TMS MQTT
DEVELOPERS GUIDE

Nov 2018

Web: http://www.tmssoftware.com
Email: info@tmssoftware.com

Index

Introduction

The TMS MQTT component as a full-featured Delphi MQTT Client that implements the **3.1.1 version** of the MQTT protocol.
http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html
The component is developed to work on all major operating systems (Windows, Mac, Linux, iOS and Android) and it supports the VCL, FMX and FPC frameworks.

It has the following Key feaures:
- Quality of Service 0, 1 and 2
- Automatic pinging
- Automatic reconnect
- Last Will and Testament (LWT)
- SSL connections
- Authentication

Usage

Installation

Prerequisites

The TMS MQTT Library has a dependency on Indy so make sure you have a working version of Indy installed.

Installation in Delphi

To install TMS MQTT in RAD studio, download and install the appropriate installer for your version of the IDE.

Installation in Lazarus/FPC

To install TMS MQTT in lazarus, download and open the TMS.MQTT.lpi package and install it manually into the IDE.

Getting started

At Design Time

The `TTMSMQTTClient` comes as a non-visual component that, after successful installation, can be found in the tool palette under **TMS MQTT**. Just add an instance of the client to your form to get started.
All necessary settings to connect the client will be available through the **Object Inspector**.

At Runtime

The `TTMSMQTTClient` can also be created at runtime. See below for an example on how to do that.

```
procedure TMQTTExampleForm.FormCreate(Sender: TObject);
begin
  MQTTClient := TTMSMQTTClient.Create(Self);
end;
```

Free Brokers

Instead of having to install your own broker first, note that to get started with MQTT, you can use one of the public free brokers listed on the following page:
http://moxd.io/2015/10/public-mqtt-brokers/

Connecting

Connection settings

Before connecting the client to a broker the following parameters can be set.

Property	Type	Description	Default value	Mandatory
`ClientID`	`string`	This is the unique ID for the Client	random string	no
`BrokerHostName`	`string`	Hostname of the broker you want to connect to		yes
`BrokerPort`	`integer`	The port to use when connecting to the broker	`1883`	no
`UseSSL`	`boolean`	Whether or not to connect through SSL	`false`	no
`Credentials`	`TMQTTCredentials`	The credentials to use when connecting (more info below)		no
`KeepAliveSettings`	`TMQTTKeepAliveSettings`	Setting to keep the connection alive (more info below)		no
`LastWillSettings`	`TMQTTLastWillSettings`	The LWT settings (more info below)		no

Connecting the client is done using the `Connect` prodecure on the `TTMSMQTTClient` instance.
This procedure takes one optional parameter to state if it should start a new session or continue with a previous session.
Connecting the client is an asynchronous process. That means that you will have to subscribe to the `OnConnectedStatusChanged` event to know when the connection was successful.

See below for a typical example on how to connect the client from code, the same thing can of course be achieved by using the Object Inspector in design-time.

```
procedure TMQTTExampleForm.ConnectButtonOnClick(Sender: TObject);
begin
  MQTTClient.ClientID := 'MyUniqueClientID';
  MQTTClient.BrokerHostName := 'broker.mydomain.com';
  MQTTClient.OnConnectedStatusChanged := ClientOnConnectedStatusChanged;
  MQTTClient.Connect;
end;

procedure TMQTTExampleForm.ClientOnConnectedStatusChanged(ASender: TObject; const
AConnected: Boolean; AStatus: TTMSMQTTConnectionStatus);
begin
  if (AConnected) then
  begin
    // The client is now connected and you can now start interacting with the broker.
    ShowMessage('We are connected!');
  end
  else
  begin
    // The client is NOT connected and any interaction with the broker will result in
an exception.
    case AStatus of
      csConnectionRejected_InvalidProtocolVersion,
      csConnectionRejected_InvalidIdentifier,
      csConnectionRejected_ServerUnavailable,
      csConnectionRejected_InvalidCredentials,
      csConnectionRejected_ClientNotAuthorized:
        ; // the connection is rejected by broker
      csConnectionLost:
        ; // the connection with the broker is lost
      csConnecting:
        ; // The client is trying to connect to the broker
      csReconnecting:
        ; // The client is trying to reconnect to the broker
    end;
  end;
end;
```

Username and Password

Some broker connections require a client to provide a **username** and **password** when connecting.
This can be achieved by editing the `Credentials` property on the `TTMSMQTTClient` instance.

Property	Type	Description	Default value	Mandatory
`Username`	`string`	The username		no
`Password`	`string`	The password		no

Code example

```
procedure TMQTTExampleForm.ConnectCredentialsButtonClick(Sender: TObject);
begin
  MQTTClient.ClientID := 'MyUniqueClientID';
  MQTTClient.BrokerHostName := 'broker.mydomain.com';
  MQTTClient.Credentials.Username := 'myUsername';
  MQTTClient.Credentials.Password := 'myPassword';
  MQTTClient.Connect;
end;
```

Keeping a connection alive

The MQTT protocol requires an open connection between the client and the broker at all times.
When connecting to the broker a client must provide a **keep alive interval**, this is the maximum allowed timespan in which no messages can be exchanged between the client and the broker. If this period is exceeded, the broker must disconnect the client.
To maintain an open connection, the client must thus send a `PINGREQ` packet to the broker if no other packets has been exchanged within the keep alive timespan.

The Keep Alive Settings can be configured using the `KeepAliveSettings` property on the `TTMSMQTTClient` instance **before connecting**.

Property	Type	Description	Default value	Mandatory
`KeepConnectionAlive`	`boolean`	Whether or not the client should keep the connection alive	`true`	no
`KeepAliveInterval`	`word`	The keep alive interval in seconds	`120`	no

Code example

```
procedure TMQTTExampleForm.ConnectKeepAliveButtonClick(Sender: TObject);
begin
  MQTTClient.ClientID := 'MyUniqueClientID';
  MQTTClient.BrokerHostName := 'broker.mydomain.com';
  MQTTClient.KeepAliveSettings.KeepConnectionAlive := true;     // Enable
Keep Alive
  MQTTClient.KeepAliveSettings.KeepAliveInterval := 60;         // 1 minute
interval
  MQTTClient.Connect;
end;
```

Automatic reconnecting

The TMS MQTT Client features a way to automatically reconnect to the broker if the connection gets lost unexpectedly. This feature is disabled by default.
Enabling automatic reconnecting can be done by editing the `KeepAliveSettings` property on the `TTMSMQTTClient` instance **before connecting**.

Property	Type	Description	Default value	Mandatory
AutoReconnect	boolean	Whether or not the client should try to restore a broken connection	false	no
AutoReconnectInterval	word	The interval to try reconnecting in seconds	30	no

Code example

```
procedure TMQTTExampleForm.ConnectAutoReconnectButtonClick(Sender: TObject);
begin
  MQTTClient.ClientID := 'MyUniqueClientID';
  MQTTClient.BrokerHostName := 'broker.mydomain.com';
  MQTTClient.KeepAliveSettings.AutoReconnect := true;            // Enable
Auto-Reconnect
  MQTTClient.KeepAliveSettings.AutoReconnectInterval := 10;      // Try
reconnecting every 10 seconds
  MQTTClient.Connect;
end;
```

Last Will Testament (LWT)

The MQTT protocol allows a client to provide an optional **Last Will Testament (LWT)** when connecting to a broker.
When provided, the broker will publish a message to the given topic as soon as it lost the connection with the client and didnt recieved a proper disconnect message.
The last will is a way to notify other clients that a client has lost it's connection.

The LWT can be configured using the `LastWillSettings` property on the `TTMSMQTTClient` instance **before connecting**.

The following parameters can be provided:

Property	Type	Description	Default value	Mandatory
Topic	string	The topic that should be used to publish the LWT message		yes
WillMessage	string	The actual message		no
Retain	boolean	Whether or not the message should be retained on the broker	false	no
QoS	TMQTTQoS	the Quality of Service that should be used to send the LWT	qosAtMostOnce	no

Property	Type	Description	Default value	Mandatory
		message		

Code example

```
procedure TMQTTExampleForm.ConnectLWTClick(Sender: TObject);
begin
  MQTTClient.ClientID := 'MyUniqueClientID';
  MQTTClient.BrokerHostName := 'broker.mydomain.com';
  MQTTClient.LastWillSettings.Topic := 'clients/disconnected';
  MQTTClient.LastWillSettings.WillMessage := 'MyUniqueClientID';
  MQTTClient.Connect;
end;
```

Publishing

After a connection has been established you can start publishing messages to a specific topic.
This can be done by calling the `Publish` method on the `TTMSMQTTClient` instance.
The method takes 4 parameters of which only the first is mandatory.

Topic Name

The topic to where you publish should be a valid UTF8 string and should be at least 1 character long.
The topic name can consist of one or more levels separated by a forward slash (/) and **cannot contain any wildcard characters** (+ OR #).

Here are some examples of valid topics to publish to:
- myapp/heatsensor
- myapp/garage/temperature
- m/g/t
- humidity

*Please note that the topics are **case-sensitive**.*

Packet Payload

The payload of a packet can be sent as a string or as an array of bytes (`TBytes`).
This parameter is optional, by default a `nil` value will be sent.

Quality of Service (QoS)

This parameter defines the level of guarantee that a message will be received by the broker.
You have 3 options:

Name	Description
`qosAtMostOnce`	At most once delivery
`qosAtLeastOnce`	At least once delivery
`qosExactlyOnce`	Exactly once delivery

Retain-flag

This parameter states whether the payload of the packet should be retained by the broker or not. The broker will only store one value per topic, so only the last value will be retained and sent to the subscribers.

By default this is set to `false`.

Code example

```
procedure TMQTTExampleForm.PublishButtonClick(Sender: TObject);
var
  packetId: Word;
begin
  packetId := MQTTClient.Publish(
    'myapp/hellotopic',   // the topic to publish to
    'Hello World!',       // the content (payload) of the packet (string or TBytes) (default nil)
    qosAtLeastOnce,       // the Quality of Service that should be used (default qosAtMostOnce)
    true                  // whether or not to retain the message on the broker (default false)
    );
end;
```

Subscribing

The `Subscribe` function on the `TTMSMQTTClient` instance can be used to subscribe to one or more topics. The function return the packetId of the subscribe packet sent to the broker.

Topic Filter

A topic filter should consist of at least one character and may contain one or more wildcard character. There are two types of wildcard characters:

Single-level wildcard +

The single-level wildcard character can be used to match all topics within a single level of topics. A topic filter can contain one or more of these wildcards.

Some valid examples:
- garage/sensor1/+
- garage/+/temperature
- +/+/temperature

Multi-level wildcard #

The multi-level wildcard matches multiple levels and can only be used once in a topic filter.
It should always be the last character of the filter and it should always be preceded by the level separator (/) unless it is the only character in the filter.

Some valid examples:
- garage/#
- garage/sensor1/#

The single-level and multi-level wildcards can also be combined in a topic filter. The following examples are also valid:

- garage/+/status/#
- +/temperature/#

Subscription Quality of Service

The Quality of Service that should be used when sending the packets to the client.
By default this is set to `qosAtMostOnce`.

Code example

```
procedure TMQTTExampleForm.SubscribeButtonClick(Sender: TObject);
var
  packetId: Word;
begin
  packetId := MQTTClient.Subscribe(
    'myapp/sensors/#',      // the topic filter
    qosAtMostOnce           // the Quality of Service that should be used
(default qosAtMostOnce)
    );
end;
```

Ensure a subscription was successful

Subscribing to one or more topics is an asynchronous process. The Client will send a `SUBSCRIBE` packet to broker and, if successful, the broker will return a `SUBACK` packet containing an *accepted*-flag for each topic that was requested.
By persisting the `PacketID` returned by the `Subscribe` method and by listening to the `OnSubscriptionAcknowledged` event we can make sure that a specific subscribe was successful or not.

```
procedure TMQTTExampleForm.ValidateSubscribeButtonClick(Sender: TObject);
begin
  MQTTClient.OnSubscriptionAcknowledged := SubscriptionAcknowledged;
  FSubscribeRequestPacketId := MQTTClient.Subscribe('myapp/sensors/#');
end;

procedure TMQTTExampleForm.SubscriptionAcknowledged(ASender: TObject;
APacketID: Word; ASubscriptions: TMQTTSubscriptions);
begin
  if (APacketID = FSubscribeRequestPacketId) and ASubscriptions[0].Accepted
then
  begin
    ShowMessage('We are subscribed!');
  end;
end;
```

Unsubscribing

You can use the `Unsubscribe` to cancel the subscription on one or more topics.

```
procedure TMQTTExampleForm.UnsubscribeButtonClick(Sender: TObject);
begin
  FUnSubscribeRequestPacketId := MQTTClient.Unsubscribe('myapp/#');
end;
```

To validate that the unsubscribe packet has been acknowledged by the broker you can listen to the `OnPacketReceived` and check for an incoming `UNSUBACK` packet with the same PacketID as the unsubscribe request.

```
procedure TMQTTExampleForm.FormCreate(Sender: TObject);
begin
  MQTTClient := TTMSMQTTClient.Create(Self);
  MQTTClient.OnPacketReceived := PacketReceived;
end;

procedure TMQTTExampleForm.PacketReceived(ASender: TObject; APacketInfo:
TMQTTPacketInfo);
begin
  if (APacketInfo.PacketType = mtUNSUBACK) AND (APacketInfo.PacketId =
FUnSubscribeRequestPacketId) then
  begin
     ShowMessage('We are unsubscribed!');
  end;
end;
```

Receiving published messages

After subscribing to a topic, the broker will start sending packets to the client.
To work with these packets in your application you can listen to the `OnPublishReceived` event on the `TTMSMQTTClient` instance.

```
procedure TMQTTExampleForm.FormCreate(Sender: TObject);
begin
  MQTTClient := TTMSMQTTClient.Create(Self);
  MQTTClient.OnPublishReceived := PublishReceived;
end;

procedure TMQTTExampleForm.PublishReceived(ASender: TObject; APacketID: Word;
ATopic: string; APayload: TBytes);
begin
  ShowMessage('Message received on topic: ' + ATopic + sLineBreak +
TEncoding.UTF8.GetString(APayload));
end;
```

Please note that due to compatibility issues with generics in C++ builder, there is a separate event `OnPublishReceivedEx` that should be used in C++ projects.
The event handler in c++ would look something like the code below:

```
void __fastcall TForm2::TMSMQTTClient1PublishReceivedEx(TObject *ASender,
WORD APacketID, UnicodeString ATopic, TTMSMQTTBytes APayload)
{
   ShowMessage(TEncoding::UTF8->GetString(APayload));
}
```

Pinging

If you enable the **keep alive** functionality the client will periodically send *ping request* (`PINGREQ`) packets to the broker to keep the connection alive. You can however send a manual *ping request* by calling the `Ping` procedure on the `TTMSMQTTClient` instance. If the client receives a *ping request* it will automatically respond with a *ping response* (`PINGRESP`) packet, you don't need to do that manually.

Monitoring in- and outgoing packets

You can monitor all outgoing and incoming packets by subscribing to the `OnPacketReceived` and `OnPacketSent` events on the `TTMSMQTTClient` instance. These events provide basic information about the packets in the form of a `TMQTTPacketInfo` record. The record contains the following information:

Property	Type	Description
`PacketId`	`Word`	The PacketID
`PacketType`	`TMQTTPacketType`	The type of the packet
`PacketQos`	`TMQTTQoS`	The Quality of Service
`IsDuplicate`	`Boolean`	Whether it is a duplicate message (in case of Qos > 0)
`IsRetained`	`Boolean`	Whether it is a retained message

***Please note** that this information does not contain any payload information, subscribe to one of the other events to know more about specific types of packets that are received.*

Logging

When debugging your application it might be handy to enable logging on the MQTTClient.
This can be done by creating a logger instance and assigning it to the `Logger` property of the `TTMSMQTTClient` component.
By default there are 2 Loggers available in the tool palette, the `TTMSMQTTLogger` and the `TTMSMQTTFileLogger`.

The `TTMSMQTTLogger` will write the log messages in the output window, the `TTMSMQTTFileLogger` will write to a file.
You can of cource create your own logger by inheriting from the existing classes.

The logger has a property `Verbosity` that can be adjusted to manipulate the amount of details you want to see in the logs.

Demo

The TMS MQTT Components comes with a demo application for VCL, FMX and FPC.
The demo applications are very simple and allow you to, after entering your name and a message, put a marker on a world map.
The map can be viewed on http://www.tmssoftware.com/mqtt/demo/ and by clicking on the markers you can send messages back to the client applications.