Scenario 2

…

1

2

3

…

4

Figure 2. Scenario 2

There is only a temperature sensor which has granularity small enough that it fits in one byte. E.g., it can only send integer numbers between 0 and 100 (Celsius). Every now and then it sends over BLE a packet („1“, later „…“) which consists only of one byte '\x<number to hex>' as it is described in the Protocol specification in the **QUICK** MT. Everything else is inferred from already described parameters. The Gateway is configured similarly like in the Scenario 1. There is a local server that is wire-connected.

On the Gateway it is configured that over the wireless link it only receives messages with imm\_SRC equal to the BLE EUI of the sensor ‒ not used for testing. There is also a table created such that the virtual column called "d(efault)" maps to the real column "temp". This can be accomplished by a message-modify rule on receiving which expands one-byte messages (to "temp=<value>") or by creating virtual columns (*GENERATED* columns) in more advanced databases – the former is used for testing. As the data type of the column *INTEGER* is put (a 4-byte integer). The local server when it boots sends a **SELECT** query („2“)

"SELECT ALL a.temp

‗FROM tabababababababab AS a

‗WHERE a.temp > 20 SUBSCRIBE 1;" (76 bytes)

coded as

"\xDF\x85a.temp

\xADt\xAB\xAB\xAB\xAB\xAB\xAB\xAB\xAB\x87a

\xFCa.temp>20\xE71" (32 bytes).

When it shuts down it can **UNSUBSCRIBE** („4“), which can be accomplished by "UNSUBSCRIBE 1;" like "\xF71". Or it can be configured that if the **ACKNOWLEDGMENT** is not received a few times, there is an automatic unsubscription – not used for testing.

To send Messages the server had subscribed to („3“, later „…“), e.g., on the Gateway it is configured that the word **SUBSCRIBE** means that a check is expected every time a **DATA** Message is received. Then it is best to automatically create some view (like SQL's *CREATE VIEW*) and then some trigger (like SQL's *CREATE TRIGGER*) on its change. That is not needed to be done by the User. He only has to configure the meaning of **SUBSCRIBE**.

Currently, as mentioned before, BLE is untestable so UDP is used (TCP must have LEN so QUICK is impossible). Also, as mentioned before, there is no discerning of Messages by protocol or interface (there are no rules operating on raw messages, but will be added). The meaning of **SUBSCRIBE** is currently not configurable. Also, automatic unsubscription is not yet supported as of now.

Scenario 3

…

1

2

3

4

5

6

7

8

9

10

11

12

13

Figure 3. Scenario 3

A Thing wants to find out how often it needs to send a reading to its Gateway and it sends

"SELECT ALL min FROM t0102030401020304

‗FETCH FIRST 1 ROW ONLY;"

to a server on the DST 0x0102030401020304 („1“). On the link layer BC is set, on the Protocol the DST is set, also the SRC is set or injected by a Gateway (the former is used for testing) in order to get a reply since imm\_SRC cannot be used, a Gateway catches the packet by a wireless protocol and it forwards it to the Cloud („2“).

The Cloud knows where the destination is and it sends the packet to its Gateway („3“) and it forwards it to the destination („4“). Routing is possible because either the route is given through manual Node configuration, or the needed **HELLO**s were sent – the former is used for testing. The destination is able to recognize the query (e.g., it just scans for "SELECT" and "min") and it sends the current configuration of the minute number as a **DATA**

"min=15"

to the SRC by the reverse route („5“-„8“). The return trip is possible because imm\_SRCs for the SRC get remembered and used as imm\_DSTs when it becomes the DST now. The source recognizes (e.g., it just strips the first 4 characters) and reconfigures. It begins to send **DATA** Messages to its Gateway every 15 minutes („13“). The source can here also send an **ACKNOWLEDGMENT** to the destination („9“-„12“) if the SRC was set, K bit was raised, and ID was either set or calculated (the former is used for testing).

It can be seen that the part of the **DATA** with column names is unnecessary (e.g., if the first character of the PL is not a-z, then the columns are evidently left out, as described elsewhere – not used for testing). Also the *FROM* clause of the *SELECT* might be unnecessary (e.g., if there is not one, it is read from the DST, as described elsewhere – not used for testing).

Wireless is still emulated using TCP.

Scenario 4

1

2

3

5

6

…

4

Figure 4. Scenario 4

A Thing does not send Data unless someone asks it for them. It has, e.g., readings r1, r2, and r3. On a mobile phone it is configured that every 15 minutes all readings are requested. The mobile phone connects to the Gateway and asks the Gateway for all readings from the Thing 0x0102010201020102 („1“, later „…“). The Gateway knows what it is about and it sends a **SELECT** query to the Thing („2“)

"SELECT \* FROM t0102010201020102;".

This is accomplished by a message-modify rule on receiving (when someone asks for the table in question, forward the request by changing DST). The Thing does not have a database implemented, but a such simple query it understands (e.g., it just scans for "SELECT" and "\*") and it sends all readings as one row of a table. The Gateway receives **DATA** (3)

"r1,r2,r3=1,2,3"

and it forwards it to the mobile phone („6“). The mobile phone displays something to the User. The Thing can here, e.g., send an **UNSUPPORTED** Message if it gets a too complex query – not used for testing.

The Gateway can also cache the response from the Thing („4“) for some further purposes (e.g., if it knows that the Thing always sends almost completely the same thing). This is accomplished by a message-inject rule on receiving (when the Thing responds, inject a copy of the Message with DST equal to the Gateway's address). If it is configured so, the Gateway can also do other things, e.g. automatically forward („5“) to someone else (e.g., if it wants to archive the Data on the Cloud). This is accomplished by a message-send rule on receiving (when the thing responds, send a copy of the Message with DST equal to the Cloud's address).

Currently, caching is simply storing and it is done without any special purpose.

Scenario 5

10

16

17

1

2

3

4

5

6

7

8

9

11

12

13

14

15

18

Figure 5. Scenario 5

A Thing wants to send Data to another Thing which is on the other side of the world and it also wants that the Message is resistant to on-line reading and tampering. The thing securely connects to its BLE Gateway and it sends a Message also specifying C and A bits are needed, not forgetting to include DST and SRC („1“).

The Gateway knows how to send the Message securely over TCP+TLS to its Cloud („2“). The Cloud then decides to send the Message to another Cloud („3“). The other Cloud realizes it does not have a secure link to the other Thing's Gateway and it silently drops the message. Routing is done like in the Scenario 4.

The original Thing realizes after some time that the Message was not received because the K bit had been raised and an **ACKNOWLEDGMENT** has not been received. It then pings the Thing with a **HELLO** with the K bit raised and then realizes no secure connection exists after receiving the **ACKNOWLEDGMENT** (4-13). The return trip is done like in the Scenario 4. Then it can decide whether to send the message insecurely („14“-„18“) or wait and try again another time – the former is used for testing.

Currently, Thing connects unsecurely to the Gateway 1, BLE is emulated using TCP without TLS, and the Message is dropped in Gateway 2 instead of the Cloud 2 because TLS cannot be turned off in Cloud 2 for now.

Scenario 6

1

…

Figure 6. Scenario 6

There are two Nodes – a smart sensor (Sensor) and a smart display (Display). Both of them run our Gateway software. The Sensor has a rule that every 20 seconds it executes a command for reading the physical sensor and using the software for sending Data to the Display („1“, later „…“). The Display has a rule that whenever a Message with format "sensor=<value>" is received, it forwards that value to the command for physically displaying the value (e.g., it just strips the first 7 characters).

This configuration does not need any external programs to be run; only to define a timed rule at one Node and a command-execute rule at another. Since these rules can be defined from the Web interface, nothing apart from the software being run is needed. Of course, commands for reading the sensor and writing the value need to be defined.

It would be the best if the Protocol remained the same, so the physical sensor understood "\0SELECT \*" and returned "\0sensor=<value>", which the physical display would understand. That would need to be described in their documentation as an implementation of our Protocol. Unfortunately, hoping to unify the interface of all the sensors in the world currently is completely far-fetched. Still, benefits of the unification would be great.

Currently, timed rules are not a part of the software but emulated using SQL commands calling external shells for timed loop. That would of course be changed in future iterations of the system. Reading the sensor is currently a script that generates and returns a random value and writing the display is currently a script that writes a message with the forwarded value to the standard output.