# MQTT/UDP Documentation

***Release 0.4-1***

**Dmitry Zavalishin**

**Jan 15, 2019**

# 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.

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*

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. 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.

CHAPTER 2

# Possible topologies

Here is a list of more or less obvious use cases 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.

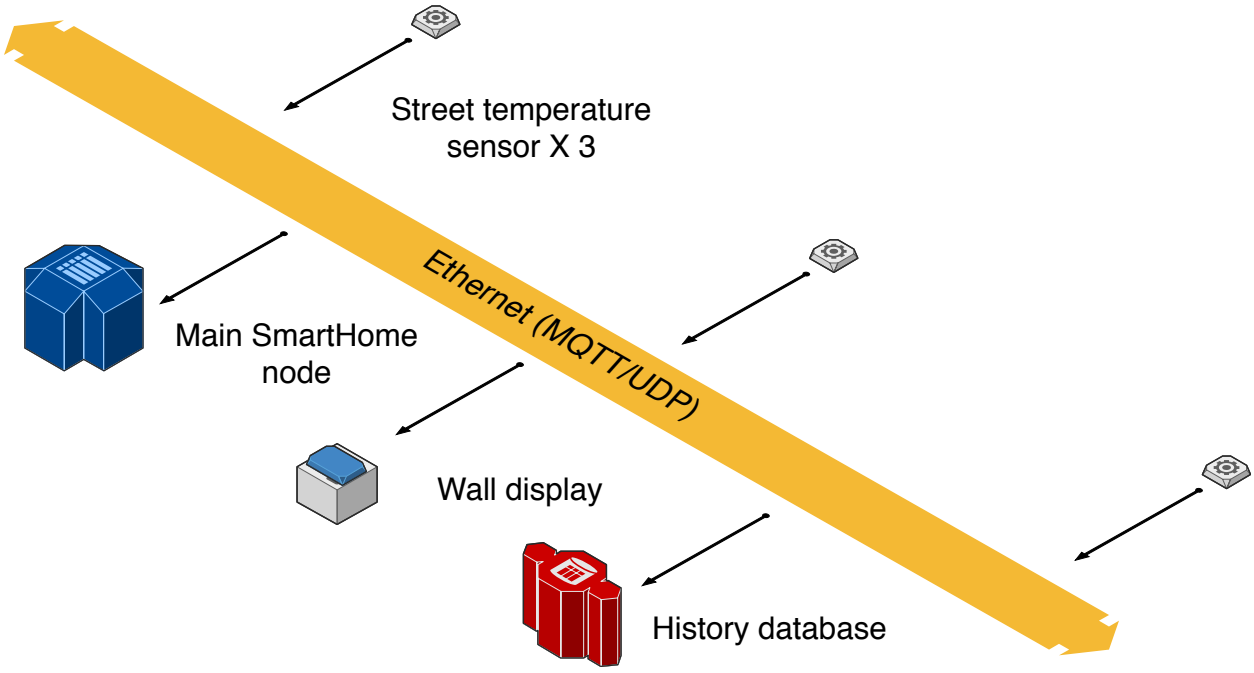


Fig. 1: Typical MQTT/UDP use case.
This diagram shows three sensors duplicating each other. For example, three outer temperature sensors. Wall display, history database and main smarthome unit get copy of all data from sensors. Malfunction of any unit does not make any problemn for others.

## 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 All the data is visible

That is a topology issue too. 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, I made simple test[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.

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

CHAPTER 4

# Packets and general logic

## 4.1 Packet types and use

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 to be implemented later.

**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.

I'm going to use **PUBACK** packet later to support reliable delivery.

## 4.2 Topic names

One important thing about topics is **$SYS** topic. MQTT/UDP is a broadcast environment, so each node which wants to use **$SYS** ,ust distinguish itself by adding IP address or host name as first subtopic under **$SYS**: **$SYS/192.168.1.33**. Topic name **$SYS/hostname/config** is to be used for configurable from network 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 5

# API Reference

## 5.1 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 get through 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;          // length of the rest of pkt down from here

    int         pkt_id;

    size_t      topic_len;
    char *      topic;

    size_t      value_len;
    char *      value;
};
```

### 5.1.1 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.

### 5.1.2 Includes

There's just one:

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

### 5.1.3 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 );
```

### 5.1.4 UDP IO interface

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

Receive UDP packet. 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 )
```

### 5.1.5 Service

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

```
mqtt_udp_match( wildcard, topic name )
```

## 5.2 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;
    }

});
```

### 5.2.1 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
```

(continues on next page)

(continued from previous page)

```
    {
        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.

### 5.2.2 Packet classes

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

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

### 5.2.3 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).

## 5.3 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(ptype,topic,value,pflags,addr):
    if ptype != "publish":
        print( ptype + ", " + topic + "\t\t" + str(addr) )
        return
    print( topic+"="+value+ "\t\t" + str(addr) )

mqttudp.engine.listen(recv_packet)
```

### 5.3.1 Module mqttudp.engine

Main package, implements MQTT/UDP protocol.

- `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")
```

### 5.3.2 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.
```

(continues on next page)

(continued from previous page)

```
[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

# 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
```

### 5.3.3 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.

### 5.3.4 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
```

## 5.4 Lua Language API Reference

**Note:** Lua API is not final.

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 "mqtt_udp_lib"
mq.send_publish( topic, val );
```

Listen for data:

```
local mq = require "mqtt_udp_lib"

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

mq.listen( listener )
```

## 5.5 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;
        m_SAddress    : SOCKADDRESS;

END_VAR

CASE STEP OF

        0:
                socket := SysSockCreate( SOCKET_AF_INET, SOCKET_DGRAM, SOCKET_IPPROTO_
↪UDP );

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

        1:  MQ_SEND_REAL( socket, m_SAddress,  'PLK0_WarmWaterConsumption', GLOBAL_
↪WarmWaterConsumption );
        2:  MQ_SEND_REAL( socket, m_SAddress,  'PLK0_ColdWaterConsumption', GLOBAL_
↪ColdWaterConsumption );

        3:  MQ_SEND_REAL( socket, m_SAddress,  'PLK0_activePa', GLOBAL_activePa_avg *␣
↪10 );
```

(continues on next page)

(continued from previous page)

```
        4:  MQ_SEND_REAL( socket, m_SAddress,  '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;
END_IF

END_PROGRAM
```

CHAPTER 6

# Integration and tools

## 6.1 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.

### 6.1.1 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.

### 6.1.2 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.

## 6.2 Scripts

There are Python scripts I made to help myself testing MQTT/UDP library. Some of them are written in C and Lua too but most exist just in Python version.

- **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.

## 6.3 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.

## 6.4 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.

### 6.4.1 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
```

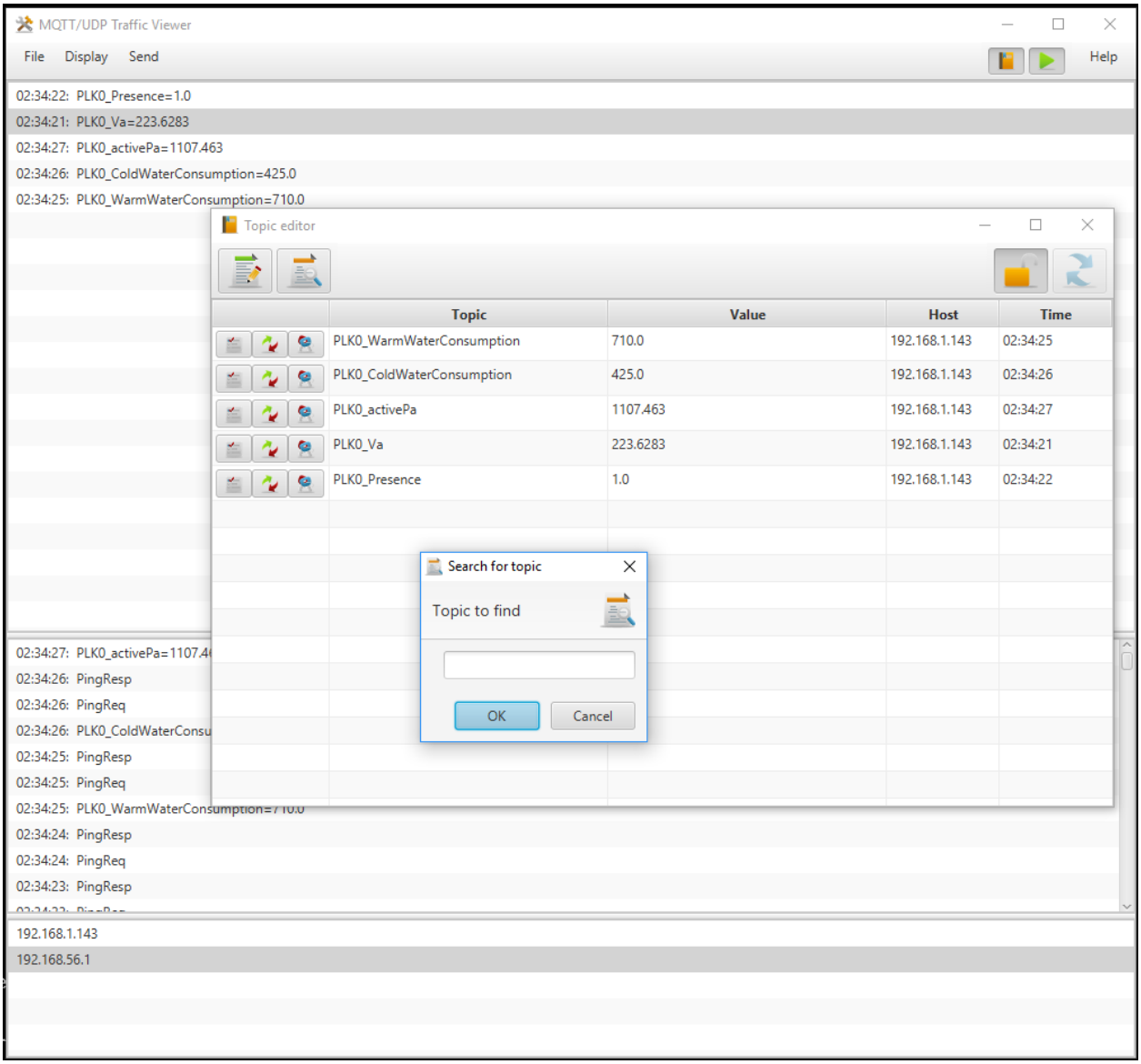


Fig. 1: Screenshot of MQTT/UDP viewer tool (Windows)

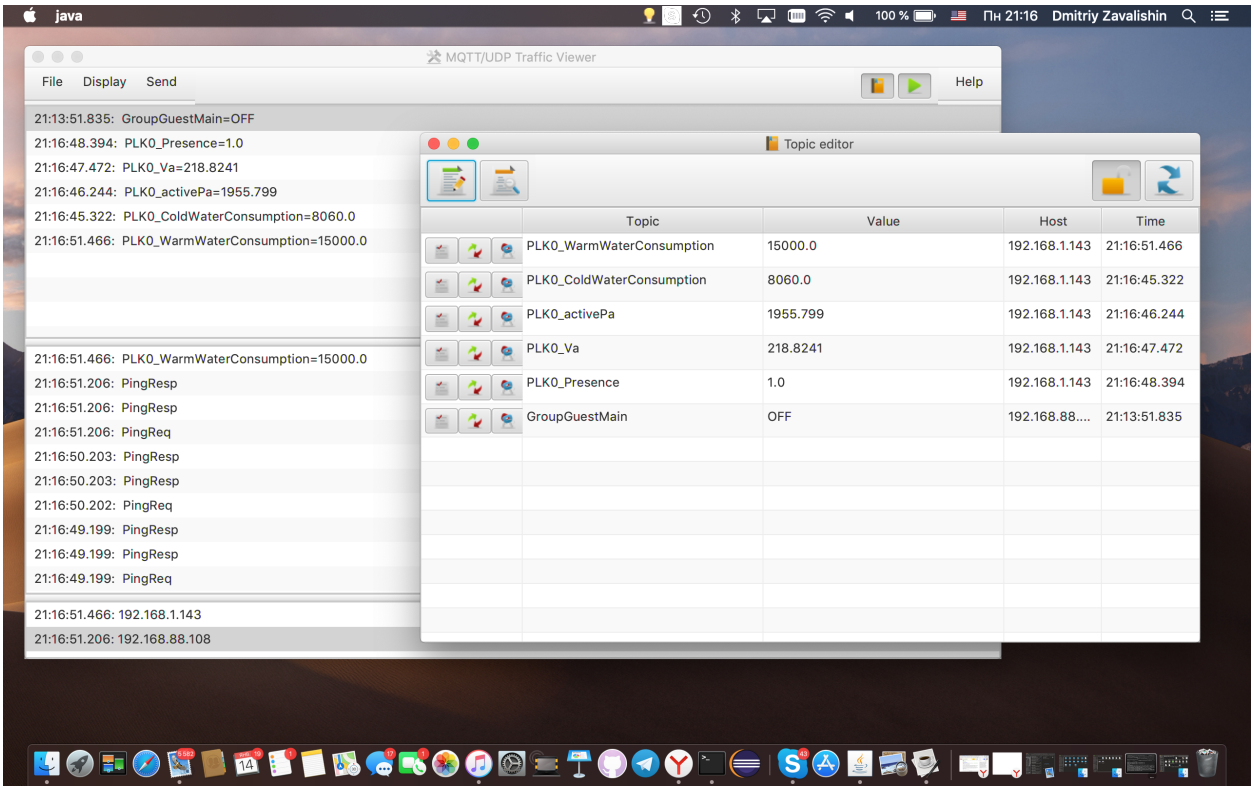


Fig. 2: Screenshot of MQTT/UDP viewer tool (Mac OS)
Being written in Java viewer works on Mac OS. It also must run on other operating systems with Java, but I did not tried it yet.

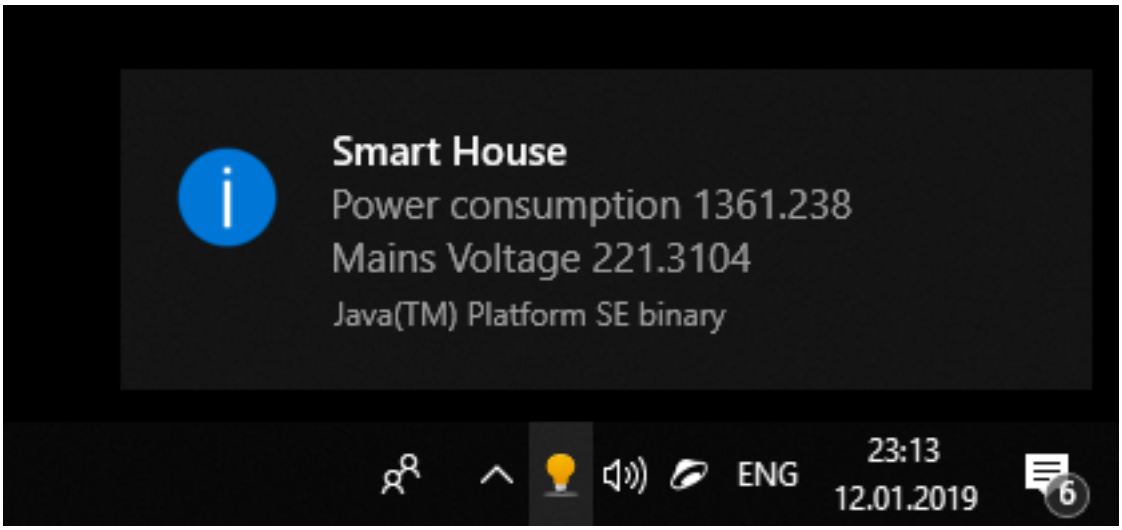


Fig. 3: Windows: tray icon informer
This informer is shown when you press right mouse button. It shows two items defined in .ini file, see reference. In this example mains voltage and total power consumption are shown.

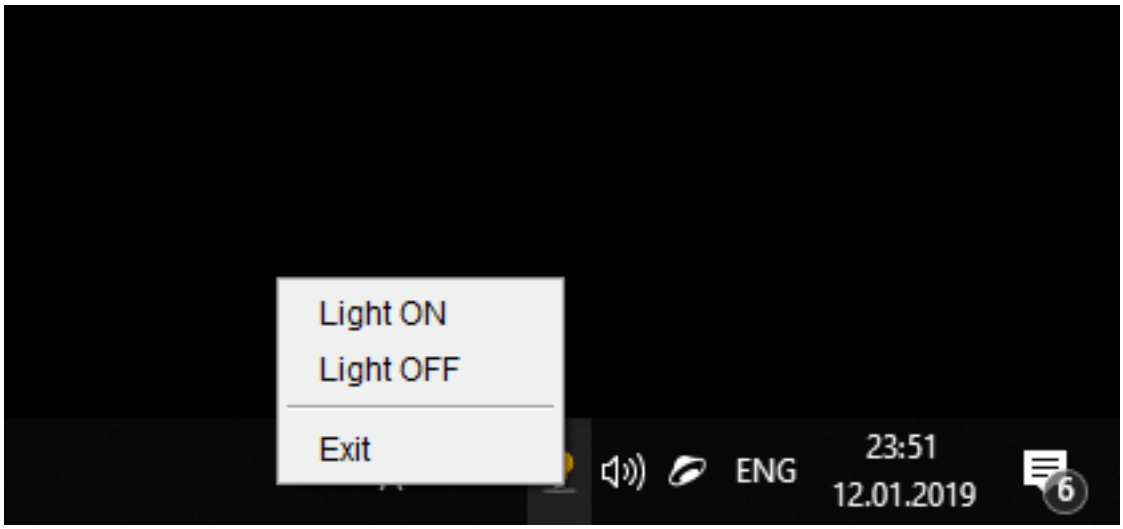


Fig. 4: Windows: tray icon menu
Menu is shown when right mouse button is pressed.

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.

### 6.4.2 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.

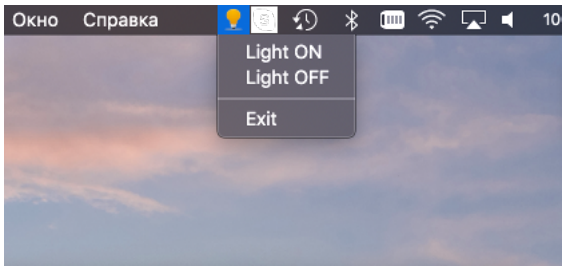


Fig. 5: Tray icon menu
Menu is shown when left mouse button is pressed.

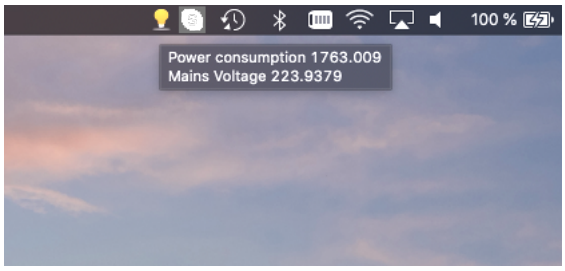


Fig. 6: Tray icon on mouse over
Tooltip is shown when mouse is over the icon.

CHAPTER 7

# Addendums

## 7.1 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

### 7.1.1 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 (see `tools/tray`) 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.

### 7.1.2 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.

## 7.2 Network

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.

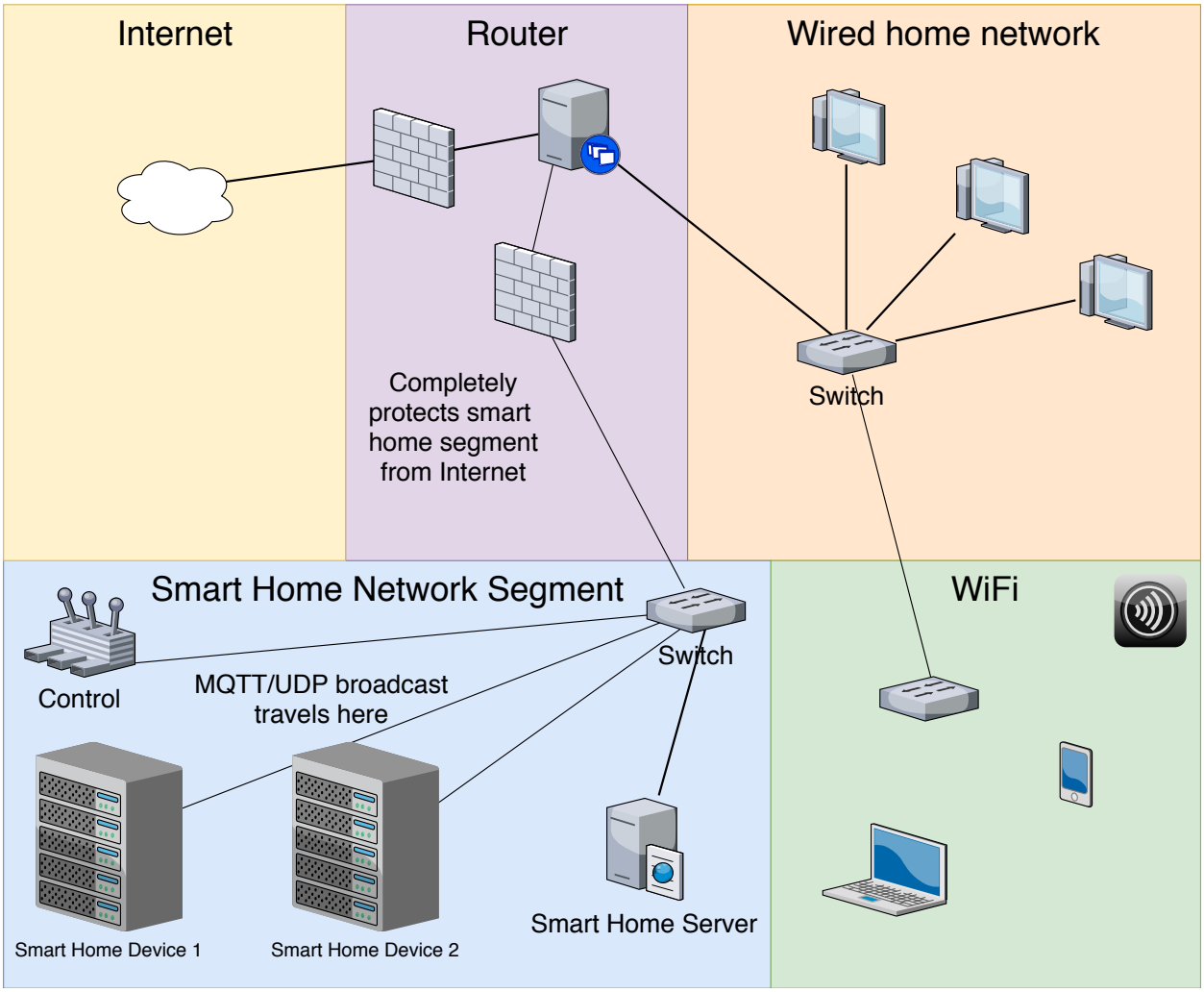


Fig. 1: Ideal structure of network.
Segment for a smart home is separated from local network for usual computers. MQTT/UDP data can be forwarded there on firewall, but not backwards.

## 7.3 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.

## 7.4 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. :)