A Framework for Modeling and Verifying IoT Communication Protocols

Maithily Diwan and Meenakshi D'Souza

International Institute of Information Technology, Bangalore
maithily.diwan@iiitb.org, meenakshi@iiitb.ac.in

Abstract. Communication protocols are integral part of the ubiquitous IoT. There are numerous light-weight protocols with small footprint available in the Industry. However, they have no formal semantics and are not formally verified.
Since these protocols have many common features, we propose a unified approach to verify these protocols through a framework in Event-B. We begin with an abstract model of an IoT Communication protocol which encompasses common features of various protocols. The abstract model is then refined into concrete models for individual IoT protocols using Refinement and decomposition techniques of Event-B. Using the above framework, we present models of MQTT, MQTT-SN and CoAP protocols, and verify communication properties like connection-establishment, persistent-sessions, caching, proxying, message-ordering, QoS, etc.
Our protocol models can be integrated-with or extended-to other formal models of IoT systems using machine-decomposition within Event-B. Thus paving way for formal modeling and verification of IoT systems.

Keywords: IoT, MQTT, MQTT-SN, CoAP, Formal Modeling and Verification, Event-B

1 Introduction

IoT is prevalent in various industries like health care, automotive, manufacturing, power grid and domotics to name a few. IoT not only connects different computing devices but sensors, actuators, people and virtually any object. With the prediction that there will be over 20 billion devices by 2020 [7], IoT will be an integral part of our lives. The end nodes in the IoT are usually sensors or small devices which have limited processing capability and low memory. In such cases, the devices send unprocessed data to cloud which is then shared with other devices/systems subscribing to this large amount of data (either raw data or processed by server), making communication between these devices an important aspect of IoT.

Various protocols are used for communication in an IoTsystem. TCP/IP is a popular protocol used in lower layers. Several protocols are adapted for the application layer in an IoT system - Message Queue Telemetry Transport Protocol (MQTT) [9], Message Queue Telemetry Transport Protocol Sensors

(MQTT-SN) [10], The Constrained Application Protocol (CoAP) [11], eXtensible Messaging and Presence Protocol (XMPP) [12], Advanced Message Queuing Protocol (AMQP) [13] to name a few. Most of them are being used for IoT systems as they are bandwidth efficient, light-weight and have small code footprint [8]. Features like publish-subscribe, messaging layer, QoS levels, resource discovery, re-transmission, etc are prevalent in these protocols.

Our framework for an IoT protocol modeling and verification is realized through an abstract model of the protocol. The abstract model consists of commonalities among various application layer protocols like communication modes, connection establishment procedure, message layer, time tracking and attacker modules. We then decompose these various modules and refine them into more concrete models for individual protocols. Properties that hold true for these protocols are verified in both abstract and concrete Models. We use Event-B to model the communication channel and the client and server side communication entities, all of which together implement the protocol. By verifying the accuracy of the model through simulations, invariant checking and LTL properties satisfiability, we are able to conclude that our Models of various protocols are correct.

Messages/streams are used as basic entities of communication between multiple clients and servers. Structure of a message apart from payload, usually consists of many fields of various types. Event-B provides record datatypes [3] through which complicated message structure with multiple attributes and sub-attributes can be expressed succinctly. All the properties of the protocol to be proved are expressed as invariants which are essentially predicates that are always true. The automatic proof discharge in Event-B using the Rodin tool [4] verifies if these invariants (properties) are satisfied for all the events in the model. It is to be noted that we verify several different properties, including functional and non-functional properties of the protocols.

The paper is organized as follows. We begin with a brief discussion on IoT protocols and their properties in Sec. 2. Section 3 highlights the features of Event-B that we use for modeling. Our Event-B model and their refinements for different protocols are detailed in Sec. 4. Verified properties and their results are presented in Sec. 5. Section 6 discusses related work and Sec. 7 presents the conclusion and on-going work.

2 IoT Communication Protocols: MQTT, MQTT-SN and CoAP

Most of the applications in IoT need a reliable network and use existing Internet to communicate with the cloud/servers and with other nodes. Hence it is common to use the existing TCP/IP stack. The underlying physical, DLL, network and transport layer of this backbone is used. However the TCP/IP stack has a larger footprint and memory requirement. UDP, a lighter transport layer protocol is used in some cases where the reliability has to be built in the application layer protocol. Other communication requirements of an IoT system which need

to be possessed by the application layer protocol are: low bandwidth, low memory consumption, small code foot-print, self recovery from loss of connection, resource discovery, light-weight, low message overhead, low power consumption, authentication, security requirements, appropriate Quality of Service(QoS).

Following is brief description of some of the application protocols highlighting the features and properties which we verify in this paper.

2.1 MQTT

MQTT [9] is a publish-subscribe protocol designed for constrained devices connected over unreliable, low bandwidth networks. It gives flexibility to connect multiple servers to multiple Clients. The protocol has low message overhead which makes it bandwidth efficient and can be easily implemented on a low powered device. Significant features offered by MQTT are explained below:

1. QoS: A Quality of Services(QoS) for a message is decided between the sender and the receiver. MQTT offers three levels of QoS: QoS0 is a fire and forget kind of message delivery, where the sender sends a publish message only once and does not wait for any acknowledgement. QoS1 guarantees delivery at least once by seeking for an acknowledgement for every message published. If the sender does not receive an acknowledgement within a defined time due to loss of either publish or its acknowledgement message, it transmits a duplicate publish message. QoS2 guarantees exactly once delivery where neither loss nor duplication of any message is allowed. It is achieved by a two step acknowledgement process.
2. Subscribe: A client can subscribe to a topic by sending subscribe message with desired QoS and receiving an acknowledgement. Similarly it can unsubscribe from any topic. The QoS with which the data is received is minimum of the sending and receiving QoS for that topic.
3. Keep-alive: To differentiate between connection drop and inactivity on a channel, a fixed time called keep-alive is configured. If there is no exchange of application messages for keep-alive time, then the client sends a ping request and server has to acknowledge. On failure of reception of the ping packets at server or client side, appropriate error handling is triggered.
4. Persistent Session: Persistent session can be maintained in lossy networks with frequent disruption by storing the session state. If a client attempts to reconnect as a persistent session, all the previous configurations of the session are restored including any old subscriptions and keep-alive information. Upon re-connection, the client and server resume communication by first sending packets for all unacknowledged messages, thus guaranteeing required QoS.
5. Retain Message: A publish message marked as "retain" is stored by the server. If any subscriber goes offline and reconnects, then all the retained messages are sent to the subscriber upon re-connection with required QoS. In case of a new subscription, if a retained message for a particular topic is already stored, the server sends this information to the subscriber immediately after the subscription request is sent.

6. Will Message: With "will" message, a client can inform all other subscribing clients that it is offline or transmit any other message to its subscribers after loss of connection. When connection between publisher and server is lost, the server transmits the will message to all the clients which subscribe to any topic being published by the publisher.
7. Authentication: MQTT also offers user-password feature for authentication. Implementations can use these fields or provide any other external authentication mechanisms. The applications can also use TLS for data encryption [9].

2.2 MQTT-SN

MQTT-SN is another data centric protocol and is based on MQTT with adaptations to suit the wireless communication environment. Unlike MQTT, MQTT-SN does not require an underlying network like TCP/IP making it a low complexity, light weight protocol. Some of the significant differences between MQTT and MQTT-SN include

1. Gateway Advertisement and Discovery: An MQTT-SN client connects to a MQTT server via gateways and forwarders. A gateway implements translation between MQTT-SN and MQTT protocols. It can be integrated within the server or can be an independent node. Forwarders in the network simply encapsulate the MQTT-SN frames it receives on the wireless side and forwards them unchanged to the gateway and vice versa. This architecture allows minimum implementation and complexity at the client side. A discovery procedure is used by the clients to discover the actual network address of an operating server/gateway. A single wireless network may have multiple gateways which can co-operate in a load-sharing or stand-by mode.
2. Topic Registration: To avoid long topic names as in MQTT, MQTT-SN offers topic registration procedure to reduce bandwidth. During the registration, a topic is assigned with a short topic ID by the server, which is then used in all further communication on that topic. Pre-defined topic IDs and short topic names can also be used for which no registration is required.
3. QoS−1: In addition to QoS 0, 1 and 2, MQTT-SN offers QoS −1 where the client communicates with the server without a formal connection establishment procedure. No topic registration or subscription is required for publishing such messages.
4. Persistent Session: MQTT's persistent session is extended with 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.
5. Support of Sleeping Clients: To save power in battery operated clients, MQTT-SN allows the client to go to sleep mode during which time all messages destined to the client are buffered at its corresponding server/gateway. The client periodically wakes up using keep-alive message, and the buffered messages are sent by the server to the client.

2.3 CoAP

CoAP is a specialized web transfer protocol based on REST architecture, fulfilling Machine to Machine(M2M) requirements in constrained environments. CoAP has low header overhead, parsing complexity, and has uri based addressing. Like HTTP it has request-response model and it also supports publish-subscribe architecture in extended mode. It is stateless HTTP mapping, allowing proxies to be built providing access to CoAP resources via HTTP. Following are significant features of CoAP:

1. Layered Architecture: CoAP implements a request-response model with asynchronous message exchanges at lower layer. In the two layer approach, CoAP's messaging layer is used to deal with UDP and the asynchronous nature of the interactions, and the request-response interactions on top of it for method and response codes. CoAP defines four types of messages: confirmable, non-confirmable, acknowledgement, reset. The basic exchanges of the four types of messages are somewhat orthogonal to the request-response interactions. A Request can be carried in confirmable and non- confirmable message, and a response can be carried in these as well or as piggybacked in acknowledgement messages.
2. Message Correlation and Deduplication: A message and its acknowledgement are correlated through a message ID and request-response are correlated through token numbers. A recipient of a message can deduplicate a message by using the message ID and should process any request only once.
3. Unicast/multicast requests: CoAP can communicate in both unicast and multicast modes. For discovering resources and services in the network, CoAP uses multicast request. After a connection is established with a server, unicast mode is used. A multicast request is characterized by being transported in a CoAP non-confirmable message that is addressed to an IP multicast address instead of a CoAP endpoint.
4. Reliability: CoAP uses UDP for transport layer which by default does not support reliability mechanisms. CoAP uses a layer of messages that supports optional reliability. A QoS of "at least once" is supported in CoAP. To avoid congestion in network, CoAP end point re-transmits with an exponential back-off mechanism.
5. Proxying and Caching:The protocol supports caching of responses in order to efficiently fulfil requests. Caching helps to limit network traffic, improve performance, help sleeping devices and also for security. Simple caching is enabled using freshness and validity information carried with CoAP responses. A cache could be located in an endpoint or an intermediary called proxy. Freshness and Validation model: A max-age option in a response indicates that the response is to be considered not fresh after its age is greater than the specified time. To avoid caching a server can set max-age option to 0. A proxy can validate a stored response after its max-age is over by communicating with the server to update the "freshness" and reuse the response
6. Resource Discovery: Like MQTT-SN, CoAP uses multicast requests to discover services and resources in the network. A CoAP node an send a multi-

Table 1. Comparison of IoT communication protocols

Sl.No.	Protocol Feature	MQTT	MQTT-SN	CoAP
1	Architecture	Asynchronous Message exchange	Asynchronous Message exchange	REST architecture Layered Approach
2	Underlying Transport Layer	TCP	Any	UDP
3	Communication type	UniCast	UniCast Multicast	UniCast Multicast
4	Addressing	ClientID Server address	ClientID Server address	Uri Based
5	Messaging pattern	Publish Subscribe	Publish Subscribe	request-response Publish-Subscribe through "Observe"
6	QoS Levels	No guarantee, atleast once, exact once	No guarantee, atleast once, exact once	No guarantee, atleast once
7	Pub-Sub Multi Client QoS	No guarantee, atleast once, exact once	No guarantee, atleast once, exact once	No guarantee, atleast once
8	Persistent Session	Yes	Yes	Yes
9	Retained Message /Offline/Caching	Yes	Yes	Yes
10	Proxying and Caching	No	Yes	Yes
11	Resource Discovery	No	Yes	Yes
12	Sleep Mode	No	Yes	Yes
13	Security	Optional TLS	Optional TLS	Optional DTLS

cast request to a group of addresses in non-confirmable message to discover if a particular service is offered by any server and those with a valid response will send a required information in a non-confirmable message.

7. Observe feature: Similar to subscribe feature of the MQTT, a client in a CoAP can subscribe to a topic by sending an observe message to the server. The server sends notifications to the subscribing client whenever the particular resource has a new value.

8. Security: CoAP provides provides optional security by binding to Datagram Transport Layer Security (DTLS)

3 Event-B

Event-B [1] is based on B-Method which provides a formal methodology for system-level modeling and analysis. Event-B uses set theory as a modeling notation and first order predicate calculus for writing axioms and invariants. It uses step by step refinement to represent systems at different abstraction levels and provides proofs to verify consistency of refinements. Initially the model is constructed on basis of known requirements. As and when required, one can refine and add the new properties while satisfying the requirements in the underlying model.

An Event-B model has two types of components: contexts and machines. Contexts contain all the data structures required for the system which are expressed as sets, constants and relations over the sets. A machine "sees" a context to use the data structures or types. A machine has several events and can also define variables and its types. A machine can refine another machine to introduce new events, refine events, split events or merge events.

An event in a machine can be seen as a "transition function" of a state machine representing the system. The states in a machine are implicit and are not explicitly defined in the model. An event consists of guards which need to be satisfied before the actions in events are executed. When an event is enabled and executed, the variables are updated as per the actions in the event. This internally leads to transition into another state where some other event(s) may be enabled.

An invariant is a condition on the state variables that must hold permanently. In order to achieve this, it is required to prove that, under the invariant $I(s, c, v)$ in question and under the guards of each event, the invariant still holds after being modified according to the transition associated with that event. In the machine M with variables v, seeing a context C with carrier sets s and constants c, the properties of constants are denoted by *P(s,c)*. Let E be an event of M with guards denoted by *G(s,c,v)* and before-after predicate *R(s,c,v,v')*. Under the properties *P(s,c)*, the invariant *I(s,c,v)*, and the guard *G(s,c,v)*, the feasibility statement shows that the before-after predicate indeed yields at least one after value v' defined by the before-after predicate *R(s,c,v,v')*, as in (1). The invariant preservation statement can then be expressed as in (2). Refer [5] for further details.

$$P(s, c) \wedge I(s, c, v) \wedge G(s, c, v) \Rightarrow \exists v \cdot R(s, c, v, v') \tag{1}$$

$$P(s, c) \wedge I(s, c, v) \wedge G(s, c, v) \wedge R(s, c, v, v') \Rightarrow I(s, c, v') \tag{2}$$

Rodin

Rodin [4] implements Event-B and is based on Eclipse platform. It provides an environment for modeling refinements and discharges proofs. It has sophisticated automatic provers like PP, ML and SMT, which automatically discharge

proofs for refinements, feasibility, invariants and well-definedness of expressions within guards, actions and invariants. Event-B also provides interactive proving mechanism for manual proofs which can be used when the automatic proof discharge fails. Rodin offers various plug-ins for development including different text editors, decomposition/modularization tools, simulator ProB, etc.

ProB

ProB [6] provides a simulation environment through animation for Event-B model. A given machine can be simulated with all its events. In the animation environment, one can select and run the given events by selecting parameters or execute with random solution. During simulation, the state of the system before and after every event execution can be observed. The state gives values of all the variables in the machine, evaluates invariants, axioms and guards for all the events. Additionally any expression can be monitored in the animator. The model can also be checked for deadlocks, invariant violations and errors in the model which will help to construct an accurate model.

4 Protocol Modeling and Decomposition using Event-B

A communication channel is a network connection which is established between a client and a server or between two clients or between two servers. In an IoT system there could be multiple channels connecting several clients and servers. Our Event-B model consists of communication channels of the IoT system which implement a communication protocol. As shown in the Fig. 1, the model has Event-B contexts and machines. The contexts have all the data structures and axioms required to setup a machine. The machine includes communication part of client and server implemented as events, and the properties required to be verified are written as invariants.

The protocol modeling is done in two major steps:

1. Building a common abstract model encompassing the common features of various protocols.
2. Refining this common abstract model into a concrete model of a particular IoT protocol.

To achieve the above methodology we use Machine Decomposition Technique [14], Refinement Strategy [2] and Atomicity Decomposition of Event-B [15].

4.1 Common Abstract Model

The common abstract model implements the commonalities among various protocols as mentioned in Table 1. Figure 2 is a diagrammatic representation of the abstract model.

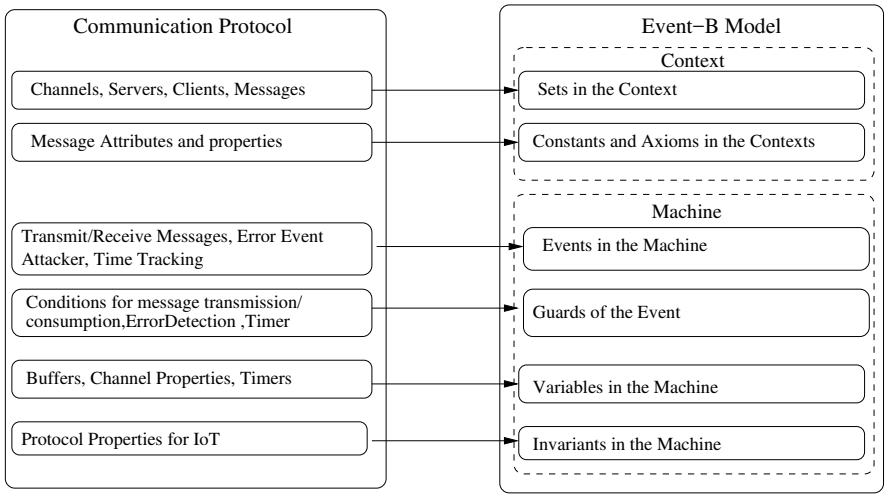

Fig. 1. Mapping between communication protocol and Event-B model

Context A basic communication entity is modelled as a message. Set named MSG and all its attributes are defined as relations over the set MSG and the sets defined for the attributes. A projection function is used to extract the value of an attribute for a given message [3].

Machine Refinements The atomicity of event Communication Channel is broken into two events representing modes of communication: Unicast and Broadcast/Multicast. Similarly a further refinement of the model breaks down the atomicity of these events. Broadcast/Multicast events are used for Service and Resource Discovery in the Network. A UniCast event is broken into ChannelEstablishment and ChannelConversation events. Since these events are not yet atomic, they can be further broken. Figure 3 shows an example of how an atomic event - ChannelConversation of previous refinement is further broken into many more events. Figure 2 and Figure 3 together show the three refinement steps done in the common abstract model. It is to be noted that our common abstract model does not breakdown to the lowest atomic level of events. This is achieved in the next step of building concrete model for a particular protocol.

Machine Decomposition The leaves of the atomicity decomposition diagram give us the events of the final refinement of the common model. Further on when we build models of particular protocols, these events further explode into more atomic events blowing up the size of the model. It has been observed that many of these events have very few interfaces among them and they can be independently be refined. This allows us to use the technique of machine decomposition in Event-B. Figure 4 gives such a decomposition of our abstract

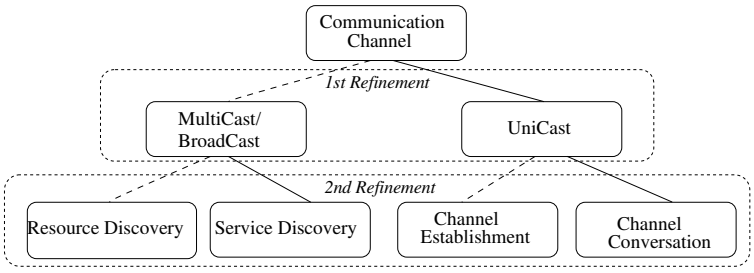

Fig. 2. Atomicity decomposition of common abstract model

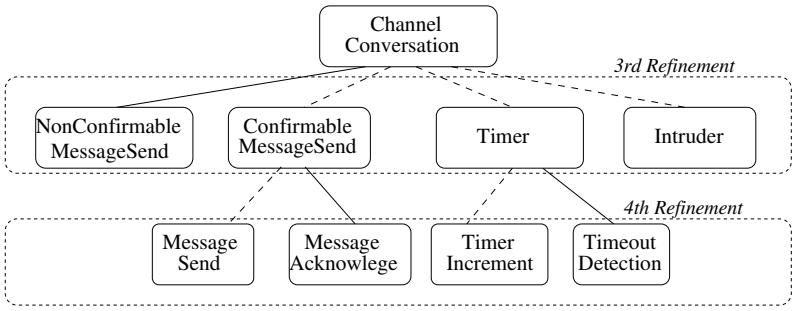

Fig. 3. Atomicity decomposition of ChannelConversation module

model. In Sec. 4.2 we give an example of how these modules of decomposed machines are further refined to give more concrete model of MQTT.

Events in Decomposed Modules There are following events in the final refinement of the common abstract model:

1. Multicast/Broadcast: Multicast/Broadcast is used when a node has to communicate to more than one peer node. Either a broadcast message is sent to all the nodes in the network or a multicast request is sent to a group of nodes in the network. The Multicast/Broadcast event is broken down into atomic events Service Discovery and Resource Discovery which are used to find the nodes that can publish the required information on the network. Once the nodes with required resources/services are discovered, the information is shared with ChannelEstablishment module.
2. ChannelEstablishment: For any two nodes to communicate, it is essential to establish a logical connection between them. The List of Resources/Services that are discovered is used to establish connection with the desired node. Events ConnectRequest and ConnectAcknowledgement are used for connection establishment. The channelEstablished interface is shared with Channel-Conversation Modules. After the communication is over the connection can be disconnected to release the limited resources through Disconnect event.

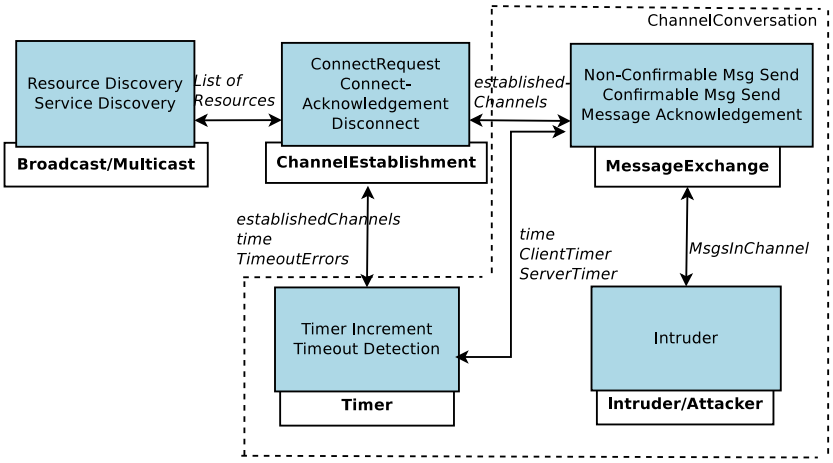

Fig. 4. Machine decomposition of common abstract model

In event of errors, the communication is forcefully disconnected. Error handling events detect errors and appropriately terminate connections as per the session configurations. In our model, Error detection events are related to connection time-out and reconnecting an existing channel. Timeout Error information is communicated through Timeout interface with Timer Module. To verify the desired properties Disconnect event is made convergent to avoid live lock in the model.

3. ChannelConversation: This is a pseudo module which contains the Message-Exchange, Timer and Intruder modules. These sub-modules are described below.
4. MessageExchange: This module includes all the application message transfer events i.e., all the transmit/receive events for message send and acknowledgement. These events update the message buffers and track time for message transmission and reception.
5. Timers: There is a global time ticking through an event called "Timer" and there are local timers maintained by client and server. These timers are incremented when either there is a Send event happening or through a special event Timer which is used to delay time when there is no activity in the channel. Every Transmission and Reception event will store the time at which each message was sent or received. Time tracking is used for "keep-alive" where ping requests have to be sent at given time interval in case of inactivity. It is also used in time-out handling for verifying time related properties like monitoring if acknowledgement for a message is received within a specified time. In further refinements of concrete protocols, timers can also be used for strategies like exponential back off in case of failed acknowledgement.
6. Intruder: This module is introduced to emulate disturbance in channel which leads to loss of messages. A malicious Intruder event can consume any message in the channel that is not yet received by the intended client or server.

Intruder can simulate attackers, connection drops, or any other disturbances in the network that can lead to loss of the application message. This is a convergent event and does not run forever.

4.2 Concrete Protocol Models

From the common abstract model, the decomposed machines are refined further to add details specific to a protocol. Some of the features which are not used in the protocol need not be used or refined. For example there is no broadcast or multicast support in MQTT protocol. Hence this module does not need any refinement in MQTT model. The contexts from the abstract model are extended to add detailed attributes. Channel variables and internal buffers are introduced to track the dynamic behaviour of the channel that include messages in channel, topics subscribed, payload counters, send and receive buffers, timers, configuration settings, etc. Following is a detailed description of MQTT protocol model created from the abstract model. We then briefly describe the other two protocol models (MQTT-SN and CoAP) which follow similar procedures.

MQTT Protocol Model MQTT protocol is modeled by abstracting communication network in an IoT system consisting of two channels. For illustrative purpose, we have modeled the channels with two servers and two clients.

ChannelEstablishment Module - From the abstract module containing events ConnectRequest, ConnectAcknowledgement and Disconnect, MQTT specific refinement is done to include configuration details and disconnection due to errors. When a channel is established, the configuration settings of the channel communicated between the client and the server are stored in channel variables.

MessageExchange Module - MessageExchange module is refined to include publish and subscribe message and their acknowledgement events. These events are further refined to send original message, duplicate message and reception of the message at both client and server sides. Figure 5 and Figure 6 give the refinement steps and atomic decomposition for transmit messages in this module. Similar model is built for acknowledgement messages.

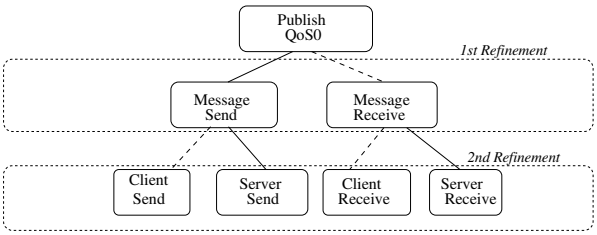

Fig. 5. Atomicity decomposition of non-confirmable message transmission - QoS0

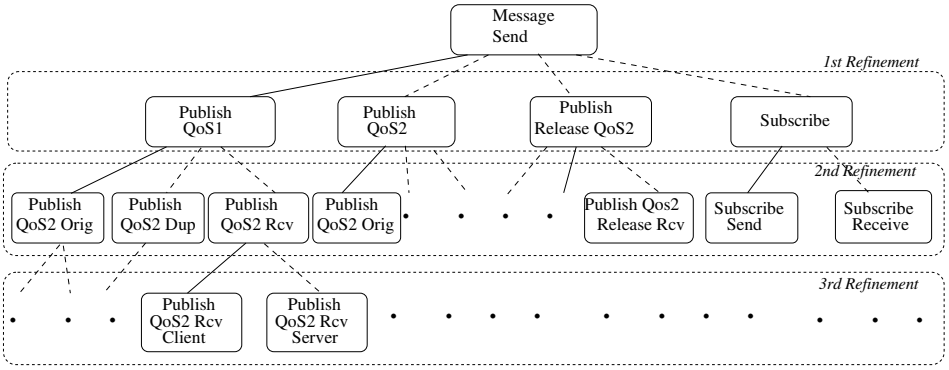

Fig. 6. Atomicity decomposition of confirmable message transmission - QoS1 and QoS2

To track if the correct message is delivered with required QoS and time, the "Payload" is implemented as a counter with a range of 0 to 9. The range of the counter can be extended to any number without affecting our model. This method of using counter as payload allows us to uniquely identify every message transmitted. By keeping a track of how many times the message with a given payload value is received, we can verify interesting properties related to QoS, message ordering, retained message and persistent sessions. Figure 7 is an example of the QoS0 Publish event transmitted by an MQTT client. The guards ensure that a message of type publish with QoS0 is transmitted on the channel which is already established. In the actions, the channel is populated with a new message carrying unique payload, *client_timer* is initialized, direction of the message is set, PayloadCounter is incremented and Timer-increment event is triggered.

Event Client_Publish_QoS0 ⟨ordinary⟩ $\widehat{=}$
any
M10
ch
Topic
where
grd1: $Msg\_Topic(M10) = Topic$
grd2: $IncrementTime = FALSE$
grd3: $M10 \in MSG \land Topic \in TOPIC$
grd4: $PayloadCounter \in 0..9$
grd5: $(M10 \mapsto ((Publish \mapsto AtmostOnce) \mapsto PayloadCounter)) \in Msg\_Type\_QoS$
then
act1: $Client\_timer(ch) := 1$
act2: $IncrementTime := TRUE$
act3: $PayloadCounter := PayloadCounter + 1$
act4: $LimitTimer := 1$
act5: $Channel\_Direction(ch) := Client\_To\_Server$
act6: $MsgsInChannel(ch) := MsgsInChannel(ch) \cup \{M10 \mapsto PayloadCounter\}$
end

Fig. 7. Event for publishing message with QoS0

In the MQTT model, Timer Module is refined to include ClientSide and ServerSide Timer events and corresponding Timeout events. Intruder Module does not have any particular refinement for MQTT.

MQTT-SN Protocol Refinement MQTT-SN model reuses MessageExchange, ChannelEstablishment, Timer and Intruder Modules from MQTT. The Multicast/Broadcast Module is refined from common abstract model to add events related to gateway discovery in the network using search gateway messages. New topic registration procedure is added to the ChannelConversation Module.

CoAP Protocol Refinement ChannelConversation module from abstract model is refined to include request-response layer by adding events that are enabled to send a request and receive a corresponding response. Each of these events then trigger the message layer events to transmit confirmable or non-confirmable messages and receive corresponding acknowledgements. At the request-response layer events related to separate-response or piggybacked-response are added. Token ID matching and message ID matching is carried out to ensure every request receives its response within a defined time. ChannelEstablishment module is refined to add multi-hop connection consisting of multiple channels. Multicast/Broadcast module is refined to discover resources and services in the network. Timer and Intruder modules are directly used from the common abstract model. Figure 8 is an example of an event to send a request through a confirmable message.

4.3 Model Validation

ProB is used for validating our model through simulation of events and checking LTL properties for common abstract model. Accuracy of the model can be obtained by executing different runs and observing the sequence of events and variable values in each of these events. ProB also reports any invariant violation or error in events which is then corrected in the model. Model validation is also done by writing invariants and seeing that these invariants are satisfied through the refinements.

Event RequestConfirmMsgSend ⟨ordinary⟩ $\widehat{=}$
any
ch
Payload
M10
token
Request
MsgID
where
grd1: $Payload \in \mathbb{N} \land time \in \mathbb{N} \land M10 \in MSG \land Payload \in \mathbb{N}$
grd2: $Request \in \{PUT, GET, POST, DELETE\}$
grd3: $(M10 \mapsto ((Confirmable \mapsto Request) \mapsto (token \mapsto MsgID))) \in Msg\_Type\_Code\_Token\_MsgID$
grd4: $MsgID \in \mathbb{N} \land MsgID \notin PendingMsgs(ch)$
grd5: $LimitTimer \neq 1$
grd6: $ch \in establishChannel$
grd7: $token \in \mathbb{N} \land token \notin PendingRequests(ch)$
then
act1: $MessageToken(M10) := token$
act2: $PendingRequests(ch) := PendingRequests(ch) \cup \{token\}$
act3: $Msg\_Payload(M10) := Payload$
act4: $MsgsInChannel(ch) := MsgsInChannel(ch) \cup \{M10 \mapsto Payload\}$
act5: $LimitTimer := 1$
act6: $SendTRange := SendTRange \lhd\!\!\!\!- \{Payload \mapsto time\}$
act7: $RetransmissionCounter(Payload) := 0$
act8: $TokenSent(ch) := TokenSent(ch) \cup \{Payload\}$
act9: $PayloadCounter := PayloadCounter + 1$
act10: $MsgID(M10) := MsgID$
act11: $PendingMsgs(ch) := PendingMsgs(ch) \cup \{MsgID\}$
act12: $MsgSentCh(ch) := MsgSentCh(ch) \lhd\!\!\!\!- \{M10 \mapsto Payload\}$
end

Fig. 8. Event for sending a confirmable message for a CoAP Request

5 Verification of IoT Properties using Event-B

Following are some of the significant properties that are verified through the model by writing them as invariants that have to be satisfied for all the events in protocol specific models. Properties 1 to 7 are verified in MQTT and MQTT-SN models and 8 to 11 are verified in CoAP model.

A property required to be satisfied by the model has been expressed as an invariant. The property invariant contains two parts : Well-definedness expressions and the actual property to be proved. For example, in the invariant for Duplicate Channel Property, for every quantified variable, there is a range defined to which it belongs: $\forall chnl \cdot (chnl \in Server\_establishedChannel)$, followed by well-definedness conditions for the projection function:
$chnl \in dom(Channel\_Server)$ and the actual property to be verified is written at the end of the equation: $(Channel\_ClientID(ch) = Channel\_ClientID(chnl) \Rightarrow (ch = chnl)$. We omit the well-definedness conditions and state only the actual property to be proved.

For every property listed, the property is stated in English language as written in the protocol specification followed by the property written in our Event-B model. The variable names given in our model are self explanatory and are not

exhaustively described. For example ClientID associated with a channel has a variable name, $Channel\_ClientID$.

1. Duplicate channel: If the ClientID represents a client already connected to the server then the server must disconnect the existing client. i.e., If a client and server are already connected through a channel, then they cannot establish another channel.

$$\begin{aligned}\forall ch \cdot \forall chnl \cdot ((& ch \in Server\_establishedChannel \\ & \wedge chnl \in Server\_establishedChannel) \\ & \wedge (Channel\_Server(ch) = Channel\_Server(chnl)) \\ & \wedge (Channel\_ClientID(ch) = Channel\_ClientID(chnl)) \\ & \Rightarrow (ch = chnl))\end{aligned} \tag{3}$$

2. Timers validity: The time tracker variable is always greater than or equal to any local time counter.

$$\begin{aligned}\forall pc \cdot ((& pc1 \in 0 \cdot \cdot 9 \\ & \Rightarrow (time \geq RcvTRange(pc1) \wedge (time \geq SendTRange(pc1)))\end{aligned} \tag{4}$$

3. Message Ordering: If both client and server make sure that no more than one message is "in-flight" at any one time (by not sending a message until its predecessor has been acknowledged), then no QoS1 message will be received after any later one. For example a subscriber might receive them in the order $1, 2, 3, 3, 4$ but not $1, 2, 3, 2, 3, 4$. Setting an in-flight window of 1 also means that order will be preserved even if the publisher sends a sequence of messages with different QoS levels on the same topic. Refer to Sec. 4.6 in [9].

$$\begin{aligned}\forall ch \cdot \forall pc1 \cdot \forall pc2 \cdot ((& pc1 \in 0 \cdot \cdot 9 \wedge pc2 \in 0 \cdot \cdot 9 \wedge ch \in establishChannel \\ & \wedge (pc1 \in Client\_MsgSentQoS2(ch) \vee pc1 \in Client\_MsgSentQoS1(ch)) \\ & \wedge (pc2 \in Client\_MsgSentQoS2(ch) \vee pc2 \in Client\_MsgSentQoS1(ch)) \\ & \wedge (time > SendTRange(pc2) + Response\_Timeout) \\ & \wedge pc1 \neq pc2 \wedge (SendTRange(pc1) < SendTRange(pc2)) \\ & \Rightarrow (RcvTRange(pc1) \leq RcvTRange(pc2))\end{aligned} \tag{5}$$

4. Persistent Session: When a client reconnects with "CleanSession" set to 0, both the client and server must re-send any unacknowledged publish packets (where QoS > 0) and publish release packets using their original packet Identifiers. Refer to Normative Statement number MQTT-4.4.0-1 in [9]. Hence a transmit message with QoS > 0 should always receive an acknowledgement. The variable RcvTRange is updated with current time only after the message is received. Hence it should be greater than the SendTRange time.

$$\begin{aligned}\forall ch \cdot \forall pc \cdot ((& pc \in 0 \cdot \cdot 9 \wedge ch \in establishChannel \\ & \wedge Channel\_CleanSess(ch) = FALSE \\ & \wedge ((pc \in Client\_MsgSentQoS1(ch)) \vee (pc \in Client\_MsgSentQoS2(ch)) \\ & \wedge (time > (SendTRange(pc) + Response\_Timeout)) \\ & \Rightarrow (RcvTRange(pc) > SendTRange(pc)))\end{aligned} \tag{6}$$

5. QoS of message in client to server channel:
   – QoS1: Publish from client to server with QoS = 1. If client sends a packet with QoS = 1 to the server, then at least one copy of the packet should be received at the server side even in case of loss or duplicate transmission of messages. Refer to Sec. 4.3.2 in [9].

$$\begin{aligned}\forall ch \cdot \forall pc \cdot ((pc \in 0 \cdot \cdot 9 \wedge ch \in establishChannel \\ \wedge (pc \in Client\_MsgSentQoS1(ch)) \\ \wedge (time > (SendTRange(pc) + Response\_Timeout)) \wedge QC > 0 \\ \wedge ((time - t1) \geqslant Response\_Timeout)) \\ \Rightarrow (\exists QC \cdot ((QC \geq 1) \wedge Server\_MsgReceived\_1(pc) = QC)))\end{aligned} \tag{7}$$

   – QoS2: Publish from client to server with QoS = 2. If client sends a packet with QoS = 2 to the server, exactly one copy of the packet should be received at the server side even in case of loss or duplicate transmission of messages. This is the highest quality of service and increases overhead. Refer to Sec. 4.3.3 in [9].

$$\begin{aligned}\forall ch \cdot \forall pc \cdot ((pc \in 0 \cdot \cdot 9 \wedge ch \in establishChannel \\ \wedge (pc \in Client\_MsgSentQoS2(ch)) \\ \wedge (time > (SendTRange(pc) + Response\_Timeout))) \\ \Rightarrow (Server\_MsgReceived\_2(pc) = 1))\end{aligned} \tag{8}$$

6. Retained Message: If the Retain flag is set to 1 in a publish Packet sent by a client to a server, then the server must store the application message and its QoS, so that it can be delivered to future subscribers whose subscriptions match its topic name. Refer to Sec. 3.3.1.3 in [9]. For Retained QoS1 Message: Publish of retained message to the subscriber which subscribed the topic with QoS1 and the transmit message was QoS > 0.

$$\begin{aligned}\forall ch \cdot \forall pc \cdot \forall chnl \cdot \forall msg \cdot \forall QoS \cdot ((ch \in establishChannel \\ \wedge chnl \in establishChannel \wedge QoS \in QOS \wedge QC > 0 \wedge pc \in 0 \cdot \cdot 9 \\ \wedge (pc \in Client\_MsgSentQoS1(ch) \wedge Msg\_Retain(msg) = TRUE \\ \wedge (msg \mapsto ((PUBLISH \mapsto QoS) \mapsto pc)) \in Msg\_Type\_QoS \\ \wedge ((Msg\_Topic(msg) \mapsto ATLEASTONCE) \in Channel\_TopicQoS(chnl)) \\ \wedge ((time - SendTRange(pc)) \geqslant Response\_Timeout)) \\ \Rightarrow (\exists QC \cdot ((QC \geq 1) \wedge Client\_MsgReceived\_2(chnl) = QC)))\end{aligned} \tag{9}$$

   Similar equation is written for Retained QoS2 Message: Publish of retained message to the subscriber which subscribed the topic with QoS2 and the transmit message was QoS > 0.

7. QoS of a message from Client1 to Client2: The effective QoS of any message received by the subscriber depends on the QoS with which the publishing client transmits this message and the QoS set by the subscriber while subscribing for the given topic. The effective QoS with which message is delivered

is the minimum of the two QoS. Below is an example of the message which is transmitted with QoS1 but was subscribed with QoS2. In this case the message is received by the subscriber with the effective QoS of 1. The below property states, if a message which is transmitted from Client1 on channel ch with QoS1 (ATLEASTONCE) and if the cleint2 has subscribed to this topic on channel chnl with QoS2 (EAXACTONCE) then Client2 will receive this message from the Server1 on channel chnl within a configured amount of time, at least once. Refer to Sec. 4.3 in [9].

$$\begin{aligned}
&\forall ch \cdot \forall pc \cdot \forall chnl \cdot \forall msg \cdot ((pc \in 0 \cdot\cdot 9 \land ch \in establishChannel \\
&\quad \land msg \in MSG \land chnl \in establishChannel \\
&\quad \land (pc \in Client_MsgSentQoS1(ch) \\
&\quad \land (msg \mapsto ((PUBLISH \mapsto ATLEASTONCE) \mapsto pc)) \in Msg_Type_QoS \\
&\quad \land ((Msg_Topic(msg) \mapsto EXACTONCE) \in Channel_TopicQoS(chnl)) \\
&\quad \land ((time - SendTRange(pc)) \geqslant Response_Timeout))) \\
&\quad \Rightarrow (\exists QC \cdot ((QC \geq 1) \land Client_MsgReceived_2(chnl) = QC)))
\end{aligned} \tag{10}$$

Similarly all combinations of the QoS at publisher and subscriber end have been verified.

8. Response to a Request Matching: Regardless of how a response is sent, it is matched to the request by means of a token that is included by the client in the request. Refer to Sec. 5.3 in [11].

$$\begin{aligned}
&\forall ch \cdot \forall pc \cdot ((pc \in 0 \cdot\cdot 11 \land ch \in establishChannel \\
&\qquad \land pc \in TokenSent(ch) \land time \geqslant 1 \\
&\qquad \land (time - SendTRange_Token(pc) \geqslant Response_Timeout)) \\
&\qquad \Rightarrow (RcvTRange_Token(pc) - SendTRange_Token(pc) \\
&\qquad\quad \geqslant Response_Timeout))
\end{aligned} \tag{11}$$

9. Reliable message transfer: An acknowledgement or reset message is related to a confirmable message or non-confirmable message by means of a message ID along with additional address information of the corresponding endpoint. Every confirmable message has a matching acknowledgement. Refer to Sec. 4.4 in [11].

$$\begin{aligned}
&\forall ch \cdot \forall pc \cdot ((pc \in 0 \cdot\cdot 11 \land ch \in establishChannel \\
&\quad \land pc \in MsgSent(ch) \land time \geqslant 1 \\
&\quad \land (time - SendTRange_Payload(pc) \geqslant Response_Timeout)) \\
&\quad \Rightarrow (RcvTRange_Payload(pc) - SendTRange_Payload(pc) \\
&\qquad \geqslant Response_Timeout))
\end{aligned} \tag{12}$$

10. Exponential Backoff: The sender retransmits the Confirmable message at exponentially increasing intervals, until it receives an acknowledgement or

runs out of attempts. Refer to Sec. 4.2 in [11].

$$\begin{aligned}\forall ch \cdot \forall pc \cdot ((pc \in 0 \cdot\cdot 11 \wedge ch \in establishChannel \wedge pc \in MsgSent(ch) \\ \wedge RetransmissionCounter(pc) \geqslant Max\_Retransmit(ch) \\ \Rightarrow ((SendTRange(pc) - SendTPrev(pc)) \leq Ack\_Timeout(pc) \\ \wedge (SendTRange(pc) - SendTPrev(pc)) > 0))\end{aligned} \tag{13}$$

5.1 Proof Obligations Results

Our validated models of MQTT, MQTT-SN and CoAP have together discharged 1840 proof obligations, of which 88% proof obligations were automatically discharged through AtlierB, SMT, PP and ML provers. The proof obligations include well-definedness of predicates and expressions in invariants, guards, actions, variant and witnesses of all the events, feasibility checks, variable reuse check, guard strengthening and witness feasibility in refinements, variant checks for natural number and decreasing variants for convergent and anticipated events, theorems in axioms and invariant preservation for refinements and invariants used for verification of required properties. About 30% of proofs discharged in the models are for verification of properties written as invariants. Table 2 gives a summary of the properties verified.

Table 2. Proof obligation statistics for verified properties of IoT protocols

Sl.No.	Protocol Property	Proof Obligations	Result
1	Duplicate Channel	10	Passed
2	Message Ordering	34	Passed
3	Persistent Session	34	Passed
4	QoS1 in single channel	26	Passed
5	QoS2 in single channel	26	Passed
6	Retained QoS1 message	24	Passed
7	Retained QoS2 message	24	Passed
8	Effective QoS0 in Multi channel(3 cases)	66	Passed
9	Effective QoS1 in Multi channel(3 cases)	66	Passed
10	Effective QoS2 in Multi channel(3 cases)	72	Passed
11	Request-Response Matching and Timeout	39	Passed
12	Confirmable Message ID Matching and Timeout	39	Passed
13	Exponential Backoff	39	Passed

6 Related Work

Communication protocols for IoT have been used for over a decade now, but there has been no attempt to provide formal semantics for these protocols. A recent paper shows that there are scenarios where MQTT has failed to adhere to the QoS requirement [16]. However the paper is limited to partial model of MQTT protocol for QoS properties. In another work, a protocol used for IoT - Zigbee is verified for properties related to connection establishment properties [17] using Event-B. In [19] and [20], the authors give methods to evaluate performance of MQTT protocol with regards to different QoS levels used and compare with other IoT protocol CoAP. In [18] the author again tests connection properties using passive testing for XMPP protocol in IoT.

We differ from the above mentioned approaches by proposing a framework comprising of a common model for IoT protocols which can be used to build models of different IoT protocols. These models verify properties required for IoT like connection establishment, persistent sessions, retained-message transmission, will messages, message ordering, proxying, caching and QoS and provide proof obligations for these properties through automatic proof discharge and interactive proof discharge methods.

7 Conclusion and Future work

In this paper we have proposed a framework using Event-B to model IoT protocols. We then have used this framework and went on to model some of the widely used IoT protocols viz., MQTT, MQTT-SN and CoAP. Through simulation and proof obligation discharge in Rodin, we have formally verified that the properties related to QoS, persistent session, will, retain messages, resource discovery, two layered request-response architecture, caching, proxying and message deduplication. We show that the protocols work as intended in an uninterrupted network as well as with an intruder which consumes messages in the network. The three protocols modeled in this paper implement simple mechanisms to provide reliable message transfer over a lossy network. They are also able to reduce overhead by providing features like persistent connections, retain messages, caching and proxying which are essential for IoT systems. Our work is a stepping stone towards providing formal semantics of IoT protocols and behaviour of IoT systems.

Future research would focus on modeling the other aspects of protocols like security, user authentication, encryption and different attacker modules. We would also like to move verification of more properties from the concrete protocol models to the common abstract model. We would like to further compare other protocols for IoT like AMQP and XMPP by modeling them using our framework. It would also be interesting to integrate the protocol model into an existing model of IoT system and verify the properties required at the system level.

References

1. Event-B, http://www.Event-B.org/
2. Abrial J.R., Modeling in Event-B: System and Software Engineering, Cambridge University Press(2010)
3. Evans, N., Butler, M.: A proposal for records in Event-B. In: International Symposium on Formal Methods, pp. 221-235. Springer, Berlin Heidelberg (2006)
4. Rodin Tool, http://wiki.Event-B.org/index.php/Rodin_Platform
5. Rodin Hand Book, https://www3.hhu.de/stups/handbook/rodin/current/pdf/rodin-doc.pdf
6. ProB Tool, https://www3.hhu.de/stups/prob/index.php/Main_Page
7. Gartner newsroom, http://www.gartner.com/newsroom/id/3165317
8. Karagiannis, V., Chatzimisios, P., Vazquez-Gallego, F., Alonso-Zarate J.: A survey on application layer protocols for the internet of things. In: Transaction on IoT and Cloud Computing, 3(1):11-7 (2015)
9. MQTT Version 3.1.1 Specification, http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
10. MQTT-SN Version 1.2 Specification, http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf
11. The Constrained Application Protocol (CoAP) Specification RFC7252, https://tools.ietf.org/html/rfc7252
12. Extensible Messaging and Presence Protocol (XMPP) Core RFC6120, http://xmpp.org/rfcs/rfc6120.html
13. Advanced Messaging Queuing Protocol Version 1.0 Specification, http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf
14. Pascal, C., and Renato, S.: Event-B model decomposition, DEPLOY Plenary Technical Workshop (2009)
15. Salehi Fathabadi A., Butler M., Rezazadeh A.: A Systematic Approach to Atomicity Decomposition in Event-B. In: 10th International Conference, SEFM. Thessaloniki, Greece (2012)
16. Aziz, B.: A Formal Model and Analysis of the MQ Telemetry Transport Protocol. In: Ninth International Conference, Availability, Reliability and Security (ARES), pp. 59-68. Fribourg (2014)
17. Gawanmeh, A.: Embedding and Verification of ZigBee Protocol Stack in Event-B. In: Procedia Computer Science, Volume 5, pp. 736-741. ISSN 1877-0509, (2011)
18. Che, X., Maag, S.: A Passive Testing Approach for Protocols in Internet of Things. In: Green Computing and Communications (GreenCom), IEEE and Internet of Things (iThings/CPSCom), IEEE International Conference on and IEEE Cyber, Physical and Social Computing, pp. 678-684. IEEE Press (2013)
19. Lee, S., Kim, H., Hong, D. K., Ju, H.: Correlation analysis of MQTT loss and delay according to QoS level. In: The International Conference on Information Networking(ICOIN), pp. 714-717. IEEE (2013)
20. Thangavel, D., Ma, X., Valera, A., Tan, H.X., Tan, C.K.: Performance evaluation of MQTT and CoAP via a common middleware. In: IEEE Ninth International Conference, Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), pp. 1-6. IEEE Press (2014)