A Framework for Modeling and Verifying IoT Communication Protocols

Maithily Diwan and Meenakshi D'Souza(✉)

International Institute of Information Technology, Bangalore, India
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 protocols · 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

K.G. Larsen et al. (Eds.): SETTA 2017, LNCS 10606, pp. 266–280, 2017.
https://doi.org/10.1007/978-3-319-69483-2_16

(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 foot-print [8]. Features like publish-subscribe, messaging layer, QoS(Quality of Service) 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 these 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 and interactive proof discharge in Event-B using the Rodin tool [4] verifies if these invariants (properties) are satisfied for all the events in the model.

The paper is organized as follows. IoT protocols and their properties are described in Sect. 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 Sect. 4. Verified properties and their results are presented in Sect. 5. Section 6 discusses related work and Sect. 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 with underlying physical, DLL, network and transport layer. However TCP/IP is heavy weight as compared to a lighter UDP, in which case reliability has to be built in the application layer protocol. Other IoT communication requirements are: low bandwidth, low memory consumption, small code foot-print, self recovery, resource discovery, light-weight, low message overhead, low power consumption, authentication, security, appropriate QoS.

We briefly describe 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. 3 levels of QoS: "At most once"- no acknowledgement is expected for a publish message, "at least once" - every message receives an acknowledgement and "exact once" - guaranteed message delivery without duplicates.
2. Subscribe: Clients can subscribe/unsubscribe to a topic with desired QoS.
3. Keep-alive: In absence of application messages within keep-alive time, client sends a ping request to keep the network active.
4. Persistent Session: Persistent session is achieved by storing the session state of channel and can be restored upon re-connection. It includes previous configurations, subscriptions, unacknowledged messages, etc.
5. Retain Message: When a new subscriber or an offline subscriber re-connects, the retained message is immediately published with configured QoS.
6. Will message: Pre-configured "will" message is sent by the server when a publishing client goes offline and wants to inform the subscribing clients.
7. Authentication: A user-password feature is used for authentication. TLS (Transport Layer Security) is optional 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. Significant differences between MQTT and MQTT-SN are listed below:

1. Gateway Advertisement and Discovery: A MQTT-SN client conntects to MQTT server via a gateway, implementing translation between the two protocols. A discovery procedure is used by the clients to discover the actual network address of an operating server/gateway.
2. Topic Registration: To reduce bandwidth, a client can use pre-defined short topic names/IDs or register a long topic name with the server and use a corresponding topic ID for further communication.
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 and topic registration procedures.
4. Support of Sleeping Clients: Power saving clients can to go to sleep mode and wake up periodically using keep-alive message. A server/gateway buffers messages destined to the client and send them to client when they wake up.

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. 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. The messaging layer deals with UDP and asynchronous nature of interactions, and the request-response interactions use method and response codes. Requests and responses are carried in confirmable and non-confirmable messages. A response can be piggybacked in acknowledgement or separate message.
2. Unicast/multicast requests: For discovering resources and services in the network, CoAP uses multicast request. After a connection is established with a server, unicast mode is used.
3. Reliability: CoAP uses a layer of messages that supports optional reliability of "at least once" with an exponential back-off mechanism.
4. Proxying and Caching: A cache could be located in an endpoint or an intermediary called proxy. Caching is enabled using freshness and validity information carried with CoAP responses. A max-age option in a response indicates its not fresh after its age is greater than the specified time. A proxy can however validate the stored response with server even after max-age expiry.
5. Resource Discovery: Like MQTT-SN, CoAP uses multicast requests to discover services and resources in the network.
6. Observe feature: CoAP can be used in publish-subscribe mode by using observe and notification options.
7. Security: Optional security using 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

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	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
6	QoS Levels	AtmostOnce, AtleastOnce, ExactOnce	AtmostOnce, AtleastOnce, ExactOnce	AtmostOnce, AtleastOnce
7	Persistent Session	Yes	Yes	Yes
8	Retained Message /Offline/Caching	Yes	Yes	Yes
9	Proxying/Caching	No	Yes	Yes
10	Resource Discovery	No	Yes	Yes
11	Sleep Mode	No	Yes	Yes
12	Security	Optional TLS	Optional TLS	Optional DTLS

variables and its types. A machine can refine another machine to introduce new events, refine events, split events or merge events. 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.

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 in question and under the guards of each event, the invariant still holds after being modified according to the transition associated with that event [5].

Rodin and ProB

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 [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.

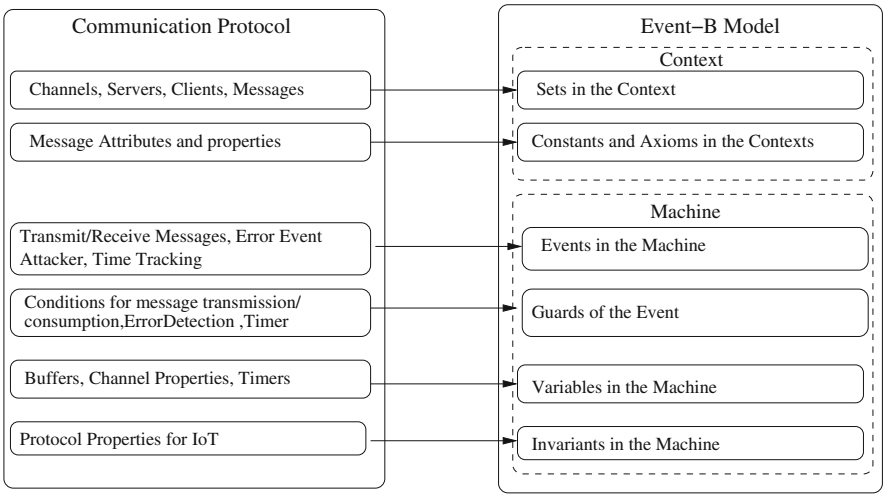

Fig. 1. Mapping between communication protocol and Event-B model

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.

Our modeling is done using the techniques of machine decomposition [14], refinement [2] and atomicity decomposition in 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.

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 into Service and Resource Discovery. A UniCast event is broken into ChannelEstablishment and ChannelConversation events. Since these events are not yet atomic, they can be further split as shown in Fig. 3 where ChannelConversation of previous refinement is further broken into many more events. Figures 2 and 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.

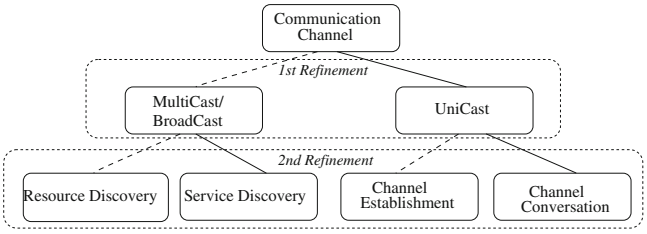

Fig. 2. Atomicity decomposition of common abstract model

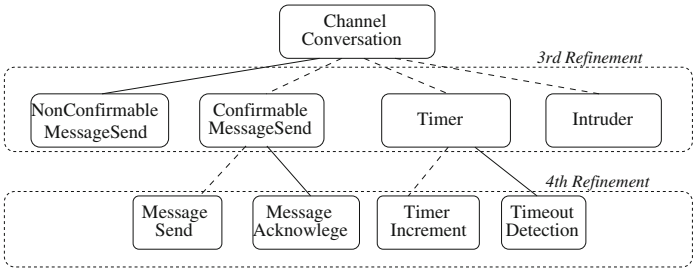

Fig. 3. Atomicity decomposition of ChannelConversation module

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

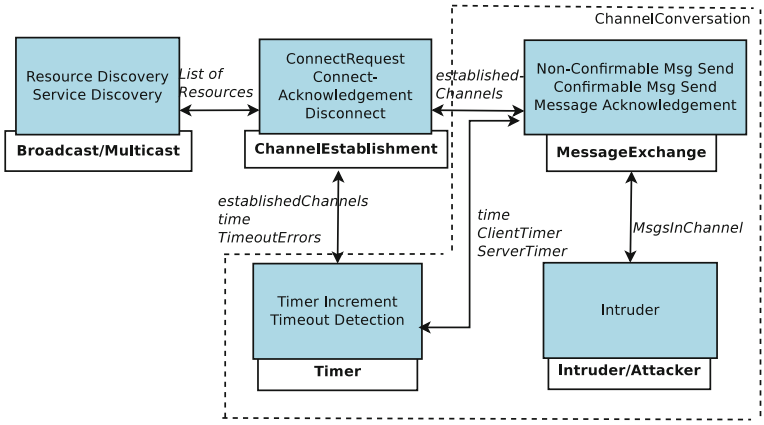

Fig. 4. Machine decomposition of common abstract model

Events in Decomposed Modules

1. Multicast/Broadcast: It is used when a node has to communicate to more than one peer node. 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: The List of Resources/Services is used to establish connection with the desired node. Events ConnectRequest and ConnectAcknowledgement are used for connection establishment. After the communication is over the connection can be disconnected to release the limited resources through Disconnect event. Disconnect event is made convergent to avoid live lock in the model. 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 and the channelEstablished interface is shared with ChannelConversation modules.

3. ChannelConversation: This is a pseudo module which contains the Message-Exchange, Timer and Intruder modules.
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 to just delay time in case of channel inactivity. Every transmission and reception event will store the time at which each message was sent or received. Time tracking is used for keep-alive mechanism, time-out handling and for verifying time related properties. 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 - First refinement of the module introduces publish and subscribe message with 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 gives the refinement steps and atomic decomposition for transmit messages in this module. Similar model is built for acknowledgement messages.

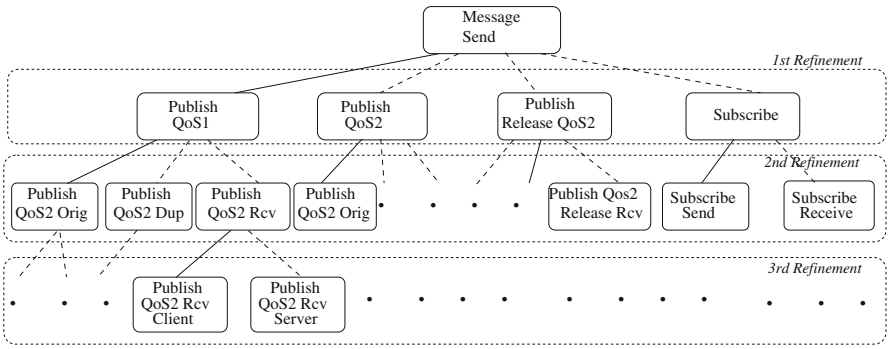

Fig. 5. Atomicity decomposition of confirmable message transmission - QoS1 and QoS2

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 which allows us to uniquely identify every message transmitted. The range of the counter can be extended to any number without affecting our model. 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 6 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, ClientTimer is initialized, direction of the message is set, PayloadCounter is incremented and Timer-increment event is triggered.

Timer and Intruder modules - Timer Module is refined to include ClientSide and ServerSide Timer, 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

Event Client_Publish_QoS0 ⟨ordinary⟩ $\widehat{=}$
any
M10
ch
Topic
where
grd1: $Msg\_Topic(M10) = Topic$
grd2: $IncrementTime = FALSE$
grd3: $M10 \in MSG \wedge Topic \in TOPIC$
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. 6. Event for publishing message with QoS0

enabled to send a request and receive a corresponding response either piggybacked or separate. Each of these events then trigger the message layer events to transmit confirmable or non-confirmable messages and receive corresponding acknowledgements. Token ID matching and message ID matching is carried out to ensure every request receives its response. 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.

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 and verifying invariants.

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. The property invariant contains two parts well-definedness expressions and the actual property to be proved. We omit the well-definedness conditions and state only the actual property to be proved. Properties 1 to 7 are verified in MQTT and MQTT-SN models and 8 to 11 are verified in CoAP model.

1. Message Ordering: If both client and server make sure that no more than one message is "in-flight" at any one time, 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 $1, 2, 3, 2, 3, 4$.
Refer to Sect. 4.6 in [9].

$$\begin{aligned}
&\forall ch \cdot \forall pc1 \cdot \forall pc2 \cdot ((pc1 \in 0 \cdot \cdot 9 \wedge pc2 \in 0 \cdot \cdot 9 \wedge ch \in establishChannel \\
&\quad \wedge (pc1 \in Client_MsgSentQoS2(ch) \vee pc1 \in Client_MsgSentQoS1(ch)) \\
&\quad \wedge (pc2 \in Client_MsgSentQoS2(ch) \vee pc2 \in Client_MsgSentQoS1(ch)) \\
&\quad \wedge (time > SendTRange(pc2) + Response_Timeout) \\
&\quad \wedge pc1 \neq pc2 \wedge (SendTRange(pc1) < SendTRange(pc2)) \\
&\quad \Rightarrow (RcvTRange(pc1) \leq RcvTRange(pc2))
\end{aligned} \tag{1}$$

2. 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]. The variable RcvTRange is updated with current time only after the message is received. Hence it should be greater than the SendTRange time.

$$\begin{aligned}
&\forall ch \cdot \forall pc \cdot ((pc \in 0 \cdot \cdot 9 \wedge ch \in establishChannel \\
&\quad \wedge Channel_CleanSess(ch) = FALSE \\
&\quad \wedge ((pc \in Client_MsgSentQoS1(ch)) \vee (pc \in Client_MsgSentQoS2(ch)) \\
&\quad \wedge (time > (SendTRange(pc) + Response_Timeout)) \\
&\quad \Rightarrow (RcvTRange(pc) > SendTRange(pc)))
\end{aligned} \tag{2}$$

3. QoS of a message from Client1 to Client2: The effective QoS of any message received by the subscriber is minimum of QoS with which the publishing client transmits this message and the QoS set by the subscriber while subscribing for the given topic. E.g. Publishing client sends with QoS1 oand subscribing client has subscribed with QoS2, with effective QoS being 1. Refer to Sect. 4.3 in [9].

$$\begin{aligned}
&\forall ch \cdot \forall pc \cdot \forall chnl \cdot \forall msg \cdot ((pc \in 0 \cdot \cdot 9 \wedge ch \in establishChannel \\
&\quad \wedge msg \in MSG \wedge chnl \in establishChannel \\
&\quad \wedge (pc \in Client_MsgSentQoS1(ch) \\
&\quad \wedge (msg \mapsto ((PUBLISH \mapsto AtleastOnce) \mapsto pc)) \in Msg_Type_QoS \\
&\quad \wedge ((Msg_Topic(msg) \mapsto ExactOnce) \in Channel_TopicQoS(chnl)) \\
&\quad \wedge ((time - SendTRange(pc)) \geqslant Response_Timeout))) \\
&\quad \Rightarrow (\exists QC \cdot ((QC \geq 1) \wedge Client_MsgReceived_2(chnl) = QC)))
\end{aligned} \tag{3}$$

4. Exponential Backoff: The sender retransmits the Confirmable message at exponentially increasing intervals, until it receives an acknowledgement or runs out of attempts. Refer to Sect. 4.2 in [11].

$$\begin{aligned}\forall ch \cdot \forall pc \cdot ((pc \in 0 \cdot\cdot 11 \wedge ch \in establishChannel \wedge pc \in MsgSent(ch) \\ \wedge RetransmissionCounter(pc) \geqslant Max\_Retransmit(ch) \\ \Rightarrow ((SendTRange(pc) - SendTPrev(pc)) \leq Ack\_Timeout(pc) \\ \wedge (SendTRange(pc) - SendTPrev(pc)) > 0))\end{aligned} \tag{4}$$

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 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, Cambridge (2010)
3. Evans, N., Butler, M.: A proposal for records in Event-B. In: Misra, J., Nipkow, T., Sekerinski, E. (eds.) FM 2006. LNCS, vol. 4085, pp. 221–235. Springer, Heidelberg (2006). doi:10.1007/11813040_16
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. Trans. IoT Cloud Comput. **3**(1), 11–7 (2015)
9. MQTT Ver. 3.1.1. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/os/mqtt-v3.1.1-os.html
10. MQTT-SN Ver. 1.2. http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf
11. The Constrained Application Protocol (CoAP) RFC7252. https://tools.ietf.org/html/rfc7252
12. Extensible Messaging and Presence Protocol (XMPP) Core RFC6120. http://xmpp.org/rfcs/rfc6120.html
13. Advanced Message Queuing Protocol ver. 1.0. http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf
14. Pascal, C., 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: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) SEFM 2012. LNCS, vol. 7504, pp. 78–93. Springer, Heidelberg (2012). doi:10.1007/978-3-642-33826-7_6
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, vol. 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)