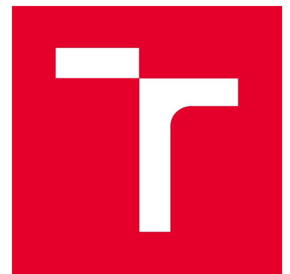
An IPFIX Extension for MQTT Protocol Monitoring

Ondřej Ryšavý
Petr Matoušek



Technical Report no. FIT-TR-2019-01
Faculty of Information Technology, Brno University of Technology

Abstract

Message Queuing Telemetry Transport is a client-server publish-subscribe protocol used for M2M communication in IoT environments. The present report aims at the analysis of protocol communication principles and protocol structure yielding to design a simplified MQTT parser and IPFIX extension providing MQTT specific attributes in Netflow records. The appendices contain i) Kaitai parser specification for MQTT and ii) the IPFIX extension for MQTT.

This report was created with the financial support of TA ČR in the frame of project TN01000077, TRACTOR: TRaffic Analysis and seCuriTy OpeRations for ICS/SCADA.

1. Introduction

MQTT (Message Queuing Telemetry Transport) is a client-server publish-subscribe protocol used for M2M communication in IoT environments. Originally, MQTT was developed for low-bandwidth and high latency networks in the late 1990s. Currently, the standard assumes to be used on TCP/IP stack. Applications either sending or receiving messages use specified TCP ports for MQTT message transport.
Nowadays, the following protocol specifications are available:

- MQTT v5.0 is an OASIS Standard. It replaces and supersedes MQTT v3.1.1. MQTT v5.0 adds a significant number of new features to MQTT while keeping much of the core in place. The major functional objectives are i) enhancements for scalability and large scale systems, ii) improved error reporting, iii) formalization of common patterns including capability discovery and request/response, iv) extensibility mechanisms including user properties, and v) performance improvements and support for small clients.
  https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf
- MQTT v3.1.1 is an older ISO/IEC 20922:2016 standard and the OASIS Standard. It defines the core principles and features of the MQTT protocol.
  http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.pdf
- MQTT-SN v1.2, formerly known as MQTT-S, is MQTT for Sensor Networks aimed at embedded devices on non-TCP/IP networks, such as Zigbee. MQTT-SN is a publish/subscribe messaging protocol for wireless sensor networks (WSN), with the aim of extending the MQTT protocol beyond the reach of TCP/IP infrastructure for Sensor and Actuator solutions.
  http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf

This document was created based on MQTT v3.1.1 and MQTT v5.0 standards. Non TCP/IP networks are not considered because of the target network monitoring technology is based on IPFIX and thus considers the environment of IP networks.

2. MQTT Principles

The core principle of MQTT is the client/server deployment of the Publish/Subscribe communication model. The MQTT system thus recognizes clients that can produce and consume messages and a server that provides message broker services. Application messages are organized into topics. The topic is a string label that enables the broker to route application messages to the corresponding subscribers.

The MQTT protocol operates by exchanging a series of MQTT Control Packets in a defined way. The MQTT packet uses a compact binary representation.

2.1. Publish/Subscribe Message Pattern

In Publish/Subscribe communication a collection of publishers produces messages consumed by subscribers. MQTT provides a decoupled model that implements a client/server architecture in which publishers and consumers are all clients and the server is represented by a message broker.

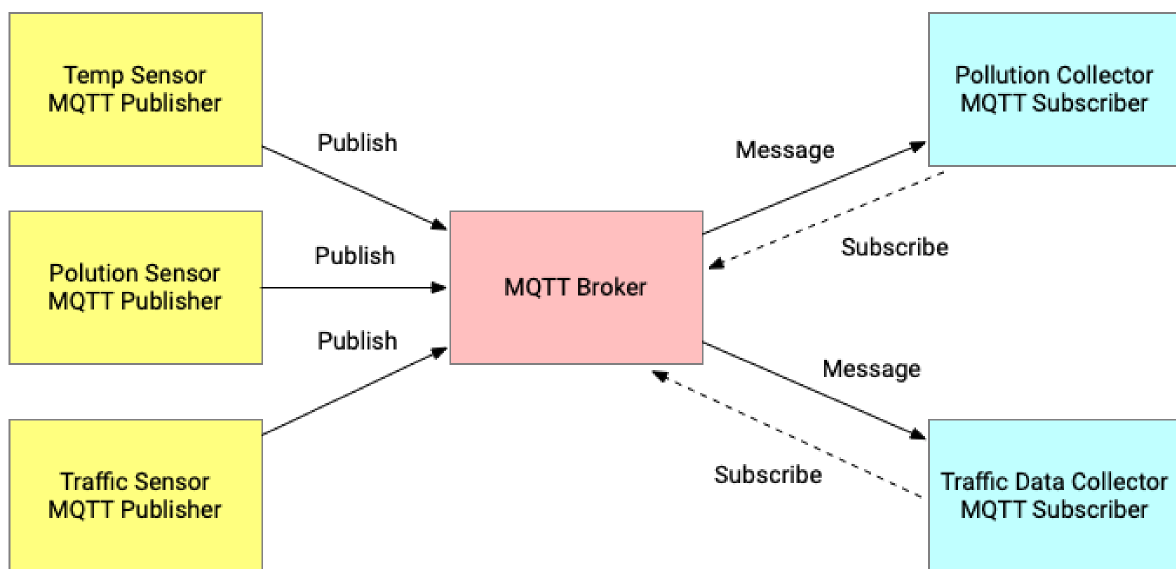


Figure 1: MQTT Architecture

All communication is thus transported through the MQTT broker that manages subscriptions of clients. The data carried by the MQTT protocol across the network for the application are called **Application Messages** and have always associated **Quality of Service** and a **Topic Name**. The topic name is a label attached to an Application Message which is matched against the Subscriptions known to the Server. The Application message is sent to each subscriber interested in the topic.

A client always establishes the TCP network session to the server - MQTT Broker. The client can:

- Publish messages that other clients might be interested in.
- Subscribe to the topic in order to receive corresponding messages
- Unsubscribe to stop receiving messages

A server provides services to all connected MQTT clients:

- Accepts Application Messages published by Clients.
- Processes Subscribe and Unsubscribe requests from Clients.
- Forwards Application Messages that match Client Subscriptions.

A client and a server establish and maintain a session. The session can be realized by one or more TCP connections.

2.2. MQTT Packet Structure

All MQTT packets share the same MQTT Control Packet structure, that consists of:

- Fixed Header
- Variable Header
- Payload

The fixed header is common to all control packets while the remaining parts depend on the type of the packet.

Fixed Header

The fixed header has at least two bytes that contain control packet type and its flags, and the length of the rest of the packet.

<table>
<tr><td>Bit</td><td>7</td><td>6</td><td>5</td><td>4</td><td>3</td><td>2</td><td>1</td><td>0</td></tr>
<tr><td>byte1</td><td colspan="4">MQTT Control Packet Type</td><td colspan="4">Specific Flags</td></tr>
<tr><td>byte2</td><td colspan="8">Length</td></tr>
</table>

MQTT Control Packet Type determines the purpose of the MQTT packet:

Name	Value	Direction	Description
Reserved	0	Forbidden	Reserved
CONNECT	1	Client to Server	Connection request
CONNACK	2	Server to Client	Connect acknowledgment
PUBLISH	3	Both	Publish message
PUBACK	4	Both	Publish acknowledgment
PUBREC	5	Both	Publish received
PUBREL	6	Both	Publish release

PUBCOMP	7	Both	Publish complete
SUBSCRIBE	8	Client to Server	Subscribe request
SUBACK	9	Server to Client	Subscribe acknowledgment
UNSUBSCRIBE	10	Client to Server	Unsubscribe request
UNSUBACK	11	Server to Client	Unsubscribe acknowledgment
PINGREQ	12	Client to Server	PING request
PINGRESP	13	Server to Client	PING response
DISCONNECT	14	Both	Disconnect notification
AUTH	15	Both	Authentication exchange

Flags are specific to each type of packet. In most of the cases, the flags are reserved and thus should have their predefined default value. The only PUBLISH packet has flags that carry additional information.

The *Remaining Length* is a Variable Byte Integer that represents the number of bytes remaining within the current Control Packet, including data in the Variable Header and the Payload. The Variable Byte Integer is encoded using an encoding scheme that uses a single byte for values up to 127. The principle of encoding is clear from the following table:

Digits	From	To
1	0 (0x00)	127 (0x7F)
2	128 (0x80, 0x01)	16,383 (0xFF, 0x7F)
3	16,384 (0x80, 0x80, 0x01)	2,097,151 (0xFF, 0xFF, 0x7F)
4	2,097,152 (0x80, 0x80, 0x80, 0x01)	268,435,455 (0xFF, 0xFF, 0xFF, 0x7F)

The MQTT specification provides also an algorithm for encoding/decoding values as Variable Byte Integer.

Variable Header

The presence of a variable header depends on the packet type. The MQTT specification provides a comprehensive definition of the possible content of variable headers for individual types of control packets. For this reason, we only provide summary information for the CONNECT packet for the sake of demonstration. The CONNECT Packet may have the following items in the variable header:

Field	Type/Length	Description
PROTOCOL NAME	UTF-8 Encoded String	The Protocol Name is a UTF-8 Encoded String that represents the protocol name “MQTT”. Some implementations use different string.
PROTOCOL VERSION	Unsigned Byte	The value of the Protocol Version field. For version 5.0 of the protocol is 5 (0x05). For the previous version, it is 3.
CONNECT FLAGS	Byte	Connect flags represents specified settings of the session as requested by the client.
KEEP ALIVE	Two Byte Integer	An integer value which is a time interval measured in seconds.
PROPERTIES	List of properties	Represents an optional set of properties provided for the connect packet. It begins with the property length field (Variable Byte Integer) that gives the total length of properties section in bytes.

The following example represents the CONNECT packet as dissected by Wireshark:

```
▼ MQ Telemetry Transport Protocol, Connect Command
  ▼ Header Flags: 0x10 (Connect Command)
      0001 .... = Message Type: Connect Command (1)
      .... 0000 = Reserved: 0
    Msg Len: 37
    Protocol Name Length: 6
    Protocol Name: MQIsdp
    Version: MQTT v3.1 (3)
  ▼ Connect Flags: 0x02
      0... .... = User Name Flag: Not set
      .0.. .... = Password Flag: Not set
      ..0. .... = Will Retain: Not set
      ...0 0... = QoS Level: At most once delivery (Fire and Forget) (0)
      .... .0.. = Will Flag: Not set
      .... ..1. = Clean Session Flag: Set
      .... ...0 = (Reserved): Not set
    Keep Alive: 5
    Client ID Length: 23
    Client ID: paho/34AAE54A75D839566E
```

The dissected packet represents an older MQTT v3 message. It can be also seen that it uses a non-standard protocol name MQIsdp. Note that Client Identifier (Client ID) is not part of the variable message but it is CONNECT packet payload.

Payload

Some MQTT Control Packets contain a Payload as the final part of the packet. The payload starts after the variable header and fills the packet until its end. The Payload of PUBLISH packet contains the Application Message that is being published. The content and format of the data are application-specific. The length of the Payload can be calculated by subtracting the length of the Variable Header from the Remaining Length field that is in the Fixed Header. It is valid for a PUBLISH packet to contain a zero-length Payload.

3. MQTT Protocol Parser

The MQTT protocol contains 16 different types of messages. Each message can have a variable header and payload. However, for the purpose of the IPFIX extension plugin, the parser does not necessarily consider all possible variable headers nor payload. The parser must be able to fully decode the fixed header and the selected protocol types. The definitions of MQTT structures are presented in Kaitai syntax.

The fixed header contains message type in the first half of the first byte. We are not interested in the other half of this byte. The next byte(s) stand for the length of the remaining data. The length uses the variable byte integer format (mqtt_varbyte).

```
mqtt_fixed_header:
  seq:
    - id: message_type
      type: b4
      enum: mqtt_message_type
    - id: flags
      type: b4
    - id: length
      type: mqtt_varbyte
enums:
  mqtt_message_type:
    0: reserved_0
    1: connect
    2: connack
    3: publish
    4: publish_ack
    5: publish_rec
    6: publish_rel
    7: publish_comp
    8: subscribe
    9: subscribe_ack
    10: unsubscribe
    11: unsubscribe_ack
    12: ping_request
    13: ping_response
    14: disconnect
    15: authentication
```

Fixed Header Format

The string encoding used by MQTT starts with the two byte length value followed by bytes representing the ASCII string.

```
mqtt_string:
  seq:
    - id: length
      type: u2
```

```
      - id: value
        type: str
        encoding: ascii
        size: length
```

MQTT String Format

The variable byte integer encoding format uses 1-4 bytes to represent an integer value. In each byte, only 7 bits are used to represent a number. The most significant bit of the byte defines whether the next byte is used or not. If the bit is 0 then it means that the byte is the last one in the number representation. Thus values less than 128 are represented by a single byte. Greater values need multiple bytes. For instance, number 200 is represented as two bytes: 72 and 1, because of 72 + 128 * 1 = 200.

```
  mqtt_varbyte:
    seq:
      - id: bytes
        type: u1
        repeat: until
        repeat-until: '(_ & 128) == 0'
    instances:
      value:
        value: '(bytes[0] & 127)
                + (bytes.size > 1 ? (bytes[1] & 127) * 128 : 0)
                + (bytes.size > 2 ? (bytes[2] & 127) * 128 * 128 : 0)
                + (bytes.size > 3 ? (bytes[3] & 127) * 128 * 128 * 128  : 0)'
```

MQTT Variable Byte Integer Format

Depending on the message type field in the fixed header, the next part of the message is one of the specific MQTT variable message headers. As we are not interested in a message payload, the parsing can finish after processing the required information from the variable header. One of the most informative messages is CONNECT:

```
mqtt_message_connect:
    seq:
      - id: protocol_name
        type: mqtt_string
      - id: protocol_version_number
        type: u1
      - id: connect_flags
        type: u1
      - id: keep_alive_timer
        type: u2
      - id:  client_id
        type: mqtt_string
      - id: rest_of_message
        size-eos: true
```

MQTT Connect Message Variable Header

Another source of information for IPFIX extension is the answer to the CONNECT message. Connect Acknowledgment message provides the return code:

```
mqtt_message_connack:
   seq:
     - id: topic_name_compression_response
       type: u1
     - id: connect_return_code
       type: u1
       enum: mqtt_connect_return_code

mqtt_connect_return_code:
   0: connection_accepted
   1: connection_refused_unacceptable_protocol_version
   2: connection_refused_identifier_rejected
   3: connection_refused_server_unavailable
   4: connection_refused_bad_username_or_password
   5: connection_refused_not_authorized
```

MQTT Connect Acknowledgment

4. Testing Environment

The purpose of an experimental environment is to verify that MQTT implementations do not significantly deviate from the standard and to obtain communication samples necessary for testing developed parsers and Flow Collector Plugin module. The environment uses the MOSQUITTO test server and the Tavern test client. The environment is used to generate MQTT data samples necessary to test the parser and IPFIX extension plugin. The various scenarios were implemented and the communication was captured.

Capture file	**Description**
mqtt_basic.pcap	A publisher connects to a broker and sends a few publish messages to several topics. The session is ended by the client by sending DISCONNECT.
mqtt_authfail.pcap	A CONNECT request to a broker fails (wrong authentication).
mqtt_publish.pcap	A client sends a lot of PUBLISH messages with many topics.
mqtt_ping.pcap	The scenario includes PING messages to preserve the activity.

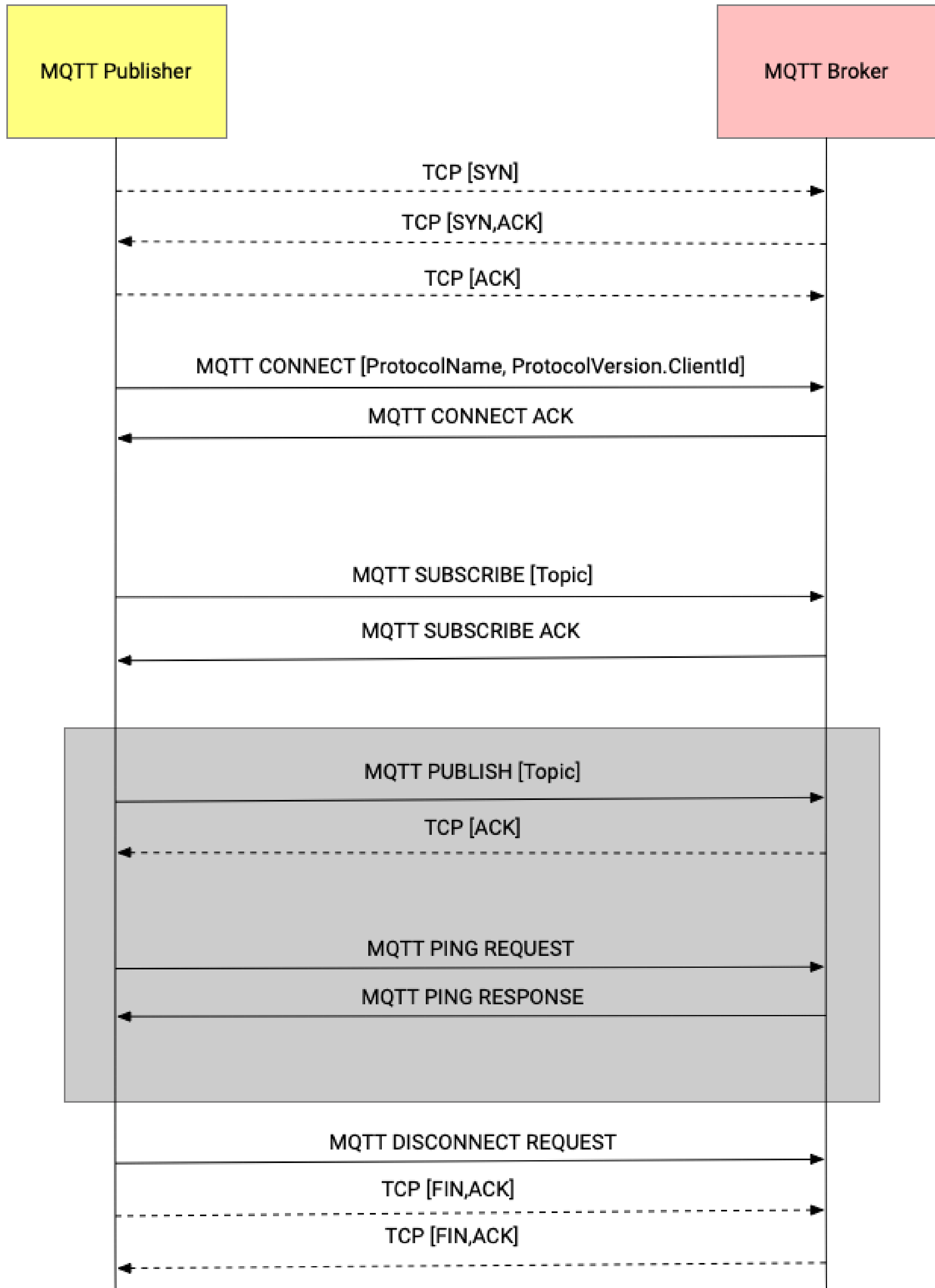


Figure 2: A TCP session between MQTT Publisher and Broker

5. Key MQTT Fields

The proposed IPFIX extension considers the scenario in which MQTT communication is monitored in order to provide visibility to network activities of MQTT enabled devices. The standard Netflow monitoring can only provide summary information on each TCP connection. As TCP connection is used in most cases for exchanging all MQTT messages between Client and Server the summary information does not provide much insight in the MQTT operations. Using MQTT IPFIX extension it should be possible:

- Identify the MQTT Client and Server
- Detect invalid attempts to connect
- Identify the MQTT protocol in use and its version
- See the number of MQTT messages and their types
- Optionally, see the topics of published messages and topics subscribed

MQTT Publisher/subscriber (Client) and MQTT Broker (Server) establish a durable TCP session that is used for MQTT message exchange. A typical MQTT communication that can be observed within a single TCP connection is demonstrated in Figure [TCP session]. Each session should start with **CONNECT** message that provides the following key information:

Name	Type	Description
MQTT_CLIENT_ID	STRING	The client identifier used to register a client in the Broker. This ID must be unique for the client. If the client needs to register again then it can use the same ID.
MQTT_VERSION	BYTE	The version of MQTT.
MQTT_PROTOCOL_NAME	STRING	The protocol name should be MQTT according to the standard, but existing implementations use different names, e.g., “MQIsdp”.

After receiving CONNECT message the server validates the client’s information and answer with CONNECT ACK, which carries the results of the connection request:

Name	Type	Description
MQTT_CONNECT_ACK	BYTE	The return code for CONNECT request. It can be one of the following: [0] connection_accepted, [1] connection_refused_unacceptable_protocol_version. [2] connection_refused_identifier_rejected, [3] connection_refused_server_unavailable, [4] connection_refused_bad_username_or_password, [5] connection_refused_not_authorized.

Depending on the client type, the subscriber clients then usually send **SUBSCRIBE** message that specifies a set of topics to be registered by the broker:

Name	**Type**	**Description**
MQTT_SUBSCRIBE_TOPIC	STRING	The topic to be registered. The topic can be specified on either specific topic or can be given using wildcards. For instance “mqttdotnet/subtest/#”, subscribes the client to all topics that start with “mqttdotnet/subtest”.

When the client is in the publisher’s role, it can send PUBLISH messages to the broker. If the client is subscribed to some topic it receives PUBLISH message from the broker. Thus, within the single TCP session the **PUBLISH** message can be sent in both directions:

Name	Type	Description
MQTT_PUBLISH_TOPIC	STRING	The topic of a message sent in PUBLISH.

The TCP session transfers MQTT messages between the MQTT Client and the MQTT server. A single TCP flow usually contains a single MQTT CONNECT message but many MQTT PUBLISH messages that may have different Topics see Figure 3. For IPFIX-based monitoring, it means that the IPFIX flow record needs to handle possible different Topics within a single TCP connection or abstract away the information about published topics.

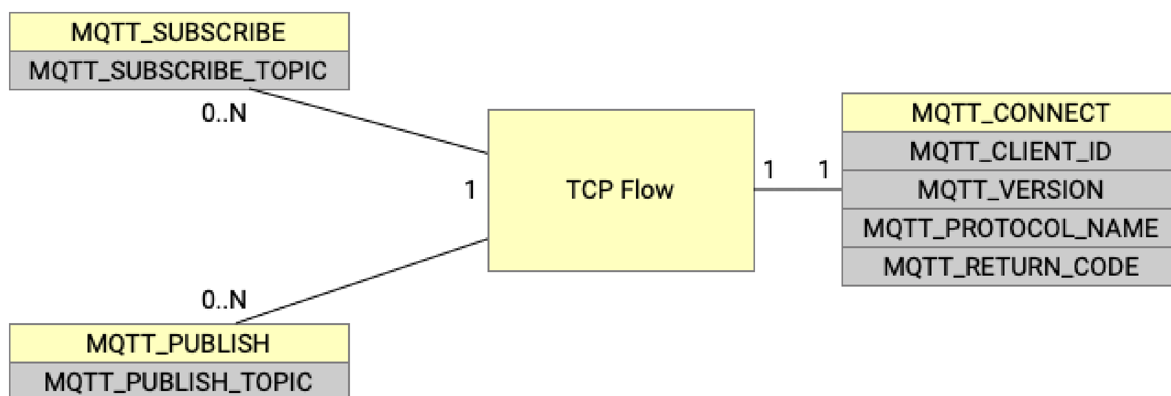


Figure 3: Relation of MQTT messages and TCP flow

6. IPFIX Extensions

In this section, the two IPFIX extensions for MQTT are proposed. The first of these extensions only collects statistical information about observed messages while the second one also identifies the message topics providing finer information but requiring to deal with sub-flows.

6.1. Basic MQTT IPFIX Extension

When the Topic information is not considered to be a source for IPFIX MQTT extension data, it is possible to create a single IPFIX flow for the entire TCP connection. In this case, only some information from CONNECT is captured and the counters for each MQTT type observed that may serve as the possible source of additional information for profiling and anomaly detection methods.

Name	Type
MQTT_CLIENT_ID	STRING
MQTT_VERSION	BYTE
MQTT_PROTOCOL_NAME	STRING
MQTT_CONNECT_ACK	BYTE
MQTT_PUB_COUNT	INTEGER
MQTT_SUB_COUNT	INTEGER

An example of IPFIX records for TCP connection between Client and Server is given in Figure 4 and Figure 5. The advantage of this extension is that the aggregated information fits TCP flow, but it does not provide any information about the Topics. Still, it is possible to create a simple profile describing the typical behavior of the system, but without topic information, it is not possible to distinguish different data points in the system.

```
FLOW: Tcp 192.168.11.32:55221 -> 5.196.95.208:1883
MQTT_CLIENT_ID: 7038a2a1-b97a-4faf-b9df-7270b32c9792
MQTT_VERSION: 3
MQTT_PROTOCOL_NAME: MQIsdp
MQTT_PUB_COUNT: 89
MQTT_SUB_COUNT: 2
```

Figure 4: MQTT IPFIX Flow for Client to Server direction

```
FLOW: Tcp 5.196.95.208:1883 -> 192.168.11.32:55221
```

```
MQTT_CONNECT_ACK: 0 (Connection Accepted)
MQTT_PUB_COUNT: 684
```

Figure 5: MQTT IPFIX Flow for Server to Client direction

6.2. MQTT IPFIX Extension considering Topics

Adding the Topic field to IPFIX extension enables us to observe what individual topics data are published. As a single TCP flow carries the PUBLISH messages for all subscribed topics it is necessary to create sub-flows. Each sub-flow corresponds to a single Topic published by the Client. Then the MQTT_PUBLISH_TOPIC is a key of the sub-flow.

Name	Type	Description
MQTT_PUBLISH_TOPIC	STRING	The topic of a message sent in PUBLISH message.

An example of IPFIX records for MQTT sub-flows within a single TCP connection between Client and Server is given in Figure 6 and Figure 7.

```
FLOW: Tcp 192.168.11.32:55221 -> 5.196.95.208:1883
MQTT_CLIENT_ID: 7038a2a1-b97a-4faf-b9df-7270b32c9792
MQTT_VERSION: 3
MQTT_PROTOCOL_NAME: MQIsdp
MQTT_SUB_COUNT: 2

SUBFLOW: a5d81be9b6/status
  MQTT_PUB_COUNT: 14

SUBFLOW: a5d81be9b6/senzors/temp
  MQTT_PUB_COUNT: 45

SUBFLOW: a5d81be9b6/senzors/humidity
  MQTT_PUB_COUNT: 30
```

Figure 6: MQTT IPFIX Flow for Client to Server direction

```
FLOW: Tcp 5.196.95.208:1883 -> 192.168.11.32:55221
MQTT_CONNECT_ACK: 0 (Connection Accepted)

SUBFLOW: master/control/service1
```

```
  MQTT_PUB_COUNT: 322

SUBFLOW: backup/control/service0
  MQTT_PUB_COUNT: 362
```

Figure 7: MQTT IPFIX Flow for Server to Client direction

7. Summary

This report provides an analysis of the MQTT connection. The principles of MQTT communication are explained. Also, the structure MQTT message is presented to provide a basis for the design of MQTT parser. Based on the analysis the two IPFIX templates were proposed. The Basic MQTT IPFIX Extension aggregates information for the whole MQTT session, providing summary information on different types of messages. While some important application-level information is missing, it is possible to use the captured information for creating a communication profile and detect possible deviations. The second extension preserves information about topics, which however requires to create sub-flows. While more complicated, it provides a deeper insight into MQTT communication. It enables, for instance, to monitor and perform anomaly detection on the level of individual topics.

References

- MQTT Version 5.0. Edited by Andrew Banks, Ed Briggs, Ken Borgendale and Rahul Gupta. 07 March 2019. OASIS Standard. Latest version: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf.
- MQTT Version 3.1.1. Edited by Andrew Banks and Rahul Gupta. 29 October 2014. OASIS Standard. Latest version: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.
- Mosquitto MQTT Broker Test site https://test.mosquitto.org
- MQTT Topics & Best Practices - MQTT Essentials: Part 5 https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/

Appendix

A. Kaitai MQTT Protocol Specification

This specification is a simplified parser of MQTT protocol.

```
meta:
  id: mqtt_packet
  title: MQTT is a Client Server publish/subscribe messaging transport protocol
  license: MIT
  endian: be
seq:
  - id: header
    type: mqtt_fixed_header
  - id: body
    size: header.length.value
    type:
      switch-on: header.message_type
      cases:
          'mqtt_message_type::reserved_0' : mqtt_message_reserved_0
          'mqtt_message_type::connect' : mqtt_message_connect
          'mqtt_message_type::connack' : mqtt_message_connack
          'mqtt_message_type::publish' : mqtt_message_publish
          'mqtt_message_type::publish_ack' : mqtt_message_publish_ack
          'mqtt_message_type::publish_rec' : mqtt_message_publish_rec
          'mqtt_message_type::publish_rel' : mqtt_message_publish_rel
          'mqtt_message_type::publish_comp' : mqtt_message_publish_comp
          'mqtt_message_type::subscribe' : mqtt_message_subscribe
          'mqtt_message_type::subscribe_ack' : mqtt_message_subscribe_ack
          'mqtt_message_type::unsubscribe' : mqtt_message_unsubscribe
          'mqtt_message_type::unsubscribe_ack' : mqtt_message_unsubscribe_ack
          'mqtt_message_type::ping_request' : mqtt_message_ping_request
          'mqtt_message_type::ping_response' : mqtt_message_response
          'mqtt_message_type::disconnect' : mqtt_message_disconnect
          'mqtt_message_type::authentication’ : mqtt_message_authentication
types:
  mqtt_fixed_header:
    seq:
      - id: message_type
        type: b4
        enum: mqtt_message_type
      - id: dup
        type: b1
      - id: qos
        type: b2
        enum: mqtt_qos
      - id: retain
        type: b1
      - id: length
        type: mqtt_varbyte

  mqtt_varbyte:
    seq:
      - id: bytes
        type: u1
        repeat: until
        repeat-until: '(_ & 128) == 0'
    instances:
      value:
        value: '(bytes[0] & 127)
```

```
              + (bytes.size > 1 ? (bytes[1] & 127) * 128 : 0)
              + (bytes.size > 2 ? (bytes[2] & 127) * 128 * 128 : 0)
              + (bytes.size > 3 ? (bytes[3] & 127) * 128 * 128 * 128  : 0)'

mqtt_connect_flags:
  seq:
    - id: username
      type: b1
    - id: password
      type: b1
    - id: will_retain
      type: b1
    - id: will_qos
      type: b2
      enum: mqtt_qos
    - id: will
      type: b1
    - id: clean_session
      type: b1
    - id: reserved
      type: b1
mqtt_subscribe_qos:
  seq:
    - id: reserved
      type: b6
    - id: qos
      type: b2
      enum: mqtt_qos
mqtt_subscribe_topic:
  seq:
    - id: topic_name
      type: mqtt_string
    - id: reserved
      type: b6
    - id: requested_qos
      type: b2
      enum: mqtt_qos
mqtt_string:
  seq:
    - id: length
      type: u2
    - id: value
      type: str
      encoding: ascii
      size: length

mqtt_message_reserved_0:
  seq:
    - id: content_of_message
      size-eos: true

mqtt_message_connect:
  seq:
    - id: protocol_name
      type: mqtt_string
    - id: protocol_version_number
      type: u1
    - id: connect_flags
      type: mqtt_connect_flags
    - id: keep_alive_timer
      type: u2
    - id:  client_id
      type: mqtt_string
    - id: will_topic
      type: mqtt_string
```

```
      if: connect_flags.will
    - id: will_message
      type: mqtt_string
      if: connect_flags.will
    - id: username
      type: mqtt_string
      if: connect_flags.username
    - id: password
      type: mqtt_string
      if: connect_flags.password

mqtt_message_connack:
    seq:
    - id: topic_name_compression_response
      type: u1
    - id: connect_return_code
      type: u1
      enum: mqtt_connect_return_code

mqtt_message_publish:
    seq:
    - id: topic
      type: mqtt_string
    - id: message_id
      type: u2
      if: '_parent.header.qos == mqtt_qos::at_least_once
          or _parent.header.qos == mqtt_qos::exactly_once'
    - id: payload
      size-eos: true

mqtt_message_publish_ack:
    seq:
    - id: message_id
      type: u2

mqtt_message_publish_rec:
    seq:
    - id: message_id
      type: u2

mqtt_message_publish_rel:
    seq:
    - id: message_id
      type: u2

mqtt_message_publish_comp:
    seq:
    - id: message_id
      type: u2

mqtt_message_subscribe:
    seq:
    - id: message_id
      type: u2
    - id: topics
      type: mqtt_subscribe_topic

mqtt_message_subscribe_ack:
    seq:
    - id: message_id
      type: u2
    - id: garanted_qos
      type: mqtt_subscribe_qos

mqtt_message_unsubscribe:
```

```
      seq:
      - id: message_id
        type: u2

  mqtt_message_unsubscribe_ack:
      seq:
      - id: message_id
        type: u2

  mqtt_message_ping_request:
      seq:
      - id: payload   # usually empty
        size-eos: true

  mqtt_message_response:
      seq:
      - id: payload   # usually empty
        size-eos: true

  mqtt_message_disconnect:
      seq:
      - id: payload   # usually empty
        size-eos: true

  mqtt_message_authentication:
      seq:
      - id: content_of_message
        size-eos: true
enums:
  mqtt_message_type:
    0: reserved_0
    1: connect
    2: connack
    3: publish
    4: publish_ack
    5: publish_rec
    6: publish_rel
    7: publish_comp
    8: subscribe
    9: subscribe_ack
    10: unsubscribe
    11: unsubscribe_ack
    12: ping_request
    13: ping_response
    14: disconnect
    15: authentication

  mqtt_connect_return_code:
    0: connection_accepted
    1: connection_refused_unacceptable_protocol_version
    2: connection_refused_identifier_rejected
    3: connection_refused_server_unavailable
    4: connection_refused_bad_username_or_password
    5: connection_refused_not_authorized

  mqtt_qos:
    0: at_most_once
    1: at_least_once
    2: exactly_once
    3: reserved
```

B. IPFIX Extension for MQTT

This appendix represents the IPFIX extension to be implemented in netflow monitoring tool. The IPFIX extension for MQTT adds the new fields to the standard IPFIX record. The fields depend on the direction of the flow. In addition to the basic extension specified in Section 6.1, the client IPFIX extension contains fields carrying information on an authenticated user, presence of password and the method in case of an extended authentication.

- Client (Publisher) to the server (Broker) MQTT flow:

Name	**Type**	**Description**
MQTT_CLIENT_ID	STRING	The client identifier used to register a client in the Broker. This ID must be unique for the client. If the client needs to register again then it can use the same ID.
MQTT_VERSION	BYTE	The version of MQTT.
MQTT_PROTOCOL_NAME	STRING	The protocol name should be MQTT according to the standard, but existing implementations may use different names, e.g., “MQIsdp”.
MQTT_PUB_COUNT	INTEGER	The total number of PUBLISH messages.
MQTT_SUB_COUNT	INTEGER	The total number of SUBSCRIBE messages.
MQTT_AUTH_USER	STRING	A username used for authentication specified in CONNECT message (optional).
MQTT_AUTH_PASS	BYTE	Presence of password in CONNECT message. Values = 1 (true), 0 (false)
MQTT_AUTH_METHOD	STRING	Name of the authentication method if extended authentication is used.

- Server (Broker) to a client (publisher) MQTT flow:

Name	**Type**	**Description**
MQTT_CLIENT_ID	STRING	The client identifier if not provided by the client can be generated by the server.
MQTT_VERSION	BYTE	The version of MQTT.
MQTT_CONNECT_ACK	BYTE	The return code for the CONNECT

		request. It can be one of the following: [0] connection_accepted, [1] connection_refused_unacceptable_protocol_version. [2] connection_refused_identifier_rejected, [3] connection_refused_server_unavailable, [4] connection_refused_bad_username_or_password, [5] connection_refused_not_authorized.
MQTT_PUB_COUNT	INTEGER	The total number of PUBLISH messages.

C. Tavern Test

The TAVERN testing framework is used to generate sample PCAP files.

```
test_name: Basic mqtt test
paho-mqtt:
  client:
    transport: tcp
    client_id: tavern-tester
  connect:
    host: test.mosquitto.org
    port: 1883
    timeout: 3
stages:
  - name: step 1 - ping
    mqtt_publish:
      topic: /device/123/ping
      payload: ping
  - name: step 2 - pong
    mqtt_publish:
      topic: /device/123/pong
      payload: pong
  - name: step 3 - ping
    mqtt_publish:
      topic: /device/123/ping
      payload: ping
  - name: step 4 - pong
    mqtt_publish:
      topic: /device/123/pong
      payload: pong
```

The content of test_mqtt_basic.tavern.yaml configuration file