ref-14)
16. [] IETF, “The Constrained Application Protocol (CoAP)”, RFC 7252, 2014. [↑](#endnote-ref-15)
17. [] IETF, “Extensible Messaging and Presence Protocol (XMPP): Core”, RFC 6120, 2011. [↑](#endnote-ref-16)
18. [] OASIS Group, “MQTT Version 5.0”, OASIS, 2019. [↑](#endnote-ref-17)
19. [] M. Asim, “A survey on appliaction layer protocols for Internet of Things (IoT)”, *Int. J. Adv. Res. In Comput. Sci.*, vol. 8, no. 3, pp. 996-1000, 2017. [↑](#endnote-ref-18)
20. [] ISO/IEC, “Information technology - Database languages - SQL - Part 2: Foundation (SQL/Foundation) Ed 5”, ISO/IEC DIS 9075-2, 2016. [↑](#endnote-ref-19)
21. [] W3C, “SPARQL 1.1 Protocol”, W3C Recommendation, 2013. [↑](#endnote-ref-20)
22. [] N. Park, H. Lee, and J. Jang, “Rule-based modeling tool for web of things applications”, in *Proc. 2015 IEEE 5th Int. Conf. Consumer Elect. - Berlin (ICCE-Berlin)*, Berlin, Germany, 2015, pp. 515-518. [↑](#endnote-ref-21)
23. [] C. E. Kaed, I. Khan, and A. Van Der Berg, “SRE: Semantic Rules Engine for the Industrial Internet-Of-Things Gateways”, *IEEE Trans. On Indust. Inf.*, vol. 14, no. 2, pp. 715-724, 2018. [↑](#endnote-ref-22)
24. [] D. Mazzei, G. Fantoni, and G. Montelisciani, “Internet of Things for designing smart objects”, in *Proc. 2014 IEEE World Forum Internet of Things (WF-IoT)*, Seoul, South Korea, 2014, pp. 293-297. [↑](#endnote-ref-23)
25. [] L. Yao, Q. Z. Sheng, and S. Dustdar, “Web-Based Management of the Internet of Things”, *IEEE Internet Comput.*, vol. 19, no. 4, pp. 60-67, 2015. [↑](#endnote-ref-24)
26. [] L. Yangqun, “A Light-Weight Rule-Based Monitoring System for Web of Things”, in *Proc. 2013 Int. Conf. Cyber-Enabled Dist. Comput. and Knowledge Discovery*, Beijing, China, 2013, pp. 251-254. [↑](#endnote-ref-25)
27. [] T. Toumisto, T. Kymäläinen, and J. Plomp, “Simple Rule Editor for the Internet of Things”, in *Proc. 2014 Int. Conf. Intel. Environments*, Shanghai, China, 2014, pp. 384-387. [↑](#endnote-ref-26)
28. [] A. Monge Roffarrello, “End User Development in the IoT: A Semantic Approach”, in *Proc. 2018 14th Int. Conf. Intel. Environments (IE)*, Rome, Italy, 2018, pp. 107-110. [↑](#endnote-ref-27)
29. [] I. Hwang, M. Kim, and H. J. Ahn, “Data Pipeline for Generation and Recommendation of the IoT Rules Based on Open Text Data”, in *Proc. 2016 30th Int. Conf. Adv. Inf. Netw. and Appl. Workshops (WAINA)*, Crans-Montana, Switzerland, 2016, pp. 238-242. [↑](#endnote-ref-28)
30. [] H. Hossayni, I. Khan, and C. E. Kaed, “Embedded Semantic Engine for Numerical Time Series Data”, in *Proc. 2018 Global Internet of Things Summit (GIoTS)*, Bilbao, Spain, 2018, pp. 1-6. [↑](#endnote-ref-29)
31. [] B. R. Baricelli and S. Valtolina, “A visual language and interactive system for end-user development of internet of things ecosystems”, *J. Vis. Lang & Comput.*, vol. 40, no. 1, pp. 1-19, 2017. [↑](#endnote-ref-30)
32. [] Microsoft. “Azure IoT - Internet of Things Platform | Microsoft Azure.” microsoft.com. https://azure.microsoft.com/en-us/overview/iot/ (accessed Feb. 1. 2021). [↑](#endnote-ref-31)
33. [] Amazon. “AWS IoT - Amazon Web Services.” amazon.com. http://aws.amazon.com/iot/ (accessed Feb. 1, 2021). [↑](#endnote-ref-32)
34. [] IBM. “Internet of Things | IBM.” ibm.com. http://www.ibm.com/cloud/internet-of-things (accessed Dec. 1, 2020). [↑](#endnote-ref-33)
35. [] L. Milić. “IoT.” github.com. https://github.com/lukamilicfoi/IoT (accessed Feb. 1, 2021). [↑](#endnote-ref-34)