MQTT/UDP Documentation

Release 0.5-0

Dmitry Zavalishin

Apr 25, 2022

Contents

CHAPTER 1

Introduction

MQTT/UDP is a simplest possible protocol for IoT, smart home applications and robotics. As you can guess from its name, it is based on MQTT (which is quite simple too), but based on UDP and needs no broker.

> ***Network is a broker***
>
> Your network does most of the broker's work. That is why MQTT/UDP implementation can be so simple, but full featured.

Fast track for impatient readers: MQTT/UDP native implementations exist in Java, Python, C, Lua and PLC specific ST language. See corresponding references:

- *C Language API Reference*
- *Java Language API Reference*
- *Python Language API Reference*
- *Lua Language API Reference*
- *CodeSys ST Language API Reference*

If you want to test MQTT/UDP on a real hardware, take a look at *Sketches* part. Ready made software is described in *Integration and tools* part.

Now some words on MQTT/UDP idea. It is quite simple. Broker is a single point of failure and can be avoided. Actual traffic of smart home installation is not too big and comes over a separated (by firewall) network. There are many listeners that need same data, such as:

- main UI subsystem (such as OpenHAB installation)
- special function controllers (light, climate units)
- per-room or per-function controllers (kitchen ventilation, bath room sensors, room CO2 sensors, etc)
- in-room displays (room and outdoor temperature)

All these points generate some information (local sensors, state) and need some other information.

> ***CAN for Ethernet***
>
> By the way, CAN bus/protocol is made for quite the same requirements, but is not good for TCP/IP and Ethernet. Actually, to some extent, MQTT/UDP is CAN for Ethernet.

So, MQTT/UDP is sending data with UDP broadcast. It means that every message is simuloneusly sent to all possible recipients with just one network packet.

Every listener selects packets it wants to listen to and processes them as it wishes.

As a result, minmal MQTT/UDP implementation is extremely simple. Though, there are more options exist which are described later.

Main use cases for MQTT/UDP are covered below.

1.1 Data exchange

Main and, for most applicartions, the only use case. It is really simple. Sender transmits one PUBLISH packet per message. Packet contains topic name (such as "rooms/dinner/temperature") and value. Value can be text string or binary data, but most programs will wait for text as packet value.

As there is no broker, parties do not need such things as CONNNECT, SUBSCRIBE or anything else but PUBLISH message.

All the MQTT/UDP programs on the network will receive message and decide if they need it.

1.2 Reliable exchange

Sender transmits PUBLISH message with non-zero QoS field. Receiver replies with PUBACK packet. If no acknowledge received, sender re-sends message.

Current libraries do not support this scenario out of the box, but it can be implemented by user code. Later versions of libraries will have this case implemented.

1.3 Data request

There is request-reply scenario possible. Requesting party sends SUBSCRIBE message, one that is responcible for requested topic replies with PUBLISH message.

This scenario can be used for remote configuration use case: configuration daemon keeps set of topics and configuration settings per topic, but does not send them to not to spam network with rarely needed data. Some IoT device turns on and requests topics that contain needed configuration parameters, gets needed settings and continues working.

If configuration settings are changed, config server re-publishes corresponding topics to update device settings.

Please see Java `config.Provider` and `config.Requester` classes for further info.

There is also a ready simple remote configuration server in `tools/config_server`.

1.4 Topic request

There is reverse scenario possible. Remote configuration program can send SUBSCRIBE message for topic, that is a wildcard for all possible configuration topics for device or all devices. Devices should respond back with PUBLISH messages for all the configurable items.

See *Passive remote configuration* for more details.

1.5 Discovery

Party that needs to find who is on the network sends PINGREQ request. All the others reply with PINGRESP messages, and requester builds a map of all active MQTT/UDP hosts on the network.

CHAPTER 2

Possible topologies

Here is a list of more or less obvious scenarios for MQTT/UDP

2.1 Fault-tolerant sensors

Some 2-4 temperature sensors are placed in one room and send updates every 10 seconds or so. Update topic is the same for all the sensors, so that every reader gets mix of all the readings.

Reader should calculate average for last 4-8 readings.

Result: reader gets average temperature in room and failure of one or two sensors is not a problem at all.

Trying to build corresponding configuration with traditional MQTT or, for example, Modbus you will have to:

- Setup broker
- Setup transport (topic names) for all separate sensors
- Setup some smart code which detects loss of updates from sensors
- Still calculate average
- Feed calculated average back if you want to share data with other system nodes

2.2 One sensor, many listeners

IoT network is a lot of parties operating together. It is usual that many of them need one data source to make a decision. Just as an example, my house control system consists of about 10 processing units of different size. Many of them need to know if it is dark outside, to understand how to control local lighting. Currently I have to distribute light sensor data via two possible points of failure - controller it is connected to and OpenHub software as a broker. I'm going to swithch to MQTT/UDP and feed all the units directly.

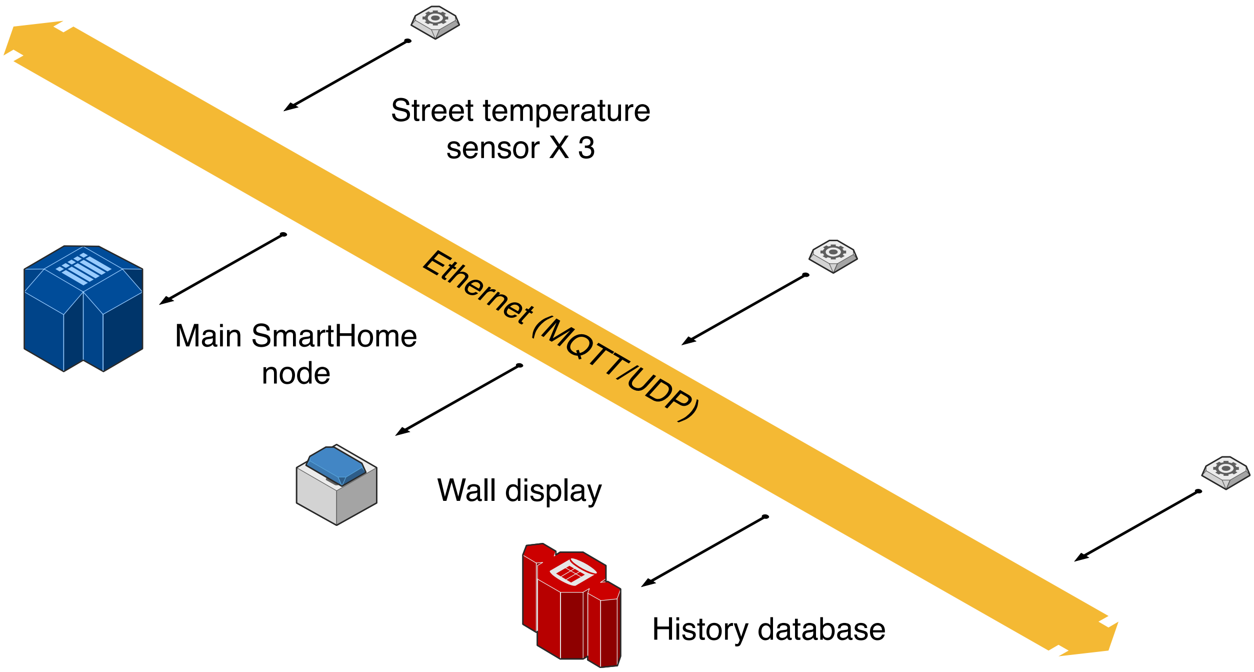

Fig. 1: Typical MQTT/UDP use case.
This diagram shows three sensors duplicating each other. For example, three outer temperature sensors. Wall display, history database and main smarthome unit get copy of all data from sensors. Malfunction of any unit does not make any problemn for others.

2.3 Multiple smart switches

Some wall switches are controlling the same device. All of them send and read one topic which translates on/off state for the device.

Of course, if one switch changes the state, all others read the state broadcast and note it, so that next time each switch knows, which state it should switch to.

It is possible, of course, that UDP packet from some switch will be lost. So when you switch it, nothing happens. What do you do in such a situation? Turn switch again, of course, until it works!

In this example I wanted to illustrate that even in this situation UDP transport is not really that bad.

2.4 Roadside processor

Data processors such as triggers, unit converters, calculators of different kinds can be easily implemented with MQTT/UDP as standalone script or a small program that just listens for required incoming data, performs calculations and sends results back to MQTT/UDP.

Other script or IoT/SmartHome component can then use resulting data.

There is an example of such combination in MQTT/UDP repository. Java program in `tools/tray` is setting up desktop tray informer which displays some MQTT/UDP parameters if user clicks on tray icon.

Companion script `lang/python3/examples/trigger.py` is listening to some topic and if topic value is out of range sends information on `tray/message` topic with a worning. Tray program listens for that topic and displays a warning to user on reception of such a message.

2.5 System debugging

Broadcast/multicast nature of MQTT/UDP lets you see what is going on on the "bus" exactly the same way as all the parties see. There is a simple tool exist for that in this repository, but you can use, for example well known WireShark as well.

CHAPTER 3

Reliability

Note: There's QoS support for MQTT/UDP is in development, which makes it as reliable as TCP version.

As MQTT/UDP is based on UDP protocol, which does not guarantee packet delivery, one can suppose that MQTT/UDP is not reliable. Is it?

Not at all.

If we use it for repeated updates, such as sensor data transfer, UDP is actually more reliable, than TCP! Really. If our network drops each second packet, TCP connection will be effectively dead, attempting to resend again and again outdated packets which are not needed anymore. And MQTT/UDP will just loose half of readings, which is not really a problem for 99% of installations. So, TCP helps just if packet loss rate is quite low.

Actualy, simple test was made[1] to ckeck UDP reliability. One host in my house's local net was generating MQTT/UDP traffic as fast as possible and other checked packets to be sequent, counting speed and error rate. Two IPTV units was started to show HD content and one of the computers was copying some few GBytes to file server. Result was quite surprising: MQTT/UDP error rate grew to… 0.4% with about 50K packets/second, but TV sets stopped showing, being, obviusly, starved.

Anyway, I'm going to add completely reliable mode to MQTT/UDP in near future.

3.1 Speed limit

There is one more reliability issue exist when we use UDP. Low power microcontrollers are quite slow and their ability to receive lots of UDP packets per second are limited. There is possible packet loss due to low processing power of some slow nodes, not because of network delivery is not reliable.

That's why protocol implementations include throttling subsystem, which limits amount of packets sent per time interval.

[1] Corresponding tools are in repository and you can run such test yourself.

By default it is tuned for maximum of 10 packets per second. Java and Python implementations use millisecond timing and send max of 3 packets with no speed limit, and then add 300 msec pause. C implementation currently uses 1 second time granularity and lets application send up to 10 packets with no limit and then waits for a second.

Actual tests of reception speed capability were done with Wemos D1 Mini unit programmed with MQTT/UDP Lua implementation.

There is `set_throttle`/`setThrottle` function in all languages but Lua, which lets you set speed limit according to your hardware capabilities, or disable it at all by setting to 0.

CHAPTER 4

Remote configuration

MQTT/UDP can be used as a remote configuration protocol. There are two basic modes of remote configuration: acive and passive. Active remote configuration mode is needed when node to be configured has no local storage at all and has to request all settings at each restart. Passive mode is good for nodes that keep settings from start to start (in local FS or NVRAM) and configuration process is executed for them once or rarely.

4.1 Passive remote configuration

Passive remote configuration is divided in three steps. First, when user starts configurator GUI application, application sends (and repeats from time to time) a SUBSCRIBE request. Second, any MQTT/UDP nodes that support remote configuration see that request and answer with PUBLISH packet for all items that can be set up. Last, configurator application sends back message with a new item value.

All the configurable items must have special topic name: `$SYS/conf/{node-id}/{kind}/{name}`. For example: `$SYS/conf/024F55A20001/info/uptime`.

4.1.1 Node id

Node id is any string that will not change at least during one program run or node start. If just one MQTT/UDP program is run on this hardware, you can use net MAC address as id. Otherwise some GUID will do. Note that node-id can be configurable item too!

4.1.2 Item kind

There are four predefined kinds:

info This kind is a readonly description which can not be configured but tells other side about us. There are `info/soft` - name of program, `info/ver` - version of program, `info/uptime` - as is.

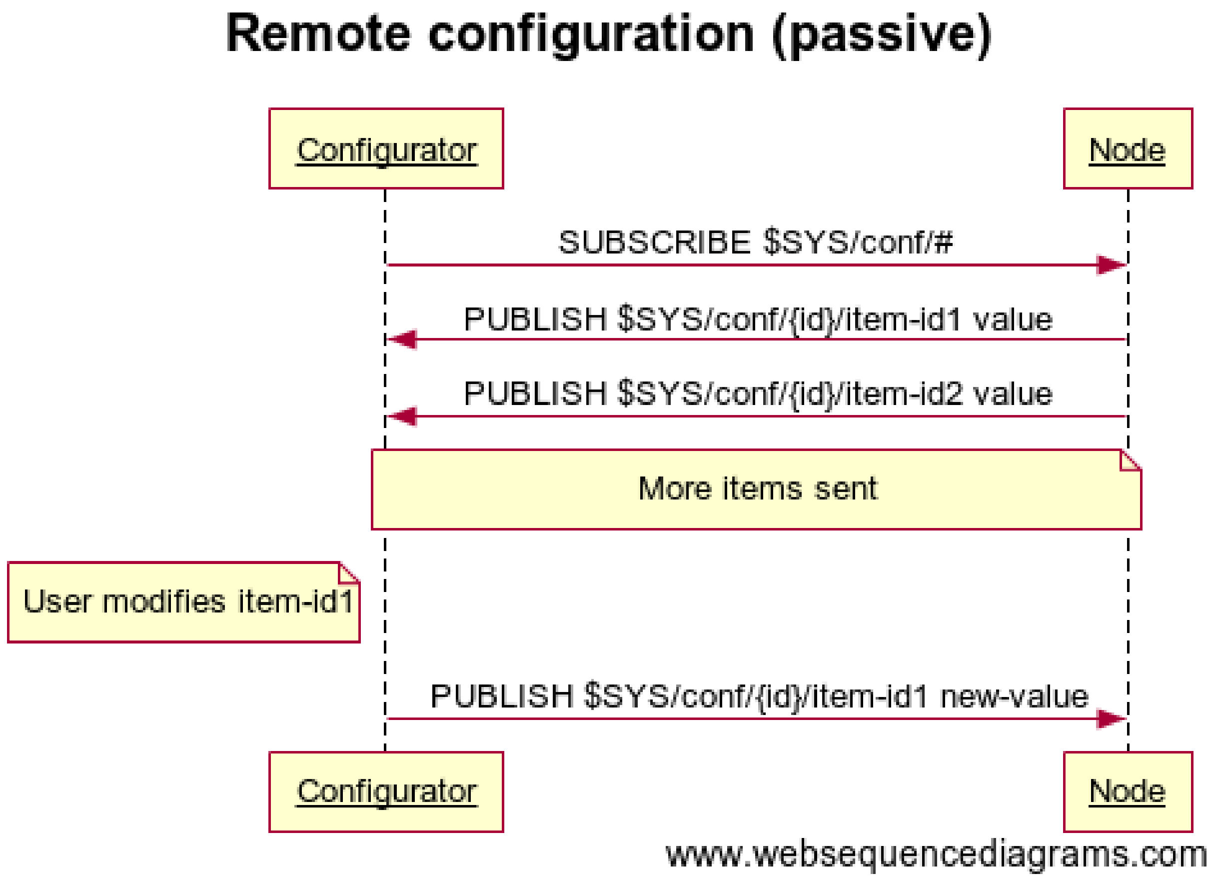

Fig. 1: Passive remote configuration state diagram

Basic remote configuration process. Configurator (which usually is a desktop application with GUI) sends SUBSCRIBE request, all nodes that support remote configuration answer with a list of items that can be configured. User sets new value for an item and configurator updates it with PUBLISH message.

node This kind describes setting for program/device in general and is editable by user. Name and location are obvious items to be in this kind. Any other global (not related to specific function or input/output) setting can be added to this kind.

topic This kind is treated as configurable value that is a topic name used to send or receive data. Item name must be descriptive so that user can understand what it is about.

net Reserved for network settings.

You can use other kind strings as you wish. They will not be interpreted in any way.

4.1.3 Item name

Just a descriptive name of an item, quite short but understandable by human. In a later version there will be added long human readable description, but in any case item name must be reasonable.

4.1.4 Predefined items

Protocol implementations expect some item kind/name pairs to mean special thins. Here is a list of know ones.

info/soft Readonly, name of software running in this node.

info/ver Readonly, version of software.

info/uptime Text string, uptime: "255d 02:12:34" or "5y 2m 13d 23:55:01"

node/name Name of this MQTT/UDP node. Human readable, not interpreted. "Weather sensors" or "A/C control"

node/location Where you can find it: "Kitchen", "building 12, 2nd floor, room 212", whatever.

net/mac Network MAC address, if not hardwired.

net/ip Network IP address, if static.

net/mask Netmask.

net/router Network default route.

Other network settings can be put to `net` kind.

4.1.5 Implementations

Configurator GUI tool exists as part of big Java MQTT/UDP viewer program, see `tools/viewer` and `build/mqtt_udp_view.*`. Clien implementations examples are done for most project languages.

Java `RemoteConfig` class in `ru.dz.mqtt_udp.config` package. Simple example code is in `main` function of this class.

Python Example code is in `lang/python3/examples/mqtt_udp_rconfig.py`.

C Simple example code is in `lang/c/examples/rconfig.c`. More advanced embedded atmega microcontroller example application is in a separate repository, see https://github.com/dzavalishin/smart-home-devices/tree/master/mmnet_mqt_udp_server/main

Lua Lua example is up and running, but as Lua has no threads, adding this example to a real program will require quite a clever tailoring. See `lang/lua/examples/mqtt_rconfig.lua`

4.2 Active remote configuration

Active remote configuration is even simpler than passive one. Starting node just requests in a loop all the items it needs to run. Configuration server replies with settings. Node continues working when all critical data is received. Server can proactively update settings later if some of them are changed.

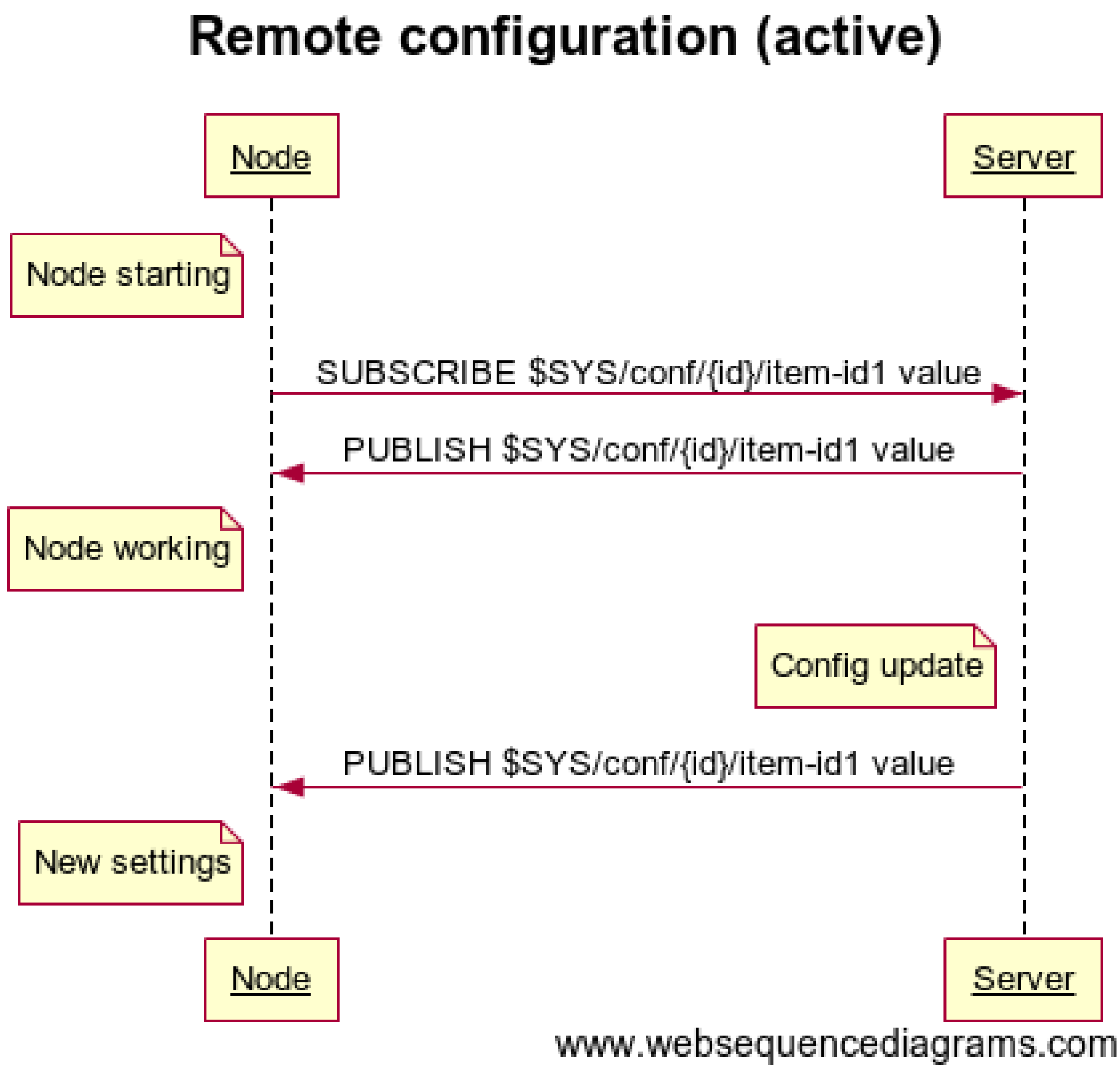

Fig. 2: Active remote configuration state diagram

4.2.1 Implementations

Currently this mode is implemented in Java as classes `Requester` and `Provider` (`ru.dz.mqtt_udp.config` package) and server application `tools/config_server`. See also `build/config_server.sh`.

CHAPTER 5

Packets and general logic

5.1 Differences from MQTT

MQTT/UDP is based on classic MQTT, but differs a bit. First of all, just a subset of packet types used and, of course, as there is no broker there is no need for CONNECT, CONNACK or DISCONNECT.

Additionally, MQTT/UDP does not send or expect variable header (packet ID field) present in some MQTT packets.

Current implementation also ignores packet flags completely, but it will change later.

Most implementations support Tagged Tail Records addition to the protocol, which extends and replaces variable header in an extensible and flexible way.

Tagged tail records can be used to add any kinds of additional information to the classic MQTT packets, but the most noticeable use of TTRs in MQTT/UDP is digital signature.

Please read detailed description at project Wiki.

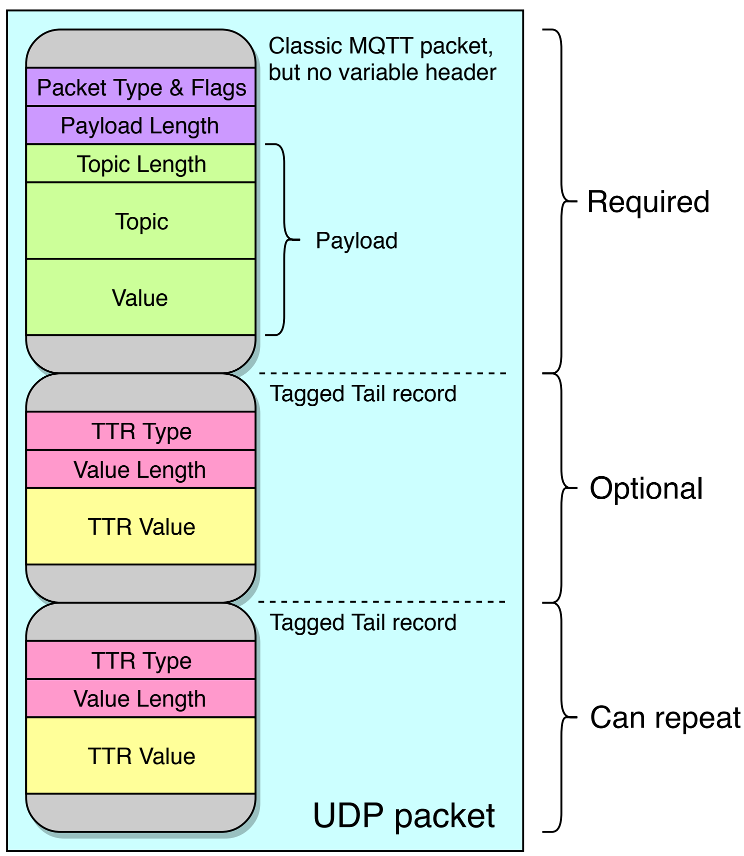

Fig. 1: General MQTT/UDP packet structure

5.2 Packet types and use

PUBLISH It is extremely simple to use MQTT/UDP. Basic use case is: one party sends **PUBLISH** packets, other receives, selecting for itself ones with topics it needs. That is all. No connect, no subscribe, no broker address to configure - we're broadcasting.

For most applications it is all that you need. But there are 3 other packet types that possibly can be used.

SUBSCRIBE MQTT/UDP uses this as a request to resend some topic value. It is not automated in any way by library code (but will be), so you have to respond to such a packet manually, if you want. It is intended for remote configuration use to let configuration program to request settings values from nodes. This is partially implemented.

PINGREQ Ping request, ask all nodes to reply. This is for remote configuration also, it helps config program to detect all nodes on the network. Library code automatically replies to `PINGREQ` with `PINGRESP`.

PINGRESP Reply to ping. You don't need to send it manually. It is done automatically.

It is supposed to use **PUBACK** packet later to support reliable delivery.

5.3 Topic names

One important thing about topics is **$SYS** topic. MQTT/UDP is a broadcast environment, so each node which wants to use **$SYS** must distinguish itself by adding MAC address or other id as a subtopic under **$SYS**: **$SYS/{group}/02AF03E6235C**. Topic name **$SYS/conf/{host-id}** is to be used for configurable parameters.

One more special thing I'm going to use is **$META** topic name suffix. It will possibly be used to request/send topic metadata. For example, if we have **kitchen/temperature** topic, then **kitchen/temperature/$META/name** can be used to pass printable topic name, and **kitchen/temperature/$META/unit** - to send measuring unit name.

CHAPTER 6

API Reference

MQTT/UDP is implemented in five languages, but implementations differ. Most complete and fast developing are Java and Python versions. Others follow a bit later. Please see map of languages and features on a project Wiki.

6.1 C Language API Reference

There is a native MQTT/UDP implementation in C. You can browse sources at https://github.com/dzavalishin/mqtt_udp/tree/master/lang/c repository.

Lets begin with a simplest examples.

Send data:

```
int rc = mqtt_udp_send_publish( topic, value );
```

Listen for data:

```
int main(int argc, char *argv[])
{
    ...
    int rc = mqtt_udp_recv_loop( mqtt_udp_dump_any_pkt );
    ...
}

int mqtt_udp_dump_any_pkt( struct mqtt_udp_pkt *o )
{
    printf( "pkt %x flags %x, id %d", o->ptype, o->pflags, o->pkt_id );

    if( o->topic_len > 0 )
        printf(" topic '%s'", o->topic );

    if( o->value_len > 0 )
        printf(" = '%s'", o->value );
```

(continues on next page)

(continued from previous page)

```
    printf( "\n");
}
```

Now lets look at the packet structure definition:

```
struct mqtt_udp_pkt
{
    int         from_ip;

    int         ptype;          // upper 4 bits, not shifted
    int         pflags;         // lower 4 bits

    size_t      total;

    int         pkt_id;

    size_t      topic_len;
    char *      topic;

    size_t      value_len;
    char *      value;

    char        is_signed;
};
```

from_ip Ip address of message sender. Usually ignored.

ptype Packet type. You will be interested in `` `PTYPE_PUBLISH` `` most of time. See `` `mqtt_udp_defs.h` `` for more.

pflags Flags specific for each type. Ignore. Current version of MQTT/UDP does not use them at all, and in any case everything critical will be processed by library.

total This field is internal for library.

pkt_id Packet id. Leave zero for outgoing packets, and sequential number will be provided. In incoming packets it will be filled if sender supports TTRs (extended packet structure).

topic and topic_len Message topic, NULL terminated. Length of topic in bytes.

value and value_len Message value, also NULL terminated. Length of value in bytes.

is_signed This packet has correct digital signature.

6.1.1 Listen for packets

See Example C code.

For listening for data from the network you need just some of fields. First, you have to check that packet is transferring item data:

```
struct mqtt_udp_pkt p;

if( p->ptype == PTYPE_PUBLISH )
{
// Got data message
}
```

For the first implementation just ignore all other packets. Frankly, there's not much for you to ignore.

Now get topic and data from packet you got:

```
strlcpy( my_value_buf, p->value, sizeof(my_data_buf) );
strlcpy( my_topic_buf, p->topic, sizeof(my_topic_buf) );
```

And you're done, now ypou have topic and value received.

6.1.2 Includes

There's just one:

```
#include "mqtt_udp.h"
```

6.1.3 Functions

Send PUBLISH packet:

```
int mqtt_udp_send_publish( char *topic, char *data );
```

Send SUBSCRIBE packet:

```
int mqtt_udp_send_subscribe( char *topic );
```

Send PINGREQ packet, ask others to respond:

```
int mqtt_udp_send_ping_request( void );
```

Send PINGRESP packet, tell that you're alive:

```
int mqtt_udp_send_ping_responce( void );
```

Start loop for packet reception, providing callback to be called when packet arrives:

```
typedef int (*process_pkt)( struct mqtt_udp_pkt *pkt );

int mqtt_udp_recv_loop( process_pkt callback );
```

Dump packet structure. Handy to debug things:

```
int mqtt_udp_dump_any_pkt( struct mqtt_udp_pkt *o );
```

Set minimal time between outgoing packets (msec), control maximum send speed:

```
void mqtt_udp_set_throttle(int msec);
```

Set callback to handle internal errors such as net IO error:

```
typedef enum {
    MQ_Err_Other,
    MQ_Err_Establish,    // open socket
    MQ_Err_IO,           // net io
    MQ_Err_Proto,        // broken pkt
    MQ_Err_Timeout,
```

(continues on next page)

(continued from previous page)

```
} mqtt_udp_err_t;

typedef int err_func_t( mqtt_udp_err_t type, int err_no , char * msg, char * arg );

void mqtt_udp_set_error_handler( err_func_t *handler );
```

User error handler can:

- Return zero: caller must attempt to ignore error, if possible.
- Return err_no: caller must return with error in turn, if possible.
- Exit (or restart application completely) if error is supposed to be fatal.

Handler can also be used for logging.

6.1.4 Digital signature

There is implementation of digital signature implemented. To use it call `mqtt_udp_enable_signature` passing encryption key. Same key must be used on all nodes that use signature. Nodes that have no signature turned on will not sign outgoing messages and will ignore incoming signatures.

Start using signature:

```
int mqtt_udp_enable_signature( const char *key, size_t key_len );
```

key Key used to sign outgoing messages and check signature on incomnig ones. Usually just an ASCII string, but can be any binary data.

key_len Number of valid bytes in key.

If signature is turned on and incoming packet is correctly signed, it will have nonzero `is_signed` field.

6.1.5 Service

Match topic name against a pattern, processing + and # wildcards, returns 1 on match:

```
int mqtt_udp_match( const char *wildcard, const char *topic_name );
```

6.1.6 Remote configuration

This part of API lets user to configure program/device by network. There is a detailed description in *Passive remote configuration* and in the Python part of this book, see *Module mqttudp.rconfig*. Here is description of C implementation.

Set up remote config subsystem:

```
#include "runtime_cfg.h"

mqtt_udp_rconfig_item_t rconfig_list[] =
{
    { MQ_CFG_TYPE_STRING, MQ_CFG_KIND_TOPIC, "Switch 1 topic",       "topic/sw1", { .
↪s = 0 } },
    { MQ_CFG_TYPE_STRING, MQ_CFG_KIND_TOPIC, "Switch 2 topic",       "topic/sw2", { .
↪s = 0 } },
```

(continues on next page)

(continued from previous page)

```
    { MQ_CFG_TYPE_STRING, MQ_CFG_KIND_TOPIC, "Di 0 topic",          "topic/di0", { .
↪s = 0 } },
    { MQ_CFG_TYPE_STRING, MQ_CFG_KIND_TOPIC, "Di 1 topic",          "topic/di1", { .
↪s = 0 } },

    { MQ_CFG_TYPE_STRING, MQ_CFG_KIND_OTHER, "MAC address",         "net/mac",   { .
↪s = 0 } },

    { MQ_CFG_TYPE_STRING, MQ_CFG_KIND_INFO, "Switch 4 topic",       "info/soft",   {␣
↪.s = DEVICE_NAME } },
    { MQ_CFG_TYPE_STRING, MQ_CFG_KIND_INFO, "Switch 4 topic",       "info/ver",    {␣
↪.s = 0 } },
    { MQ_CFG_TYPE_STRING, MQ_CFG_KIND_INFO, "Switch 4 topic",       "info/uptime", {␣
↪.s = 0 } },

    { MQ_CFG_TYPE_STRING, MQ_CFG_KIND_OTHER, "Name",                "node/name",
↪{ .s = 0 } },
    { MQ_CFG_TYPE_STRING, MQ_CFG_KIND_OTHER, "Location",            "node/location",
↪{ .s = 0 } },
};

int rconfig_list_size = sizeof(rconfig_list) / sizeof(mqtt_udp_rconfig_item_t);

int rc = mqtt_udp_rconfig_client_init( mac_string, rconfig_rw_callback, rconfig_list,␣
↪rconfig_list_size );
if( rc ) printf("rconfig init failed, %d\n", rc );
```

Each array item is one parameter to be set up remotely. The only type supported now is `MQ_CFG_TYPE_STRING`. Kinds:

MQ_CFG_KIND_INFO Read-only information about this instance (program or device)

MQ_CFG_KIND_TOPIC Is a configurable topic name, used to publish or receive information.

MQ_CFG_KIND_OTHER Any other parameter type. (R/W and not topic)

Third item field is human-readable item description, currently it is not used, but will be translated to configuration tool. Fourth item is identification of configurable item, both for local and remote side. For remote side it is sent as part of configuration message topic and is shown to user as configuration iem description. Last field is current parameter value. For read-only parameters you can just put any string pointer here. For R/W string must be malloc'ed (or set with `mqtt_udp_rconfig_set_string()`).

To be precise:

```
/// Definition of configuration parameter
typedef struct
{
    mqtt_udp_rconfig_item_type_t        type;   ///< Item (.value field) data type␣
↪(string, bool, number, other)
    mqtt_udp_rconfig_inetm_kind_t       kind;   ///< Item kind, not processed by␣
↪network code
    const char *                        name;   ///< Human readable name for this␣
↪config parameter
    const char *                        topic;  ///< MQTT/UDP topic name for this␣
↪config parameter
    mqtt_udp_rconfig_item_value_t       value;  ///< Current value
    mqtt_udp_rconfig_item_value_t       opaque; ///< user data item, not processed by␣
↪MQTT/UDP code at all
```

(continues on next page)

(continued from previous page)

```
} mqtt_udp_rconfig_item_t;
```

type Data type for `.value`, must be MQ_CFG_TYPE_STRING as for now.

kind Kind of item, see above. If kind is MQ_CFG_KIND_TOPIC, `.topic` field must begin with "topic/".

name Human-readable description, unused now.

value Current value. You will be using `.value.s` union field.

opaque Not used or interpreted, use as pointer to external storage for this item, internal item index or function pointer to read/set item as you wish.

Now lets look at available functions.

Init subsystem:

```
int rc = mqtt_udp_rconfig_client_init( mac_string, rconfig_rw_callback, rconfig_list,␣
↪rconfig_list_size );
```

mac_string Id string (12 bytes) used as unique id of this configurable instance. MAC address of device is a good candidate.

rconfig_rw_callback Callback called by subsystem to ask you provide current value for item or get new setting after instance item was remotely set up. Prototype is `int rconfig_rw_callback( int pos, int write )`, where `pos` is item position (index) in array and `write` is nonzero if callback shall get new setting from instance array and save it somewhare for next run. If zero, callback must read saved instance value and call `mqtt_udp_rconfig_set_string()` for it.

Set item value:

```
int mqtt_udp_rconfig_set_string( int pos, char *string );
```

pos Item position (index) in array

string New value

Get item value checking kind:

```
const char * rconfig_get_string_by_item_index( int pos, mqtt_udp_rconfig_inetm_kind_t␣
↪kind );
```

pos Item position (index) in array

kind Expected kind for item. If not, global error callback is called and `NULL` is returned. This function is supposed to be used to get configurable topic for outgoing message so usually this parameter is `MQ_CFG_KIND_TOPIC`.

Find item by `.value` string:

```
int rconfig_find_by_string_value( const char *search, mqtt_udp_rconfig_inetm_kind_t␣
↪kind );
```

search String value to be found.

kind Only lines of this kind will match. This function is supposed to look up incoming items topics to find if some of configurable topics match. So this parameter usually is `MQ_CFG_KIND_TOPIC`.

Please study this API use example in sample remote config C application.

6.1.7 UDP IO interface

Default implementation uses POSIX API to communicate with network, but for embedded use you can redefine corresponding functions. Here are things to reimplement.

Receive UDP packet. Returning value is number of bytes in packet received or negative error code. Must return sender's address in `src_ip_addr`:

```
int mqtt_udp_recv_pkt( int fd, char *buf, size_t buflen, int *src_ip_addr );
```

Broadcast UDP packet:

```
int mqtt_udp_send_pkt( int fd, char *data, size_t len );
```

Send UDP packet (actually not used now, but can be later):

```
int mqtt_udp_send_pkt_addr( int fd, char *data, size_t len, int ip_addr );
```

Create UDP socket which can be used to send or broadcast:

```
int mqtt_udp_socket(void);
```

Prepare socket for reception on MQTT_PORT:

```
int mqtt_udp_bind( int fd )
```

Close UDP socket:

```
int mqtt_udp_close_fd( int fd )
```

6.2 Java Language API Reference

There is a native MQTT/UDP implementation in Java. You can browse sources at https://github.com/dzavalishin/mqtt_udp/tree/master/lang/java repository.

Again, here are simplest examples.

Send data:

```
PublishPacket pkt = new PublishPacket(topic, value);
pkt.send();
```

Listen for data:

```
PacketSourceServer ss = new PacketSourceServer();
ss.setSink( pkt -> {
    System.out.println("Got packet: "+pkt);

    if (p instanceof PublishPacket) {
        PublishPacket pp = (PublishPacket) p;
    }

});
```

6.2.1 Listen for packets

See Example Java code.

Here it is:

```
package ru.dz.mqtt_udp.util;

import java.io.IOException;
import java.net.SocketException;

import ru.dz.mqtt_udp.IPacket;
import ru.dz.mqtt_udp.MqttProtocolException;
import ru.dz.mqtt_udp.SubServer;

public class Sub extends SubServer
{

    public static void main(String[] args) throws SocketException, IOException,
↪MqttProtocolException
    {
        Sub srv = new Sub();
        srv.start();
    }

    @Override
    protected void processPacket(IPacket p) {
        System.out.println(p);

        if (p instanceof PublishPacket) {
            PublishPacket pp = (PublishPacket) p;

            // now use pp.getTopic() and pp.getValueString() or pp.getValueRaw()
        }
    }
}
```

Now what we are doung here. Our class `Sub` is based on `SubServer`, which is doing all the reception job, and calls `processPacket` when it got some data for you. There are many possible types of packets, but for now we need just one, which is `PublishPacket`. Hence we check for type, and convert:

```
if (p instanceof PublishPacket) {
    PublishPacket pp = (PublishPacket) p;
```

Now we can do what we wish with data we got using `pp.getTopic()` and `pp.getValueString()`.

Listen code we've seen in a first example is slightly different:

```
PacketSourceServer ss = new PacketSourceServer();
ss.setSink( pkt -> {
    System.out.println("Got packet: "+pkt);

    if (p instanceof PublishPacket) {
        PublishPacket pp = (PublishPacket) p;
    }

});
```

Used here `PacketSourceServer`, first of all, starts automatically, and uses `Sink` you pass to `setSink` to pass packets received to you. The rest of the story is the same.

There is another, more complex listen server class, `` `PacketSourceMultiServer` ``. Instance of it can provide incoming packets to more than one listener:

```
PacketSourceMultiServer ms = new PacketSourceMultiServer();
ms.addPacketSink( first_listener );
ms.addPacketSink( next_listener );
ms.start(); // Does not start automatically
```

6.2.2 Packet classes

There are `PublishPacket`, `SubscribePacket`, `PingReqPacket` and `PingRespPacket`. Usage is extremely simple:

```
new PingReqPacket().send();
```

6.2.3 Service

Match topic name against a pattern, processing + and # wildcards, returns true on match:

```
TopicFilter tf = new TopicFilter("aaa/+/bbb");
boolean matches = tf.test("aaa/ccc/bbb") );
```

TopicFlter is a Predicate (functional interface implementation).

6.2.4 Control

setMuted(boolean mute) PacketSourceServer and PacketSourceMultiServer can be swiched to not to reply to any incoming packet (such as PING) automatically.

Engine.setThrottle(int msec) Set average time in milliseconds between packets sent. Set to 0 to turn throttling off.

6.2.5 Error handling

There is a global error handler which is called from library code on IO, protocol and some other errors. Default handler just prints error message on `stderr`. You can install your own error handler with call to `GlobalErrorHandler.setHandler(BiConsumer<ErrorType, String> h)`.

6.2.6 Digital Signature

Note: Development is in progress, not a final implementation.

Java `ru.dz.mqtt_udp.Engine` class has preliminary controls for message digital signature. It is implemented with HMAC MD5 technology.

Set signature secret key to sign and check signature:

```
void setSignatureKey(String key)
```

Set requirement for all incoming packets to be signed:

```
void setSignatureRequired(boolean req)
```

At this moment other language implementations ignore and do not generate signature at all.

6.3 Python Language API Reference

As you already guessed, python implementation is native too. You can browse sources at https://github.com/dzavalishin/mqtt_udp/tree/master/lang/python3 repository. There is also lang/python directory, which is for older 2.x python environment, but it is outdated. Sorry, can't afford to support it. If you need python 2.x, you can backport some python3 code, it should be quite easy.

Let's begin with examples, as usual.

Send data:

```
mqttudp.engine.send_publish( "test_topic", "Hello, world!" )
```

Listen for data:

```
def recv_packet(pkt):
    if pkt.ptype != me.PacketType.Publish:
        print( str(pkt.ptype) + ", " + pkt.topic + "\t\t" + str(pkt.addr) )
        return
    print( pkt.topic+"="+pkt.value+ "\t\t" + str(pkt.addr) )

mqttudp.engine.listen(recv_packet)
```

6.3.1 Module mqttudp.engine

Main package, implements MQTT/UDP protocol.

Paccket class:

```
class PacketType(Enum):
    Unknown     = 0
    Publish     = 0x30
    Subscribe   = 0x80
    PingReq     = 0xC0
    PingResp    = 0xD0


class Packet(object):
    def __init__( self, ptype, topic, value, pflags, ttrs ):
        self.ptype  = ptype
        self.pflags = pflags
        self.topic  = topic
        self.value  = value
        self.ttrs   = ttrs
        self.addr   = None
```

Functions:

- `send_ping()` - send PINGREQ packet.

- `send_ping_responce()` - send PINGRESP packet. It is sent automatically, you don't have to.
- `listen(callback)` - listen for incoming packets.
- `send_publish( topic, payload)` - this what is mostly used.
- `send_subscribe(topic)` - ask other party to send corresponding item again. This is optional.
- `set_muted(mode: bool)` - turn off protocol replies. Use for send-only daemons which do not need to be discovered.

Match topic name against a pattern, processing + and # wildcards, returns True on match:

```
import mqttudp.engine as me
me.match("aaa/+/bbb", "aaa/ccc/bbb")
```

Turn of automatic protocol replies:

```
set_muted(mode: bool)
```

Set minimum time between packets sent, msec:

```
set_throttle(msec: int)
```

Set network address to listen at (choose incoming packets network interface). Address must be equal to address of some network interface:

```
set_bind_address( "192.168.1.1" )
```

Set network address to broadcast to (choose outgoing packets network interface). Address must be broadcast address for some of existing network interfaces. Ask local network administrator if unsure:

```
set_broadcast_address( "192.168.1.255" )
```

6.3.2 Module mqttudp.config

Additional module, sets up configuration file reader. Most command line utilities use it to get settings. It reads `mqtt-udp.ini` file in current directory. Here is an example:

```
[DEFAULT]
host = smart.

[mqtt-gate]             # Settings for MQTT to MQTT/UDP gate
login =
password =

subscribe=#
#host = smart.          # See [DEFAULT] above

#blacklist=/topic       # Regexp to check if topic is forbidden to relay
#blacklist=/openhab

[openhab-gate]
#port=8080              # There's builtin default
#host = smart.          # Settings for MQTT/UDP to OpehHAB gate

#blacklist=/topic       # Regexp to check if topic is forbidden to relay
```

(continues on next page)

(continued from previous page)

```
# which sitemap to use for reading data from openhab
#sitemap=default
```

Usage:

```
import mqttudp.config as cfg

cfg.setGroup('mqtt-gate')            # set ours .ini file [section]

blackList=cfg.get('blacklist')       # read setting
```

6.3.3 Module mqttudp.rconfig

Additional module implemening passive remote configuration client (party that is being configured) implementation.

There is a complete demonstration example exist.

To see example working please run `mqtt_udp_rconfig.py` first and mqtt_udp_view after it. In viewer please press middle toolbar button to open remote configuration window. This window will show all running MQTT/UDP instances that can be configured. There must be `Python test node` among them. Select its tab. You will see all the configurable items (from `init_items` dictionary) as text fields. Meanwhile `mqtt_udp_rconfig.py` will be sending a random number with "test" topic. Enter new topic name in a field near "topic: test" description and press nearest button to send new setting to program. Notice that now it sends random data with a topic you just set up.

Now lets look at example code (see examples/mqtt_udp_rconfig.py):

```
import mqttudp.rconfig as rcfg

init_items = {
    ## read only
    "info/soft"          : "Pyton example",
    "info/ver"           : "0.0",
    "info/uptime"        : "?",

    ## common instance info
    "node/name"          : "Unnamed",
    "node/location"      : "Nowhere",

    # items we want to send out
    "topic/test"         : "test",
    "topic/ai0"          : "unnamed_ai0",
    "topic/di0"          : "unnamed_di0",

    "topic/pwm0"         : "unnamed_pwm0",
}

def send_thread():
    while True:
        n = str(random.randint(0, 9))
        print( "Send "+n )
        rcfg.publish_for( "test", n )
        time.sleep(2)
```

(continues on next page)

(continued from previous page)

```
if __name__ == "__main__":
    print( "Will demonstrate remote config" )
    rcfg.init( init_items )

    st = threading.Thread(target=send_thread, args=())
    st.start()

    mq.listen( rcfg.recv_packet )
```

This example shows how to use remote configuration subsystem. Dictionary `init_items` contains list of items which can possiblly be configured remotely. Different elements are used in a different ways.

In general each item in a list is a configurable thing. For example, `"node/location" : "Nowhere"` is item which name is `node/location` and initial value is `Nowhere`. (It is supposed as a memo for user to know where an appliance or computer running this code is installed.) Another example is `"topic/ai0" : "unnamed_ai0"` - it is supposed to be a configurable topic name that device uses to send data from some analogue input. User must configure topic name and it will be used by node to send data.

Generally item keys consist of two parts separated with slash: `"topic/pwm0"`, `"info/uptime"` or `"node/name"`. Left part is named **kind** and defines the way item is processed. Here is a list of known kinds.

`info` Read only description of node/program.

`node` General node information or settings.

`topic` This kind is a configurable value that is a topic name.

`net` Reserved for network settings.

Please see more on kinds in *Passive remote configuration*.

Lets go on with code. Line `rcfg.init( init_items )` sets subsystem up. Remote config subsystem first loads previous settings from .ini file (you can set file name with `set_ini_file_name(fn)` function) and fills all items from given `init_items` dictionary that was not read from .ini file. Items with `info` kind are taken from `init_items` in any way.

After init your program continues working, but must call `recv_packet` function of `mqttudp.rconfig` for each incoming MQTT/UDP packet for remote configuration to work.

There are three ways to use configured parameters.

Just read parameter You can just call `get_setting( name )` for needed item to get current configured value. For example, `get_setting( "net/mac" )` or, say, `lcd_print( get_setting( "node/location" ) )`. If your node will be reconfigured in run time, on next call there will be new value.

Send data for configurable topic By calling `publish_for( item_of_topic, data )` you will send data to a topic which is configured by item with, guess what, `topic` kind. See above `rcfg.publish_for( "test", n )` - this line looks up config item named `topic/test`, and uses its value as a topic to publisn value of varible `n` to.

Check incoming packet topic On receiving incoming PUBLISH packet, you can use `is_for( topic_item, topic )` function, which checks `topic` parameter to be equal to value of config item named `"topic/"+topic_item`, such as `is_for( "pwm0", topic )` will return `True` if `topic` variable contains string equal to value of config item `"topic/pwm0"`.

Only thing left to mention is that you can set callback with call to `set_on_config( callback )` and it will be called if remote configuration happens. Config item name and new value will be passed as parameters.

6.3.4 Module mqttudp.interlock

Additional module, implements two classes: `Bidirectional` and `Timer`.

`Bidirectional` is used by bidiractional gateways to prevent loop traffic:

```
# Init interlock object which will
# forbid reverse direction traffic
# for 5 seconds after message passed
# in one direction.

ilock = mqttudp.interlock.Bidirectional(5)

# Check if we can pass forward

if ilock.broker_to_udp(msg.topic, msg.payload):
    mqttudp.engine.send_publish( msg.topic, msg.payload )
    print("To UDP: "+msg.topic+"="+str(msg.payload))
else:
    print("BLOCKED to UDP: "+msg.topic+"="+str(msg.payload))

# and back

if ilock.udp_to_broker(topic, value):
    bclient.publish(topic, value, qos=0)
    print( "From UDP: "+topic+"="+value )
else:
    print( "BLOCKED from UDP: "+topic+"="+value )
```

Value is not actually used in current implementation. It is passed for later and smarter versions.

`Timer` prevents updates from coming too frequently:

```
it = mqttudp.interlock.Timer(10)

if it.can_pass( topic, value ):
    print("From broker "+topic+" "+value)
    mqttudp.engine.send_publish( topic, value )
else:
    print("From broker REPEAT BLOCKED "+topic+" "+value)
```

It checks if value is changed. Such values are permitted to pass through. Unchanged ones will pass only if time (10 seconds in this example) is passed since previous item come through.

6.3.5 Module mqttudp.mqtt_udp_defs

This module is not for user code, it is used internally. But you can get library release version from it:

```
PACKAGE_VERSION_MAJOR = 0
PACKAGE_VERSION_MINOR = 4
```

6.4 Lua Language API Reference

You can browse sources at https://github.com/dzavalishin/mqtt_udp/tree/master/lang/lua repository.

Basic examples in Lua.

Send data:

```
local mq = require "mqttudp"
mq.send_publish( topic, val );
```

Listen for data:

```
local mq = require "mqttudp"

local listener = function( ptype, topic, value, ip, port )
    print("'"..topic.."' = '"..val.."'".."  from: ", ip, port)
end

mq.listen( listener )
```

6.4.1 Send packets

There are functions to send different kinds of packets:

```
local mq = require "mqttudp"

mq.send_pingreq()
mq.send_pingresp()
mq.send_subscribe( topic )
mq.send_publish( topic, value )
```

6.4.2 Service

Match topic name against a pattern, processing + and # wildcards, returns `true` on match:

```
local mu = require "mqttudp"
local ok = mu.match( wildcard, topic_name )
```

6.4.3 NodeMCU

There is a version of Lua library for NodeMCU microcontroller firmware. See `lang/lua/nodemcu` for examples.

6.5 CodeSys ST Language API Reference

Note: This implementation ise currently send only.

Sorry, due to PLC limitations, there is no clear API in this code example, just integrated protocol and client code example.

PLC is specific: it runs all its programs in loop and it is assumed that each program is running without blocking and does not spend too much time each loop cycle. There's usually a watch dog that checks for it. Hence, ST implementation is cycling, sending just one topic per loop cycle.

Actual API is simple:

```
FUNCTION MQTT_SEND : BOOL

VAR_INPUT
    socket          : DINT;

    topic           : STRING;
    data            : STRING;

    sock_adr_out    : SOCKADDRESS;
END_VAR


FUNCTION MQ_SEND_REAL : BOOL
VAR_INPUT
    socket          : DINT;
    m_SAddress      : SOCKADDRESS;

    topic           : STRING;
    data            : REAL;
END_VAR
```

Here is how it is used in main program:

```
PROGRAM MQTT_PRG
VAR
    STEP         : INT  := 0;
    socket       : DINT := SOCKET_INVALID;
    wOutPort     : INT  := 1883;
    addr         : SOCKADDRESS;

END_VAR

CASE STEP OF
    0:
        socket := SysSockCreate( SOCKET_AF_INET, SOCKET_DGRAM, SOCKET_IPPROTO_UDP );

        addr.sin_family := SOCKET_AF_INET;
        addr.sin_port   := SysSockHtons( wOutPort );
        addr.sin_addr   := 16#FFFFFFFF; (* broadcast *)

    1:  MQ_SEND_REAL( socket, addr,  'PLK0_WarmWater', WarmWaterConsumption );
    2:  MQ_SEND_REAL( socket, addr,  'PLK0_ColdWater', ColdWaterConsumption );

    3:  MQ_SEND_REAL( socket, addr,  'PLK0_activePa', activePa_avg );
    4:  MQ_SEND_REAL( socket, addr,  'PLK0_Va', Va );

ELSE
    IF socket <> SOCKET_INVALID THEN
        SysSockClose( socket );
    END_IF
    socket := SOCKET_INVALID;
END_CASE

STEP := STEP + 1;

IF socket = SOCKET_INVALID THEN
    STEP := 0;
```

(continues on next page)

(continued from previous page)

```
END_IF

END_PROGRAM
```

CHAPTER 7

Integration and tools

7.1 Connectors

Project includes two simple connectors. One joins MQTT/UDP with classical MQTT, other connects to OpenHAB.

All the tools read `mqtt-udp.ini` file, see *Module mqttudp.config* for detailed description. You have, at least, to set host name for both tools.

7.1.1 Classic MQTT

It is obvious that MQTT/UDP can be used together with traditional MQTT, so there's a simple gateway to pass traffic back and forth. It is written in Python and copies everything from one side to another and back. There's interlock logic introduced that prevents loops by not passing same topic message in reverse direction for some 5 seconds.

To run connector go to `lang/python3/examples` directory and start `mqtt_bidir_gate.py` program.

There are also unidirectional gates `mqtt_broker_to_udp.py` and `mqtt_udp_to_broker.py`.

There is an example of service configuration file `mqttudpgate.service` for Unix `systemctl` service control tools.

This bridge is also can be used to integrate with cloud MQTT servers. Just set up corresponding host/port/login/password in section `[mqtt-gate]` of `mqtt-udp.ini` and run `mqtt_bidir_gate.py`.

7.1.2 OpenHAB

At the moment there are two one way gateways, from MQTT/UDP to OpenHAB and back, and one complete bidirectional gateway.

To run connector go to `lang/python3/examples` directory and start `mqtt_udp_to_openhab.py`, `openhab_to_udp.py`, or `openhab_bidir_gate.py` program.

Minimal configuration required is to set OpenHAB host name in section `[openhab-gate]` of `mqtt-udp.ini` file. Gateway uses OpenHAB sitemap to get list of items to read. By default it uses sitemap named `default`. If your OpenHAB setup most populated sitemap is not default one, please set sitemap name in .ini file too.

7.1.3 CCU825 GSM Controller

There is a connector for a CCU825 controller in a separate repository.

7.2 Programs

There are some programs and scripts made to help testing MQTT/UDP library. Some of them are written in C and Java but most exist just in Python version.

7.2.1 C programs

- **mqtt_udp_clock** - sends date and time value to network once a minute. Can be used to set clock in IoT/smarthome peripheral devices. NB! Use SNTP if you need high accuracy.

7.2.2 Java programs

- **tools/config_server** - simple remote configuration server. See corresponding README for details.

7.2.3 Python programs

- **random_to_udp.py** - send random numbers with 2 sec interval, to test reception.
- **dump.py** - just show all traffic.
- **ping.py** - send ping and show responces. By using set_muted(mode: bool) function it turns off protocol replies so it will not resond to itself.
- **subscribe.py** - send subscribe request.
- **seq_storm_send.py** - send sequential data with no speed limit (use -s to set limit, though).
- **seq_storm_check.py** - check traffic sent by *seq_storm_send.py* and calculate speed and error rate.

7.3 Traffic viewer

A GUI tool to view what's going on and send data too.

It is supposed that this tool can be used as remote configuration for MQTT/UDP nodes on the network.

To run program go to project root directory and start `mqtt_udp_view.cmd` or `mqtt_udp_view` depending on your OS. You will need Java 8 and JavaFX installed for it to run. Please download it from http://java.com or try to use OpenJDK. (I did not yet.)

Actual user guide is at project Wiki: https://github.com/dzavalishin/mqtt_udp/wiki/MQTT-UDP-Viewer-Help

To run viewer you will need MqttUdpViewer.jar - on any OS `java -jar MqttUdpViewer.jar` will start program. For Windows there is MqttUdpViewer.exe which is a starter for MqttUdpViewer.jar, so in widows you can start it with `MqttUdpViewer` command.

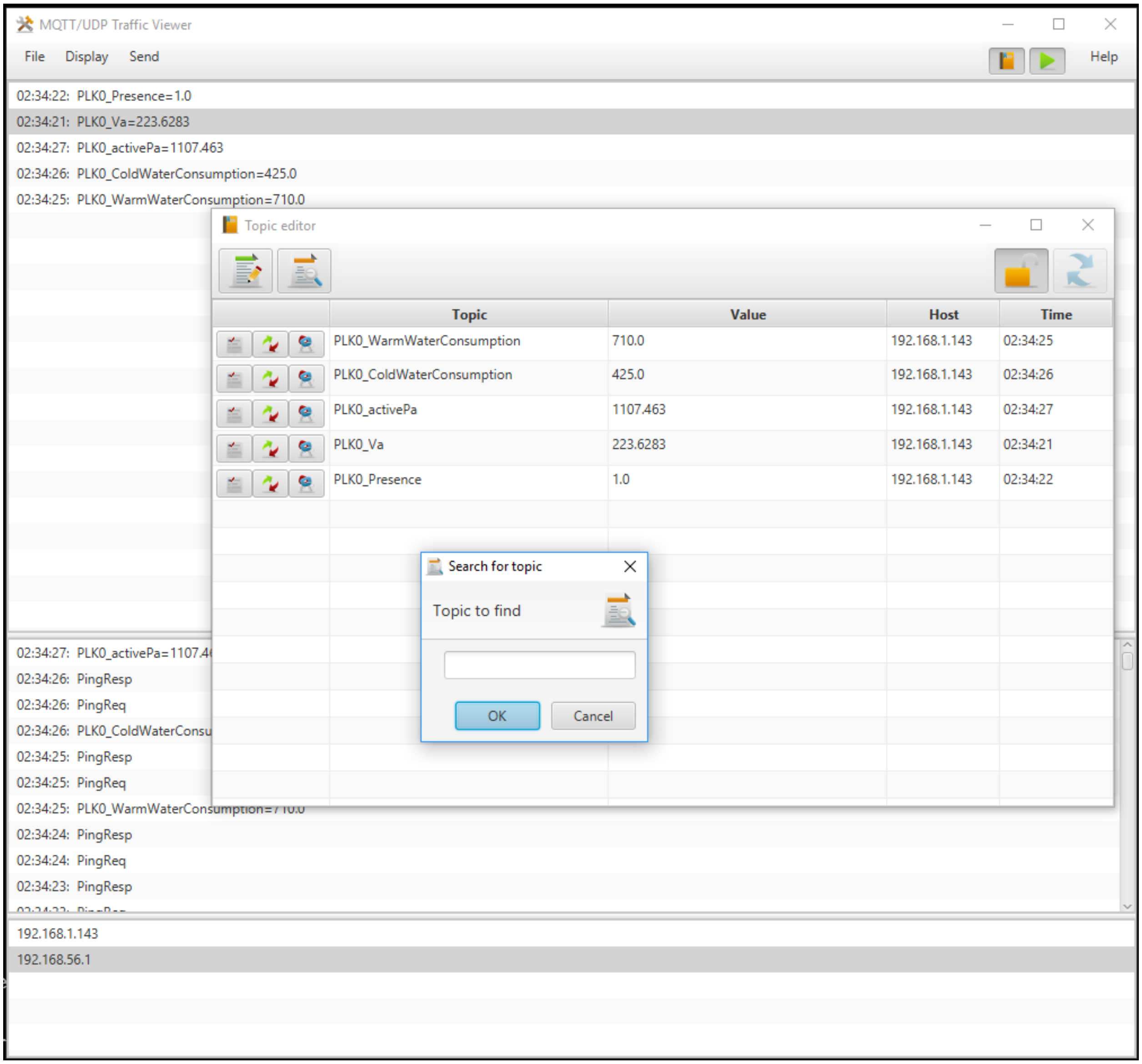

Fig. 1: Screenshot of MQTT/UDP viewer tool (Windows)

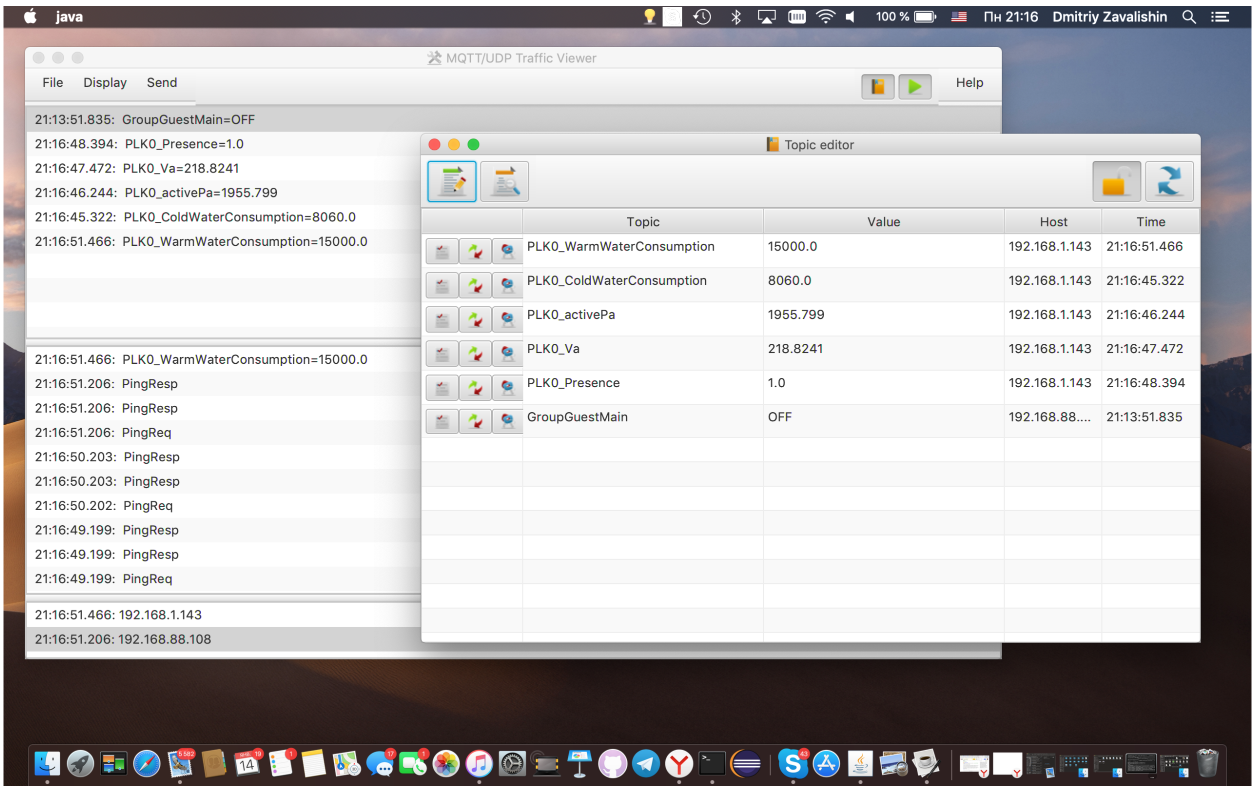

Fig. 2: Screenshot of MQTT/UDP viewer tool (Mac OS)

Being written in Java viewer works on Mac OS. It also must run on other operating systems with Java, but it was not yet tested.

For details please read wiki, but in short, viewer has following parts:

7.3.1 Value view

Top list displays current value of all topics that was transmitted since program start. There is also time of last update.

7.3.2 Log view

Shows each message passing. You can choose if you will see ping/reply packets or no.

7.3.3 Host list

Displays list of network hosts sending MQTT/UDP traffic.

7.3.4 Message Editor

Can be used to send messages to network. It is possible to send message just to one host or broadcast them. It is also possible to send **SUBSCRIBE** messages to request topic data to be sent.

7.3.5 Remote configurator

Remote configuration is described in *Passive remote configuration* in detail. This program implements passive remote configuration mode.

Programs or devices that use MQTT/UDP passive remote configuration feature can be configured by network with this tool. Instances provide list of configurable items and each tab of config window lets you set instance parameters. There is a complete example made for Python, see *Module mqttudp.rconfig* for description. Implementations of client side also made for C, Java and Lua languages.

7.4 System Tray Informer

There is a simple program that adds an icon to a system tray. This icon lets you see some data from MQTT/UDP or control one OpenHAB item. Being a Java program it should run on MacOS and Linux, but it was not tested with Linux yet. Illustrations show how it looks in Windows and Mac OS.

7.4.1 Setting up

This program reads an `mqttudptray.ini` configuration file on start:

```
topic1=PLK0_activePa
topic2=PLK0_Va

topic1header=Power consumption
topic2header=Mains Voltage

# experimental
#
controltopic=GroupGuestMain
```

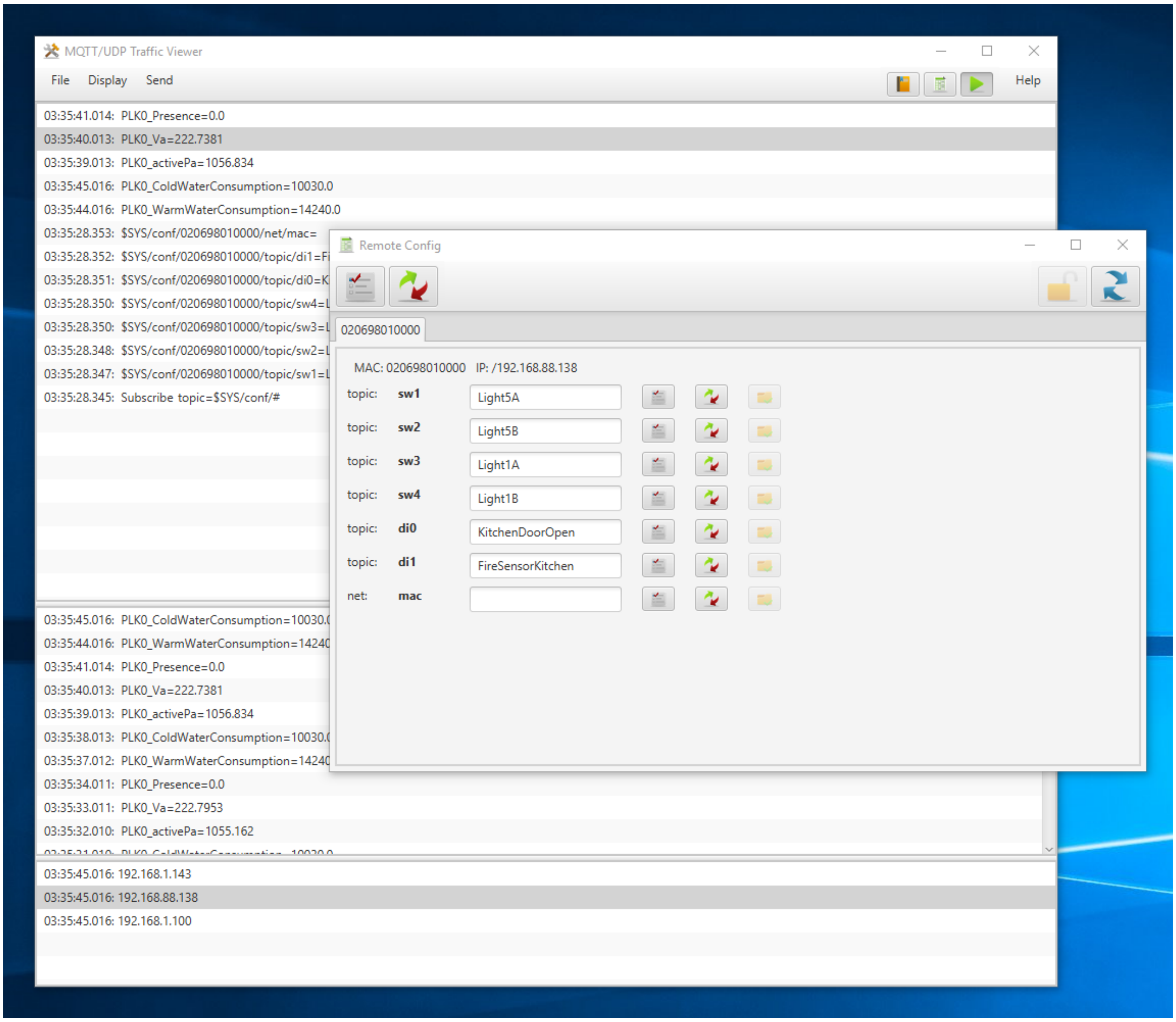

Fig. 3: Screenshot of MQTT/UDP viewer tool remote configuration window

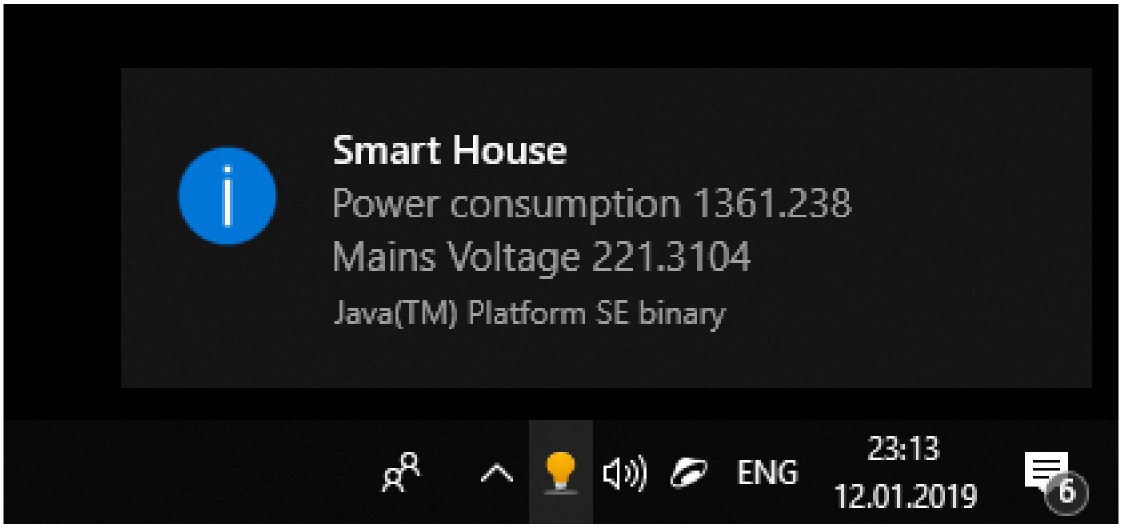

Fig. 4: Windows: tray icon informer
This informer is shown when you press right mouse button. It shows two items defined in .ini file, see reference. In this example mains voltage and total power consumption are shown.

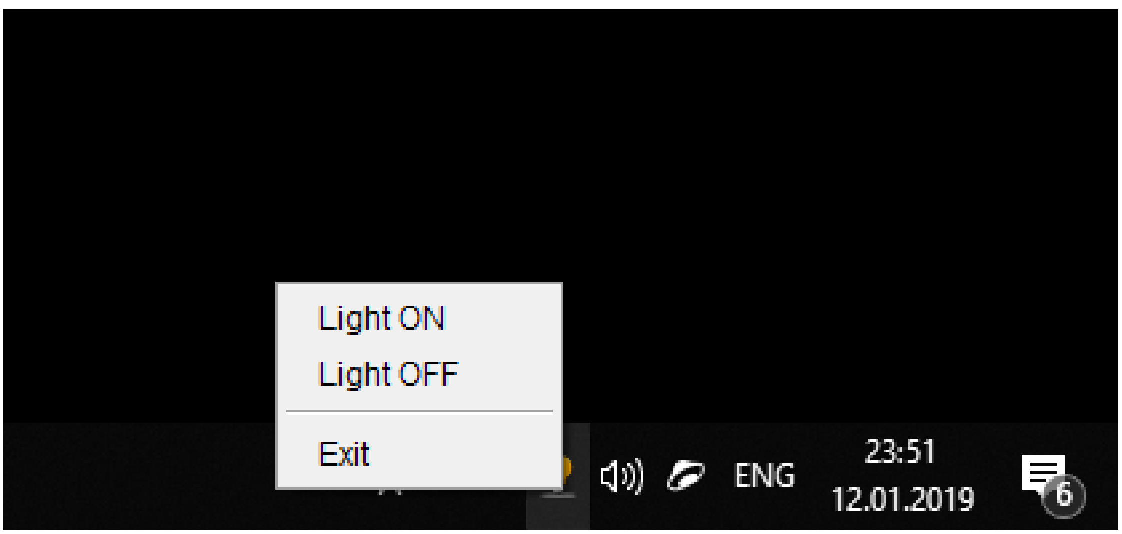

Fig. 5: Windows: tray icon menu
Menu is shown when right mouse button is pressed.

You can define which two topics will be displayed, and what human readable names they have. The `controltopic` setting is for controlling light (or other ON/OFF switch) via OpenHAB. If defined, *Light on* and *Light off* menu items of a tray icon will send ON and OFF values to corresponding topic.

Current version of MQTT/UDP does not support QoS, and, possiblly on/off message can be lost. That is why this function is marked as experimental.

7.4.2 Running

In any OS you will need `MqttUdpTray.jar` and `mqttudptray.ini`. There is `MqttUdpTray.exe` for windows. In other systems (with Java 8 installed) please execute `javaw -jar MqttUdpTray.jar` or `java -jar MqttUdpTray.jar` command. All the files are in the `build` directory.

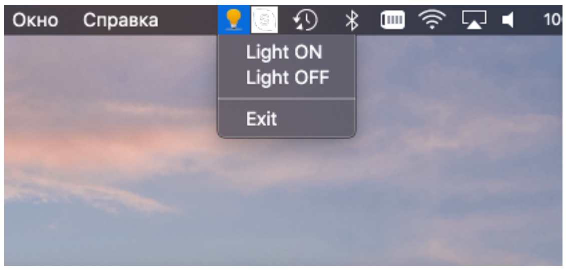

Fig. 6: Tray icon menu
Menu is shown when left mouse button is pressed.

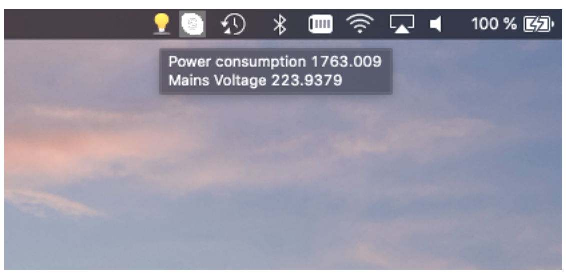

Fig. 7: Tray icon on mouse over
Tooltip is shown when mouse is over the icon.

CHAPTER 8

Addendums

8.1 Cook Book

Even if you think that MQTT/UDP is not for you and can't be used as primary transport in your project, there are other possibilities to use it together with traditional IoT infrastructure

8.1.1 Displays

Send a copy of all the items state to MQTT/UDP and use it to bring data to hardware and software displays. For example, this project includes an example program (`tools/tray` directory, see figure *Tray icon on mouse over*) to display some MQTT/UDP items via an icon in a desktop tray. Being a Java program it should work in Windows, MacOS and Unix.

8.1.2 Sensors and integrations

It is not really easy to write a native Java connector for OpenHAB. Write it in Python for MQTT/UDP and translate data from MQTT/UDP to OpenHAB. It is really easy.

By the way, there is quite a lot of sensors drivers in Python for Raspberry and clones.

Don't like Raspberry? Use Arduino or some ARM CPU unit and C version of MQTT/UDP.

8.2 Sketches

There are more or less complete demo implementations exist.

8.2.1 Wemos D1 Mini Pro

This sketh must also run on any NodeMCU hardware.

See lang/lua/nodemcu for source code and instruction.

8.2.2 Arduino

This sketch must run on any Arduino device as long as it has ENC28J60 ethernet module connected.

See lang/arduino for source code and instructions for this one.

8.3 Network

Note: Basic digital signature subsystem for MQTT/UDP is in development now. Java implementation is already supporting it, contact us if you want to test it or take part in development. See `ru.dz.mqtt_udp.Engine` class.

Current implementation of MQTT/UDP has no security support. It is supposed that later some kind of packet digital signature will be added. At the moment I suppose that protocol can be used in comletely secure networks or for not really important data.

Actually I personally use MQTT/UDP in typical home network, separated from Internet with NAT but with no separation between smart home and other computers. I do think that would my home network be hacked into, intervention into the smart home system is the lesser of possible evils.

8.4 Work in progress

There are parts of protocol or additional components design that not finished completely, and are subject of discussion.

8.4.1 QoS Server

Nowadays UDP is quite reliable (typical loss rate is < 0.5% even in a busy network), but not 100% reliable. Things are easy in point to point communications. Send a message, wait for acknowledgement, resend if none received in a reasonable time. But MQTT/UDP is broadcast protocol. Do we have to wait for ack from all nodes in a network? Some nodes? Which ones?

It is makes sense that we can build a map of nodes that listen to us by collecting their responces. But we want to keep MQTT/UDP implementation simple and this is not that simple. And not any node needs such high a reliability.

The idea is to add separate server on a network that will build lists of listeners for each topic, collect low-QoS ack packets and sent one high-QoS ack packet to topic publisher(s).

Note that such server is not a single point of failure. First of all, there can be more than one instance of QoS server. Second, even if QoS server fails, nodes continue to send data. Though, each packet will be resent few times, but it is not a communications failure. Last, but not least, sending node can stop resending after few acks with lower QoS. For example, sending node can take for acknowledge one QoS 3 ack message and 2 or 3 QoS 2 ones.

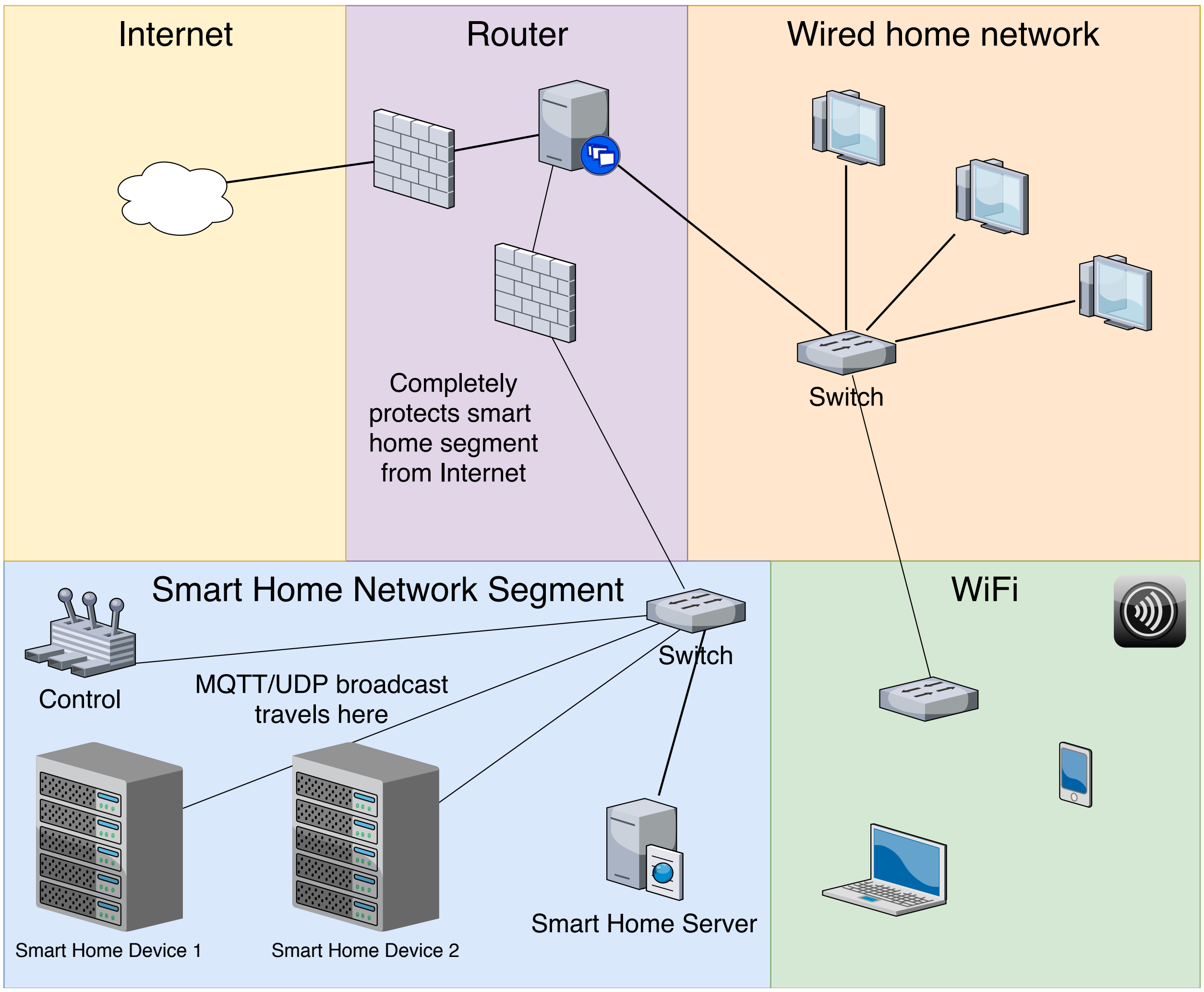

Fig. 1: Ideal structure of network.
Segment for a smart home is separated from local network for usual computers. MQTT/UDP data can be forwarded there on firewall, but not backwards.

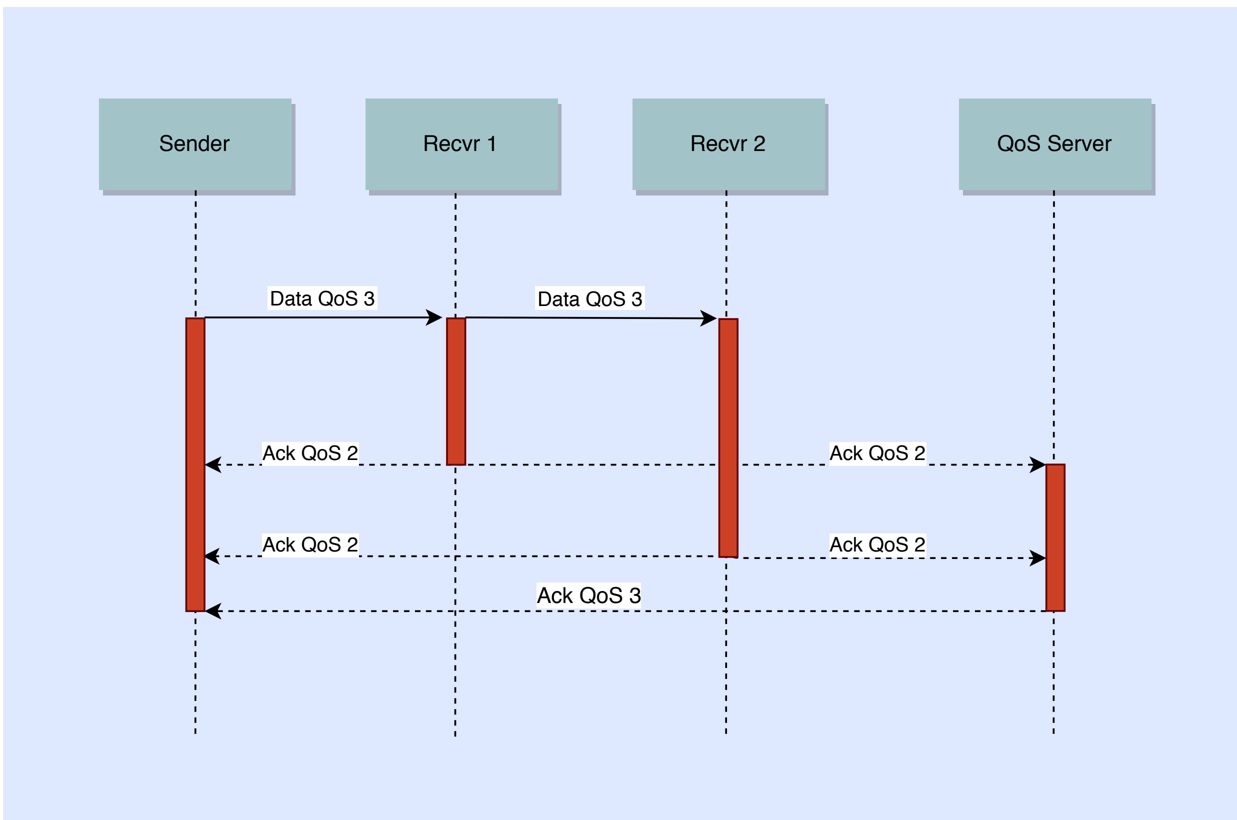

Fig. 2: QoS server sequence diagram

8.5 FAQ

Q: There's MQTT-SN, aren't you repeating it?

A: MQTT-SN still needs broker. And MQTT/UDP still simpler. :)

Q: Why such a set of languages?

A: **C** is for embedded use. I want it to be easy to build smert sensor or wall display/control unit based on MQTT/UDP.

Python is for gateways and scripting. Writing small command line program or daemon in Python is easy. Also, there is a lot of Python drivers for various sensors and displays on Raspberry/Orange/Banana/whatever PI.

Java is for serious programming and GUI apps. Viewer was easy thing to do with JavaFX.

Lua is for NodeMCU and, possibly, other embedded platforms.

CodeSys is evil you can't escape.

8.6 Links

GitHUb: https://github.com/dzavalishin/mqtt_udp

Error reports and feature requests: https://github.com/dzavalishin/mqtt_udp/issues

If you use MQTT/UDP, please let me know by adding issue at GitHub. :)

8.6.1 Additional repositories

These projects use or extend MQTT/UDP.

CCU825 connector, Java https://github.com/dzavalishin/ccu825modbus

AtMega128 IO Unit https://github.com/dzavalishin/smart-home-devices/tree/master/mmnet_mqt_udp_server

Python Module Index

m

Index