IMPLEMENTATION OF MQTT-SN PROTOCOL

INTRODUCTION:

There is a recent increase of interest in Wireless Sensor Networks (WSNs), both from commercial and technical point of view, due to their simplicity, low cost and easy deployment. Those networks can serve different purposes, from measurement and detection, to automation and process control. A typical WSN consists of a large number of battery-operated sensors and actuators (SAs), which are usually equipped with a limited amount of storage and processing capabilities. It is important that those devices communicate wirelessly, since the number of SA-nodes is typically very large, and the cost of deployment of a wired infrastructure is prohibitively expensive. Such a network is by nature very dynamic: the wireless links may temporarily break at any time, and nodes may fail and be replaced very often. In such situations the conventional approach of using addresses for communicating with the individual nodes may become a nightmare.

Applications residing on the fixed network and requiring interactions with the wireless SA devices would need to manage and maintain means to communicate with a large number of nodes. In most cases they do not need to know the address or identity of the devices which deliver the information, but are more interested in the content of the data. For example, an asset tracking application is more interested in the current location of a certain asset than in the network address of the GPS receivers that deliver that information. In addition, several applications may have interest in the same sensor data but for different purposes. In this case the SA nodes would need to manage and maintain communication means with multiple applications in parallel. This might exceed the limited capabilities of the simple and low-cost SA devices. Another problem is the difference in the addressing schemes between the networks involved. For example, how does an application residing on a TCP/IP-based network address a SA device running on a ZigBee based wireless network.

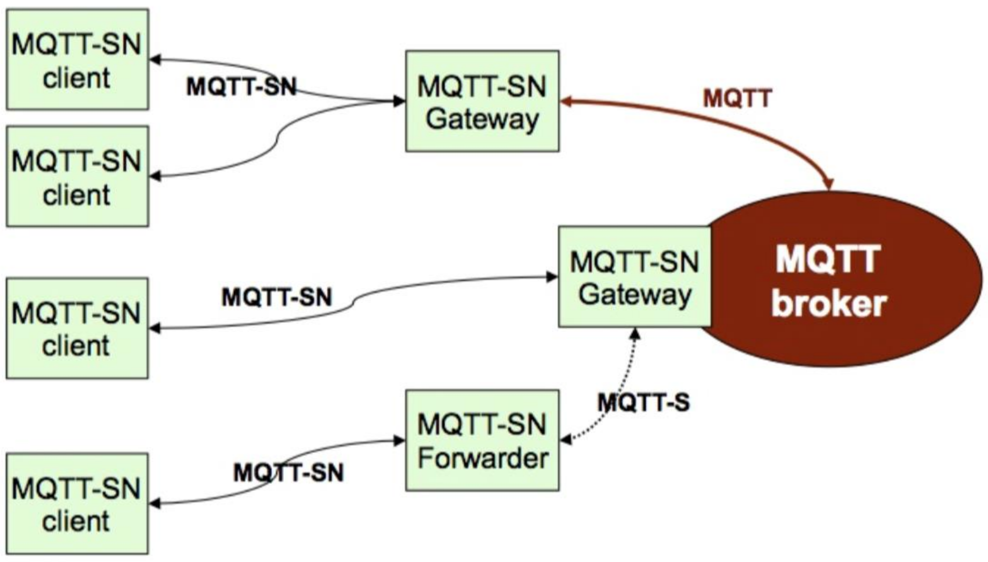
The problem described above may be overcome by using a data-centric communication approach, in which information is delivered to the receivers not based on their network addresses but rather as a function of their contents and interests. One well-known example of data-centric communication is the “Publish/Subscribe” (pub/sub) messaging system which is already being widely used in enterprise networks, mainly due to their scalability and support of dynamic network topology. Extending the enterprise pub/sub system into the WSNs also enables a seamless integration of the WSNs into the enterprise network, thus making the field data collected by the SAs available to all applications as any other enterprise information and enabling the control of the SAs from any enterprise application.

This can be for example achieved by using the MQTT protocol, where MQTT stands for **Message Queue Telemetry Transport**, which is an open and lightweight publish/subscribe protocol designed specifically for machine-to-machine and mobile applications. It is optimized for communications over networks where bandwidth is at a premium or where the network connection could be intermittent. However MQTT requires an underlying network, such as TCP/IP, that provides an ordered lossless connection capability and this is too complex for very simple, small footprint, and low-cost devices such as wireless SAs.

Wireless radio links have in general a higher failure rates than wired ones due to their susceptibility to fading and interference disturbances. They have also a lower transmission rate. For example, WSNs based on the IEEE 802.15.4 standard provide a maximum bandwidth of 250 kbit/s in the 2.4 GHz band. Moreover, to be resistant against transmission errors, their packets have a very short length. In the case of IEEE 802.15.4, the packet length at the physical layer is limited to 128 bytes. Half of these 128 bytes could be taken away by the overhead information required by supporting functions such as MAC layer, networking, security, etc.

The purpose of this document is to specify MQTT-SN, a pub/sub protocol for wireless sensor networks. MQTT-SN can be considered as a version of MQTT which is adapted to the peculiarities of a wireless communication environment. MQTT-SN is also optimized for implementation on low-cost, battery-operated devices with limited processing and storage resources. MQTT-SN allows to build up a network of constrained devices with a central broker connected to many clients. Message distribution is controlled by a message bus like mechanism and topic registration.

The architecture of MQTT-SN protocol is depicted below:



In accordance with the MQTT-SN protocol, a broker node is responsible for managing subscriptions as well as storing and sending publications to corresponding subscriber nodes. In addition, the broker is able to provide the same services to nodes in external networks (Internet) by using a gateway node, This node is located at the edge of the network and is in charge of message mapping between the MQTT (for external network) and MQTT-SN protocols. A MQTT-SN gateway may or may not be integrated within the broker. The sensor nodes are able to act as publisher, subscriber or relay nodes in case of a multi-hop scenario, in order to establish connection with the broker node.

There are three kinds of MQTT-SN components: MQTT-SN clients, MQTT-SN gateways (GW), and MQTT server. MQTT-SN clients connect themselves to a MQTT server via a MQTT-SN GW using the MQTT-SN protocol. A MQTT-SN GW may or may not be integrated with a MQTT server. In case of a stand-alone GW the MQTT protocol is used between the MQTT server and the MQTT-SN GW. Its main function is the translation between MQTT and MQTT-SN and thus separating the clients and server as two separate entities.

IMPLEMENTATION OF MQTT-SN PROTOCOL

MQTT vs MQTT-SN Protocol:

MQTT-S, which is now typically referred to as MQTT-SN (MQTT for Sensor Networks) is a version of MQTT that has been adapted to better function on devices where low power usage is a primary concern. The protocol has also be adapted for communication over low bandwidth links that are capable of only short messages and where network interruptions are common. Such systems would include low power microcontrollers communicating via low bitrate wired serial links, low power wireless links (e.g. 802.15.4, ZigBee, proprietary wireless serial, etc.), IR, ultrasonic, or some other type of link.

Both MQTT and MQTT-SN are client-server protocols, for which a server is needed to distribute messages between the client applications. They also both use the publish-subscribe paradigm, rather than queueing: a receiving application subscribes to topics of interest, and the sending application publishes messages to topics.

Advantages of MQTT-SN over MQTT:

- MQTT-SN supports **topic ID** instead of topic name. First client sends a registration request with topic name and topic ID (2 octets) to a broker. After the registration is accepted, client uses topic ID to refer the topic name. This saves media bandwidth and device memory - it is quite expensive to keep and send topic name in memory for each publish message.
- The **CONNECT** message is split into three messages. The two additional ones are optional and used to transfer the Will topic and the Will message to the server.
- **“Pre-defined” topic ids and “short” topic names** are introduced, for which no registration is required. Predefined topic ids are also a two-byte long replacement of the topic name, their mapping to the topic names is however known in advance by both the client’s application and the gateway/server. Therefore both sides can start using pre-defined topic ids; there is no need for a registration as in the case of “normal” topic ids mentioned above.
- A discovery procedure helps clients without a **pre-configured server/gateway’s** address to discover the actual network address of an operating server/gateway. Multiple gateways may be present at the same time within a single wireless network and can co-operate in a load-sharing or stand-by mode.
- The semantic of a “**clean session**” is extended to the Will feature, i.e. not only client’s subscriptions are persistent, but also Will topic and Will message. A client can also modify its Will topic and Will message during a session.
- A new offline **keep-alive** procedure is defined for the support of **sleeping clients**. With this procedure, battery-operated devices can go to a sleeping state during which all messages destined to them are buffered at the server/gateway and delivered later to them when they wake up.

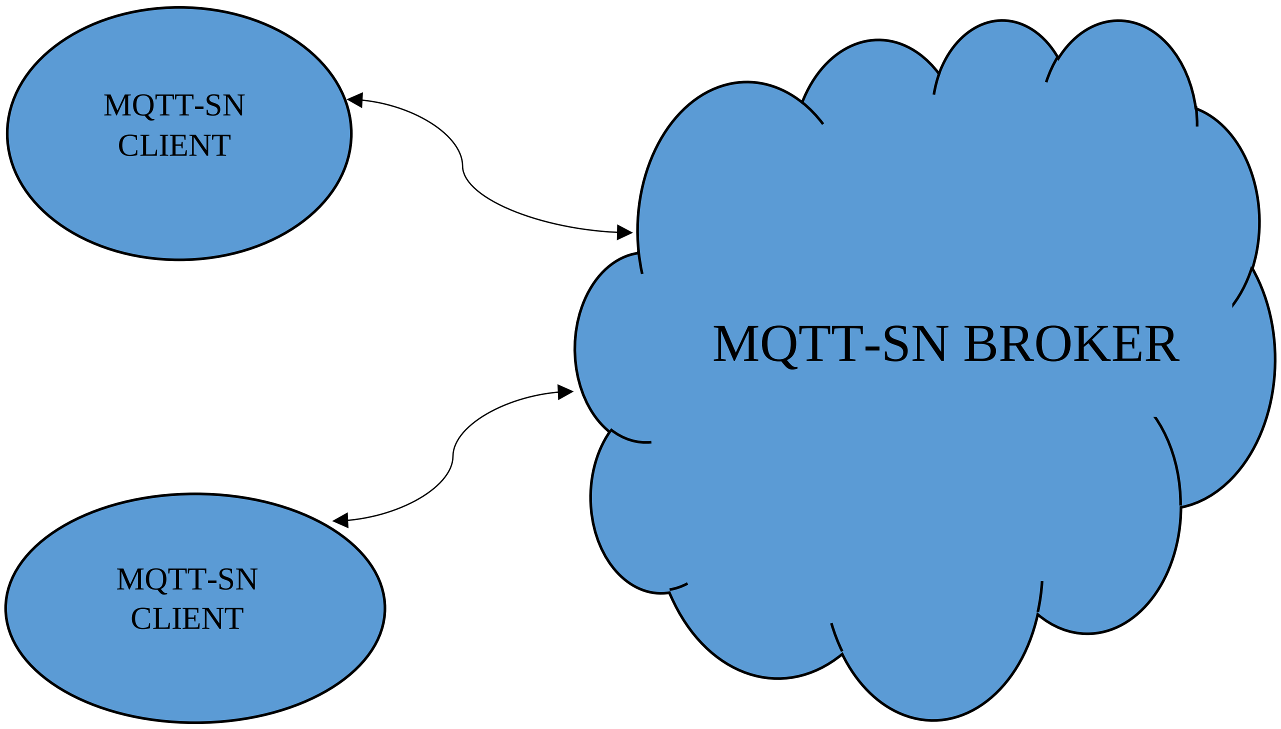
Disadvantages of MQTT-SN over MQTT:

- Disadvantage is that you need some sort of gateway, which is nothing else than a **TCP or UDP** stack moved to a different device. This can also be a simple device (e.g.: Arduino Uno) just serving multiple MQTT-SN devices without doing other job.
- MQTT-SN is not well supported. Doing datagrams directly to the cloud is inefficient and error prone, hence the gateway in the middle which basically collects these datagrams, and moves stuff to the cloud via reliable, TCP-based MQTT.

IMPLEMENTATION OF MQTT-SN PROTOCOL

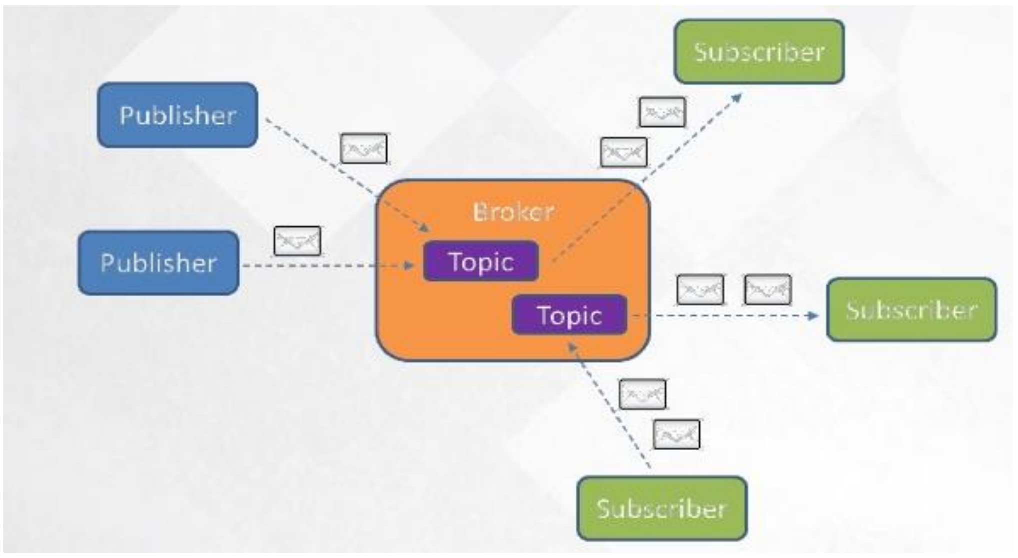
MQTT-SN IMPLEMENTATION:

Here in this project, we have eliminated the MQTT-SN gateway and only communication between Client and Broker has been implemented. Hence the need for the translation device from MQTT-SN to MQTT has been eliminated. With this implementation, the necessity of an actual hardware device which serves as gateway has been carved out and client can send communicate by sending messages in the MQTT-SN protocol defined format and can expect a response in the same format as described by the protocol.



All message exchanges are end-to-end between the MQTT-SN client and the MQTT SN server, all functions and features that are implemented by the server can be offered to the client. The broker acts as a conduit and controls the message to and from the client based on pub-sub mechanism and registration of topics. Even if very few clients are connected, the above said mechanism is necessary to support robustness. To reduce the size of the payloads, the data packets are numbered by numeric topic ids rather than long topic names. This particular feature which is different from the MQTT reduces the readability of the topics, but elegantly reduces the size of the packets. The negotiation for the topic ids has to happen from the client side. The protocol does not guarantee any kind of operation when a restart happens, which might have erased all the topics. These kind of operations has to be taken care at the application level.

MQTT-SN defines 27 type of control messages for the communication between clients and server such as CONNECT, PUBLISH, SUBSCRIBE, UNSUBSCRIBE, DISCONNECT etc. Here the client implements 4 major functionalities such as CONNECT, REGISTER, SUBSCRIBE & PUBLISH. For each of this message sent, appropriate responses from the server is sent as an Acknowledgment to requests obtained based on the response codes generated. The acknowledge messages are CONNACK, REGACK, SUBACK and PUBACK respectively. Every MQTT-SN message contains a 2 or 4 bytes message header and variable payload. Size of the smallest MQTT-SN message is 2 bytes. Largest MQTT-SN message is of size 65535 bytes.



MESSAGE INTERFACE:

Here in this section we discuss the format of messages sent out by the client and responses provided by the server. The general message format of all messages are given below:

MESSAGE HEADER (2 to 4 Octets)	MESSAGE VARIABLE PART (n Octets)

A MQTT-SN message consists of two parts: a 2- or 4-octet long header and an optional variable part. While the header is always present and contains the same fields, the presence and content of the variable part depend on the type of the considered message.

Message Headers:

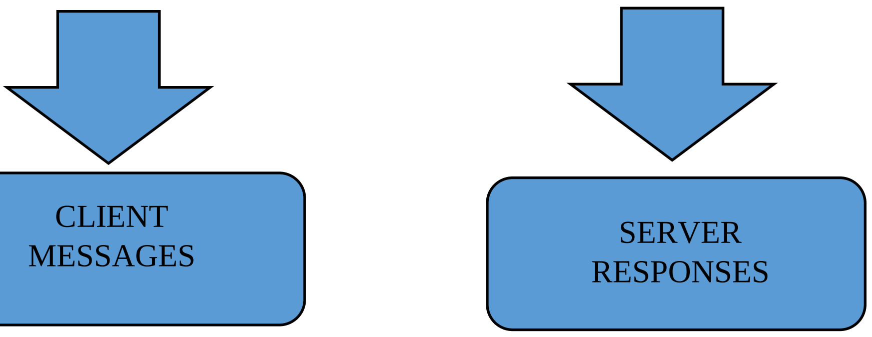
The message headers contain the information about the length of the total message and also the kind of message received/sent.

LENGTH (1 to 3 Octets)	MESSAGE TYPE (1 Octets)

- **Length:** The Length field is either 1- or 3-octet long and specifies the total number of octets contained in the message (including the Length field itself). If the first octet of the Length field is coded “0x01” then the Length field is 3-octet long; in this case, the two following octets specify the total number of octets of the message (most-significant octet first). Otherwise, the Length field is only 1-octet long and specifies itself the total number of octets contained in the message. MQTT-SN does not support message fragmentation and reassembly

- **MessageType:** The MsgType field is 1-octet long and specifies the message type. Some of the MsgType field values which are implemented in this project is given below:

MsgType field value	MsgType	MsgType field value	MsgType
0x04	CONNECT	0x05	CONNACK
0x0A	REGISTER	0x0B	REGACK
0x0C	PUBLISH	0x0D	PUBACK
0x12	SUBSCRIBE	0x13	SUBACK



MESSAGE VARIABLE PART:

The content of the message variable part depends on the type of the message. Few are given below:

- **CLIENTID:** the CLIENTID field has a variable length and contains a 1-23 character long string that uniquely identifies the client to the server.
- **DATA:** The DATA field corresponds to payload of an MQTT PUBLISH message. It has a variable length and contains the application data that is being published.
- **DURATION:** The DURATION field is 2-octet long and specifies the duration of a time period in seconds.
- **FLAGS:** The FLAGS field is 1-octet and contains many flags such as:
    - DUP: value '0' – Message transmitted once & '1' – retransmission.
    - QOS: 3 types of QOS are present.
    - CleanSession: For a clean connect connection.
    - TopicIDType: Indicates whether topic ID or topic name is present.
- **MSGID:** The MSGID field is 2-octet long and it allows the sender to match a message with its corresponding acknowledgment.
- **PROTOCOLID:** The PROTOCOLID is 1-octet long. It is only present in a CONNECT message and corresponds to the MQTT 'protocol name' and 'protocol version'. It is coded 0x01. All other values are reserved.

- **TOPICID:** The TOPICID field is 2-octet long and contains the value of the topic id. The values "0x0000" and "0xFFFF" are reserved and therefore should not be used.
- **TOPICNAME:** The TOPICNAME field has a variable length and contains an UTF8-encoded string that specifies the topic name.
- **RETURNCODE:** This field is 1 octet long and the meaning is as follows:

RETURN CODE VALUE	MEANING
0x00	ACCEPTED
0x01	REJECTED: CONGESTION
0x02	REJECTED: INVALID TOPICID
0x03	REJECTED: NOT SUPPORTED
0x04 – 0xFF	RESERVED

MESSAGE STRUCTURE:

- **CONNECT:** The CONNECT message is sent by a client to setup a connection.

Length (Octet0)	MsgType (1)	Flags (2)	Protocol ID (3)	Duration (4,5)	ClientId (6:n)

Here in our setup we have initialized the ClientId to not be more than 4 octets. Hence the overall Connect message length is 10 Octets. This value is stored in Length field.

- **CONNACK:** The CONNACK message is sent by the server in response to a connection request from a client.

Length (Octet 0)	MsgType (1)	Return Code (2)

- **REGISTER:** The REGISTER message is sent by the client to the server to register the short TopicID and hence mapping it with TopicName. Here in this message, only TopicName has variable size of length and the rest other message fields have fixed length and hence by initializing it to a TopicName of given octets, we can achieve a short message transmission with a low message size.

Length (Octet0)	MsgType (1)	TopicID (2, 3)	MsgID (4, 5)	TopicName (6:n)

- **REGACK:** The REGACK message is sent by the Broker as an acknowledgment to the receipt and processing of a REGISTER message.
  The length of this Acknowledge message is fixed to 7 Octets.

Length (Octet0)	MsgType (1)	TopicID (2, 3)	MsgID (4, 5)	ReturnCode (6)

- **PUBLISH:** This message is used by both clients and gateways to publish data for a certain TopicID. Based on the Subscriptions sent from the clients which requests data to be published to them based on the TopicID. When the TopicID matches with client's subscription TopicID, the data will be published.

Length (Octet0)	MsgType (1)	Flags (2)	TopicID (3, 4)	MsgID (5, 6)	Data (7:n)

Here in Publish message, all the message fields are fixed except the Data field. By initializing the data to a minimal number of bytes, we can send a short message and transmit. Hence accounting low data occupancy and shorter memory usage.

- **PUBACK:** The PUBACK message is sent by the Broker as an acknowledgment to the receipt and processing of a PUBLISH message. It can also be sent as response to a PUBLISH message in case of an error; the error reason is then indicated in the ReturnCode field.
  This Acknowledge message has a fixed size of 7 Octets.

Length (Octet0)	MsgType (1)	TopicID (2, 3)	MsgID (4, 5)	ReturnCode (6)

- **SUBSCRIBE:** The SUBSCRIBE message is used by a client to subscribe to a certain topic. All the message fields of this message are fixed except TopicID or TopicName. When TopicID is used in the subscription message, we can restrict the message length to 7 Octets. If TopicName is used in subscription message, then we'll need to restrict the TopicName to a fixed number of octets. Then we'll be able to transmit messages of shorter length.

Length (Octet0)	MsgType (1)	Flags (2)	MsgID (3, 4)	TopicID or TopicName (5,6 or 5-n)

- **SUBACK:** The SUBACK message is sent by the Broker to a client as an acknowledgment to the receipt and processing of a SUBSCRIBE message. The SUBACK message is fixed for 7 octets. The return code indicates if the subscription was successful.

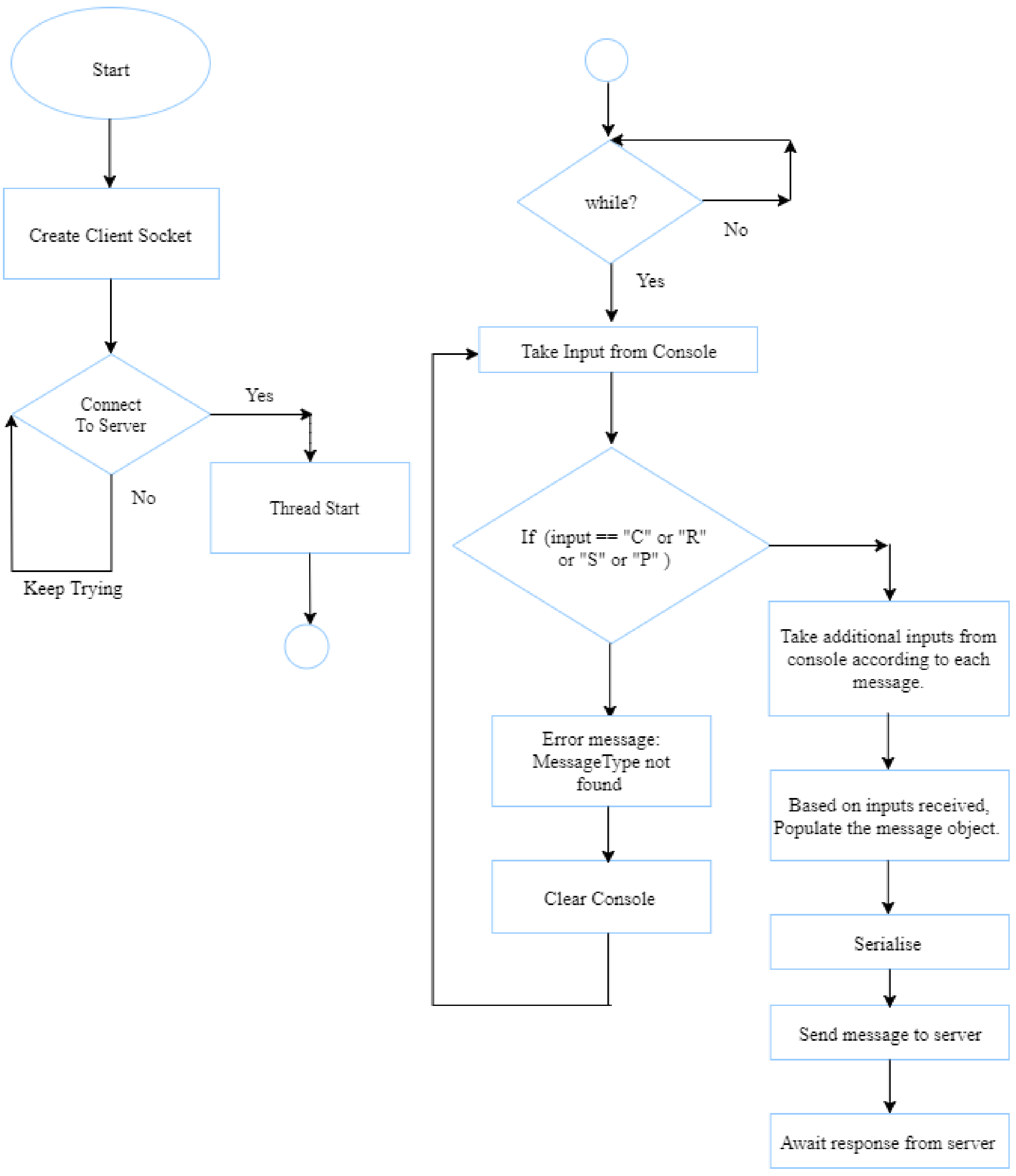
Length (Octet0)	MsgType (1)	Flags (2)	TopicID (3, 4)	MsgID (5, 6)	Return Code (7)

These are the few important messages implemented in our project. The workflow and working is explained in the coming topics.

IMPLEMENTATION OF MQTT-SN PROTOCOL

MQTT-SN CLIENT:

The MQTT-SN functionalities is represented in the flowchart shown below.


Start
Create Client Socket
Connect To Server
Yes
No
Keep Trying
Thread Start
while?
No
Yes
Take Input from Console
If (input == "C" or "R" or "S" or "P")
Take additional inputs from console according to each message.
Error message: MessageType not found
Clear Console
Based on inputs received, Populate the message object.
Serialise
Send message to server
Await response from server

IMPLEMENTATION OF MQTT-SN PROTOCOL

MQTT-SN BROKER:

MQTT-SN Broker enables applications to interact with sensor and other things. It executes information queries on behalf of applications. Broker collects and aggregates the received information. Here in this project we have implemented an MQTT-SN Broker which uses TCP as the transport layer protocol. Although, MQTT-SN preaches to use a lightweight server as it is easy to communicate in a wireless environment like ZigBee etc., we have opted against it as we are not implementing any middleware such as a transparent or an aggregating gateway. Since the gateway is not implemented, in order to stabilise the system and keep it secure we have chosen to implement a TCP based server.

Why TCP over UDP as server?

- TCP as we know is a connection based protocol, meaning that a connection needs to be setup before the transfer of data can start. To be able to do that TCP has been designed with the 3-way handshake system. In this system a user who wants to send data initializes the connection and is acknowledged by the receiving end. Once acknowledged, the sender acknowledges the Acknowledgement, thus completing the 3-way handshake.
- TCP is a reliable protocol, meaning that the data that is sent is reached by the receiving party, which is not an entity in UDP.
- TCP enables data to be received in an ordered way, meaning if 5 data packets are sent, then data packet 1 should be received before data packet 2. This doesn't happen in UDP which is a connection less and works on the principle of shoot the data. The working principle of UDP is to send the data without taking care whether it reaches its destination or not.
- UDP does not give guarantee that data will reach the destination. It does not gives guarantee that data will be in the same order and it also does not give guarantee that data will reached on destination without any duplication.
- TCP provides are stream data transfer, reliability, efficient flow control, full-duplex operation, and multiplexing.
- TCP provides Flow and Congestion control whereas UDP does not.
- In TCP, data is read as a byte stream, no distinguishing indications are transmitted to signal message (segment) boundaries whereas in UDP, Packets are sent individually and are checked for integrity only if they arrive.

SQLite DATABASE:

Client/server SQL database engines strive to implement a shared repository of enterprise data. They emphasis scalability, concurrency, centralization, and control. SQLite strives to provide local data storage for individual applications and devices. SQLite emphasizes economy, efficiency, reliability, independence, and simplicity. SQLite does not compete with client/server databases.

Features of SQLite:

- SQLite is a very simple and fast open source SQL engine.

- SQLite is focused on simplicity. Because it is completely internal, it is often significantly faster than alternatives.
- For features like portability (with regards to both languages and platforms), simplicity, speed, and a small memory footprint--SQLite is ideal.
- SQLite can only support one writer at a time, and the normally high file system latency may be inconvenient if there is a need for many clients to access a SQLite database simultaneously.
- Fields where SQLite suits well are in embedded devices, Internet of things, Application file format, Websites, Data Analysis.

Why SQLite is used as Server Side Database?

- SQLite is very fast. It runs on the same machine, so there is no networking overhead when executing queries or reading results. It runs in the same address space, so there is no wire protocol, serialization or need to communicate via Linux sockets.
- SQLite runs on mobile devices where resources are scarce and efficiency is crucial. SQLite also supports a huge number of compilation flags that allow you to remove features you don't plan to use.
- SQLite's speed makes up for one of its (perceived) greatest shortcomings, which is database-file locking for writes. By writing data incredibly quickly, database locking only becomes an issue when there is a great deal of concurrent writers.
- SQLite is extensible. Because SQLite is embedded by your application, it runs in the same address space and can execute application code on your behalf. SQLite is very easy to configure and it's very easy to manage as it is only one file.

Database Structure:

The database contains 4 tables called Clients, Register, Subscriber, Publisher for 4 messages receiving from the client respectively. The tables will be updated as and when the client messages are received and data is extracted.

Clients Table Structure:

Client_ID (string)	Status (bool)	KeepAlive (integer)	UpdateTime(DateTime)

Register Table Structure:

TopicID(string)	TopicName(string)

Subscriber Table Structure:

SubscriberID (string)	SubscriberTopicID(string)

Publisher Table Structure:

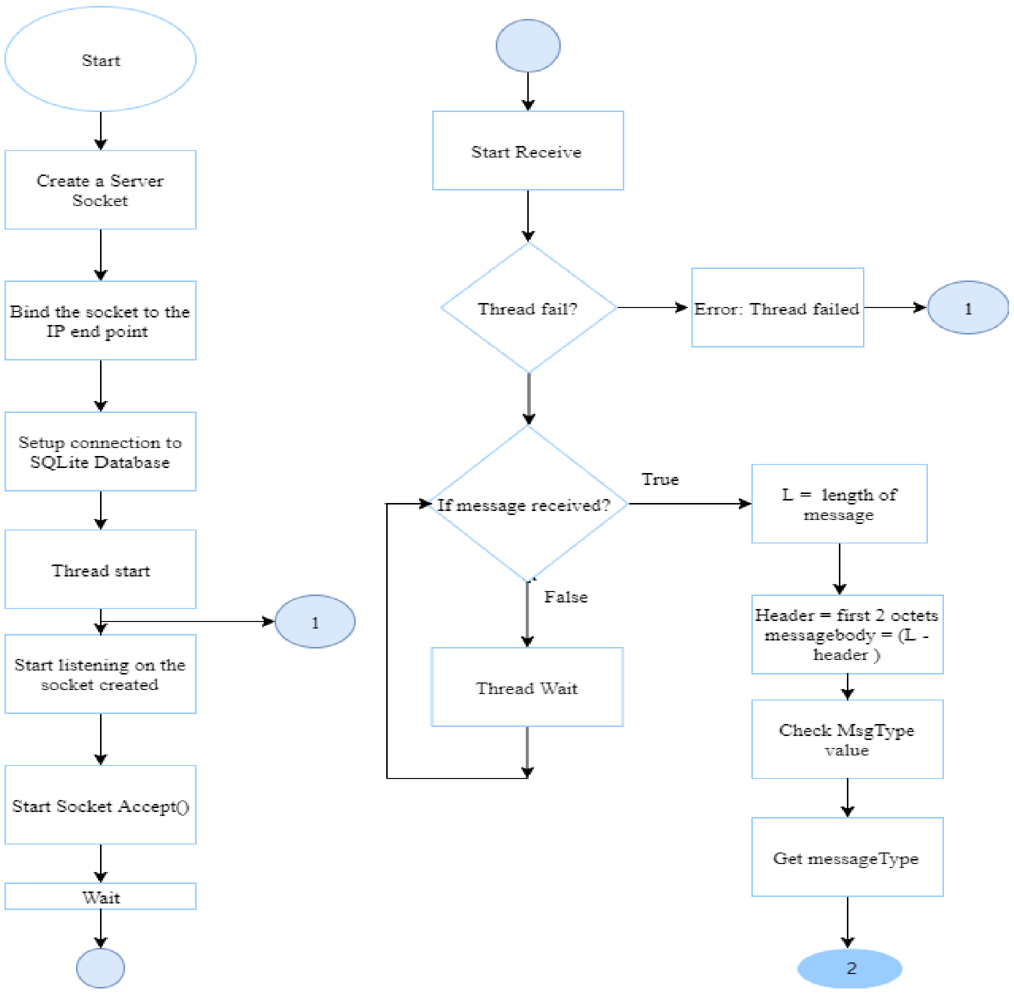
TopicID (string)	TopicName(string)	Message(string)	SubscriberPendingID (string)

IMPLEMENTATION OF MQTT-SN PROTOCOL

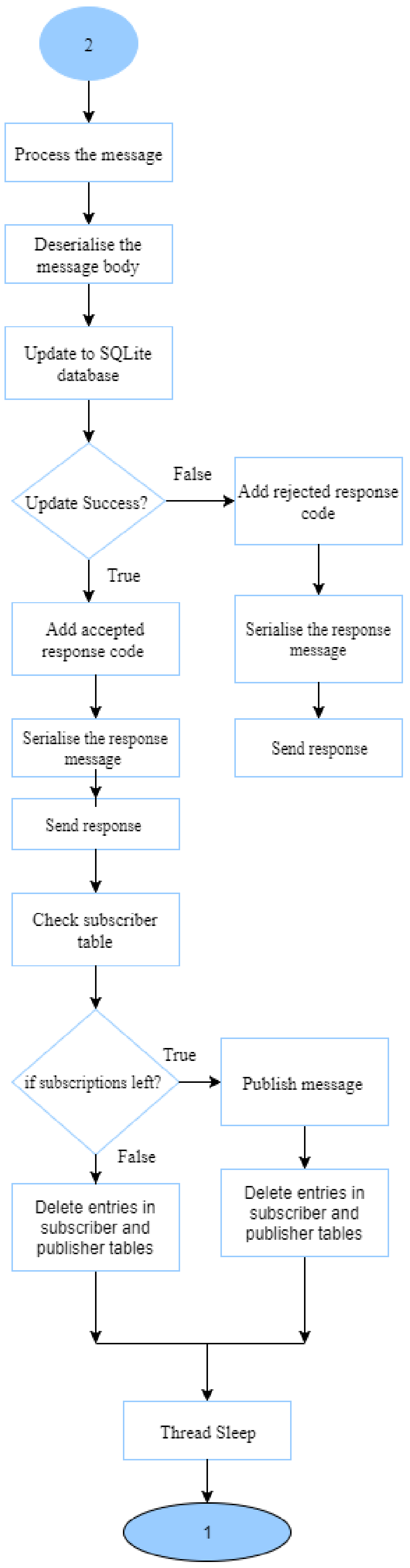
WORKING FLOW:

The server functionalities are explained in the following flowcharts.

- For Connect message, based on the client authentication process we update the Clients Table in the database. Once the data entry is done, we send a CONNACK message back. Also if the client asks for a clean session, it should be noted that the database will be wiped completely and a fresh session will be started.
- For a Register message, we receive the short 2 byte topicID and a fixed topic name. Once the data is updated on the database, REGACK is sent back. Similarly same functionality is implemented for Subscriber as well and SUBACK response will be obtained. Here the subscriber ID is nothing but the client ID.
- For a publish message, subscriber table is check and for all subscriptions present data is provided by cross verifying the Topic ID.



IMPLEMENTATION OF MQTT-SN PROTOCOL


2
Process the message
Deserialise the message body
Update to SQLite database
Update Success?
False
Add rejected response code
True
Add accepted response code
Serialise the response message
Send response
Serialise the response message
Send response
Check subscriber table
if subscriptions left?
True
Publish message
False
Delete entries in subscriber and publisher tables
Delete entries in subscriber and publisher tables
Thread Sleep
1

Strengths:

- Avoids UTF-8 encoded string payloads to carry topic names in every publish message by using pre-defined topic IDs, short topic names and dynamic topic IDs. This reduces the size of publish messages considerably.
- For low powered device adopting offline keep alive mechanism to buffer data to be delivered by the server/broker.
- Apart from this the messages are received in order as the broker used is TCP. Packet loss is reduced.
- Database is hosted along with the application and hence operating it is easy and fast process.

Weakness:

- Cannot subscribe to multiple topics in one SUBSCRIBE message.
- As it's a TCP implementation, it's a heavy weight protocol for sensor networks.

Reference: MQTT-SN protocol v1.2.

IMPLEMENTATION OF MQTT-SN PROTOCOL