MQTT-S – A Publish/Subscribe Protocol For Wireless Sensor Networks

Urs Hunkeler & Hong Linh Truong
IBM Zurich Research Laboratory, Switzerland
Email: {hun,hlt}@zurich.ibm.com

Andy Stanford-Clark
IBM UK Laboratories, Hursley, UK
Email: andysc@uk.ibm.com

***Abstract*— Wireless Sensor Networks (WSNs) pose novel challenges compared with traditional networks. To answer such challenges a new communication paradigm, data-centric communication, is emerging. One form of data-centric communication is the publish/subscribe messaging system. Compared with other data-centric variants, publish/subscribe systems are common and wide-spread in distributed computing. Thus, extending publish/subscribe systems into WSNs will simplify the integration of sensor applications with other distributed applications. This paper describes MQTT-S [1], an extension of the open publish/subscribe protocol Message Queuing Telemetry Transport (MQTT) [2] to WSNs. MQTT-S is designed in such a way that it can be run on low-end and battery-operated sensor/actuator devices and operate over bandwidth-constraint WSNs such as ZigBee-based networks. Various protocol design points are discussed and compared. MQTT-S has been implemented and is currently being tested on the IBM wireless sensor networking testbed [3]. Implementation aspects, open challenges and future work are also presented.**

I. Introduction

In the past few years, Wireless Sensor Networks (WSNs) have been gaining increasing attention, both from commercial and technical point of views, because of their potential of enabling of novel and attractive solutions in areas such as industrial automation, asset management, environmental monitoring, transportation business, etc. Many of these applications require the transfer of data collected by the sensors to applications residing on a traditional network infrastructure (e.g Internet, LAN, enterprise network, etc.). Thus the WSNs need to be *integrated* with these traditional networks. Figure 1 shows the typical structure of such an integrated network, in which *gateways* are used to connect multiple WSNs to a traditional network. Within the WSNs, a large number of battery-operated Sensor/Actuator (SA) devices, usually equipped with a limited amount of storage and processing capabilities, collect information about their environment and send them to the gateways for further transfer to the applications. Even for networks without actuators, information also flows in the opposite direction, e.g., for sensor management and configuration as well as for software updates.

The entire network is very dynamic. On the WSN side, SA devices may change their network addresses at any time. Wireless links are quite likely to fail. Furthermore, SA nodes could also fail at any time, and rather than being repaired, it is expected that they will be replaced by new ones. Applications can be hosted and run on any machines anywhere in the traditional network. Our work on the IBM wireless sensor networking testbed [3] have shown that even in a static network some networking protocols (e.g., ZigBee) change the device address from time to time. In such situations, the conventional approach of using *network addresses* as communication means between the SA devices and the applications may be very problematic because of their dynamic and temporal nature. Applications requiring interactions with the wireless SA devices would need to manage and maintain the addresses of a large number of nodes. In most cases they do not need to know the address or identity of the devices that deliver the information; they are more interested in the content of the data. For example, an asset-tracking application is more interested in the current geographical location of a certain asset than in the network address of the GPS receivers that deliver this information. Moreover, several applications may have an 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.

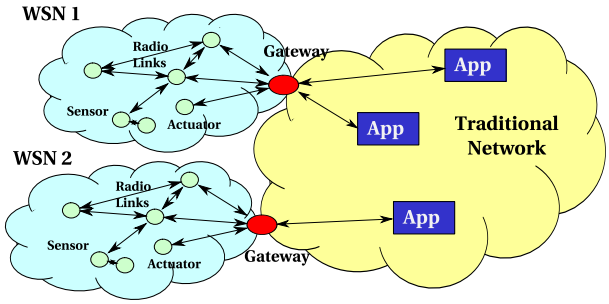

Fig. 1. Integrated Wireless Sensor Networks

The problem described above can be overcome by using a data-centric communication approach [4], in which information is delivered to the consumers not based on their network addresses, but rather as a function of their contents and interests. Publish/Subscribe (pub/sub) messaging systems [5] are well-known examples of data-centric communication and are widely used in enterprise networks, mainly because of their scalability and support of a dynamic application topology. These features are achieved by decoupling the various communicating components from each other such that it is

easy to add new data sources/consumers or to replace existing modules [6].

This paper describes the pub/sub protocol *MQTT-S* [1]. MQTT-S is an extension of the open publish/subscribe protocol Message Queuing Telemetry Transport (MQTT) [2]. It is designed especially for operation on low-cost and low-power SA devices and running over bandwidth constrained WSNs such as ZigBee [7] or TinyOS [8] based networks. ZigBee [9] is an open and global communication standard for WSNs. ZigBee is based on the IEEE 802.15.4 standard [10] for wireless personal area networks (WPANs). It adds on top on this standard the required network, security and application layers, thus providing interoperability between products from different vendors. The processing and storage capabilities of the SA devices are assumed to be equivalent to the original Berkeley Mica mote [11]. Our solution not only provides a simple but scalable communication means for interacting with a large number of SA devices, but also enable a seamless integration of the WSNs into traditional networks.

We begin with an overview on what a pub/sub messaging system is and which advantages it provides for WSNs. In Section III we then briefly present pub/sub protocols already known in the area of sensor networks, with special focus on the open protocol MQTT [2]. MQTT-S is then described in Section IV, with a discussion of the most important design points. Our implementation of MQTT-S is presented in Section V. Open challenges and future work are presented in Section VI, and the conclusions are given in Section VII.

II. Publish/Subscribe Systems

The principle of the publish/subscribe (pub/sub) communication model is that components which are interested in consuming certain information register their interest. This process of registering an interest is called subscription, the interested party is therefore called a *subscriber*. Components which want to produce certain information do so by publishing their information. They are thus called *publishers*. The entity which ensures that the data gets from the publishers to the subscribers is the *broker*. The broker coordinates subscriptions, and subscribers usually have to contact the broker explicitly to subscribe.

There are three principal types of pub/sub systems: topic-based, type-based and content-based [5]. With topic-based systems, the list of topics is usually known in advance, e.g., during the design phase of an application. Subscriptions and publications can only be made on a specified set of topics. In type-based systems, a subscriber states the type of data it is interested in (e.g., temperature data). Type-based systems are not very common. Content-based systems are the most versatile ones. The subscriber describes the content of messages it wants to receive. Such a subscription could be for any messages containing both temperature and light readings where the temperature is below a certain threshold and the light is on.

A form of content-based messaging is TinyDB [12]. The user issues an SQL-like query that describes the data the user is interested in. TinyDB even allows the aggregation of data inside the network. On the other hand, TinyDB is not a general-purpose communication platform but rather a querying system for sensors. To implement a general-purpose content-based publish/subscribe system, the data has to be augmented with meta-data to identify the different data fields. We believe that adding such meta-data incurs too high an overhead for the very constrained platforms we target.

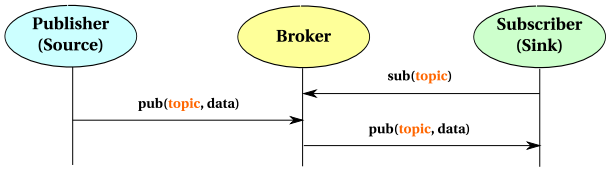

Fig. 2. Topic-based Pub/Sub Communication Model

Owing to its simplicity when compared with the other types of pub/sub systems, we believe that topic-based systems are most appropriate for WSNs based on hardware similar to the original Berkeley mote [11]. The communication model of a topic-based pub/sub system is shown in Figure 2. A subscriber sends a *sub(topic)* message to inform the broker of its interest in the indicated *topic*, whereas a publisher sends a *pub(topic,data)* message which contains the *data* to be published together with the related *topic*. If there is a match between the publisher's and the subscriber's topics, the broker transfers the *pub(topic,data)* message to the subscriber. A single *pub* message may be distributed to multiple subscribers if its topic matches the topics of these subscribers.

Figure 3 shows the resulting architecture of the same integrated network as the one in Figure 1, when a pub/sub system is used as communication middleware. A broker is introduced in the traditional network, and all other components are connected to it and communicate with each other using the broker's pub/sub service. The main function of the gateways is to provide the SA devices with access to the broker. The broker is located in the traditional network because of its higher performance in terms of bandwidth and capabilities.

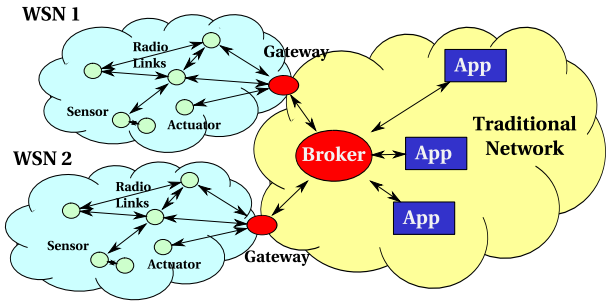

Fig. 3. Integrated Wireless Sensor Networks with Pub/Sub Communication

An application or a SA device can be both subscriber and publisher. Subscribers and publishers are decoupled from each other by the broker, even if they reside on the same device. For example a temperature sensor may need to be monitored by

multiple applications for different reasons. When the data is available, the sensor simply adds the appropriate topic and publishes it to the broker. The broker will then distribute the published data to those applications that have subscribed to that topic. The set of publishers and subscribers that are coupled together by matching topics can dynamically change over time without the subscribers or publishers being aware of this. This is particularly interesting for WSNs, where devices might fail and new ones could be added to replace failed nodes or to extend the network at anytime. Applications do not need to be aware of the failures and changes; they just receive their data when the new devices begin to operate. The same applies to the SA publisher nodes: they do not need to know which applications are interested in their data. They just send their data to the broker, which will then take care of the data distribution to the applications.

For application developers, the pub/sub system hides the complexity of the underlying network and lets them concentrate on the design of the application itself. To receive data of a certain SA device, the only thing that they need to know is the topic the SA device uses when it publishes its data. Similarly, if they want to send control information to a SA device, they only have to know the topics to which the SA device has subscribed, not its actual network addresses. Even if a SA device is moved to another WSN (e.g., because of network congestion), no change needs to be done to the applications and gateways as long as the SA is still using the same topics for its publications and subscriptions. Moreover, these topics are defined by the application developers themselves, and not by the networks or by the broker.

If the enterprise networks already use pub/sub systems as communication middleware, extending the pub/sub protocols into the WSNs will significantly simplify the interconnection of the two networks. Existing broker infrastructure can be used for interacting with and managing the SA devices. Field data collected by the SAs can be made seamlessly available to all applications like any other enterprise information, and in the same way also the control and management of the SA nodes can be performed from any application located in the enterprise network.

III. Related Work

A. Publish/Subscribe Protocols for WSNs

The Global Sensor Network (GSN) [13] sees WSNs as black boxes. It offers a unified way to query WSNs independently of where or how they are connected to the backbone network. Similarly, IrisNet [14] aims at unifying data from sensor networks worldwide. MQTT-S differs in that it aims at hiding the end-point details. An application running on either the backbone network or inside the WSN does not know whether the data is coming from a device in a WSN or the backbone network. Devices from different WSNs can intercommunicate. MQTT-S not only allows data to be collected from a WSN, it also allows data to be sent to devices inside WSNs.

TinySIP [15] is similar to MQTT-S in that it is an extension of a well-known protocol into the world of WSNs. TinySIP makes the functionality of the Session Initiator Protocol (SIP) [16] available to WSNs. TinySIP supports session semantics, publish/subscribe, and instant messaging. TinySIP offers support for multiple gateways. Most communication is done by addressing individual devices. As device addresses are related to the gateway being used, changing the gateway on the fly is difficult.

Asene [17] is an implementation of an active database inside a WSN. It uses publish/subscribe to communicate among nodes. The basic principle of Asene is to wait for events, then evaluate a condition and, if the condition is true, execute a given action. Asene is not a generic transport mechanism.

Mires [18] is a publish/subscribe architecture for WSNs. Basically sensors only publish readings if the user has subscribed to the specific sensor reading. Messages can be aggregated in cluster heads. Subscriptions are issued from the sink node (typically directly connected to a PC), which then receives all publications.

DV/DRP [19] is another publish/subscribe architecture for WSNs. DV/DRP stands for Distance Vector/Dynamic Receiver Partitioning. Subscriptions are made based on the content of the desired messages. Subscriptions are flooded in the network. Intermediate nodes aggregate subscriptions. They forward publications only if there is an interest for this publication. Because of the complexity of matching subscriptions to arbitrary data packets it would be difficult to implement this protocol on the devices we target.

Messo and Preso [20] are two complementary publish/subscribe protocols for WSNs. Messo allows data to be collected from sensors in a WSN, whereas Preso allows data to be sent to actuators in the WSN. Messo and Preso rely, as does MQTT-S, on an external broker. Messo and Preso differ from MQTT-S in that they do not establish individual connections between the devices and the broker. Their implementation takes advantage of the possibility of processing data inside the WSN. Each node decides locally whether to forward a message. If data is collected with Messo, nodes that relay messages can also combine multiple messages. Currently, Messo and Preso rely on predefined topics. They cannot dynamically add new topics. They also require a single gateway.

B. MQTT ("Message Queuing Telemetry Transport")

As mentioned, MQTT is an open pub/sub protocol [2] designed for constrained devices used in telemetry applications. However, it does not consider the case of SA devices; its extension for sensor networks, "MQTT-S", will be described in Section IV.

MQTT is designed in such a way that its implementation on the client's side (i.e., the SA's side) is very simple. All of the system complexities reside on the broker's side. MQTT does not specify any routing or networking techniques; it assumes that the underlying network provides a point-to-point, session-oriented, auto-segmenting data transport service with in-order delivery (such as TCP/IP) and employs this service for the exchange of messages.

MQTT is a topic-based pub/sub protocol that uses character strings to provide support of hierarchical topics. This also facilitates the subscription to multiple topics. For example, a temperature sensor located on floor "F2", room "R248" could publish its data using the hierarchical topic *"wsn/sensor/F2/R248/temperature"*. The forward slash character *"/"* is used to separate each part of the topic. Wildcard characters can then be used to replace any part of the topic, e.g., the string *"wsn/sensor/F2/+/temperature"* could be employed to subscribe to data generated by all temperature sensors on floor F2. In this example the character *"+"* was used as wildcard for any pattern at the 4th level of the topic.

MQTT supports basic end-to-end Quality of Service (QoS) [21]. Depending on how reliably messages should be delivered to their receivers, MQTT distinguishes between three QoS levels. QoS level 0 is the simplest one: it offers a best-effort delivery service, in which messages are delivered either once or not at all to their destination. No retransmission or acknowledgment is defined. QoS level 1 provides a more reliable transport: messages with QoS level 1 are retransmitted until they are acknowledged by the receivers. Consequently, QoS level 1 messages are certain to arrive, but they may arrive multiple times at the destination because of the retransmissions. The highest QoS level, QoS level 2, ensures not only the reception of the messages, but also that they are delivered only once to the receiving entities. It is up to the application to select the appropriate QoS level for its publications and subscriptions. For example, a temperature-monitoring application could decide to use QoS level 0 for the publication of normal and regular measurement reports, but QoS level 1 for transferring alarm messages when the temperature exceeds a certain threshold.

MQTT is a connection-oriented protocol in the sense that it requires a client to setup a connection with the broker before it can exchange publications and subscriptions with the broker. To this end, a *"CONNECT"* message is defined. It contains, among other connection parameters, a *Client Id*, which enables the broker to identify the connected client. This Client Id is used by the broker, for example to make sure that QoS level 1 and 2 publications are delivered correctly when the client reconnects after a network failure. The broker supervises the liveliness of the client/connection by a *"keep-alive"* timer, which defines the maximum time interval that may elapse between two messages received from that client. If during this time interval the client has no data-related messages to be transmitted, it will send a *PING* message to the broker, which is acknowledged by the broker. Thus the keep-alive timer enables the broker to detect the failure of either the client or the network link.

A related and interesting MQTT feature is its support of the so-called *"Will"* concept. At connection time, a client could ask the broker to store a "Will" message together with a "Will" topic. The broker will only send this "Will" publication to the subscribers when it abnormally loses the connection with the client. Applications could use this feature to detect failures of devices and links.

IV. MQTT-S, a MQTT-based pub/sub protocol for sensor networks

In this section we will describe MQTT-S, a pub/sub protocol based on MQTT and especially designed for WSNs. MQTT-S is developed based on the following design points:

1) **As close as possible to MQTT:** This allows a seamless connection of the SA devices to an MQTT broker, thus enabling a smooth integration of the WSNs with the existing communication infrastructure. This also enables a very simple and lossless implementation of the gateways. As a consequence, MQTT-S supports not only all MQTT features (e.g., those described in Section III-B) but also almost all the message flows and contents defined by MQTT.
2) **Optimized for tiny SA devices:** The protocol is designed in such a way that it can be implemented for low-cost, battery-operated devices with limited processing and storage. Whenever complexities are required, they reside on the gateway/broker's side; the client running on the SA devices is kept as simple as possible.
3) **Consideration of wireless network constraints such as high link failure rates, low bandwidth, and short message payload:** Wireless radio links in general have higher failure rates than wired links, owing to their susceptibility to fading and interference disturbances. They have also a lower transmission capacity. For example, WSNs based on the IEEE 802.15.4 standard provide an aggregate (shared) bandwidth of a theoretic maximum of 250 kbit/s in the 2.4 GHz band [10]. In practice the bandwidth is even lower because of free channel assessment and retransmissions. Procedures should be defined to reduce the risk of having SAs disconnected from the infrastructure owing to link failures or network congestion.
   Moreover, to be resistant against transmission errors, wireless networks have a much shorter packet length than wired networks. In the case of IEEE 802.15.4, the physical layer provides a maximum packet length of 128 bytes. Half of these 128 bytes could be taken away by the overhead information required by other supporting layers and functions such as MAC, network, security, etc., see for example [22] for the case of ZigBee. That means, MQTT-S messages should be shorter than 64 bytes. This is very little if human-readable data formats (such as topic names) are to be supported.
   How MQTT-S copes with these issues is described below.
4) **Network independent:** MQTT-S is designed to run on any network that provides the two following services:
   a) Point-to-point data transfer service (unicast service): A datagram service that allows the transport of messages between any two points based on their network address. The two points involved may be multiple hops away from each other.

b) One-hop broadcast data transfer service: This is in principle supported by all wireless networks; messages sent by a node can be received by all nodes within the transmission range.

In contrast to MQTT, MQTT-S does not assume a connection-oriented service, and does not rely on message segmentation, nor in-order delivery of those segments.

A. MQTT-S Architecture

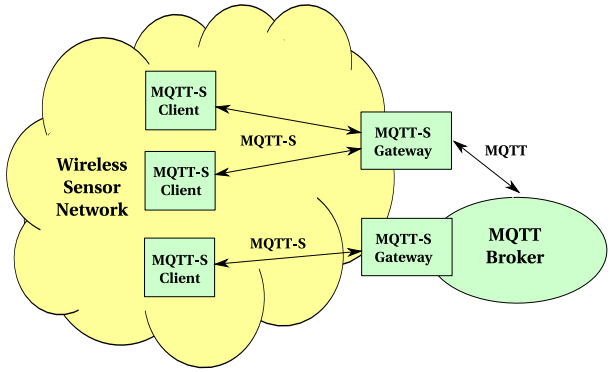

Fig. 4. MQTT-S Architecture

The architecture of MQTT-S is shown in Figure 4. There are two types of components: MQTT-S *clients* and MQTT-S *gateways (GWs)*. MQTT-S clients are on the WSN side and enable the SA devices to access the pub/sub services of a MQTT broker located on the traditional network. They connect to the gateway using the MQTT-S protocol, and the gateway connects to the broker. The main function of the gateway is to translate between the MQTT and MQTT-S protocols. A MQTT-S gateway may or may not be integrated with the broker. In the case of stand-alone operation, i.e., the gateway is not integrated into the broker, the gateway uses the MQTT protocol to communicate with the broker.

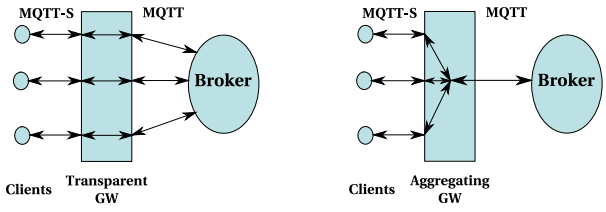

Fig. 5. Transparent and Aggregating Gateways

Depending on how a gateway performs the protocol translation between MQTT and MQTT-S, we can differentiate between two types of gateways, namely, *transparent* and *aggregating* GWs, see Fig. 5:

1) Transparent Gateway: For each connected MQTT-S client, a transparent GW sets up and maintains a MQTT connection to the MQTT broker. There will be as many MQTT connections between the GW and the broker as there are MQTT-S clients connected to the GW. The transparent GW will perform a "translation" between the two protocols. As all MQTT-S messages can be mapped to MQTT ones and vice versa, all functions and features that are implemented by the broker can be offered to the client.
2) Aggregating Gateway: Instead of having an MQTT connection for each connected client, an aggregating GW will have only one MQTT connection to the broker. All message exchanges between a MQTT-S client and an aggregating GW end at the GW. The GW then decides which information will be transported further to the broker.

Although the implementation of a transparent GW is simpler than that of an aggregating GW, the main issue with a transparent GW is the system scalability in terms of number of connected clients: because the gateway and broker have to maintain a connection to every active client, its performance may deteriorate in networks with a very large number of SA devices. An aggregating GW may be helpful because it allows a significant reduction of the number of MQTT connections that the broker has to support concurrently.

B. Support for multiple gateways

As mentioned above, one of the weaknesses of wireless networks is their high link failure rates. Links between a SA device and a gateway may fail at anytime, thus disconnecting the SA device from the broker. It is therefore highly desirable that a SA device has access to at least two gateways, so that if its connection to one gateway fails it can re-connect via the other one. Another reason for requiring the presence of at least two gateways is the low transmission capacity of the wireless links. Links in the proximity of a gateway could become congested if a large number of SAs exchange their messages with that gateway. Having more than one gateway helps remedy that situation because the traffic toward the broker can be spread evenly between the gateways. The Tenet architecture [23] argues that future large-scale sensor network deployments will have a back-end network to render the WSN more stable. MQTT-S provides the means of doing exactly this: the WSN is linked to a more powerful traditional network through the gateways.

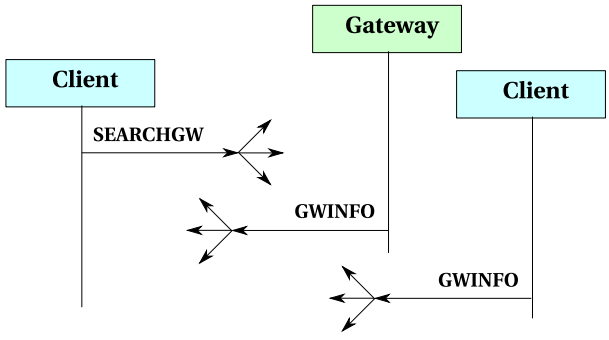

Fig. 6. Gateway Discovery Procedure

MQTT-S supports the presence of multiple gateways via

a gateway discovery procedure. As shown in Figure 6, its message flow is kept as simple as possible. To find a gateway, a client broadcasts a SEARCHGW message, which is replied-to by a gateway by means of a GWINFO message. A client also answers with a GWINFO message if it has the address of a gateway. To give priority to the gateways and to reduce the number of messages, clients delay their transmissions for a random time; if during this delay they receive a GWINFO message sent by another node (gateway or client), they do not send their reply. The procedure is thus very bandwidth-efficient; a single SEARCHGW and GWINFO message exchange could already provide the required information to multiple clients that are concurrently searching for a gateway. If there is no response, the SEARCHGW message is retransmitted, with the time interval between two consecutive transmissions being increased exponentially.

Both SEARCHGW and GWINFO messages are locally broadcasted, i.e., they are not repeated by the receiving nodes. With this one-hop broadcasting, we exploit the inherent broadcast nature of radio transmissions to transfer the messages to all nodes within the transmission range of the sender. There is no "extra" bandwidth consumed by these broadcasts!

C. Support of short message payload

Another issue with wireless networks is that their packets are very short. As mentioned above, the payload provided by ZigBee to an application is limited to around 60 bytes. MQTT uses human-readable strings for client IDs and topic names. In particular two messages may become quite long. The first is the CONNECT message, which may contain three large parameters: the Client Id, the Will topic, and the Will message (see Section III-B for an explanation of these concepts). The second is the PUBLISH message, which contains the topic name, and a potentially large data field.

There are in principle three approaches for resolving this issue. The first and obvious one is to define a generic segmentation and reassembly procedure. Segmentation and reassembly are, for instance, used for ATM, TCP/IP, Bluetooth, etc. Although protocols and algorithms for implementing this method are well-known, we rejected it for the following reasons: it increases the memory footprint of the client; it requires additional overhead in the message, thus reducing the application payload further; and it does not reduce the wasted bandwidth consumed by such messages like the PUBLISH ones, which in most cases contain the same topic name.

The second method is to split a long message into multiple shorter ones. We applied this idea to the CONNECT message, and divide this message into three smaller ones: the first carrying the string for the Client Id, the second for the Will topic, and the third for the Will message. The resulting message flow is shown in Figure 7.

However, we did not apply this approach to the PUBLISH message, because it still does not help us reduce the bandwidth wasted by sending the same topic name every time. To get rid of that redundant information, we replace the topic name by a short, two-byte long "topic id" and define a registration

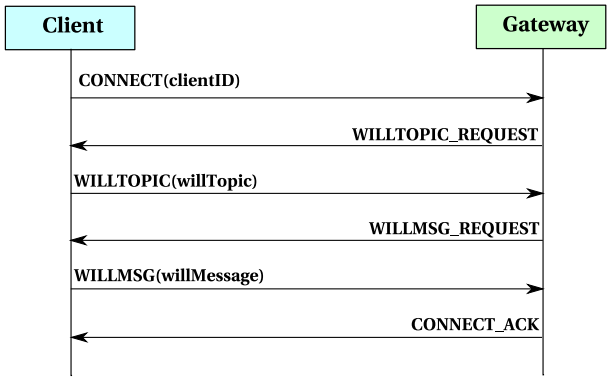

Fig. 7. Connect Procedure

procedure to allow clients to register their topic names with the gateway and obtain the assigned topic ids, see Figure 8. After this registration of the topic name, PUBLISH messages sent by the client will contain the assigned short topic id instead of the longer topic name. The same procedure is also used by the gateway in the opposite direction to inform the client about a new topic name before sending PUBLISH messages with this topic to the client.

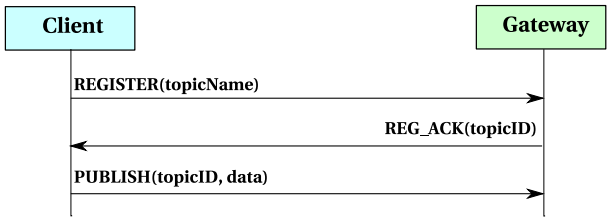

Fig. 8. Register Procedure

With the procedure according to Figure 8, there is a risk that a client wrongly uses a topic id assigned to another topic in the gateway. This may happen when the gateway uses a single table for topic id to topic name mapping. However, to keep the client's implementation simple, we refrain from defining a topic id synchronization algorithm. Instead, we prefer to implement for each client a dedicated mapping table in the gateway. With this solution, the probability that the client uses a wrong topic id is reduced to almost zero.

To further simplify the client's implementation, we introduce so-called pre-defined topic ids that clients can use immediately for publishing without prior registration of topics. In this case the topic id's mapping table in the gateway is pre-configured by an administrator.

V. Testbed Implementation

We have implemented an MQTT-S client and gateway to study MQTT-S' behavior on real systems. In our testbed [3] the gateway client is connected over a serial port to a PC, where the gateway software runs. In the following we first describe the implementation details of the client and then those of the gateway.

A. MQTT-S Client

The ZigBee client is written in C such that there is as little dependency on the hardware platform as possible. The client runs on multiple ZigBee hardware platforms currently commercially available. As of this writing no multi-hop point-to-point communication protocol for ZigBee exist that are compatible across products from different vendors. For this reason we rely on vendor-specific transport protocols that are available for vendor-specific profiles. Although this results in MQTT-S not being able to communicate across devices from different vendors, the client code should be easy to adapt to a standardized transport protocol once one is available. Moreover, it is possible to communicate between devices from different vendors if the devices from a single vendor form an independent subnetwork that is connected to the broker.

The TinyOS client is written in NesC, a variant of C for TinyOS. The client code uses the collection tree protocol (CTP)[24] as its underlying routing protocol. Essentially CTP allows to send data from any node to the closest gateway. On top of CTP we have added a reverse route table: each time a node forwards a message to the gateway, it stores the source node's address in a reverse route table. This effectively extends the routing layer to allow bidirectional communication between any node and its closest gateway. The TinyOS client currently runs on Tmote [25] and MicaZ motes. Our TinyOS network can communicate with our ZigBee network through the broker.

Most of the protocol logic is handled by the broker and the gateway. This makes the client implementation extremely lightweight. A full implementation of the client is about 12 kB. In comparison, the ZigBee protocol stack and the support functions provided by the hardware vendors are around 50 kB. On devices with only 64 kB of program memory available, this gets very close to the limit of what can be done. It is however possible to further reduce the complexity and the memory requirements of the MQTT-S client code. For instance, if the client is built for a very specific purpose, predefined topic IDs could be used. In this case the code that handles dynamic topic registration can be omitted. Similarly if it is known that the client will only ever act as either publisher or subscriber, the unneeded functionality can be removed. As the broker or the gateway will never use a functionality of the MQTT-S protocol that the client has not previously requested, the client can be stripped down even further in this way.

B. MQTT-S Gateway

The gateway is written in Java. It uses the Java Communications API to communicate with the gateway devices over the serial port. The gateway devices simply forward any packets they receive from the wireless network stack over the serial port, only adding framing information. The gateway then deframes the data it receives from the devices. The gateway connects to the broker using the MQTT Client API.

The gateway stores session information for any client device that connects through it to the broker. The session information is mostly limited to topic mappings. All other protocol messages are simply translated from one format to the other and sent on. Most of the session handling, protocol state and other protocol processing is done in the broker.

If a client changes the gateway, it merely reissues a connect. The gateway forwards the connection request to the broker. The broker then realizes that the client is already connected through a different gateway. It will close the connection to the first gateway and, if it was not instructed to clean-start a new session, transfer it to the new gateway. This feature of MQTT assures that the change of the gateway will not result in the loss of the session.

Using the MQTT Client API results in some limitations of the functionality of the gateway. For instance, the Client API automatically sends keep-alive messages. So even though a device has failed and no longer sends messages, the Client API will still continue to send ping requests. However, in this scenario the broker will never learn that the client device has disconnected abnormally, and the will message that the client might have registered will never be sent. If we detect the failure of the device in the gateway, we have no means of triggering the will message either. All we can do is disconnect from the broker. But in this case the disconnection will not be seen as abnormal, and the broker will not send the will message either.

Moreover, it is currently not possible to send messages with QoS 1 or 2 from the broker to the client device. The reason for this is that the gateway cannot delay acknowledging the message once it has received it from the Client API. To correctly implement QoS 1 and 2, the gateway would have to be able to wait with the acknowledgment to the broker until it has received the respective acknowledgment from the client device. With the current implementation of the Client API, however, this would result in the API stalling during the wait.

Apart from these small limitations, the gateway is fully functional. We have written unit tests that show that the gateway keeps working correctly even in the case of message losses and unexpected or incorrectly formed messages. Testing the protocol for several weeks on our testbed [3] has revealed additional problem cases which we did not consider while writing the original unit tests. MQTT-S is now stable and has been running for several weeks over our multi-hop test network.

VI. Open Challenges and Future Work

The current implementation of the MQTT-S gateway has two open problems: in the case of node failure it is not possible to trigger the will message and it is not possible to send message with QoS 1 or 2 from the broker to the clients. Both problems are directly related to the current implementation of the MQTT Client API. One way around this would be to implement an MQTT protocol module ourselves. We do not like this approach as it would mean that the MQTT protocol implementation would not be maintained by the MQTT developers. We instead intend to implement a MQTT-S protocol module directly into the broker.

A more critical problem is the handling of the duty cycle of the client devices. To save as much energy as possible, client devices would like to enter a sleep mode whenever they are not used. They will wake up and publish whenever they have new data. As the sleeping times could potentially be very large (on the order of several seconds to several minutes to multiple hours), the gateway and the broker need to be aware of this. ZigBee router devices can be used to store messages until the client wakes up. However there might be more messages than a ZigBee router device can store (being itself a relatively constrained device). We are currently working on how the broker could interact optimally with sleeping devices.

So far, we have only tested our implementation with a limited number of real devices. We plan to extend our test network (to more than 50 nodes) with several gateways and run it for several weeks to test the actual performance of the protocol.

VII. Conclusion

We have presented MQTT-S [1], an adaption of the MQTT protocol to the constraints of WSNs. MQTT-S is based on experience with out testbed [3]. The publish/subscribe paradigm naturally fits many requirements for communication in WSNs as it hides the topology of the network and allows data to be delivered based on interests rather than individual device addresses. A particular advantage of MQTT-S over other protocols is that it is based on a well-known publish/subscribe protocol already widely used. MQTT-S allows a transparent data exchange between WSNs and traditional networks and even between different WSNs. In addition, MQTT-S is extremely lightweight and can be further stripped down to a bare minimum.

Implementing MQTT-S revealed many challenges of WSNs that we would not have thought of otherwise. In particular, the need to support sleeping clients is going to be addressed in the next version. The implementation also demonstrates that the protocol can easily be implemented on devices with only limited resources.

Acknowledgment

The authors thank Daniel Bauer and Dave Locke for their help and discussions about MQTT. We also thank Mariot Chauvin for his excellent work on MQTT-S during his master's thesis.

References

[1] A. Stanford-Clark and H. L. Truong, *MQTT for sensor networks (MQTTs)*, http://www.mqtt.org/MQTTs_Specification_V1.0.pdf, Oct. 2007.

[2] "MQ Telemetry Transport," http://mqtt.org.

[3] S. Furrer, W. Schott, H. L. Truong, and B. Weiss, "The IBM wireless sensor networking testbed," in *2nd International Conference on Testbeds & Research Infrastructures for the DEvelopment of NeTworks & COMmunities (TRIDENTCOM'06)*, Mar. 2006.

[4] D. Estrin, R. Govindan, J. S. Heidemann, and S. Kumar, "Next century challenges: Scalable coordination in sensor networks," in *Mobile Computing and Networking*, 1999, pp. 263–270.

[5] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kernmarrec, "The many faces of publish/subscribe," *ACM Computing Surveys*, vol. 35, no. 2, pp. 114–131, June 2003.

[6] B. Oki, M. Pfluegl, A. Siegel, and D. Skeen, "The information bus: An architecture for extensible distributed systems," *ACM SIGOPS Operating Systems Review*, vol. 27, no. 5, 1993.

[7] P. Baronti, P. Pillai, V. W. C. Chook, S. Chessa, A. Gotta, and Y. F. Hu, "Wireless sensor networks: A survey on the state of the art and the 802.15.4 and ZigBee standards," *Computer Communications*, vol. 30, no. 7, pp. 1655–1695, 2007.

[8] TinyOS Alliance, "TinyOS," http://www.tinyos.net/.

[9] "ZigBee Alliance," http://www.zigbee.org.

[10] IEEE LAN/MAN Standards Committee, "IEEE Standard 802.15.4, Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY), Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)," http://www.ieee802.org/15/pub/TG4.html, 2003.

[11] J. L. Hill and D. E. Culler, "Mica: A wireless platform for deeply embedded networks," *IEEE Micro*, vol. 22, no. 6, pp. 12–24, 2002.

[12] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "The design of an acquisitional query processor for sensor networks," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD'03)*, 2003, pp. 491–502.

[13] K. Aberer, M. Hauswirth, and A. Salehi, "The Global Sensor Networks middleware for efficient and flexible deployment and interconnection of sensor networks," Swiss Federal Institute of Technology, Lausanne (EPFL), Tech. Rep., 2006.

[14] P. B. Gibbons, B. Karp, Y. Ke, S. Nath, and S. Seshan, "IrisNet: An architecture for a worldwide sensor web," *IEEE Pervasive Computing*, vol. 2, no. 4, pp. 81–88, 2003.

[15] S. Krishnamurthy, "TinySIP: Providing Seamless Access to Sensor-based Services," in *Proceedings of the 1st International Workshop on Advances in Sensor Networks 2006 (IWASN 2006)*, July 2006.

[16] J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnston, J. Peterson, R. Sparks, M. Handley, and E. Schooler, "The session initiation protocol (IETF RFC 3261)," http://www.ietf.org/rfc/rfc3261.txt, June 2002.

[17] M. Zoumboulakis, G. Roussos, and A. Poulovassilis, "Active rules for sensor databases," in *Proceeedings of the 1st International Workshop on Data Management for Sensor Networks (DMSN'04)*, 2004, pp. 98–103.

[18] E. Souto, G. Guimares, G. Vasconcelos, M. Vieira, N. Rosa, and C. Ferraz, "A message-oriented middleware for sensor networks," in *Proceedings of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing (MPAC'04)*, 2004, pp. 127–134.

[19] C. P. Hall, A. Carzaniga, J. Rose, and A. L. Wolf, "A content-based networking protocol for sensor networks," Department of Computer Science, University of Colorado, Tech. Rep., Aug. 2004.

[20] S. Rooney and L. Garces-Erice, "Messo & Preso practical sensor-network messaging protocols," in *Proceedings of the Fourth European Conference on Universal Multiservice Networks (ECUMN'07)*, 2007, pp. 364–376.

[21] D. Chen and P. K. Varshney, "QoS support in wireless sensor networks: A survey." in *International Conference on Wireless Networks*, 2004, pp. 227–233.

[22] O. Oezcelik and L. Baumstark, "Audio transmission with IEEE 802.15.4 and ZigBee," in *1st European ZigBee Developers' Conference*, Munich, Germany, June 2007.

[23] O. Gnawali, K.-Y. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler, "The tenet architecture for tiered sensor networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys'06)*, 2006, pp. 153–166.

[24] R. Fonseca, O. Gnawali, K. J. S. Kim, P. Levis, and A. Woo, *The Collection Tree Protocol (CTP)*, http://www.tinyos.net/tinyos-2.x/doc/txt/tep123.txt, Oct. 2007, draft version.

[25] Moteiv Corporation, "Tmote Sky Brochure," http://www.moteiv.com/products/docs/tmote-sky-brochure.pdf, 2005, version 1.00.