Master's Thesis

PEREZ Julio
September 2005

MQTT Performance Analysis with OMNeT++

IBM Zurich Research Laboratory, Switzerland

Technical Supervisors :
Dr. Sean Rooney
Dr. Paolo Scotton

Academic Supervisor :
Dr. Pietro Michiardi

This thesis is NOT confidential.

Networking
Institut Eurécom

Abstract

Publish/subscribe systems increasingly enjoy popularity because of their simplicity, efficiency and scalability. The Message Queue Telemetry Transport protocol (MQTT) is a scalable publish/subscribe protocol developed by IBM and targeted at networks consisting of small footprint devices such as sensors and control devices.

Up until now, no performance analysis of the MQTT protocol has been carried out, although especially for small footprint devices with restricted processing power and power supply, it is important to know how the protocol behaves in different environments.

This thesis addresses the investigation of MQTT's performance under various error conditions of the communication channels and over different protocol stacks. To perform the simulations the discrete event simulator OMNeT++ was used. Various network setups have been tested and measures such as end-to-end delay and power consumption have been estimated and compared. To accomplish this task, OMNeT++ had first to be extended by additional modules and models to allow for simulating MQTT's performance over TCP and wireless channels.

Résumé

Les protocoles publish/subscribe gagnent de plus en plus en popularité à cause de leur simplicité, efficacité et possibilité de passage à l'échelle. Le protocole Message Queue Telemetry Transport (MQTT), développé par IBM, est un protocole publish/subscribe pour les réseaux consistués de petits dispositifs comme, par exemple, des capteurs ou des dispositifs de contrôle.

Malgré le fait qu'il soit important de connaître l'efficacité du protocole sous différentes conditions, surtout si on considère des dispositifs avec des ressources limitées, la performance de MQTT n'a pas été testée jusqu'à présent.

Dans cette thèse, nous proposons une étude des performances de MQTT sous différents taux d'erreur de transmission et avec différentes piles de protocoles. Les résultats ont été obtenus avec l'outil de simulation OMNeT++. Nous avons adopté comme critères de performance le temps de transmission de bout-en-bout et la consommation d'énergie. Ces valeurs ont été mesurées ou estimées pour différents types de réseaux. Afin d'obtenir ces résultats, nous avons dû développer et mettre au point de nouveaux modules pour OMNeT++, notamment un module TCP et un module simulant un réseau sans fil.

Acknowledgments

Special thanks go to all the people at the IBM Zurich Research Laboratory who strongly supported and advised me during the last 6 months, including Sean Rooney, Paolo Scotton, Daniel Bauer, Luis Garcés-Erice and Bernard Metzler.

I also would like to thank the Eurécom Institute for making this thesis possible and giving me the opportunity to make this special experience.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Direct device-to-device or machine-to-machine communication within computer networks is increasingly enjoying great popularity. Probably the most popular technology implementing this communication paradigm is the Radio Frequency Identification (RFID) used in e.g. supply chain management. Further examples are the use of wireless sensor networks for environmental monitoring, maintenance systems for early recognition of component replacement requirements as well as real-time process-control systems for industrial automation.

The demand for real-time status information and notification requires integration of these networks into general enterprise computer networks. A main issue for machine-to-machine communication is the completely different information flow compared to conventional computer networks. Instead of large flows from central servers to clients possibly at the edge of the network, the main data flow for sensor network systems is from many devices at the edge of the network towards a few central servers.

Publish/subscribe systems implement this machine-to-machine communication. Senders label each message with the name of a topic, rather than addressing it to specific recipients. The messaging system then sends the message to all eligible systems that have asked to receive messages on that topic. This form of asynchronous messaging is a far more scalable architecture than point-to-point alternatives such as message queueing, since message senders need only concern themselves with creating the original message, and can leave the task of servicing recipients to the messaging infrastructure. It is a very loosely coupled architecture, in which senders often do not even know who the subscribers are. Moreover, publish/subscribe networks are highly dynamic, as generally any node can leave and rejoin as many times as it wants.

The rest of this document is organized as follows:

Chapter 2 In this chapter the MQTT protocol is described, including the message types and formats and the QoS levels.

Chapter 3 OMNeT++, the simulation system used to perform the actual analysis of MQTT, is shortly discussed.

Chapter 4 To actually simulate MQTT over TCP, both protocols had to be implemented for OMNeT++. This process is described in chapter 4. Also an additional OMNeT++ model implemented is discussed, namely the Gilbert-Elliot wireless channel model.

Chapter 5 This chapter is dedicated to the simulations performed, the results gathered and their interpretation.

Chapter 6 Finally, the thesis concludes with a short summary on OMNeT++ and MQTT and proposals for future work.

Chapter 2

WebSphere Message Queue Telemetry Transport Protocol

In this chapter the WebSphere Message Queue Telemetry Transport protocol (MQTT) will be described in detail. After a short introduction the message formats and the message commands will be given followed by a discussion of the quality of service message flows.

2.1 Introduction

Currently many oil and gas pipeline distribution and metering systems use traditional SCADA (Supervisory Control And Data Acquisition) systems to collect data and prepare information for use by isolated applications. Some even record information on chart recorders and meter tickets to be read and manually input into billing systems to facilitate data transfer.

First deployed over 20 years ago, the SCADA architecture has changed very little over time. A SCADA system is typically based on a poll/response model, continually interrogating devices and acquiring information for report logs. A SCADA system is a large capital investment and as such, replacement of a legacy system in order to take advantage of new technologies and to achieve increased efficiency can be a costly venture. This exercise can involve significant costs particularly when this requires effort to develop custom applications. The benefits of efficiently delivering data directly from sensors, flow computers or the legacy SCADA system are obvious.

Minimizing the time and costs of data acquisition while maximizing its utilization across multiple applications will ultimately improve business performance. What this really means is delivering data efficiently from field devices as diverse as flow meters and pressure sensors in the process con-

trol environment, to applications across many other industries such as batch counters and check weighers on a factory production line. This can then be disseminated directly into back-office applications.

Arcom in association with IBM, have worked to develop a new method for end-to-end enterprise data delivery, which offers wider distribution of operational data and allows legacy SCADA systems to be enhanced at a minimal cost. The lightweight TCP/IP based protocol MQTT was developed to deliver data directly from remote devices and data producers into the publish and subscribe 'integration broker'. A remote device can thereby consist of e.g. a conventional Linux PC running the MQTT protocol. In a RFID environment, the RFID reader device could take up the role of a remote device delivering data to the backend system. Though, on-going research focuses on integrating MQTT into small sensor devices such as Crossbow's Mote systems and it is on such devices that we will focus in this thesis.

From the broker, information can be distributed on a 'one-to-many' basis and delivered directly to multiple applications. This solution supplies event driven, real time data to any application within the enterprise such as SAP based ERP, billing, scheduling or even directly to the energy trading floors. MQTT is designed to minimize the required communication bandwidth by using a protocol with very low overhead, and providing three levels of delivery assurance or 'quality of service' (QoS). The QoS for each data message can be selected by the system programmer on the basis of its importance to the enterprise applications or because of bandwidth constraints. The low data overhead ensures it has very little impact on existing local area networks and is also cost effective over dial up, radio or communication satellite channels (the trend for telecommunication operators which offer GPRS and satellite based data services is to bill user traffic on a per byte basis). Essentially, by implementing the MQTT protocol along with the publish/subscribe data model, you can make full use of existing SCADA based reporting systems, while increasing flexibility through TCP/IP connectivity.

2.2 Protocol Specification

2.2.1 MQTT Message Format

The message header for each MQTT command message contains a fixed header. Some messages also require a variable header and a payload.

2.2.1.1 Fixed Header

The fixed header is contained in each MQTT command message. Table 2.1 shows the fixed header format. The possible message types are listed in table 2.2.

Table 2.1: Fixed Header Format

	7	6	5	4	3	2	1	0
Byte 1	Message Type				DUP Flag	QoS Level		RETAIN
Byte 2	Remaining Length							

The DUP flag is only used with QoS level greater than zero (and therefore an acknowledgment is required, see 2.2.2.2) and is set by the client or the broker when a PUBLISH message is resent.

The QoS field indicates the level of assurance for delivery of a MQTT message. The possible QoS values are shown in table 2.3.

The RETAIN flag, when set, indicates that the broker sends the message as an initial message to new subscribers to this topic. Thereby, a new client connecting to the broker can quickly establish the current number of topics. This is useful where publishers send messages on a "report by exception" basis, and it might be some time before a new subscriber receives data on a particular topic. After sending a SUBSCRIBE message to one or more topics, a subscriber receives a SUBACK message, followed by messages for each newly subscribed topic for which the publishers set the retain flag. The retained messages are published from the broker to the subscriber with the retain flag set and with the same QoS with which they were originally published, and are therefore subject to the usual QoS delivery assurances. The retain flag is set in the messages to the subscribers to distinguish them from "live" data so that they are handled appropriately by the subscriber. However it should be noted that overuse of this flag can inhibit scalability, as a new subscriber may receive a huge number of retained messages.

The second byte of the fixed header contains the remaining length field. It represents the number of bytes remaining within the current message, including data in the variable header and the payload (see following sections).

Table 2.2: Command Message Types

Mnemonic	Enumeration	Description
Reserved	0	Reserved
CONNECT	1	Client request to Connect to Broker
CONNACK	2	Connect Acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish Acknowledgment
PUBREC	5	Publish Received
PUBREL	6	Publish Release
PUBCOMP	7	Publish Complete
SUBSCRIBE	8	Client Subscribe request
SUBACK	9	Subscribe Acknowledgment
UNSUBSCRIBE	10	Client Unsubscribe request
UNSUBACK	11	Unsubscribe Acknowledgment
PINGREQ	12	Ping Request
PINGRESP	13	Ping Response
DISCONNECT	14	Client is Disconnecting
Reserved	15	Reserved

Table 2.3: QoS Levels

QoS Value	Bit 2	Bit 1	Description	
0	0	0	At Most Once	Sent only once. Receiver will get zero or one copies.
1	0	1	At Least Once	Acknowledged delivery. Receiver gets at least one copy.
2	1	0	Exactly Once	Assured delivery. Receiver gets exactly one copy.
3	1	1	Reserved	

2.2.1.2 Variable Header

The message header for some types of WebSphere MQTT command messages contains a variable header. It resides between the fixed header and the payload. The format of the variable header fields are described in table 2.4 and are listed in the order in which they must appear in the header. For some fields a more detailed explanation is given in the following paragraphs.

Table 2.4: Variable Header Fields

Field	Present In	Description
Protocol Name	CONNECT	UTF-encoded string representing the protocol name "MQIsdp" [1].
Protocol Version	CONNECT	8-bit unsigned value representing the revision level of the protocol used by the client (currently version 3).
CONNECT Flags	CONNECT	Clean start, Will Flag, Will QoS, and Will Retain, see page 7.
Keep Alive Timer	CONNECT	Defines the maximum time interval in seconds between messages received from a client. See also page 8.
CONNECT Return Code	CONNACK	Defines a one byte return code. 0: Connection accepted 1-3: Connection refused
Topic Name	PUBLISH	UTF-encoded string which identifies the information channel to which payload data is published.
Message Identifier	Several	16-bit unsigned integer (not used when QoS is zero). See page 9.

CONNECT Flags Since the CONNECT message is the first information exchanged between a MQTT client and the MQTT broker, it seems sensible to have some special fields within this message to carry extra information. Some of them have already been described in table 2.4. In the next table the remaining fields within the CONNECT variable header are shown.

The purpose of the Clean Start flag is to return the client to a known, "clean" state with the broker. If the flag is set, the broker discards any outstanding messages, deletes all subscriptions for the client, and resets the

[1] Originally called "ArgoOTWP" in version 1, the protocol name was changed to "MQIpdp" in version 2 before being set to "MQIsdp" (MQSeries Integrator SCADA Device Protocol) in the final release 3.

Message ID to 1. The client proceeds without the risk of any data from previous connections interfering with the current connection.

The remaining flags are all related to the so called Will message which may be contained in the payload of the CONNECT message. If the Will flag is set, the Will message is published on behalf of the client by the broker when either an I/O error is encountered by the broker during communication with the client, or the client fails to communicate within the Keep Alive Timer schedule. Therefore, the Will message is sent in the event that the client is disconnected unexpectedly and is published to the Will Topic. Sending a Will message is not triggered by the broker receiving a DISCONNECT message from the client.

If the Will flag is set, the Will QoS and Will Retain fields must be present in the CONNECT flag's byte, and the Will Topic and Will Message fields must be present in the payload. The Will QoS specifies at which service level a potential Will message should be published by the broker whereas the Will Retain flag indicates whether or not the broker should retain the Will message after having it published.

Table 2.5: CONNECT Flags

	7	6	5	4	3	2	1	0
Flag	Reserved	Reserved	Will Retain	Will QoS		Will Flag	Clean Start	Reserved

Keep Alive Timer The Keep Alive Timer, measured in seconds, defines the maximum time interval between messages received from a client. It enables the broker to detect that the network connection to a client has dropped, without having to wait for the long TCP timeout. The client has a responsibility to send a message within each keep alive time period. In the absence of a data-related message during the time period, the client sends a PINGREQ message, which the broker acknowledges with a PINGRESP message.

If the broker does not receive a message from the client within one and a half times the keep alive time period (the client is allowed "grace" of half a time period), it disconnects the client as if the client had sent a DISCONNECT message. This action does not impact any of the client's subscriptions. See the DISCONNECT notification in section 2.2.2.5 for more details.

The Keep Alive Timer is a 16-bit value that represents the number of seconds for the time period. The actual value is application-specific, but

a typical value is a few minutes. The maximum value is approximately 18 hours. A value of zero (0) means the client is never disconnected.

Message Identifier The message identifier is present in the variable header of the following WebSphere MQTT messages: PUBLISH, PUBACK, PUBREC, PUBREL, PUBCOMP, SUBSCRIBE, SUBACK, UNSUBSCRIBE and UNSUBACK. This field is only present in messages where the QoS bit in the fixed header indicates QoS level 1 or 2. See section 2.2.2.2 for more information.

The message ID is a 16-bit unsigned integer. It typically increases by exactly one from one message to the next, but is not required to do so. This assumes that there are never more than 65535 messages "in flight" between one particular client-broker pair at any time. The message ID 0 is reserved as an invalid message ID.

2.2.1.3 Payload

As stated previously, some MQTT command messages carry additional information in the payload part. The payload contained in the CONNECT command consists of either one or three UTF-8 encoded strings. The first string uniquely identifies the client to the broker. The second string is the Will topic, and the third string is the Will message. The second and third strings are present only if the Will flag is set in the CONNECT flag's byte.

In a SUBSCRIBE command message the payload contains a list of topic names to which the client wants to subscribe, and the QoS level. These strings are UTF-encoded.

Furthermore, there is the SUBACK message containing a list of granted QoS levels in the payload. These are the QoS levels at which the administrators for the broker have permitted the client to subscribe to a particular topic. Granted QoS levels are listed in the same order as the topic names in the corresponding SUBSCRIBE message.

With regard to the PUBLISH message, the payload part contains application-specific data only. No assumptions are made about the nature or content of the data.

2.2.2 MQTT Command Messages

In the preceding sections the general structure of MQTT messages was explained. This section explains in more detail the purpose of each MQTT command message and which fields have to be taken care of. Please refer to [1] for the exact structure of each command message.

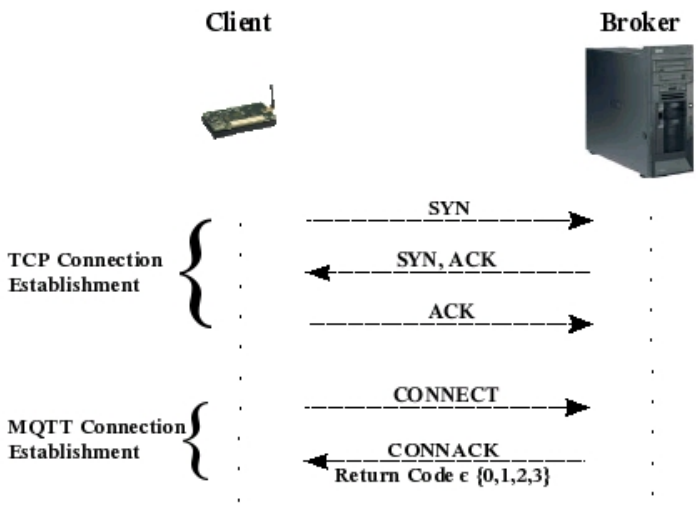

Figure 2.1: MQTT Connection Establishment

2.2.2.1 Connecting to the Broker

Assuming that a client has already established a TCP connection to the broker, the next step is to establish a MQTT connection to the broker. The two messages targeted at establishing a protocol level session are the so called CONNECT and CONNACK messages.

The CONNECT message is sent by the client to the broker to inform the latter that the client wants to set up a MQTT session. As explained in page 7 this initial message contains the CONNECT flags and the Keep Alive Timer value in the variable header. Since the broker has to be able to distinguish the clients from one another, the CONNECT message must also contain a client identifier that is unique across all connected and connecting clients. This ID is present in the payload of the message, where also the Will topic and message are put if necessary (see page 7).

The response of the broker to the CONNECT request by a client is a CONNACK message whose only purpose is to inform the client if the connection attempt was successful. From table 2.4 we can see that only return code 0 indicates a successful MQTT connection attempt. Return codes 1 to 3 indicate that the connection was refused by the broker because of an unacceptable protocol version (currently only version 3 is acceptable), an illegal client identifier that was not specified or exceeds the maximum length of 23 characters or because the broker is unavailable for any reason.

If the client does not receive a CONNACK message from the broker within

a client-specified timeout period, the client closes the TCP/IP socket connection and restarts first a TCP connection followed by the MQTT session establishment.

2.2.2.2 Publishing to Topics

A PUBLISH message is sent by a client to a broker for distribution to interested subscribers. Each PUBLISH message is associated with a topic name, which is part of the variable header. This is a hierarchical name space that defines a taxonomy of information sources for which subscribers can register an interest. A message that is published to a specific topic name is delivered to connected clients subscribed to that topic. The actual data published to the topic name is contained in the payload section of the message. The content and format of the data is application-specific. Please note that PUBLISH messages can be sent either from a publisher to the broker, or from the broker to a subscriber.

Depending on the QoS level at which a PUBLISH message is sent, the message ID and duplicate flag fields are used. For QoS 0, no message ID is included in the message and the message is never resent (best effort). For QoS greater than zero a unique message ID has to be included in the PUBLISH message and the duplicate flag is set when a retransmission is triggered because no response was received. The awaited response depends again on the QoS and is explained in the next paragraphs.

Quality of Service Flows PUBLISH messages are delivered according to the QoS level specified in the corresponding field. Depending on this value, sender and receiver take different actions.

QoS 0 With QoS 0 the message is delivered according to the best efforts of the underlying TCP/IP network. A response is not expected and no retry semantics are defined in the protocol. The message arrives at the broker/subscriber either once or not at all. Upon reception of a PUBLISH message by the broker, it forwards the message to all interested subscribers.

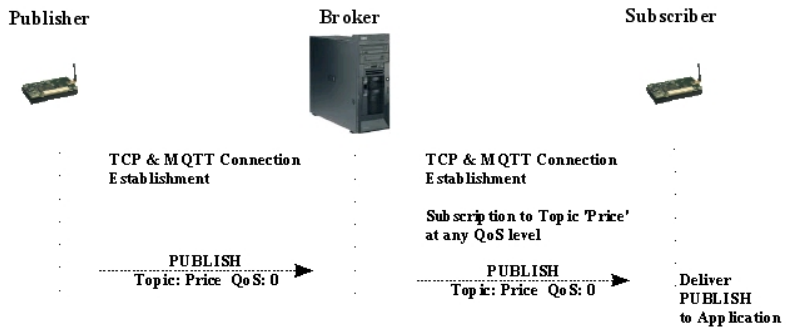

Figure 2.2: MQTT Publication with QoS 0

QoS 1 The reception of a PUBLISH message with QoS 1 by the broker or subscriber is acknowledged by a PUBACK message (PUBlish ACKnowledgment) containing the same message ID as the PUBLISH message that is acknowledged. When the acknowledgment message is not received after a specified period of time, the sender resends the message with the DUP bit set in the message header. The message arrives at the receiver at least once. The broker, upon reception of a QoS 1 PUBLISH message, logs the message to persistent storage, makes it available to any interested parties, and returns a PUBACK message to the sender. In the case where it receives a duplicate message from the client, the broker republishes the message to all interested subscribers, and sends another PUBACK message to the publisher. A subscriber receiving a duplicate PUBLISH message makes it available to the application and sends back a PUBACK to the broker.

Once the sender of the PUBLISH message (either a publisher or the broker) receives the PUBACK message, it discards the corresponding PUBLISH message that was stored persistently.

It is important to note that a PUBLISH message is sent at QoS 1 from the broker to the subscriber only if the minimum out of the QoS of the original PUBLISH message sent by the publisher and the granted QoS by the broker to the subscriber for that topic is equal to 1. See also section 2.2.2.3 for more details.

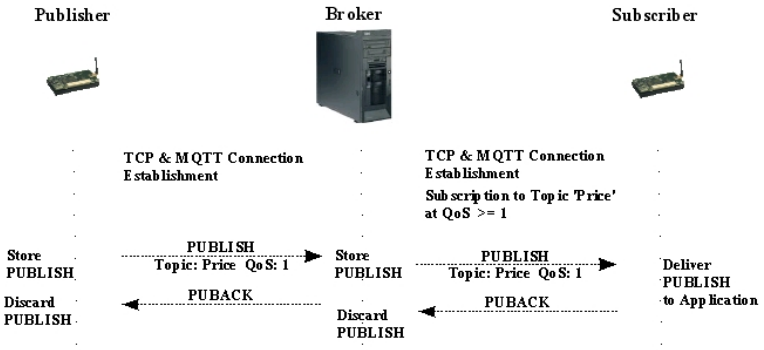

Figure 2.3: MQTT Publication with QoS 1

QoS 2 Additional protocol flows above QoS level 1 ensure that duplicate messages are not delivered to the receiving application. This is the highest level of delivery, for use when duplicate messages are not acceptable. There is an increase in network traffic, but it is usually acceptable because of the importance of the message content.

Between a publisher and the broker, the protocol flow is as follows (and analogously between a broker and an interested subscriber). Upon reception of a PUBLISH message with QoS 2, the message is stored persistently by the broker and acknowledged with a so called PUBREC message (PUBlish RECeived). It contains the message ID of the original PUBLISH message. When it receives a PUBREC message, the publisher, as a next step, sends a PUBREL message (PUBlish RELease) to the broker with the same message ID as the PUBREC message (and the original PUBLISH message). Finally, upon reception of the PUBREL message, the broker sends back a PUBCOMP (PUBlish COMPlete) to the publisher, again with the same message ID as the PUBREL that is acknowledged. It is only after receiving the PUBREL message that the broker makes the original PUBLISH message available to interested subscribers.

When the client receives a PUBCOMP message, it discards the original PUBLISH message because it has been delivered, exactly once, to the broker. If a failure is detected, or after a defined timeout period, each part of the protocol flow is retried with the DUP bit set.

Again, one has to keep in mind that the PUBLISH message is delivered at QoS 2 to the subscribers only and only if the corresponding granted QoS is 2 and the PUBLISH message was also sent out by the publisher at QoS 2. Only in this case the additional protocol flows ensure that the message is

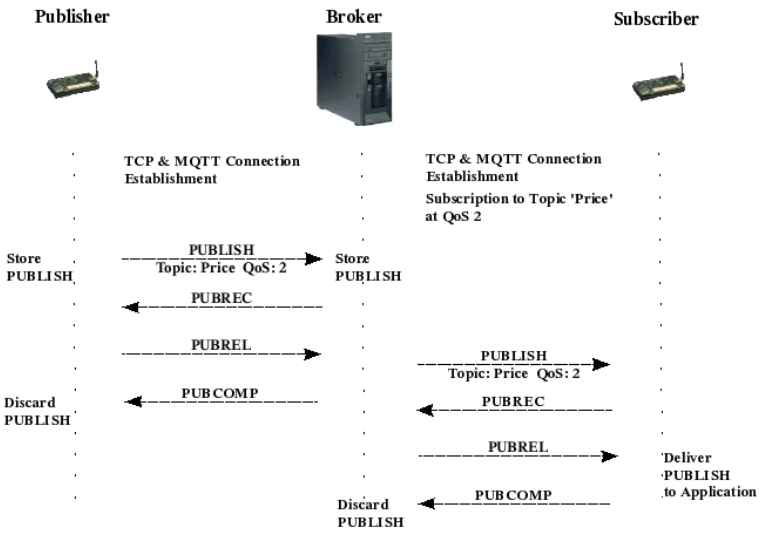

Figure 2.4: MQTT Publication with QoS 2

delivered to the application at the subscriber once only.

2.2.2.3 Subscribing to Topics

After having successfully established a MQTT connection to the broker, a client wishing to receive information on certain topics has to tell the broker about this interest. The basic data that a MQTT subscriber has to deliver to the broker is a list of topic names which interest the client. Furthermore, the QoS level at which the client wants to receive published messages can be specified. Hence, a MQTT SUBSCRIBE command message contains a list of topic names/QoS pairs.

As any other message, SUBSCRIBEs can get lost. Therefore, in MQTT the SUBSCRIBE message uses QoS 1 to ensure that the message is received by the broker, i.e. upon receiving a subscription request from a client, the broker sends back a SUBACK message. The latter contains a list of granted QoS levels. These are the levels at which the administrators for the broker permit the client to subscribe to specific topic names. In the current version of the protocol, the broker always grants the QoS level requested by the subscriber. The order of granted QoS levels in the SUBACK message matches the order of the topic names in the corresponding SUBSCRIBE message. However, the granted QoS don't necessarily mean that the subscriber will get PUBLISH messages at these QoS levels. In fact, the client receives PUBLISH messages at less than or equal to these granted QoS levels, depending on the

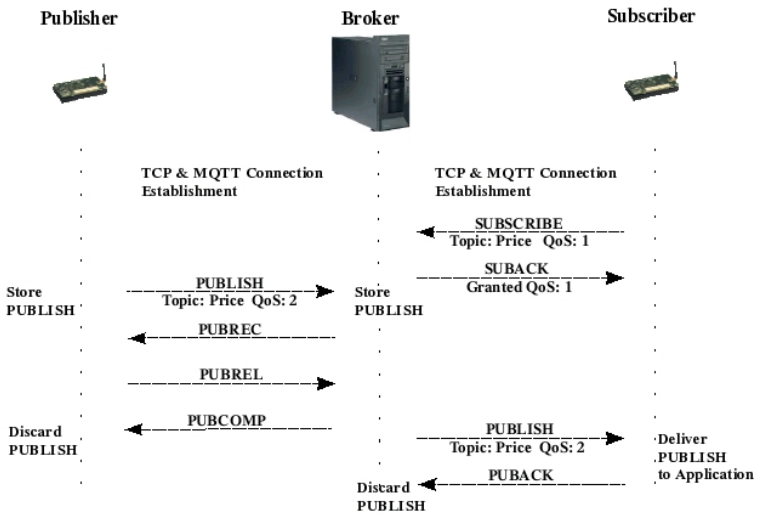

Figure 2.5: MQTT Subscription to Topics

QoS levels of the original messages from the publisher. As an example, let us assume that a publisher sends PUBLISH messages at QoS 2 to the broker and that a subscriber is subscribed to the topic at a granted QoS of 1. Then, the broker will send PUBLISH messages at QoS 1 to the subscriber. If on the other hand you have a subscription at granted QoS 2, the messages will be published at QoS 2 from the broker to the subscriber. Thus, the message is always published to the subscriber at QoS equal to the minimum of the QoS of the original PUBLISH message sent to the broker and the QoS granted by the broker to the subscriber for the topic matching the one of the PUBLISH. Figure 2.5 illustrates the first example mentioned above. It is worth noting that in the example the broker publishes the message with QoS 1, but the original PUBLISH message is left unchanged and therefore the QoS field is set to 2.

Since the SUBSCRIBE message uses QoS 1, it must contain a message ID to match an arriving SUBACK to the correct SUBSCRIBE message. Also, if after a client-specified time no SUBACK is received, a duplicate of the SUBSCRIBE is sent to the broker with the duplicate flag set.

2.2.2.4 Unsubscribing from Topics

Once a subscriber is subscribed to certain topics, it is also allowed to unsubscribe from some of the topics if the client is not interested anymore in receiving information to those topics. This is done by using the UNSUB-

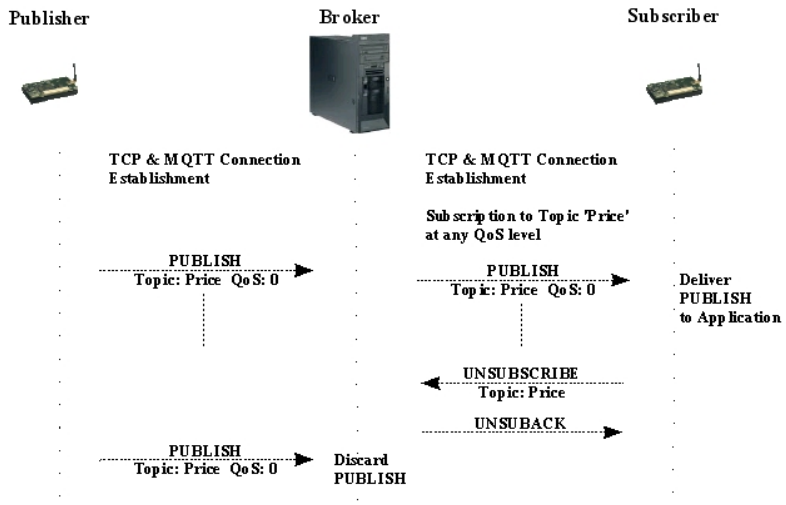

Figure 2.6: MQTT Unsubscription from Topics

SCRIBE/UNSUBACK command messages. The client sends an UNSUBSCRIBE message to the broker containing a list of topic names from which it wishes to unsubscribe. The broker responds with an UNSUBACK message to confirm that it received the unsubscription request. After having sent the UNSUBACK, the subscriber will not receive anymore PUBLISH messages to the topics specified in the corresponding UNSUBSCRIBE message.

Obviously, the UNSUBSCIBE is sent at QoS 1 and therefore a message ID is used to match UNSUBACKs to UNSUBSCRIBEs. As in the case of a subscription, timeouts are used and once expired, an UNSUBSCRIBE message is resent with the duplicate flag set.

2.2.2.5 Disconnecting

The DISCONNECT message is sent from the client to the broker to indicate that it is about to close its TCP/IP connection. This allows for a clean disconnection, rather than just dropping the line. Sending the DISCONNECT message does not affect existing subscriptions. They are persistent until either explicitly unsubscribed, or if there is a clean start. The broker retains QoS 1 and QoS 2 messages for topics to which the disconnected client is subscribed to until the client reconnects. QoS 0 messages are not retained, since they are delivered on a best efforts basis. Note that the DISCONNECT message is not acked.

2.2.2.6 Ping Request

The PINGREQ message is an "are you alive" message that is sent from or received by a connected client. A PINGRESP message is the response to a PINGREQ message and means "yes I am alive". Keep Alive messages flow in either direction, sent either by a connected client or the broker.

As explained in page 8, each client is responsable for sending at least one message within each keep alive period in order to assure the broker that the network connection has not dropped. If there is no data-related message during the time period, MQTT PINGREQ messages have to be sent to prevent the broker from DISCONNECTing the client because of a keep alive timeout. The response to a PINGREQ is a PINGRESP message which, alike the PINGREQ contains nothing but the fixed header.

Chapter 3

The Discrete Event Simulation System OMNeT++

For simulations performed in this project, the OMNeT++ simulator was used. This chapter serves as a short presentation of the main features of OMNeT++.

3.1 Introduction

OMNeT++ [2] is a discrete event simulator based on C++, is highly modular, well structured and scalable. It provides a basic infrastructure wherein modules exchange messages. The name OMNeT++ stands for Objective Modular Network Testbed in C++. It has an open-source distribution policy and can be used free of charge by academic research institutions. It runs on Windows and Unix platforms, including Linux, and offers a command line interface as well as a powerful graphical user interface. The simulator can be used, for instance, to model communication and queueing networks, multiprocessors and other distributed hardware systems as well as to validate hardware architectures.

3.2 Modeling Concept

An OMNeT++ model consists of hierarchically nested modules, which communicate by passing messages to each other. OMNeT++ models are often referred to as networks. The top level module is the system module. The system module contains submodules, which can also contain submodules themselves. The depth of module nesting is not limited; this allows the user to reflect the logical structure of the actual system in the model structure.

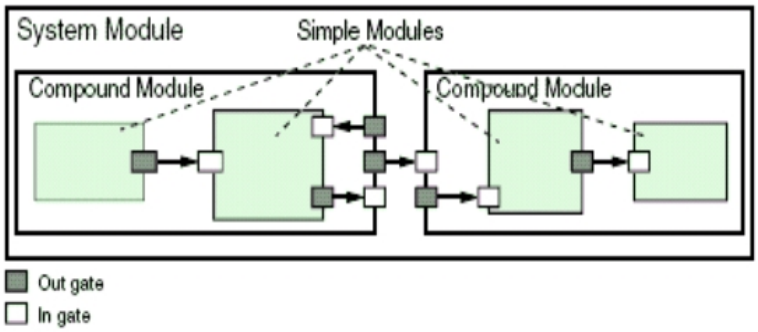

Figure 3.1: OMNeT++ Module Hierarchy

The model structure is described with OMNeT's NED language. Modules that contain submodules are termed compound modules, as opposed to simple modules which are at the lowest level of the module hierarchy. Simple modules contain the algorithms in the model. The user implements the simple modules in C++, using the OMNeT++ simulation class library.

Modules communicate by exchanging messages. In an actual simulation, messages can represent frames or packets in a computer network, jobs or customers in a queueing network or other types of mobile entities. The local simulation time of a module advances when the module receives a message. The message can arrive from another module or from the same module (self-messages are used to implement timers).

Gates are the input and output interfaces of modules; OMNeT++ supports only simplex (one-directional) connections, so there are input and output gates. Messages are sent out through output gates and arrive through input gates.

Due to the hierarchical structure of the model, messages typically travel through a series of connections, to start and arrive in simple modules. Such serieses of connections that go from simple module to simple module are called routes. Compound modules act as "cardboard boxes" in the model, transparently relaying messages between their inside and the outside world. Connections can be assigned three parameters, which facilitate the modeling of communication networks, but can be useful in other models too: propagation delay, bit error rate and data rate, all three being optional. One can specify link parameters individually for each connection, or define link types and use them throughout the whole model.

The simple modules of a model contain algorithms as C++ functions. The full flexibility and power of the programming language can be used, supported

by the OMNeT++ simulation class library. The simulation programmer can choose between event-driven and process-style description, and can freely use object-oriented concepts (inheritance, polymorphism etc.) and design patterns to extend the functionality of the simulator.

3.3 Basic Parts of an OMNeT++ Model

An OMNeT++ model physically consists of the following parts:

- NED language topology description(s)
- Message definitions
- Simple modules implementations and other C++ code

To build an executable simulation program, you first need to translate the NED files and the message files into C++, using the NED compiler (nedtool) and the message compiler (opp_msgc). NED files can also be loaded dynamically, in which case they don't need to be compiled beforehand. After this step, the process is the same as building any C/C++ program from source.

3.4 Interesting Features

As it was shown in [3] and [4], the cycle length of a random number generator (RNG) is fundamental, especially when RNGs are used for simulation purposes. OMNeT++ releases prior to 3.0 used a linear congruential generator (LCG) with a cycle length of $2^{31} - 2$. This RNG is still available but is only suitable for small-scale simulation studies. Newer OMNeT++ releases use by default the Mersenne Twister RNG (MT) by M. Matsumoto and T. Nishimura ([5]). MT has a period of $2^{19937} - 1$, and 623-dimensional equidistribution property is assured. MT is also very fast: as fast or faster than ANSI C's rand(). In addition, OMNeT++ allows to plug in own RNGs as well.

In many simulations, only the steady state performance (i.e. the performance after the system has reached a stable state) is of interest. The initial part of the simulation is called the transient period. After the model has entered steady state, simulation must proceed until enough statistical data has been collected to compute results with the required accuracy.

Detection of the end of the transient period and a certain result accuracy is supported by OMNeT++. The transient detection and result accuracy

objects will do the specific algorithms on the data fed into the result object and tell if the transient period is over or the result accuracy has been reached. The transient detection algorithm uses a sliding window approach with two windows, and checks the difference of the averages of the two windows to see if the transient period is over. The accuracy detection algorithm divides the standard deviation by the square of the number of samples and checks if this is within the accuracy range specified by the user. These algorithms were used for the experiments described in chapter 5.

3.5 Comparison with Other Simulators

Available Models Non-commercial simulation tools cannot compete with some commercial ones (especially OPNET) which have a large selection of ready-made protocol models. OMNeT++ is no exception, it clearly lacks models, also compared with non-commercial tools such as ns-2 (but it has to be considered that OMNeT++ is a rather new tool, it was originally released in 1999). On the other hand OMNeT++ provides a larger variety of models (that allows the user to simulate more than just communication networks) as compared to ns, which mainly provides TCP/IP centered models.

Model Management The OMNeT++ simulation kernel is a class library, i.e. models in OMNeT++ are independent of the simulation kernel. The user writes his components (simple modules) like using any other class library, and generates the executable by compiling and linking them against the simulation library. This means that there is no need to modify the OMNeT++ sources (this enforces reusability). ns-2 tends to be monolithic: to add new models to it, one needs to download the full source and modify it a bit, copy files to specific locations, add constants in other files etc.

Reliability As a matter of fact, models provided with simulation tools are often not validated. This also applies to OMNeT++. A good example is the TCP implementation in the INET framework for OMNeT++ (more on that later on). This is a general problem of non-commercial tools: anybody can contribute, but nobody gives any garanty. Moreover, some models are still under development and therefore represent simplified versions of what they are intended to model.

Network Topology Definition Network simulation tools naturally share the property that a model (network) consists of “nodes” (blocks, entities,

modules, etc.) connected by "links" (channels, connections, etc.). Many commercial simulators have graphical editors to define the network; however, this is only a good solution if there is an alternative form of topology description (e.g. text file) which allows one to generate the topology by program. On the other hand, most non-commercial simulation tools do not provide explicit support for topology description: one must program a "driver entity" which will boot the model by creating the necessary nodes and interconnecting them (e.g. in ns-2 the OTcl scripting language is used). Finally, a large part of the tools that do support explicit topology description support only flat topologies. OMNeT++ probably uses the most flexible method: it has a human-readable textual topology description format (the NED language) which is easy to create with any text-processing tool (perl, awk, etc.), and the same format is used by the graphical editor. It is also possible to create a "driver entity" to build a network at run-time by program. Moreover, OMNeT++ also supports submodule nesting without limitations on the depth of nesting.

Configuration of Simulation Runs Parameters of a simulation experiment are written in the omnetpp.ini, this strongly enforces the concept of separating the model from experiments. Models and experiments are usually seriously interwoven in ns-2: parameters are usually embedded in the Tcl script and thus are difficult to edit.

Debugging C++-based simulation tools rarely offer much more than the printf()-style debugging process; often the simulation kernel is also capable of dumping selected debug information on the standard output. OMNeT++ goes a different way by linking the GUI library with the debugging/tracing capability into the simulation executable. This architecture enables the GUI to be very powerful: every user-created object is visible (and modifiable) in the GUI via inspector windows and the user has tight control over the execution. To the author's best knowledge, the tracing feature OMNeT++ provides is unique among the C++-based simulation tools. In addition, this property makes OMNeT++ an excellent tool for demonstrational or educational purposes.

Performance Performance is a particularly interesting issue with OMNeT++ since the GUI debugging/tracing support involves some extra overhead in the simulation library. Simulating large networks (e.g. MQTT networks with hundreds of clients) results in unacceptable performance. But this is also a big problem with other popular simulators such as ns-2.

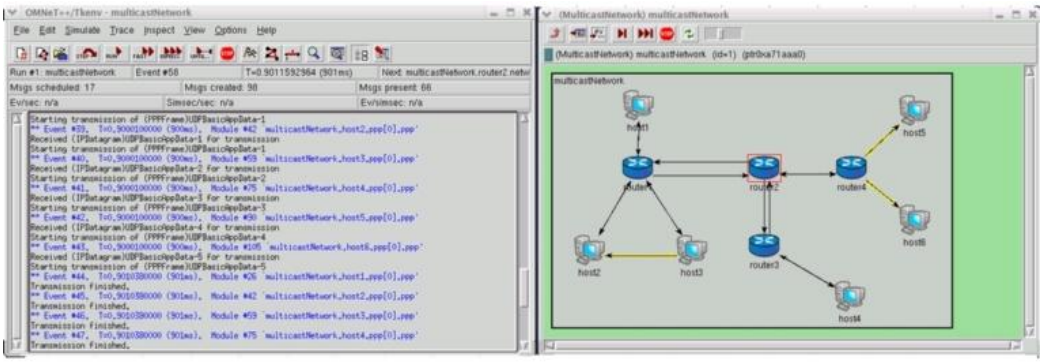

Figure 3.2: OMNeT++ Screenshot

Chapter 4

Protocols and Models Implemented for OMNeT++

This chapter contains a description of the protocols and models which had to be implemented for OMNeT++ in order to allow for using the simulator for performance tests of MQTT over TCP and wireless links. First, the MQTT implementation is described, followed by the process of extending OMNeT++ with a TCP stack. Finally, the implementation of an adequate model to simulate wireless channels is described.

4.1 MQTT Implementation

The fact that OMNeT++ is highly modular and well structured is a big advantage when it comes to implementing new protocols to be used in the simulator. The process of implementing MQTT for OMNeT++ was quite straightforward and convenient especially thanks to the NED language.

The implementation process can be summarized as developing two different compound modules: one representing a MQTT client and another one taking the role of a broker. Furthermore, the implementation of the MQTT client compound module can be subdivided into developing a MQTT publisher and a MQTT subscriber. Additionally, with the help of OMNeT's message description compiler, all the necessary message types were developed.

The MQTT client compound module is a conventional TCP/IP host extended with a simple module representing the MQTT layer on top of TCP and a MQTT application. The MQTT layer module contains the logic to perform the MQTT protocol with the broker. That is, it contains the functionality to connect to the broker, to publish or subscribe to topics depending

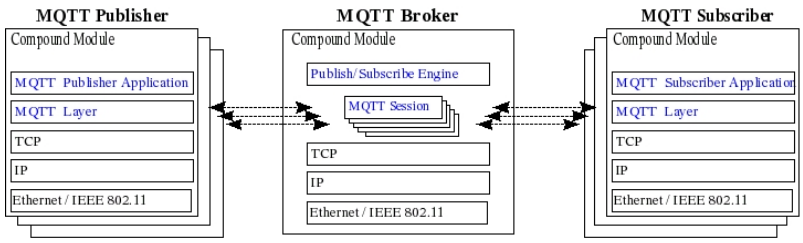

Figure 4.1: MQTT Implementation Design

on the application on top, and of course also to execute the ping process or to disconnect from the broker. Moreover, it has to be able to communicate with the application that may represent a publisher or a subscriber. To allow for this communication, an additional message type had to be introduced which would allow the MQTT layer to distinguish between MQTT messages received from the broker and messages received from the application.

On top of the MQTT protocol layer sits the MQTT application, which can be a publisher or a subscriber. The publishing application sends PUBLISHes to the MQTT layer via the special message type mentioned before. The subscriber application on the other hand contains the logic to tell the MQTT layer to which topics it wants to subscribe and at which QoS level. Moreover it is able to accept PUBLISH messages delivered by the MQTT protocol layer. In general, the two types of application modules do not differ much, hence to simplify matters they could also me merged to build one simple module representing a publisher, a subscriber or both.

The second compound module needed to make up a MQTT network is the broker. Again, this is a conventional TCP/IP host, but extended with a simple module implementing the broker functionality. Besides containing the logic to "talk" MQTT as the clients do, it needs the ability to match publications to subscriptions. This was implemented by adding an additional simple helper module which is instantiated for each client connecting to the broker. Thus, the broker associates such a helper module to each client, and this module is responsible for maintaining session information such as e.g. to which topics a client is subscribed to, when the last message was received from the client to be able to calculate the ping timeout expiration etc.

With the help of the NED language, we then define a MQTT network as a network consisting of a compound broker module, and several compound MQTT client modules, each being connected to the broker either explicitly when using a wired network, or implicitly in the case of a wireless network, where the clients just have to be in transmission range of the broker.

4.2 Integration of a TCP Stack

OMNeT++ as is does not come with any modules to simulate TCP/IP networks. This is provided by the INET framework. The INET framework is an open-source communication networks simulation package for the OMNeT++ simulation environment. It contains models for several Internet protocols: beyond TCP and IP there is UDP, Ethernet, PPP and others.

However, it has been shown in [6] that not only there are features missing in the TCP/IP models provided with the INET framework, but also the implementation has been proved not to work correctly. Simple tests carried out at the beginning of this project have reinforced the observations made in [6] and have shown that the implementation behaves abnormal when higher link error rates are introduced. Since we are particularly interested in MQTT's performance over TCP and lossy links (as we will be communicating over wireless links) it was impossible to use the TCP implementation shipped with the INET framework to perform the simulations.

There are basically two approaches as how to address this deficiency of the TCP implementation. On one hand, it is possible to take INET's TCP implementation and to try to fix possible errors found. However, this has two major disadvantages. First, it is probable to lead again to errors and malfunctioning due to newly introduced bugs or to errors not identified. Secondly, it is not an easy task to look through the code and compare it with a real implementation, especially not if the simulator's TCP stack was not based on a particular operating system (OS). After all, TCP is not a simple protocol.

On the other hand, it seems more reasonable to take an existing, validated TCP implementation and to make it fit into OMNeT's environment. Thereby, you reduce the chances to introduce new errors since you reuse code and avoid reimplementing a protocol based on a erroneous implementation. By reusing code, you base the implementation on another that has been proven to work correctly.

Clearly, the second approach seems to be the more reliable and simpler one. Hence, in the following the process of integrating an existing TCP implementation into OMNeT++ is shortly described.

4.2.1 Integration of the NetBSD TCP Stack

NetBSD [7] is a free and highly portable open source OS available for many platforms. Several protocol suites supported by this OS were inherited from BSD and subsequently enhanced and improved. A big advantage of NetBSD's TCP implementation is its proper structure that makes it easier to integrate

into OMNeT++.

To integrate the NetBSD TCP implementation into OMNeT++, the simplest and probably most efficient solution is to pack all the TCP functionality into one OMNeT++ module. This is presumably also the most safe approach, as the whole code is maintained together in one single module (which of course does not interdict to spread the code over several files).

To make the TCP stack available to the application, appropriate TCP sockets were implemented that let the application interact with the TCP stack via OMNeT++ messages. Moreover, appropriate in- and output gates were added to the simple module definition to not only allow for communication between the implemented TCP sockets and the TCP stack, but also to enable data passing between TCP and the IP layer beneath. The latter corresponds to the module provided by the INET framework.

4.2.1.1 Blocking Calls

It has already been identified in [8] that implementing blocking calls in OMNeT++ is not easily feasible. Since OMNeT++ is not a multithreaded environment, every function call that would let a process sleep or wait for a while would stop the simulation. This is due to the fact that the handleMessage() procedure, which every (non-coroutine based) OMNeT++ module implements and that is called whenever a message (event) arrives, has to finish in order to return control back to the OMNeT++ simulation kernel. This makes it nearly impossible to avoid changing applications in order to allow for blocking calls. Since in real life usually blocking socket calls are used, it is crucial to introduce somehow this functionality into our TCP stack.

The basic idea to simulate blocking calls is to have the TCP layer send back a message to the application to inform it of how much space there is in the socket buffer. Before sending any data through the socket, the application checks that TCP will have enough space to take the data. If this is the case, the data is just sent through the socket. However, if TCP would block because there is not enough free buffer space, the application stores the message to send in a FIFO queue until the TCP layer tells it to wake up. When the wake up signal arrives, the first message within the queue is dequeued and sent out. Whenever the queue is not empty and the application triggers a “send message event”, instead of directly checking if the socket would block, it verifies if there are any backlogged messages in the queue. In this case the new message has to be enqueued at the tail. In the other case of an empty queue, the application checks if there is enough buffer space for the message.

Clearly, this is an unusual and cumbersome way to simulate blocking

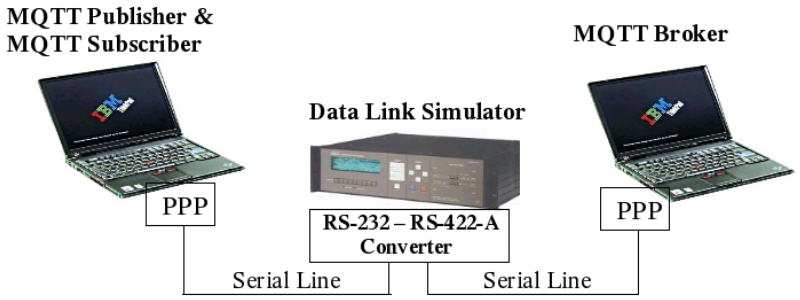

Figure 4.2: Lab setup to validate the TCP implementation integrated into OMNeT++.

calls, as it is the application that has to care about the blocking mechanism. But to the best of our knowledge there is no simpler way to introduce the blocking calls functionality into OMNeT++.

4.2.2 Validation with Real TCP

To validate up to a certain level that the newly integrated TCP stack works correctly, some tests were performed to compare it with a real TCP environment. The test scenario is depicted in figure 4.2.

To perform the tests, the SX/13a data link simulator was used, which simulates terrestrial and satellite data links for testing internetworking equipment and applications under repeatable and controllable conditions. It allows for bi-directional testing with programmable delays, random bit errors and burst errors.

The data link simulator used has two RS-422-A interfaces. By using a RS-232 to RS-422-A converter, two IBM Thinkpads were connected to it (see figure). One Thinkpad took the role of a MQTT broker, the other machine was used to run a publisher and a subscriber application. To communicate over the serial links, the Point-to-Point protocol (PPP) was used with a data rate of 115Kb/s. The tests were run with a negligible fix propagation delay of 0.1ms and random errors. Several payload sizes and data rates were tested to have some more confidence in the implementation. Figures 4.3 and 4.4 show some of the performed validations for the mean end-to-end delay measured in the laboratory setup and with OMNeT++, respectively. The shapes of the graphs are pretty similar. Clearly, the results do not match completely, but this is also comprehensible. It has to be noted that factors such as processing delays at different layers, socket read and write operations etc. are

not considered in the simulator. Moreover, in real life there are many factors that can slighty vary over time but which are constant in the simulation.

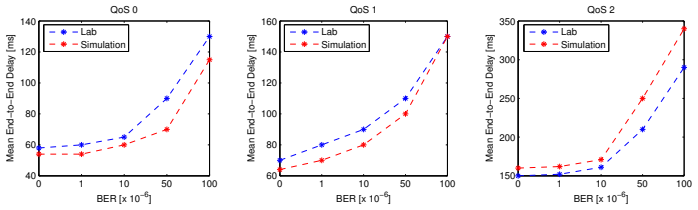

Figure 4.3: Comparison of the measured mean end-to-end delay of the lab setup and the OMNeT++ simulations. 10 messages per second with 10B MQTT payload.

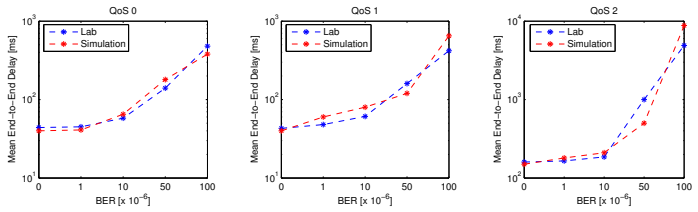

Figure 4.4: 50 messages per second with 10B payload.

4.3 Wireless Channel Model

A shortcoming of OMNeT++ is the limited number of available simulation models for different network protocols and technologies. Especially in the area of wireless networks there is still a lot of development necessary to make OMNeT++ a more powerful simulator. The latest INET framework release contains support for wireless network simulations. However, among other things, OMNeT++ still lacks models to simulate wireless channel properties.

Wireless channels differ a lot from wired channels, due to their unreliable behavior. The state of a wireless channel may change within very short time spans. On the other hand, wireless communication is known for its correlated error characteristics which lead to error bursts ([9],[10]). Therefore, we extended OMNeT++ by a simple though powerful model that enables the user to model wireless channel properties.

A very often referenced bit-level wireless channel model is the so called Gilbert or Gilbert-Elliot model ([11], [12]). It is a first-order Markov model which assumes a good and a bad channel state. Within every state, bit errors occur according to the independent model with rates e_G and e_B, respectively, with $e_G << e_B$. The bit error rates in general depend on the frequency and coding scheme used and on environmental conditions. The statistics of the bit errors are then fully characterized by the transition matrix

$$\mathbf{P} = \begin{pmatrix} p_{BB} & p_{BG} \\ p_{GB} & p_{GG} \end{pmatrix}$$

where p_{BG} is the transition probability from the bad to the good state, and the other entries in the matrix are defined analogously. The steady-state probabilities to be in a certain state are given by

$$p_G = \frac{1 - p_{BB}}{2 - (p_{GG} + p_{BB})} \qquad p_B = \frac{1 - p_{GG}}{2 - (p_{GG} + p_{BB})}$$

and the mean bit error rate is given as $\overline{e} = p_G e_G + p_B e_B$. The mean state holding times can be computed as

$$\frac{1}{1 - p_{GG}} \qquad \frac{1}{1 - p_{BB}}$$

for the good state and the bad state respectively. The Gilbert-Elliot model has short-term correlation properties for bit errors, but burst length sequences are uncorrelated.

For our purposes, this model was simple enough to implement and is sufficient to simulate wireless channels with a certain burstiness. Other algorithms found in literature, e.g. [13] may be more accurate due to their

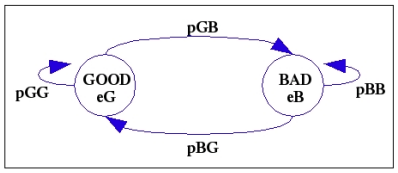

Figure 4.5: Gilbert-Elliot Channel Model

trace-based approach, however they need to be fed with a trace having the desired characteristics. Also, more accurate models mostly imply that the number of parameters to be computed is much larger.

Chapter 5

MQTT Performance Analysis

Up until now, no performance analysis of the MQTT protocol has been made, although there are various parameters that one can vary and which will have an impact on the systems behavior. This chapter discusses some first performance measurements recorded with OMNeT++ enhanced with the integrated NetBSD TCP stack. The goal is to test the protocol's behavior under different network setups and to show how and why chosen parameter settings influence the protocols performance.

In the first section, the parameters and options used and varied will be listed. Also, the performance measures of interest are discussed. Then, the findings for the simulated wired networks are presented, followed by the probably most interesting and realistic case of wireless networks using IEEE 802.11.

5.1 Parameters and Performance Measures

A host using TCP as transport protocol has usually a set of parameters and options that it can alter or use. Depending on the TCP implementation/operating system, it may be e.g. possible to use special ack mechanisms such as SACK, or to use the timestamp option etc. Having additionally on top of TCP MQTT that allows applications to choose some parameters according to their needs (e.g. the QoS) increases the "degrees of freedom". On the other hand, you also have the choice between several data link layer and especially MAC protocols such as Ethernet and IEEE 802.11. Next, OMNeT++ provides the necessary means to define the physical layer properties such as the bandwidth. This shows us that there are basically a lot of parameters for different OSI layers that can be set to appropriate values. Hence, it is important to first specify which options the simulator provides and which

of them will be used. Each of the following subsections discusses one layer and the possible modifiable parameters. The final subsection defines the performance measures of interest.

5.1.1 MQTT Parameters

The most interesting parameter from a MQTT application point of view is the QoS which can take one of three possible values: 0 (at most once semantics), 1 (at least once semantics) and 2 (exactly once semantics). The additional messages involved in QoS greater than 0, i.e. the acknowledgments at MQTT layer and the potential retransmissions after timeouts will cause extra load and influence thereby the performance of the system.

Since the MQTT protocol does not restrict the payload size of PUBLISH messages, an application is free to choose the amount of data to be published (note that the topic name is also counted as payload from the point of view of a MQTT application). This parameter again directly affects the data laod produced on layers beneath and on the system.

Moreover, a publisher may choose freely at which rate it wants to send out data. Regardless of the fact that there is no limit on the payload amount published, we should rather consider increasing the sending rate than the payload size since this will more likely correspond to publish/subscribe and sensor networks infrastructures. Considering for instance the case of a sensor reporting the current temperature of a location, the data would consist of only a few bytes.

Not only the sending rate, but also the arrival distribution of PUBLISH messages is of interest. Two very typical scenarios would be to have sensors that send whenever an event occurs (e.g. a sensor measuring the temperature detects a change), which could be modeled with a poisson distribution (exponentially distributed interarrival times). On the other hand it is also possible that a sensor reports status at fix points in time, e.g. every ten seconds. This scenario would be modeled with a constant sending rate. We will concentrate our attention on the poisson distributed arrival process, since this introduces some variance which may result in more interesting impacts.

Even if it cannot directly be considered as a MQTT parameter, the number of MQTT clients can obviously also be varied. In sections 5.2 and 5.3 we will first start with a small network and increase the number of clients afterwards.

Table 5.1 lists the parameters of interest mentioned above.

5.1.2 TCP Parameters and Options

Depending on the operating system used, the TCP implementation offers various options to be tuned to the users need. A setting that has an important impact on TCP's performance is the send and receive buffer sizes/window sizes. Most common OS use send and receive buffers of 16KB or 32KB per default. The user is subsequently allowed to change this value to his needs. However, the TCP specification limits the maximum window size to 64KB (since the window field in the TCP header is limited to 16 bits). Thus, some implementations provide the window scale option specified in [15], which allows to scale the window size to higher values. This is achieved by specifying the scale in an additional option field in the TCP header of a SYN message. This option is also available in our OMNeT++ TCP implementation.

[15] also suggests an other option to better estimate the round-trip time RTT of a connection. This timestamp option which consists of three bytes, is a countermeasure to the fact that many TCP implementations base their RTT measurements on a sample of only one packet per window. The timestamp option proposes the sender to place a timestamp in each data segment and the receiver to reflect the timestamp in the ACK. The sender can then more accurately calculate the RTT based on the timestamp.

The default maximum segment size MSS of TCP is 536, which is based on IP's default maximum datagram size of 576. Changing this (and accordingly IP's maximum datagram size) value can affect the loss probability of a segment. Moreover, for application messages larger than the MSS the fragmentation may cause higher end-to-end delays as the application data is split into different segments.

Unfortunately, an important ACK related option, the so called selective acknowledgment (SACK) mechanism described in [16] is not supported by our version of NetBSD (the SACK option was just recently added to the NetBSD stack). SACKs would inform the sender of data that has been received as opposed to cumulative ACKs where the sender is only informed which sequence number the receiver expects next.

Finally, for all simulations, the NewReno TCP version ([18]) was used. In addition to the fast retransmission algorithm used in TCP Tahoe [19] (when the sender receives three duplicate ACKs before the timeout expires the packet is considered as lost and retransmitted), TCP Reno [19] makes use of the fast recovery algorithm. It prevents the sender to slow start after executing fast retransmission. Instead, when the third duplicate ACK is received, the window threshold size ssthreshold is set to half the last seccussful congestion window (cwnd) and the cwnd is set to ssthresh + 3 (instead of 1). Each time a duplicate ACK is received the cwnd is "artificially inflated"

by one and a new segment is transmitted if possible. Once an ACK arrives that acknowledges new data, the cwnd is set to ssthresh. In the case of NewReno, fast retransmit and congestion window adaptation are as in Reno, but the loss recovery mechanism is as in Tahoe. That is, NewReno will not send the first lost packet alone and wait for its ACK like Reno. Instead it will continue with the transmission of subsequent packets like Tahoe. This allows NewReno to recover in some cases from multiple packet losses within a window (as opposed to Reno).

5.1.3 Data Link Layer Settings

Knowing that we want to test MQTT's performance under wired and wireless networks, we need adequate MAC protocols for each type of network. For a wired setup, we will use Ethernet as it best represents today's state of art. Regarding wireless networks, the INET framework (see section 4.2) provides an implementation of IEEE's 802.11 MAC layer. Also supported is the RTS/CTS mechanism that can optionally be enabled or disabled.

5.1.4 Physical Layer Settings

For the physical layer the set of parameters depends on the protocol used at the layer above. When using Ethernet, we have to choose the bandwidth of the channel, the delay and the bit error rate. Thanks to the implemented Gilbert-Elliot channel model, we can also define the error burstiness of the channel (see section 4.3). Although originally designed to model wireless channel properties, we can use this model also to introduce error burstiness in a wired network.

When simulating a wireless network, we have again the choice to set the bandwidth to our needs. For additional random errors besides collisions, we will have to use an adapted Gilbert-Elliot model which allows to set the state holding times, i.e. how long we stay in each (good or bad) state. This modification will be explained in section 5.3.2.

Actually, we have some more parameters that might be varied, including the carrier frequency, the path loss coefficient, the transmission power and the sensitivity of the WLAN cards. Nevertheless, these parameters were set to reasonable fixed values for all experiments.

Table 5.1 summarizes all the available parameters mentioned in this chapter.

5.1.5 Performance Measures

The performance of a protocol can be defined in many ways. But depending on the simulator used and on the network you would like to simulate, some things may or may not be possible to be measured. For our needs, we basically would like to test MQTT's performance in three dimensions: end-to-end delay, power consumption and where appropriate effective throughput.

The end-to-end delay measures the time it takes a PUBLISH message sent by a MQTT publisher to be received by a MQTT subscriber. For this, the publishing application puts a timestamp in the message in order to allow the receiving application to calculate by a simple subtraction how long it took the PUBLISH message to traverse the network. Please note that in OMNeT++ you can make use of a timestamp without increasing the message's size.

For wireless communication there is an extra measure which should be considered: power consumption. This measure is very important especially for small footprint devices such as sensors. In [21] the power consumption for several components of a Mica2 sensor have been measured. While the radio consumed 7mA in receiving state, the current consumed when transmitting varied between 4 and 20mA depending on the transmission power level used. At the same time, the CPU consumed only 8mA when being active. Other components tested were the EEPROM (6mA for read operations, 18mA for write operations) or the LED (2mA). This clearly shows that the power consumed by the radio makes up a very important part of the overall power consumption of a small device. Since on-going research aims at integrating MQTT with exactly this kind of "restricted" devices, it is crucial to have results indicating how good the protocol performs with respect to this aspect. Now, since we are using virtual networks where no real wireless communication is involved, it is difficult to measure the power consumed by a host. Though, it is possible to record other related statistics that, when interpreted correctly, can give an indication on how power efficient the protocol is. The main idea is to measure the number of transmitted and received packets and bytes at the MAC layer to have an estimate of the power consumed. The measurement at packet granularity would indicate how many times the radio interface switched from idle to active state. At byte granularity, we get an estimate of how long the radio has been active. Furthermore, the recorded statistics can be very useful when compared to the number of sent and received messages and bytes at the application layer. From this kind of comparison, you can deduce an estimate for the efficiency of the protocol. For instance, sending out one packet at the MAC layer per PUBLISH message sent by the application would be acceptable. Having five packets sent out per PUBLISH message is on the other hand less efficient

and may be caused by packet losses due to bit errors, retransmissions at e.g. the TCP layer etc. And since this indicator is not only useful for estimating the power consumption but also to have an idea of the efficiency of the system, we will consider this measure also for wired networks. This will also allow comparisons between running the protocol in a wired and in a wireless environment. For the rest of this document b-efficiency for the publisher will refer to the ratio sent bytes at the application versus bytes sent (and received) at the MAC layer. For the subscriber b-efficiency denotes the ratio bytes received at the application versus bytes received (and sent) at the MAC layer. Similarly, we will use the term m-efficiency for the sending/receiving ratios of application messages to frames at the MAC layer.

The effective throughput on the other hand is defined as the application data per unit of time that a MQTT publisher was able to transmit. As data we count the publish topic and the corresponding payload published to that topic, plus one additional byte for the QoS (for the MQTT protocol layer has to know with which QoS level the PUBLISH should be delivered).

For all experiments, the convergence detection algorithms provided by OMNeT++ (see section 3.4) were used and applied to the end-to-end delay. If no convergence up to 1% accuracy was detected, the simulation was stopped as soon as each subscriber had received 50000 MQTT messages.

Table 5.1: Simulation Parameters

Protocol/Layer	Parameter/Option
MQTT	QoS
	Payload Size
	Publisher's Sending Rate
	Number of MQTT Clients
TCP	Buffer/Window Sizes
	Window Scale Option
	Timestamp Option
	Maximum Segment Size (MSS)
MAC Layer	Ethernet MAC
	IEEE 802.11 MAC
Physical Layer	Bandwidth
	Propagation Delay
	Error Rate
	Error Burstiness

5.2 Wired Network

Although MQTT is a lightweight protocol mainly for remote sensors and control devices through low bandwidth communications, it is certainly interesting to see the protocol's behavior under other conditions, too. Since MQTT over TCP/IP will certainly behave differently depending on the underlying data link and physical layer properties, it is important to compare results obtained under different network assumptions. This will allow us to see how different settings influence MQTT's behavior and the results may be considered for future applications of MQTT. Nevertheless, the focus should be on wireless networks, since they represent the main application area of MQTT. Moreover, we are particularly interested in lossy links with error bursts.

5.2.1 Preliminary Observations with a Small Ethernet Network

In this section we will discuss some basic observations we make when assuming a simple Ethernet network of only two clients besides the broker. This is certainly not a realistic setup, since in practice you will usually have several tens or hundreds of clients per broker. However, this will give us a first insight into MQTT's and TCP's behavior depending on the paramater values chosen. It will also serve to choose some default values for certain parameters which will assure stability of the system.

The MQTT payload length is set to 30B (includes the topic name, the publish data and 1B for the QoS) and the interarrival time of PUBLISH messages is exponentially distributed with parameter $\lambda = 5$ms (corresponds to a mean sending rate of 200 messages per second). The MQTT timeout value for QoS > 0 is set to 10 seconds per default.

Regarding TCP, we set the buffer sizes to the default value of 32*MSS with MSS equal to 2^{10}B. All options mentioned in section 5.1.2 are disabled.

The transmission links have a bandwidth of 100Mb/s and a propagation delay of 1ms. The probability pBB of the Gilbert-Elliot channel model is set to 0.1 which corresponds to a 10% probability of staying in the bad state. This can clearly be considered as a non-bursty error distribution.

Table 5.2 shows some basic numbers regarding the efficiency of MQTT over TCP/IP for a bit error rate (BER) of 10^{-5}. It is clear that the higher the QoS level is, the lower the ratio of PUBLISH messages sent to total number of frames sent (by the publisher) will be. Figure 5.1 illustrates this property. With QoS 0, the ratio will only significantly drop below 1 when there are many retransmissions at the TCP layer, which is not the case for

Table 5.2: Small Ethernet network: Efficiency for different QoS.

	QoS 0	QoS 1	QoS 2
Publisher M-Efficiency	.99	.49	.25
Publisher B-Efficiency	.33	.19	.10
Subscriber M-Efficiency	.99	.49	.24
Subscriber B-Efficiency	.33	.21	.11

this experiment (the m-efficiency is slightly below 1 because of the initial TCP and MQTT connection establishment packets, running the simulation for a longer time would yield a ratio of 1). Increasing the QoS to 1 will inevitably bring down the ratio to below 50% as the publisher has to send back a TCP ACK for the received PUBACK. Analogously, with QoS 2 we will have to send a TCP ACK for both the PUBREC and the PUBCOMP received, therefore the ratio is supposed to fall below 25%. It is also not surprising that the b-efficiency (for the publisher this is the ratio of bytes sent by the application to overall number of bytes sent at the MAC layer, and for the subscriber analogously for received application bytes and received bytes at the MAC layer) is much lower with higher QoS.

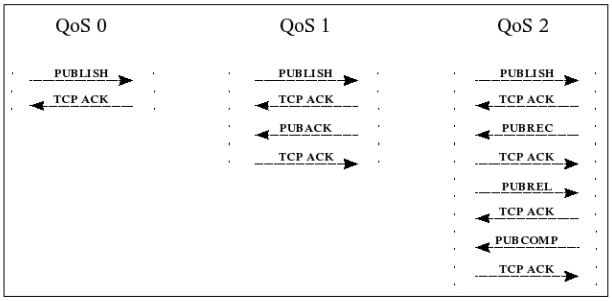

Figure 5.1: Overhead caused by higher QoS levels.

Doing the same simulations with higher application payload sizes highlights further facts. It is reasonable that increasing the payload size will lead to better b-efficiency. This is due to TCP ACKs and MQTT ACKs (PUBREC, PUBREL and PUBCOMP) still having the same size independently of the size of the PUBLISH messages. Therefore the publisher will experience a rise in the number of bytes sent that is more significant at the application layer than it is at the MAC layer. Of course, this is under the assumption that the increased payload size does not significantly affect the packet loss

rate and thereby the number of TCP retransmissions. According to the simulation results, this is clearly the case. An ideal indicator for this measure is the m-efficiency. If it stays about the same for different payload sizes, we can deduce that TCP's behavior was not importantly influenced.

However, the increase of the b-efficiency is not linear to the increase of the payload size (e.g. for QoS 0 the publisher's b-efficiency rises by 12% when the payload is incresed from 100B to 250B, but only by 6% from 250B to 500B). This is because increasing the payload and thereby also the packet size results in a higher probability of the packet being lost. This will consequently lead to more bytes being retransmitted, and thus the increase of the b-efficiency will be slowed down.

Note also the symmetry between the publisher's and the subscriber's m- and b-efficiency. Taking again a look at figure 5.1 shows that this is not surprising. Nevertheless, please note that the publisher's m- and b-efficiency refer to sent messages or sent bytes ratio, whereas the subscriber's m- and b-efficiency measure the received messages or received bytes ratio.

Table 5.3 summarizes the observations explained in this section for a BER of 10^{-4}.

Table 5.3: Small Ethernet network: Influence of the payload size on the efficiency.

	QoS 0			QoS 1			QoS 2		
	100B	250B	500B	100B	250B	500B	100B	250B	500B
Publisher M-Efficiency	.98	.97	.95	.49	.50	.49	.24	.25	.26
Publisher B-Efficiency	.56	.68	.74	.41	.58	.68	.27	.44	.58
Subscriber M-Efficiency	.97	.98	.98	.49	.50	.53	.24	.25	.28
Subscriber B-Efficiency	.57	.69	.75	.41	.58	.68	.27	.46	.59

5.2.1.1 Effect of Error Correlation

TCP was originally designed for an environment where packets are lost mostly due to congestion, and therefore the control algorithms embedded therein act accordingly. When a connection extends over wireless links, packet losses occur primarily due to channel errors or during handoff. Therefore, it is important to understand the effect of channel errors. Wireless channels are known to introduce correlated bursts of errors at the physical layer ([9], [10]). In this chapter we would like to test MQTT's behavior over TCP/IP and Ethernet while having a channel with similar characteristics to a wireless environment.

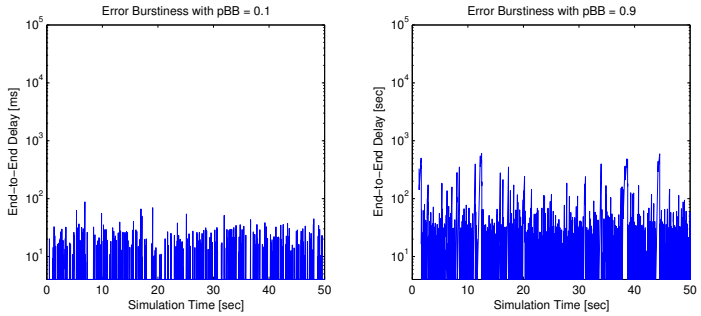

Figure 5.2: Impact of the error distribution on the end-to-end delay for a BER of 10^{-4}, 30B payload and QoS 0.

As a first approach to see the impact of error burstiness on the system's performance, we record the end-to-end delay of PUBLISH messages (time it takes a PUBLISH message to get from the publisher to the subscriber). Let's take a simple setup with just 30 bytes of payload published with QoS 0 (the other parameters are set as in the previous section). Figure 5.2 shows the recorded statistics for a BER of 10^{-4}. A clear rise of the end-to-end delay can be detected. More precisely, the mean end-to-end delay increased from 55ms to over 570ms, hence a difference of an order of magnitude. Note also that the distribution contains much more variance with high peaks. This makes clear how deep the impact of the bit error distribution on the performance can be. Doing the same experiments with higher QoS leads to the same observation: one order of magnitude increase of the mean end-to-end delay. The explanation is that back-to-back losses caused by the burstiness of the channel lead to an exponential increase of TCP's RTO. This not only implies higher end-to-end delays, but also more variance.

Observing an effective throughput of around 48Kb/s at the publishing application unambiguously shows that the performance loss is not due to congestion of TCP buffers (48Kb/s results from a sending rate of 200 messages per second with 30 bytes per message).

Considering that with a bit error rate of 10^{-5} no significant end-to-end delay increase was observed, it may be interesting to see how the mean end-to-end delay evolves for a BER in the range between 10^{-5} and 10^{-4}. Figure 5.3 plots the evolution of the mean end-to-end delay for different MQTT QoS levels and 30B payload.

The shapes of the curves for QoS 0 and 1 are the same, which is reasonable as QoS 1 does not delay messages from the point of view of MQTT. QoS 1 does only effect MQTT's return path, on which an additional message (PUBACK) is sent. The curve of QoS 2 indicates about 10ms higher mean

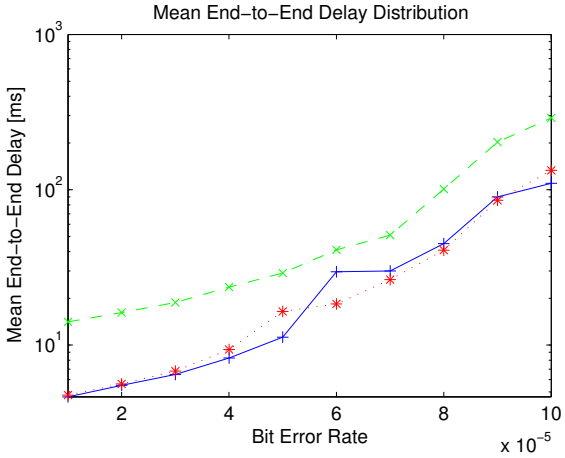

Figure 5.3: Mean end-to-end delay evolution for different QoS and 30B payload.

end-to-end delays, which is due to the additional protocol steps to ensure exactly once semantics.

Nevertheless, we will concentrate in the following on the upper "limit" of a BER of 10^{-4}.

Given the preliminary results from above, we may expect the performance to be worse when increasing the payload size. For the moment, we will increase the payload size to 65B (i.e. we double more or less the payload size) and observe the corresponding behavior. Please note that we are just doubling the payload size, not the overall frame size which includes the MQTT, TCP/IP and Ethernet headers (∼60B overhead). Thus, theoretically the packet loss probability for a PUBLISH message is increased by "only" 3% approximately.

Interestingly, by this 35B payload increment we get astonishing results. Plotting the end-to-end delay of the first 150 simulated seconds basically reveals a tremendous increase of the mean end-to-end delay for all QoS levels.

The distribution of the end-to-end delay shows that the end-to-end delay keeps increasing with simulation time advancing. Looking at the tcp socket send and receive buffers reveals that the publisher's send buffer is filled up nearly from the beginning. Hence, the packet losses produced by the error channel completely break TCP's throughput which would back pressure the application. But since the application produces load regardless of the effective throughput that it experiences, more and more application messages will be queued up, i.e. once the send buffer is full, following socket send calls will be blocked. Since TCP assumes that packets get lost because of network

congestion, the exponential backoff mechanism will be triggered. However, this is unlikely to help, for the mean channel BER will actually be around the same in the future.

As a matter of fact, only the publisher's send buffer will actually be congested, but not the broker's one which is dedicated to sending data to the subscriber. This can be easily explained if we consider that the publisher's TCP layer will send at the rate that the advertised window by the broker's TCP layer allows it to. Therefore, the broker will receive MQTT messages at the rate the publisher was restricted to, and hence the sending rate of the broker towards the subscriber will be lower or equal to the receiving rate of the broker (which is significantly lower than the publisher's sending rate because of the publisher's congested buffers).

Assuming that we cannot change the properties of the channel, i.e. it is given that the channel exhibits error correlations in the form of bursty errors, we would like to find a way to get more promising results. Let's also assume that we cannot change the application's behavior. More precisely, consider a real life situation where a sensor reports some status information whenever an event happens, and the events happen at an average rate of 200 events per second. We would not like to reduce this rate nor the size of the information data if it can be avoided.

As a first approach, one could suggest to increase the send and receive buffers either beyond the default values, or even beyond TCP's maximum allowable value of 64KB by using the window scale option. But essentially this will not give better results. The reason why we see bad performance is not because the application is not provided with large enough buffers, but because TCP's throughput decays as a response to overall network properties. Simulations carried out underline this basic fact.

The simplest solution to a bad performance mainly caused by an error-prone channel is basically to try reducing the direct effect of bit errors. If we have a given bit error rate, the packet error rate still can vary depending on the parameter settings. The packet loss rate depends on the packet sizes and more specifically, the probability of a packet getting lost (i.e. to contain a bit error) depends on its size. Therefore, varying TCP's maximum segment size MSS can have important effects. We therefore consider reducing the MSS from 1024B as set by default, to 536 as suggested by [17]. One should be aware that in most TCP implementations the socket buffer sizes are declared as multiples of the MSS, thus when varying the MSS it should not be forgotten to adjust accordingly the buffer sizes.

Table 5.4 shows the mean end-to-end delay distribution for several BER. It can clearly be seen that the impact of a lower MSS is huge for high BER. For lower BER, we get approximately the same results. However, comparing

the efficiency in terms of application bytes sent to bytes sent at the MAC layer is unnecessary since we know that we have a very bad performance in the case of a MSS of 1024B.

Table 5.4: Impact of the MSS on the mean and the standard deviation of the end-to-end delay [ms] for 65B payload and a bursty channel.

BER	QoS 0				QoS 1				QoS 2			
	536B		1024B		536B		1024B		536B		1024B	
	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
0	4.0	0	4.0	0	4.0	0	4.0	0	12.0	0	12.0	0
10^{-6}	4.1	1	4.2	1	4.1	1	4.1	1	12.3	1	12.2	1.2
10^{-5}	4.8	4	4.9	4	5.2	6	5.2	6	14.3	9	14.4	10
10^{-4}	150	170	inf	inf	159	190	inf	inf	750	920	inf	inf

Note that the mean end-to-end delay fo QoS 0 and QoS 1 does not differ much, independent of the BER. However, while for low BER the mean end-to-end delay for QoS 2 is three times higher than for lower QoS (caused by the 4-way handshake of QoS 2), with high error rates we have a factor of five (with a MSS of 536B).

The timestamp option suggested in [15] and supported by the NetBSD TCP stack is worth to be considered in more detail. The main problem is that many TCP implementations (including NetBSD) base the RTT measurements on one packet sample per window. Hence, the data sent out with a certain packet rate is sampled only at the windows rate. Neverteless, a good RTT estimator with a conservative retransmission timeout calculation can significantly improve performance. This is especially true when the number of dropped packets increases. Since Karn's algorithm prevents the RTT from being computed on retransmitted segments, at least a window's worth of time has to be waited before a new RTT measurement can be computed.

Figure 5.4 shows the impact of activating the timestamp option on the computed retransmission timeout (RTO) of TCP. The traces were collected at the publisher. First, we can see that the RTO's with timestamp option disabled are much higher than with enabled option. Also, we can detect a higher sampling rate when the option is on. An interesting fact is that with higher QoS we get lower RTOs than with QoS 0 when not using the timestamp option. For QoS 1 this could be explained by the fact that the PUBACK (in this case sent back by the broker to the publisher) can also serve as an ACK (piggybacking). If the original TCP ACK was lost (we have a BER of 10^{-4}) there is still the PUBACK which can act as a TCP ACK. In the case of QoS 2 we also have this effect, but additionally the publisher (in this case) has to send PUBRELs, which are smaller than PUBLISH messages

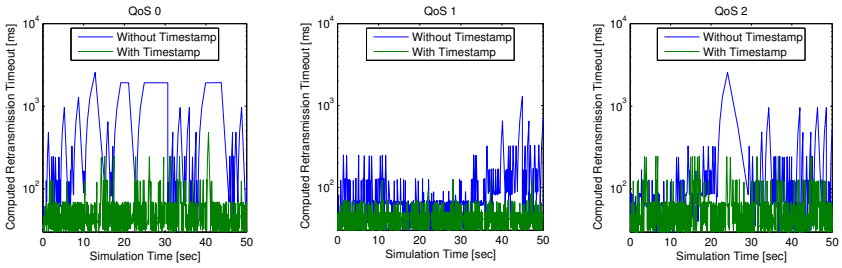

Figure 5.4: Computed TCP RTO with and without timestamp option for 65B of payload and BER = 10^{-4}.

(depending on the payload) and hence have a bigger chance to get through.

Please note that NetBSD's TCP implementation contains a parameter which, for algorithmic stability, sets the minimal allowable retransmission timeout (TCPTV_MIN). This value is set to 1 second per default. However, this has been proven to be bad for networks with bandwidths higher than a modem. Thus, in FreeBSD's TCP implementation the default value was corrected to 30ms, which is also the value we have chosen (and can be seen in figure 5.4).

Based on the results in this section, we will use a MSS of 536B and enable the timestamp option if not specified explicitly.

5.2.2 Several Publishers and Subscribers

After shortly having discussed the main issues and options, in this section the results of simulations carried out for a network of 20 MQTT clients, half of them being publishers and the other half subscribers, are discussed. To introduce some variance, different kinds of publishers and subscribers are present. Basically, we subdivide the publishers into three sets of similar size. Those publishing to let's say topic X, another set of publishers sending data to topic Y and a third group informing the subscribers about topic Z. Analogously, we have three different sets of subscribers: some subscribers being subscribed to all topics, i.e. to topics X, Y and Z, others receiving PUBLISHes to two of the three topics and finally a group of subscribers only being subscribed to one topic. Among other things we would like to see if and how the number of subscribed topics influences the performance. The data rate (100Mb/s) and the propagation delay (1ms) are kept.

Since Ethernet provides a high data rate, the focus will be on high publisher sending rates of 40 messages per second and per publisher. The downside is that we are restricted to small payloads, since otherwise TCP's buffers will not be able to handle the load. We will see in section 5.3.4 that in a

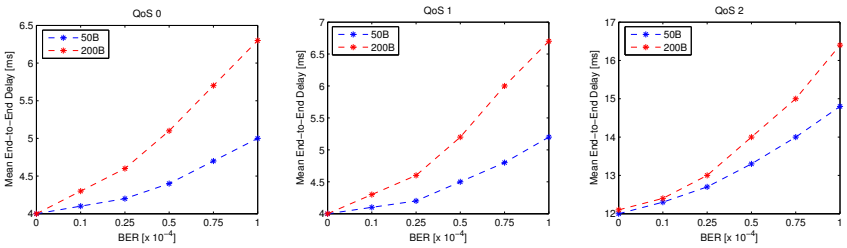

Figure 5.5: Mean end-to-end delay for pBB = 0.1.

wireless network the sending rate will have to be reduced to low values, and hence we will investigate on large payload sizes there.

The preliminary results from the previous section have shown that for small payload sizes, a BER below 10^{-5} does not significantly influence the performance measures. This makes perfectly sense since having e.g. 100B MQTT payload will result in Ethernet frames of around 175B including the MQTT, TCP/IP and Ethernet headers. For such a frame the probability to contain a bit error is only around 1%, hence very small. On the other hand, if we increase the BER up to 10^{-3}, on average each frame containing 100B MQTT payload will be lost. So the interesting BER range for small sized MQTT messages is between 10^{-5} and 10^{-4} and thus, we will use BERs in this range for the following simulations.

It can be debated what a small size for a MQTT messages is. However, considering a MSS of 536B and that we expect the MQTT clients to be small footprint devices, payload sizes up to 200B may be a reasonable assumption. Also, the additional bytes added through the headers of each layer will significantly increase the packet size.

End-to-End Delay

Figure 5.5 summarizes the mean end-to-end delays obtained for a mean bad state sojourn time of one bit duration (i.e. pBB = 0.1, see section 4.3). No interesting observations can be made, as the effect of the BER is negligible. Moreover, the influence of the payload size does not affect the results, neither. QoS 0 and 1 basically show the same statistical values. Only QoS 2 shows some bias due to the MQTT 4-way handshake which delays the MQTT PUBLISH messages by a factor of three.

Considering a PUBLISH message with 50B payload and 75B headers, with a BER of 10^{-4} we expect a 10% packet loss rate for those kind of messages. The figures indicate how insensitive TCP is to these values, on average the delay increases by only 3ms when introducing a BER of 10^{-4}.

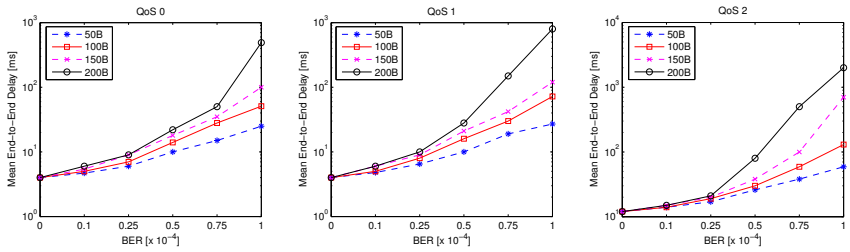

Figure 5.6: Mean end-to-end delay for pBB = 0.8.

As stated above, we would like to see the effect of lossy links with bursty error characteristics (as it is the case in wireless communications). Hence, by increasing the pBB probability of the Gilbert-Elliot model we get higher error burst lengths. When we increase the bad state sojourn time to 5 bit durations (pBB = 0.8), we actually should expect to see different results. This is shown in figure 5.6. The end-to-end delay is now very sensitive to both, the BER and the payload size. Again, we mostly see a match of QoS 0 and 1, while QoS 2 is a case for itself. To better see how the values evolve table 5.5 indicates the mean and standard deviation of the end-to-end delays for 50 and 200B.

Table 5.5: Mean and standard deviation of the end-to-end delay [ms] for pBB = 0.8.

BER	QoS 0				QoS 1				QoS 2			
	50B		200B		50B		200B		50B		200B	
	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.	Mean	Std.
0	4	0	4	0	4	0	4	0	12	0	12	0
10^{-5}	5	4	6	7	5	4	6	7	14	7	15	9
10^{-4}	25	46	485	400	27	43	574	412	59	67	2000	1000

It is clearly shown that the effect of both the BER and the payload size is very significant. However, we have taken the averages over all subscribers. Looking at the figures and the table mentioned above, it looks like the back-to-back losses introduced through the error burstiness and the resulting exponentially increasing TCP RTO's have led to congested buffers. But actually we don't have congestion between the broker and the publishers, but between the broker and some subscribers. It is worth noting that we have ten publishers sending 40 messages per second each. Assuming a client subscribed to all available topics, it will be served by the broker with 400 messages per socond. We saw in figure 5.5 that with low burstiness this

does not cause any problems, but now that we have introduced longer error bursts, we see the effects especially when increasing the payload size or the BER.

In figure 5.7 we compare the end-to-end distribution of three different subscribers: one is subscribed to all topics, one to two and another only to one topic. Obviously the number of topics plays an important role with regard to the end-to-end delay. This is due to fact that depending on the number of subscribed topics, the broker will have to send different amounts of data to a subscriber. For the client subscribed to only one topic (right plot in figure 5.7) this will lead to a lower mean arrival rate of PUBLISH messages as for the other two subscribers. Hence, if the number of subscribed topics and the sending rates of the publishers result in too high amounts of data to be transmitted by the broker to the client, the congested buffers will be responsible for poor performance. The end-to-end delay distribution for the client subscribed to all topics also points out the burstiness of the link. We will see in the next section if the number of subscribed topics also affects the efficiency.

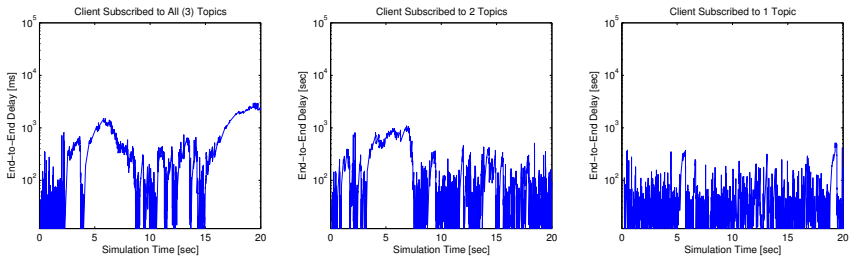

Figure 5.7: Mean end-to-end delay distribution for pBB = 0.8, 10^{-4} and 200B payload. Comparison between different kinds of subscribers.

Efficiency

In this section the efficiency of MQTT is addressed. Tables 5.6 and 5.7 contain the relevant data for pBB = 0.1. For the given payload sizes we observe the same facts as in the previous section for the end-to-end delay: the BER hardly changes the efficiency. The system is in a quite stable state, both from the point of view of the end-to-end delay but also with regard to the efficiency. The S and R ratios for QoS 0 and 200B payload are especially informative. We only loose 3% of b-efficiency when a BER of 10^{-4} is introduced. The higher the QoS, the lower is the effect of the BER on the m- and b-efficiency. This is due to the fact that errors will affect also smaller sized messages such as PUBACKs, PUBRECs etc. Loosing such messages is more

favorable for the efficiency as compared to loosing PUBLISH messages. We can actually conclude that if the error burst length is low, BER in the tested range don't significantly affect the performance.

Table 5.6: Efficiency for 50B payload and pBB = 0.1. Pub denotes the publisher, Sub the subscriber, S the ratio of application bytes sent to bytes sent at the MAC layer, R analogously for received bytes. For the publisher T_p refers to the ratio application bytes sent to *overall* bytes sent *and received* at the MAC layer, for the subscriber analogously with application bytes received (T_s).Additionally, we have included the ratios with respect to messages instead of bytes (S_m, R_m, T_{pm} and T_{sm}).

BER	QoS 0				QoS 1				QoS 2			
	Pub		Sub		Pub		Sub		Pub		Sub	
	S	T_p	R	T_s	S	T_p	R	T_s	S	T_p	R	T_s
0	.41	.26	.41	.26	.26	.15	.26	.15	.15	.08	.15	.08
10^{-4}	.4	.26	.4	.26	.26	.15	.26	.15	.15	.08	.15	.08
	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}
0	.99	.49	.99	.49	.5	.25	.5	.25	.25	.13	.25	.12
10^{-4}	.98	.5	.98	.5	.5	.25	.5	.25	.25	.12	.25	.12

Table 5.7: Efficiency for 200B payload and pBB = 0.1.

BER	QoS 0				QoS 1				QoS 2			
	Pub		Sub		Pub		Sub		Pub		Sub	
	S	T_p	R	T_s	S	T_p	R	T_s	S	T_p	R	T_s
0	.73	.58	.73	.58	.58	.41	.58	.41	.41	.26	.41	.26
10^{-4}	.7	.56	.7	.56	.56	.4	.56	.4	.4	.25	.4	.25
	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}
0	1	.5	1	.5	.5	.25	.5	.25	.25	.13	.25	.13
10^{-4}	.96	.48	.95	.48	.49	.25	.49	.25	.25	.13	.25	.12

We have seen that the results for a bad state sojourn time of one bit duration reveals a rather stable state of the system, introducing longer error bursts produces the results represented in tables 5.8 and 5.9. Considering a payload size of 50B, it is clearly shown that the efficiency decreases. For QoS 0 we get around 5% lower S and S_m ratios. It is worth noting that the T_p and T_s values experience a lower decrease than the corresponding S and R values. For the publisher this is because messages that it sends out are on average larger than the messages it receives (for QoS 0 it receives only TCP ACKs). Thus, on average the packets that it transmits have a higher loss probability than packets sent to it by the broker. This implies

that the publisher will have to retransmit more often and larger packets than the broker. The total number of bytes received is hence less affected by the effects of the error burstiness than it is the case for the total number of bytes sent. For the subscriber, analogously, the packets sent by the broker will experience a higher loss probability than the packets the subscriber sends itself.

For QoS 2 the effect of back-to-back losses is negligible. This again is due to the small MQTT ACK messages that lower the impact of lost packets. This is partially also true for QoS 1.

Table 5.8: Efficiency for 50B payload, pBB = 0.8 and BER = 10^{-4}.

QoS 0				QoS 1				QoS 2			
Pub		Sub		Pub		Sub		Pub		Sub	
S	T_p	R	T_s	S	T_p	R	T_s	S	T_p	R	T_s
.36	.24	.34	.24	.22	.13	.21	.13	.14	.08	.14	.08
S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}
.92	.48	.98	.51	.44	.22	.46	.23	.24	.12	.25	.12

In the previous section it was shown that not only the BER but also the payload size significantly affect the end-to-end delay when the channel is bursty. As we can see from table 5.9 this is also the case for the measured efficiency. This time also the T_p and T_s ratios experience a significant downstream, due to the fact that PUBLISH messages are bigger and have a higher loss probability. Retransmissions will result in more data having to be retransmitted than it is the case with 50B payload. For instance for the publisher, the number of outgoing bytes will affect the T_p ratio much stronger than it is the case with lower payload sizes.

Impressive is again the robustness of higher QoS levels, especially of QoS 2. While the T-ratios only differ by 3% from the measured values with pBB = 0.1, the m-efficiency only differs by 1%.

Table 5.9: Efficiency for 200B payload, pBB = 0.8 and BER = 10^{-4}.

QoS 0				QoS 1				QoS 2			
Pub		Sub		Pub		Sub		Pub		Sub	
S	T_p	R	T_s	S	T_p	R	T_s	S	T_p	R	T_s
.56	.47	.52	.44	.48	.36	.47	.35	.36	.23	.34	.23
S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}
.83	.44	.9	.49	.46	.23	.52	.26	.24	.12	.26	.13

According to the results obtained for the end-to-end delay the number of

subscribed topics and the resulting arrival rate of messages have a significant effect on the performance. The recorded statistics allow us to see if this is also true for the efficiency. The results are summarized in tables 5.10 and 5.11. To avoid overfilling the tables, only the more important b-efficiency is considered.

Table 5.10 serves just as a sanity check (and as a benchmark for the comparison with pBB = 0.8). The "type" of subscriber does not affect the efficiency. Hence for any subscriber, the efficiency it experiences is equal to the efficiency averaged over all subscribers.

If we increase the error burst length (see table 5.11) it shows that in such a setup, different subscribers experience different performances. But actually what was already stated earlier on, still holds: the higher the QoS level is the minor is the performance gain of subscribing to less topics. If we look at QoS 0, we see that for small payload sizes different subscribers see the same performance. However, with a payload of 200B subscriber groups S_2 and S_3 suffer an efficiency drop. The reason why we see worse efficiency of these groups is not directly obvious, though. We can argue that higher sending rates from the broker to these subscribers because of the higher number of interesting PUBLISHes can result in loaded TCP buffers. This would justify that end-to-end delays increase. Nevertheless, this does not directly justify that also the efficiency increases, since the ratio of application bytes received to bytes received (and sent) at the MAC layer is not directly affected by congested buffers. If we look at the number of bytes received and sent (instead of looking at the ratios R and T_s) we can actually conclude why we see this artifact. The recorded values reveal that, for a given amount of application bytes received, the more topics the client is subscribed to, the higher is the number of bytes received and the slightly lower the number of bytes sent at the MAC layer. The higher number of MAC bytes received is an indication that there are more retransmissions. And this can only be due to the fact that TCP packets are larger and therefore have a higher error probability, i.e. the packet error rate has increased. Why packets are larger in size can be explained by the fact that the higher number of subscribed topics fills the send buffers at the broker. And since there is data in the buffer waiting to be sent, TCP segments will be filled with more data (up to MSS). Thanks to OMNeT's powerful GUI this was easily verified, since you can inspect the size of each packet in the network.

On the other hand, the decreased number of MAC bytes sent is actually due to the same reason: since there are more bytes contained in a TCP segment, the subscribers can acknowledge more data at once, i.e. they have to send less TCP ACKs.

The fact that this feature is clearly less observable with higher QoS levels

needs further discussion. Comparing again the raw number of bytes received and sent at the MAC layer shows that the difference between different subscribers diminishes in the sense that the number of bytes received is nearly equal and the same holds for the nunmber of bytes sent. This is most probably due to the additional MQTT ACKs and the corresponding TCP ACKs that will sum up to the bytes sent and received by the subscribers. This amount of additional bytes will actually be the same for all subscribers, since each subscriber will have to send a MQTT ACK for each MQTT message that it wants to acknowledge and similarly the broker will send the corresponding TCP ACK for each such MQTT ACK received by a subscriber. And the higher the QoS, the more MQTT ACKs and corresponding TCP ACKs we will have.

Table 5.10: B-Efficiency comparison of different subscribers, pBB = 0.1. S_1 denotes the average over all clients being subscribed to only one topic, S_2 the average over subscribers to two topics and S_3 accordingly for subscribers to all three topics.

BER		QoS 0				QoS 1				QoS 2			
		R		T_s		R		T_s		R		T_s	
		50B	200B	50B	200B	50B	200B	50B	200B	50B	200B	50B	200B
0	S_1	.41	.73	.26	.58	.26	.58	.15	.41	.15	.41	.08	.26
	S_2	.41	.73	.26	.58	.26	.58	.15	.41	.15	.41	.08	.26
	S_3	.4	.73	.26	.58	.26	.58	.15	.41	.15	.4	.08	.25
10^{-4}	S_1	.41	.7	.26	.56	.26	.56	.15	.4	.15	.41	.08	.26
	S_2	.4	.7	.26	.56	.26	.56	.14	.4	.15	.4	.08	.25
	S_3	.4	.7	.26	.55	.25	.56	.15	.4	.15	.4	.08	.25

Table 5.11: B-Efficiency comparison of different subscribers, pBB = 0.8, BER = 10^{-4}.

	QoS 0				QoS 1				QoS 2			
	R		T_s		R		T_s		R		T_s	
	50B	200B	50B	200B	50B	200B	50B	200B	50B	200B	50B	200B
S_1	.35	.55	.24	.47	.22	.49	.14	.36	.14	.35	.08	.24
S_2	.34	.51	.24	.43	.21	.47	.14	.35	.14	.35	.08	.22
S_3	.34	.48	.24	.4	.20	.46	.13	.34	.14	.34	.08	.22

5.3 Wireless Network

In this chapter we will focus on MQTT's performance in a wireless environment. More specifically, the most recent INET-framework version (INET-20050720) for OMNeT++ includes modules to simulate IEEE 802.11 networks.

5.3.1 Introduction

In IEEE 802.11, in addition to MQTT's and TCP's ACKs, we also have ACKs at the MAC layer. This leads to a more complex behavior than in an Ethernet network. Furthermore, in a wireless environment, there are more error sources than in a wired network. Bit errors and packet losses will not only occur because of random error sources and collisions, but also because of signal strength weakening over distance and fading. OMNeT's INET-framework allows to set some parameters that influence these characteristics. In the following paragraphs we specify the chosen values.

Path attenuation refers to RF power loss due to e.g. link path obstructions and link distance. It is usually expressed in dB. In the following simulations the path loss coefficient is set to 3dB. We also assume that the area over which the MQTT hosts are distributed, is composed of $100m \cdot 100m$. The broker is placed within the area such that it is within transmission range of all clients and vice versa.

How strong a received signal is also depends on the sender's transmission power. A reasonable value of 2mW (3dB) was chosen, with the receivers' sensitivity set to -100dB.

Moreover, for each host we define a threshold for the signal-to-noise ratio (SNR). When a signal is received and the SNR is beneath this threshold, the frame is considered to be corrupted and will be lost. This threshold was set to 4dB.

OMNeT++ also allows to set the thermal noise, which is noise generated by the thermal agitation of electrons in a conductor. For completeness this value was set to -110dB (which is nearly negligible).

Finally, the carrier frequency was fixed to 2.4GHz, which corresponds to a IEEE 802.11b WLAN.

5.3.2 Adapted Gilbert-Elliot Model

Due to OMNeT++'s internal functionality, it does not allow the use of the developed Gilbert-Elliot model (see section 4.3) in a wireless network. However, the INET-framework includes a modified Gilbert-Elliot model which

behaves similarly. It basically has two parameters: meanGood and meanBad. These correspond to the time spent in each state before a transition to the other state, hence they represent the mean state sojourn times.

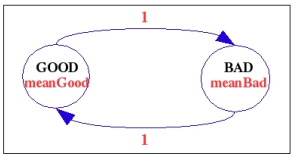

Figure 5.8: Gilbert Channel Model

The main difference to the Gilbert-Elliot model is actually that the mean error probabilities for each state are fixed such that when you are in the good state, no errors occur, when you are in the bad state the packet error probability is uniformly distributed with a bad state *packet* error probability of 0.5. The transition probabilities are also fixed in the sense that the probability to stay in a state after the state sojourn time has ellapsed is zero, thus we change state everytime the state holding time (meanGood, meanBad) indicates to do so. This is a simplified version of the Gilbert-Elliot model and corresponds to the Gilbert model ([11]), but still allows us to introduce bursty errors. However, it clearly has deficiencies. First, because of the fixed per state error and transition probabilities, and secondly because the burst length will basically depend on the data rate used. Still, up to a certain point it allows for simulating wireless channel characteristics.

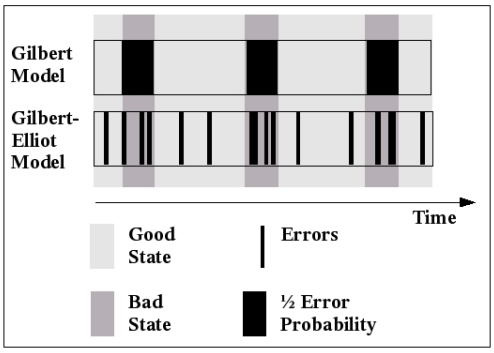

Figure 5.9: Difference between the Gilbert and the Gilbert-Elliot channel models.

In the following simulations, we will fix the ratio $meanBad/meanGood$ to 10%, i.e. around 10% of the time the channel is in the bad state. This will allow for the rest of this document to only indicate the chosen meanBad value, the meanGood parameter can then easily be computed.

5.3.3 Small IEEE 802.11 Network

We first would like to consider again a simple network of just two clients in addition to the broker. The bandwidth of the channel is set 2Mb/s and the RTS/CTS mechanism ([20]) is enabled.

As a preliminary step, an adequate sending rate for the publisher should be found. Please note that we reduced the bandwidth of the channel from 100Mb/s in the Ethernet network to 2Mb/s. But, also we should consider that we now have acknowledgments at the MAC layer in addition to the already existing acknowledgment procedures in TCP and MQTT. Moreover, when RTS/CTS is enabled, we have additional messages sent by the 802.11 MAC layer, which should lead to less collisions.

Looking at the end-to-end delay for meanBad values in the range of 0 to 60 ms and a small payload size of 30B shows that the impact of varying the sending rate is significant. Especially for QoS 2 we see a high variance between different sending rates. E.g. decreasing the mean interarrival time from 50 to 40ms increases the mean end-to-end delay by at least 2 orders of magnitude for higher sojourn times in the bad state of the channel model. Based on these observations, we choose a temporary sending rate of 25 messages/second.

We now would like to see the impact of data load on the performance of MQTT from the point of view of the end-to-end delay and especially with regard to the b-efficiency.

The focus will be on the mechanisms that actually produce more data load on the network. There are several features at different protocol levels that should be considered. On the MQTT level, we can clearly identify the MQTT payload size as one of the factors contributing to the data load. Also, we expect different QoS levels to be unequally efficient since they produce different amounts of MQTT control messages.

At the 802.11 MAC layer, the amount of data sent/received also depends on the fact that the RTS/CTS mechanism is enabled or not. Enabling this collision avoidance mechanism leads in general to an increased amount of data sent and received. But on the other hand it is supposed to lead to less collisions and hence to less retransmissions.

Table 5.12 shows the recorded values for a MQTT payload size of 200B. The results are similar for smaller sized payloads (simulations were performed for 30B up to 200B of payload). With regard to the mean end-to-end delay, a very obvious observation is made. Since we have only a few hosts (exactly 3 of them) sharing the same medium, collisions are quite unlikely to happen. Hence, the delay introduced by first sending a RTS message and secondly waiting for the corresponding CTS from the peer before sending

the data frame, clearly affects the end-to-end delay. The higher the QoS, the more significant this impact is since there are more overall packets sent (please note that any MQTT message will trigger the sending of many other packets including RTS/CTS frames, MAC layer ACKs, TCP ACKs and eventually MQTT ACKs). And more (MQTT and TCP) messages sent implies more performed RTS/CTS procedures. This also explains why for a disabled RTS/CTS mechanism the mean end-to-end delay for QoS 2 is around three to four times higher than for lower QoS levels whereas with RTS/CTS the ratio is clearly higher (say five to six).

Table 5.12: End-to-end delay [ms] with and without RTS/CTS with 200B payload. mB denotes the mean sojourn time in the bad state (= meanBad parameter).

mB [ms]	RTS/CTS	QoS 0		QoS 1		QoS 2	
		Mean	Std.	Mean	Std.	Mean	Std.
0	Enabled	6	2	11	5	49	35
20		10	31	14	28	71	80
40		11	35	16	31	90	160
60		12	26	18	35	95	162
0	Disabled	5	1	7	3	23	11
20		7	22	10	36	28	21
40		8	31	10	30	30	31
60		6	28	9	31	35	43

We can also observe how the evolution for different mB (mean bad state sojourn time) differs depending on the usage of RTS/CTS. Having this option disabled leads to minor increases in the end-to-end delay. Making usage of the RTS/CTS procedure instead lets the end-to-end delay nearly double when having a 60ms state holding time in the bad state, as compared to having no additional errors (mB = 0). This is in fact again caused by the increased number of steps involved in delivering a MQTT message over TCP and 802.11 from sender to receiver, for then it is more probable that a step has to be reexecuted because of a loss. Hence, it shows that in this setup it clearly does not pay off to use the RTS/CTS machanism, as it leads to significantly increased end-to-end delays because of the additional message flows.

Probably more interesting than the mean end-to-end delay is the efficiency in terms of bytes sent and received by the MQTT clients. Again we will discuss the results for 200B payload, but the observations made also apply for smaller sized payloads (see figure 5.10). At first, it is interesting to see that the ratios calculated don't differ much for different mean bad state sojourn times (that's why in table 5.13 only the statistics for two mB values

are included). On average they differ by 2 or 3%. It may primarily sound contradictory to the results obtained for the end-to-end delay. There, we actually experienced quite large variances of the performance for different mB values, especially for higher QoS levels. However, this can be explained as follows. First of all we know that for higher mB values the performance decreases because of retransmissions at any layer (mainly on TCP and on the MAC layer). Then, the end-to-end delay for a retransmitted packet will mainly depend on the chosen retransmission timeout. But, since retranmitted packets will encouter the same mean error probability, the probability that a retransmission is lost is non-negligible. The retransmission timeout will then increase exponentially and significantly affect the end-to-end delay. On the other hand, the retransmitted bytes will only add linearly to the b-efficiency ratios, hence the efficiency is less sensitive to retransmissions.

The table also reveals that for a given mB and QoS level with RTS/CTS enabled or disabled, the total amount of bytes sent and received by a client is independent of the client being a publisher or a subscriber, i.e. T_p and T_s (see the table) are equal for a given set of parameters. This indicates that the clients are in transmission range of each other, because then every packet sent by one client is received by the other, and assuming that both clients are in trasmission range of the broker, they both will receive the brokers data. Moreover, assuming that each PUBLISH message sent by the publisher will be delivered to the subscriber (TCP should take care of this), the number of application bytes sent by the publisher equals the number of application bytes received by the subscriber. Hence, T_p and T_s are equal.

Table 5.13: B-Efficiency with and without RTS/CTS with 200B payload. Pub denotes the publisher, Sub the subscriber, S the ratio of application bytes sent to bytes sent at the MAC layer, R analogously for received bytes. For the publisher T_p refers to the ratio application bytes sent to *overall* bytes sent *and received* at the MAC layer, for the subscriber analogously with application bytes received (T_s).

mB [ms]	RTS/CTS	QoS 0				QoS 1				QoS 2			
		Pub		Sub		Pub		Sub		Pub		Sub	
		S	T_p	R	T_s	S	T_p	R	T_s	S	T_p	R	T_s
0	Enabled	.59	.21	.25	.21	.42	.13	.16	.13	.28	.08	.1	.08
60		.56	.2	.23	.2	.4	.13	.16	.13	.27	.08	.1	.08
0	Disabled	.66	.25	.28	.25	.49	.16	.2	.16	.35	.1	.13	.1
60		.6	.22	.25	.22	.44	.15	.17	.16	.32	.09	.12	.09

It should now also be clear, why the S ratio is always higher than the R ratio. Again, this is because of the subscriber receiving everything that

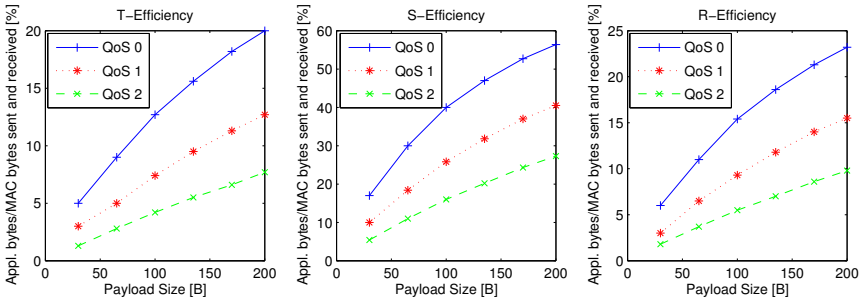

Figure 5.10: B-Efficiency for 60ms mean bad state sojourn time and enabled RTS/CTS. T-Efficiency refers to the ratio averaged over T_p and T_s, S- and R-Efficiency denote the S and R ratios.

the publisher sends (to the broker) in addition to the data received from the broker (be it data sent to the publisher or to the subscriber itself). It is nevertheless impressive to see already with QoS 0 that while for the publisher more than every second byte sent out is MQTT application data, for the subscriber at most every fourth byte received will be delivered to the application. And the higher the QoS, the lower these ratios are, both for the publisher and for the subscriber. Also, disabling the RTS/CTS mechanism will not help much, on average we will only see a 5% enhancement. Therefore we can actually state that the RTS/CTS mechanism has rather an impact on the measured end-to-end delays than on the efficiency ratios (power consumption).

With regard to the T_p and T_s respectively, we see that the lower the QoS level, the bigger is the difference of the values between enabled and disabled RTS/CTS mechanism. The reduced number of bytes sent and received is self-explanatory, it corresponds to the RTS/CTS frames that are not sent nor received. Hence, in this setup some power is saved by disabling the RTS/CTS procedure with QoS < 2.

Figure 5.10 summarizes the observations made. It clearly shows that the shape of the different b-efficiency measures is the same for all QoS, what differs is the scale.

5.3.4 Several Publishers and Subscribers

From the observations in the previous section it becomes clear that the fact that clients are in transmission range of each other or not has a clear impact on the performance measures. In the following experiments the clients are randomly distributed over the test area, some of them being in transmission range of each other and some not. In a real life situation it may be possible

to place the clients at adequate positions to reduce the interference, but this scenario will not be subject of the following simulations.

The network setup is similar to the one in section 5.2.2, i.e. we have 10 MQTT publishers and 10 MQTT subscribers subdivided into the same kinds of sets.

Since we have more hosts sharing the medium, it is sensible to increase the bandwidth to 11Mb/s, which corresponds to a 802.11b network. Similarly, the test area has been expanded to $150m \cdot 150m$, again because of the increased number of hosts. This implies that the average distance between hosts increases and hence, also the transmission power has been adapted to 6mW (∼8dB).

Preliminary tests have shown that already for sending rates of 10 messages per second (100ms mean interarrival time) per publisher the system becomes very unstable in the sense that the end-to-end delays rise to tens of seconds and buffers congest. It has to be kept in mind that having 10 publishers sending at such a rate will unevitably lead to an overall sending rate of 100 messages per second. This was discovered to be to high already in the setup of the previous section, where we only considered two clients, a publisher and a subscriber. Hence, we will focus on a sending rate of just one message per second and per publisher. It should however be considered that this will all the same lead to a highly enough used medium, since all the control messages of different OSI layers, i.e. RTS/CTS and ACKs at the 802.11 MAC, TCP and MQTT layer, will add to the PUBLISH messages sent.

5.3.4.1 RTS/CTS Enabled

We first consider that the RTS/CTS mechanism is turned on by every MQTT host. Since the sending rate has been reduced, we can effort to increase the payload sizes and vary them between 200 and 1000B (and we still have a TCP MSS of 536B).

End-to-End Delay

In the following mean end-to-end delay refers to the mean averaged over all clients. We first concentrate on QoS 0 and 1 before discussing the results for QoS 2. The results show that in general increasing the bad state holding time of the Gilbert model and thereby increasing the error burstiness of the channel is not significant for lower application payload sizes. This was also observed in the small network of the previous section. For QoS 0 we actually have a rather smooth evolution of the mean end-to-end delay for different bad state sojourn times and a fixed paylaod size. Only in the case of 1000B

payload we notice a strong impact of the error burstiness. For QoS 1 we actually see the impact of the burstiness once the payload size goes beyond the MSS.

Increasing the payload over the MSS threshold leads clearly to an increase of the mean end-to-end delay as TCP will fragment the payload and hence the fragments of the MQTT PUBLISH message may arrive delayed. However, also for higher payload sizes we have a low variation of the delay for QoS 0 (except for the case mentioned above), only the measured delay when having no additional errors differs significantly. Thus, we have a kind of robustness against error burstiness with QoS 0. This is interesting to compare with the results we obtained in section 5.3.3 (see table 5.12) where we actually experienced a doubling of the delay when increasing the bad state holding time from 0 to 60. This discrepancy basically has two reasons. On one hand it is caused by the fact that we have reduced the sending rate. On the other hand the increased number of clients and the resulting contention periods and collisions may reduce the impact of additional random packet losses.

Interestingly, QoS 1 seems to be more sensitive to error burstiness when having big payload sizes. Moreover, the scale of the mean end-to-end delay is completely different, we have values around 100% greater than the corresponding values for QoS 0. This increase cannot be explained by MQTT's operation, as the additional PUBACKs in the return paths do not directly delay the PUBLISH messages to be delivered to the subscribers' applications. However, it can be explained by the consequences of the additional MQTT PUBACK messages, namely the loading of the medium by the PUBACKs and all the resulting messages (RTS/CTS, 802.11 ACKs, TCP ACKs and potential retransmissions). In fact, the broker will send a PUBACK for each PUBLISH received from any of the ten publishers, and the ten subscribers will do the same when receiving data from the broker. This will obviously lead to contentions and/or collisions. If we compare these results with the observations made with Ethernet networks, it shows that a wireless environment is much more sensitive to additional message traffic, hence we get significant differences of the performance between QoS 0 and 1, whereas with Ethernet this was not the case.

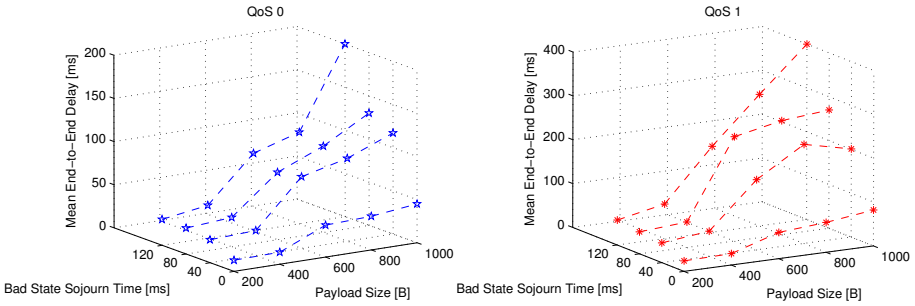

Figure 5.11: Mean end-to-end delay evolution for QoS 0 and 1. RTS/CTS enabled.

The most curious results were obtained for QoS 2. As figure 5.12 shows we have tremendous mean end-to-end values in the range of several seconds. It is actually difficult to interpret the results, because there seems to be quite a lot of randomness. Nonetheless, we should take these results as an indication that QoS 2 can lead to unstableness and poor performance as compared to lower QoS levels. Moreover, it introduces a lot of variance that makes it impossible to estimate the performance. It should also be kept in mind that we get such bad results with low sending rates of one message per second and per publisher, and that we have a simplified network of "only" 20 clients. Experimenting with a network of 100 clients or more would probably completely break the system, i.e. the effective end-to-end delay would be enormous.

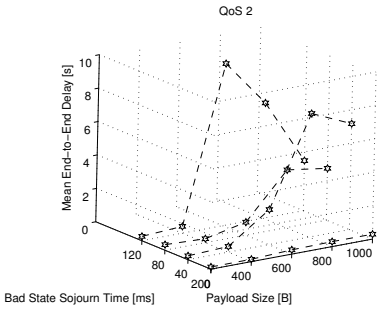

Figure 5.12: Mean end-to-end delay evolution for QoS 2. RTS/CTS enabled.

It was mentioned at the beginning of this section that we would also like to compare the end-to-end delay distributions of different types of subscribers, i.e. clients that are subscribed to different numbers of topics. Nevertheless, comparing the obtained distributions don't show any particularly interesting behavior. We would actually expect the distributions to be different when

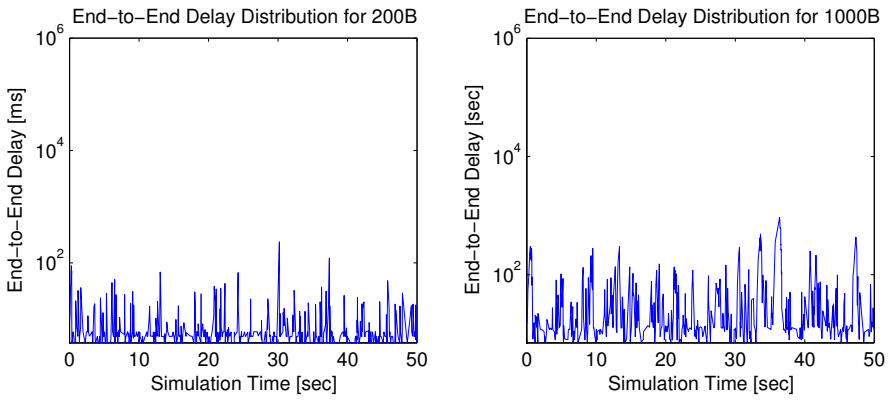

Figure 5.13: End-to-end delay distribution with QoS 0. 200 and 1000B.

the higher number of subscribed topics would result in congestions of TCP windows. But since the contentions and collisions affect all messages equally and the publishers send at a low rate, there shouldn't be any special effect when being subscribed to more topics.

But what certainly should be compared is the distribution of the end-to-end delay for low and large payload sizes with respect to the MSS. Figure 5.13 compares the distribution for payload sizes of 200 and 800B, QoS 0 and a mean bad state holding time of 120ms for a client subscribed to all topics. We observe that actually the peaks reach up to the same values with 200 and 800B. But, looking at the bottom line of the plot reveals that there is a larger "base" end-to-end delay for 800B messages. Furthermore, the frequence of the peaks is higher with larger payload size, i.e. with 200B peaks are more sporadic.

In figure 5.14 we observe that with QoS 2 there are bursty peaks already with 200B (e.g. during the period between 20 and 30 simulated seconds). Looking at the distribution for 800B, we actually don't have anymore peaks, we rather have constantly high values. We clearly see the drawback of big payload sizes when using QoS 2, and also with lower QoS levels the network is very sensitive to the payload size.

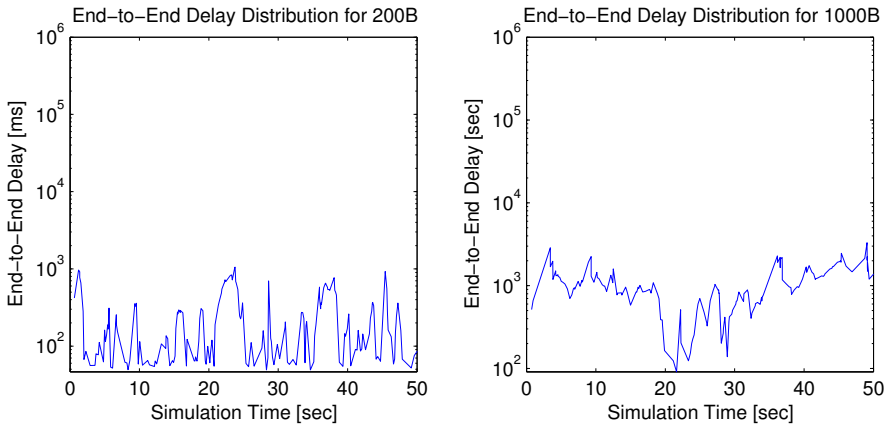

Figure 5.14: End-to-end delay distribution with QoS 2. 200 and 1000B.

Since we have seen that the impact of the payload size is huge and we argue that it depends on the MSS, we should consider increasing the MSS to a value beyond the biggest payload size used. Hence, we now will discuss the results for a MSS of 1024B, which is a common value used in most implementations.

Figures 5.15 and 5.16 show clearly that by increasing the MSS the mean end-to-end delay was significantly reduced for payload sizes larger than the old MSS. Moreover, for smaller payload sizes we get similar results as with a MSS of 536B. Therefore it seems that although larger packets have a bigger chance to get lost with respect to the random errors introduced, we get much better results if TCP can avoid fragmenting the payload.

However, we again experience a high variance when having QoS 2. Moreover, there are no huge mean end-to-end delay values anymore, but still we get significantly larger values than with lower QoS levels due to the nature of QoS 2 (4-way handshake).

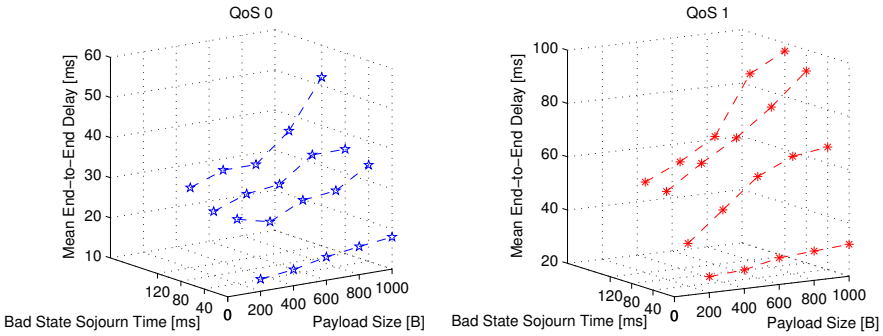

Figure 5.15: Mean end-to-end delay evolution for QoS 0 and 1 with 1024B MSS. RTS/CTS enabled.

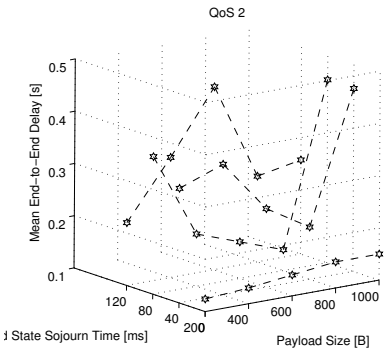

Figure 5.16: Mean end-to-end delay evolution for QoS 2 with 1024B MSS. RTS/CTS enabled.

Efficiency

After having analyzed the measured end-to-end delays, in this section the focus will be on the (power-)efficiency. Statistics were collected over all publishers on one hand, and over all subscribers on the other hand.

If we first take a look at table 5.14 it seems that introducing bursty errors via the Gilbert model does only have an impact on the S and S_m ratios, all the other values are approximately the same for different mean bad sojourn times (mB). We can also imagine why the other ratios remain the same. The number of bytes and frames received by any host, both frames addressed to the receiving host or not (it will receive both and detect on the MAC layer if it is the destination) clearly dominates over the frames sent by a host. That is the reason why e.g. the S and the T_p measures differ by two orders of magnitude. In the case of a network of only 2 clients the T_p ratio was clearly higher because the publisher only received redundant frames from 2 hosts,

Table 5.14: Efficiency for 200B payload. mB, S, R, T_p and T_s are defined as in table 5.13. Additionally, we have included the ratios with respect to messages instead of bytes (S_m, R_m, T_{pm} and T_{sm}).

mB [ms]	QoS 0				QoS 1				QoS 2			
	Pub		Sub		Pub		Sub		Pub		Sub	
	S	T_p	R	T_s	S	T_p	R	T_s	S	T_p	R	T_s
0	.58	.005	.03	.04	.41	.003	.024	.023	.26	.002	.014	.013
120	.49	.005	.03	.03	.38	.003	.023	.022	.24	.002	.014	.013
	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}
0	.24	.002	.01	.01	.12	.0008	.005	.005	.06	.0004	.003	.003
120	.18	.001	.009	.009	.1	.0007	.005	.005	.06	.0004	.003	.003

Table 5.15: Efficiency for 1000B payload.

mB [ms]	QoS 0				QoS 1				QoS 2			
	Pub		Sub		Pub		Sub		Pub		Sub	
	S	T_p	R	T_s	S	T_p	R	T_s	S	T_p	R	T_s
0	.74	.008	.06	.05	.66	.006	.05	.05	.57	.005	.04	.03
120	.67	.007	.05	.05	.63	.006	.04	.04	.54	.005	.03	.03
	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}
0	.11	.0007	.005	.005	.08	.0005	.004	.003	.05	.0003	.002	.002
120	.09	.0006	.004	.004	.07	.0004	.003	.002	.04	.0003	.002	.002

the broker and the other client. In this setup there are 19 other clients, some of them being in transmission range and hence everything sent by them will be received. In addition, every client will receive everything that the broker sends, since the network setup was chosen in such a way. Hence, if there are 20 clients talking to the broker, every client will receive the redundant data sent by the broker to any of the other clients.

Moreover, the measured ratios at message granularity (S_m, R_m, T_{pm} and T_{sm}) are much smaller than the corresponding byte ratios. This could indicate that there are a lot of small frames received by a host. Small frames may be RTS/CTS and MAC layer ACKs frames. Hence, the radio interface is "active" a significant amount of time to receive small control frames, and most of them are not even being addressed to the host itself.

With 1000B payload, we experience the same robustness against errors and their burstiness as with smaller payload sizes. If we remember that the mean end-to-end delays were significantly influenced by the additional bursty errors, this is quite astonishing. We can again observe that additional packet losses due to random errors introduced impact the end-to-end delay, but not very much the efficiency or wastefulness of the protocol.

In addition, the byte count ratios are eminently higher with 1000B payload. Some of them, e.g. the S ratio for QoS 2 are twice as high. However, the price to pay is an increased number of radio interface switches. But this is clearly due to the overstepping of the MSS. Since TCP will fragment the 1000B payload into two packets, the number of frames sent and received will be higher. And thus, we actually have around 50% lower m-efficiency ratios with 1000B than with 200B. This is clearly improved when increasing the MSS over 1000B, as table 5.16 shows.

Table 5.16: Efficiency for 1000B payload with 1024B TCP MSS.

mB [ms]	QoS 0				QoS 1				QoS 2			
	Pub		Sub		Pub		Sub		Pub		Sub	
	S	T_p	R	T_s	S	T_p	R	T_s	S	T_p	R	T_s
0	.86	.01	.07	.07	.76	.008	.06	.06	.62	.006	.04	.04
120	.78	.008	.06	.06	.7	.007	.05	.05	.59	.006	.04	.04
	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}
0	.24	.002	.01	.01	.12	.0008	.006	.005	.06	.0004	.003	.003
120	.19	.001	.009	.008	.1	.0006	.005	.005	.06	.0004	.003	.003

When comparing the ratios between different QoS levels, we notice a pretty linear evolution of the values. While we noticed a high variance introduced by QoS 2 when discussing the mean end-to-end, this is not true for the efficiency. A reason might be that the end-to-end delay is highly sensitive to contention periods caused by the additional MQTT control messages. However, the efficiency only depends on the transmission/reception of a frame, not on the time it was delayed because of a busy medium.

5.3.4.2 RTS/CTS Disabled

In this section, the results obtained when disabling the RTS/CTS mechanism are shortly discussed and compared with the results of the previous section. As already mentioned previously, the main benefit of using the RTS/CTS mechanism is when terminals are "hidden" from each other, i.e. they are not in transmission range of each other, but there is a terminal (in our case the broker) which both can hear. When having hosts distributed randomly over the test area, comparisons between results obtained with and without RTS/CTS can potentially vary a lot depending on the positioning of the hosts. Since simulations have been carried out with randomly positioned hosts with respect to the broker, the results are of course specific to that setup. Therefore, this section should rather serve to give an idea of how small or big the difference can be between using and not using RTS/CTS for given MQTT and TCP parameters.

End-to-End Delay

Table 5.17 gives an impression of how strong the impact of disabling RTS/CTS can be with regard to the end-to-end delay in a random setup. To have a better overview, only the results for the smallest and largest payload sizes have been included.

Clearly, we see a better performance when disabling the RTS/CTS mechanism for all mB values and both payload sizes. We observed this already in the network with only two MQTT clients in transmission range of each other. In general it seems that without RTS/CTS we have more stability in several dimensions. First, if we look at the variance introduced by the error burstiness factor mB, the table shows much higher proportional increases with RTS/CTS. Secondly, in the dimension of the payload size, it can be identified that in both cases the impact of having payloads bigger than TCP's MSS is high. However, it has a deeper effect when enabling RTS/CTS. Third, when increasing the QoS level, we again experience a bigger performance loss with RTS/CTS. So, in summary we cannot find any advantage of using the RTS/CTS mechanism in a network with our parameter settings.

Table 5.17: Mean end-to-end delay [ms] with and without RTS/CTS with 200B and 1000B payload.

mB [ms]	RTS/CTS	QoS 0		QoS 1		QoS 2	
		200B	1000B	200B	1000B	200B	1000B
0	Enabled	13	45	24	82	124	290
40		26	117	45	200	230	$> 1s$
80		30	130	50	270	330	$> 1s$
120		30	200	57	400	260	$> 1s$
0	Disabled	8	23	12	29	49	100
40		11	36	17	54	76	770
80		11	44	20	62	84	$> 1s$
120		11	42	22	75	86	$> 1s$

Nevertheless, there is a property that the fact of not using the RTS/CTS mechanism did not solve. With QoS 2 we still have the unstableness when using big payload sizes and having additional errors. We get high variance and mean end-to-end delays in the range of several seconds. This seems to be an artifact that is difficult to avoid.

In section 5.3.4.1 we determined that also when having a high enough TCP MSS, i.e. 1024B, we still noticed the high variance of QoS 2 with high payload sizes when varying the mean bad state sojourn time. Table 5.18 compares the measured statistics for 1000B payload and according to the

usage of the RTS/CTS option with the MSS set to 1024B. Again, it shows that with RTS/CTS the error burstiness affects the measured mean end-to-end delay much more than it is the case when the option is disabled.

Table 5.18: Mean and standard deviation of the end-to-end delay [ms] with and without RTS/CTS with 1000B payload and 1024B MSS.

mB [ms]	RTS/CTS	QoS 0		QoS 1		QoS 2	
		Mean	Std.	Mean	Std.	Mean	Std.
0	Enabled	18	11	32	26	150	110
40		34	37	58	69	450	365
80		36	45	92	138	450	410
120		52	84	90	135	280	274
0	Disabled	15	6	18	10	62	35
40		17	11	29	35	100	83
80		17	13	26	20	112	94
120		19	18	27	26	170	225

Efficiency

Finally, we will shortly discuss the results for the efficiency of the protocol without the usage of RTS/CTS frames. Tables 5.19 to 5.21 contain all the gathered statistics.

As in the case with RTS/CTS, we notice a fairly constant behavior when increasing the error burstiness. Comparing the results for no additional errors and 120ms mean bad state sojourn time evidently points out that except for the S ratio, no other measure differs by more than 1%. Therefore, according to the results and our power consumption estimates, we cannot conclude that having larger error burst lengths leads to significantly higher wastefulness of MQTT. On average, a single publisher will actually experience more TCP or MAC layer retransmissions when having longer error bursts, however the total amount of time where the radio interface is not idle, including transmission and receiving periods, is fairly constant. This again implies that the overall time spent in the receiving state is much longer than the overall time spent transmitting. Looking up the raw data recorded with OMNeT++ shows that for a publisher on average only 8 out of 1000 bytes are transmitted bytes, the rest of the data was received.

What actually interests more is the comparison with the results obtained with enabled RTS/CTS option. Comparing the S and S_m ratios makes clear that sending RTS/CTS frames will actually have an impact on the number

of times the radio switches state. This can be seen by the S_m value that is twice as high with disabled RTS/CTS for all QoS levels. Nonetheless, the S ratio is only 5% higher since RTS/CTS frames are small in size.

It is worth noting that in general it can be detected that the impact of error burstiness on the publisher is bigger with disabled RTS/CTS. E.g. comparing the S ratio for 1000B payload and QoS 1 shows only a 3% decrease with RTS/CTS enabled when the mean bad state sojourn time is changed from 0 to 120ms. But, in the absence of the RTS/CTS mechanism the publisher will experience a 15% decrease. However, it seems difficult to explain this behavior. A possible explanation may be that by using RTS/CTS frames, a significant part of the frames sent are exactly frames of that type. Then, an error burst may lead to the loss of RTS/CTS frames and their retransmissions. And since RTS/CTS frames are small as compared to data frames, loosing them instead of data frames will not have a deep impact on the number of MAC layer bytes sent. In the absence of RTS/CTS frames, errors and especially error bursts will affect data frames (or ACKs) and their retransmissions. So, the probability that the errors will affect frames of bigger size (namely data frames) is actually bigger.

Table 5.19: Efficiency without RTS/CTS and 200B payload.

mB [ms]	QoS 0				QoS 1				QoS 2			
	Pub		Sub		Pub		Sub		Pub		Sub	
	S	T_p	R	T_s	S	T_p	R	T_s	S	T_p	R	T_s
0	.64	.006	.04	.04	.48	.004	.03	.03	.31	.002	.02	.02
120	.54	.005	.03	.03	.39	.003	.02	.02	.27	.002	.01	.01
	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}
0	.47	.003	.02	.02	.24	.002	.01	.01	.12	.0008	.006	.005
120	.4	.003	.02	.02	.2	.001	.009	.008	.1	.0006	.005	.005

Table 5.20: Efficiency without RTS/CTS and 1000B payload.

mB [ms]	QoS 0				QoS 1				QoS 2			
	Pub		Sub		Pub		Sub		Pub		Sub	
	S	T_p	R	T_s	S	T_p	R	T_s	S	T_p	R	T_s
0	.79	.009	.06	.06	.73	.008	.05	.05	.6	.006	.04	.04
120	.63	.007	.05	.04	.58	.006	.04	.04	.5	.004	.03	.03
	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}
0	.23	.001	.01	.01	.16	.001	.007	.007	.09	.0006	.005	.004
120	.19	.001	.008	.008	.12	.0007	.005	.005	.08	.0005	.003	.003

Table 5.21: Efficiency without RTS/CTS, 1000B payload and 1024B MSS.

mB [ms]	QoS 0				QoS 1				QoS 2			
	Pub		Sub		Pub		Sub		Pub		Sub	
	S	T_p	R	T_s	S	T_p	R	T_s	S	T_p	R	T_s
0	89	.01	.07	.07	.79	.09	.06	.06	.66	.007	.05	.05
120	.7	.008	.06	.06	.65	.007	.05	.05	.51	.005	.04	.03
	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}	S_m	T_{pm}	R_m	T_{sm}
0	.47	.003	.02	.02	.24	.002	.01	.01	.1	.0008	.006	.005
120	.38	.002	.02	.02	.2	.001	.009	.009	.1	.0006	.004	.004

Chapter 6

Conclusion and Future Work

This chapter is split into two sections: one for the conclusion about the discrete event simulator OMNeT++, and the other for the MQTT protocol itself.

OMNeT++ as a Simulation Tool

OMNeT++ is definitely worth to be considered a powerful simulation system that can keep up with other popular simulation tools. Its clear structure, modular design and debugging capabilities are big advantages over other commercial and non-commercial simulators. Moreover, its strong GUI support makes it one-of-a-kind. With regard to the performance aspect, it seems that this represents a big challenge for most popular simulators, and OMNeT++ is no exception. On the other hand, we face a lack of models and especially the problem of not enough validated models. This is the reason why in the context of this thesis a TCP and a wireless channel model were integrated into the simulator. But considering that the OMNeT++ community is continuously expanding, it is just a matter of time when more stable and complete simulation models will increasingly satisfy the users needs.

MQTT Performance

In this thesis several performance measures and several factors affecting these measures have been looked at. Of particular interest were the end-to-end delay and the (power-) efficiency of MQTT. Parameters on several layers (MQTT, TCP, MAC including Ethernet and IEEE 802.11 and physical layer) were varied and the resulting performance observed.

On the basis of a small Ethernet network it was shown how significantly TCP's MSS, the timestamp option and the error correlation of the physical

link affect the measured performance (especially the end-to-end delay). Also, it was demonstrated how higher QoS decrease the efficiency because of the additional MQTT ACK control messages.

The transition to an Ethernet network with 20 MQTT clients revealed that when having non-bursty errors we don't experience significant changes of the performance measures when varying parameters such as the bit error rate (BER). However, introducing burstiness makes clear how important the BER and the payload size are for the resulting performance. QoS 2 seems to be a paradoxical case where the end-to-end delay is very sensitive to the values of the parameters mentioned above whereas the efficiency stays rather constant. It was also shown how the performance for different subscribers varies depending on the number of topics the clients are subscribed to.

The main focus was however on wireless IEEE 802.11 networks. This completely changes the observed performance since not only we have significantly higher probabilities of collisions and contention periods, but also we have MAC layer acknowledgments. The Gilbert channel model was used to simulate wireless links and the effect of the usage of the RTS/CTS option was tested.

For the network setups tested, much more variance of the end-to-end delay was observed than in Ethernet networks. Also, the measured performance differences between the available QoS levels was eminently higher. In general, the wireless setup proved to be much more sensitive to the parameter settings. With regard to the RTS/CTS mechanism, it was shown that disabling this option led to significantly better performance.

Future Work

This project was a first step towards evaluating MQTT's performance under several network assumptions by means of the OMNeT++ simulator. Clearly, testing MQTT under all possible parameter combinations was out of the scope of this work, only a subset of the potential setups has been focused on. For future tests (possibly with OMNeT++ and the MQTT and TCP models developed) there are however critical points that should be addressed.

One evident aspect is the network size. In practice, MQTT is used with several hundred clients per broker, and this should also be considered in the simulations. Especially in wireless networks the number of clients plays a crucial role when measuring the performance of the protocol. Nevertheless, OMNeT++ does not seem to be able to simulate such networks with reasonable performance when using adequate publisher sending rates.

In addition, more tests with different sending rates and distributions

would be senseful, as these are important parameters, too.

To simulate wireless channels, the Gilbert model shipped with the INET framework is clearly not sufficient. Another enhanced model should be developed in order to allow for specifying more accurately wireless channel characteristics.

The INET framework also provides some support to simulate mobile networks, i.e. it is possible to have moving hosts. It would be interesting to test MQTT in mobile networks where sensors change position over time. However, this has been identified to lead to more events to be handled by the simulation kernel and therefore the performance of the simulator degrades significantly.

Bibliography

[1] WebSphere MQ Telemetry Transport. http://publib.boulder.ibm.com/infocenter/wbihelp/index.jsp?topic=/com.ibm.etools.mft.doc/ac10840_.htm

[2] A. Varga. OMNeT++, http://www.omnetpp.org.

[3] K. Entacher, B. Hechenleitner, S. Wegenkittl. A simple OMNeT++ queueing experiment using parallel streams. PARALLEL NUMERICS'02 - Theory and Applications, pages 89–105, 2002.

[4] P. Hellekalek. Don't trust parallel Monte Carlo. ACM SIGSIM Simulation Digest, 28(1):82–89, July 1998.

[5] Makoto Matsumoto and Takuji Nishimura. Mersenne Twister: A 623-dimensionally equidistributed uniform pseudorandom number generator. ACM Transactions on Modeling and Computer Simulation, 8(1):3–30, 1998.

[6] Jeroen Idserda. TCP/IP modeling in OMNeT++. B-Assignment, University of Twente, the Netherlands, July 2004.

[7] NetBSD, http://www.netbsd.org.

[8] Roland Bless and Mark Doll. Integration of the FreeBSD TCP/IP-Stack into the Discrete Event Simulator OMNeT++. Proc. of the 2004 Winter Simulation Conference.

[9] Andreas Willig, Martin Kubisch, Christian Hoene, and Adam Wolisz. Measurements of a Wireless Link in an Industrial Environment using an IEEE 802.11-Compliant Physical Layer. IEEE Transactions on Industrial Electronics, 2001.

[10] David Eckhard and Peter Steenkiste. Measurement and analysis of the error characteristics of an in-building wireless network. In Proc. of ACM SIGCOMM'96 Conference, pages 243-254, Stanford University, California, August 1996.

[11] E. N. Gilbert. Capacity of a burst-noise channel. Bell Systems Technical Journal, 39:1253-1265, September 1960.

[12] E. O. Elliot. Estimates of error rates for codes on burst-noise channels. Bell Systems Technical Journal, 42:1977-1997, September 1963.

[13] Almudena Konrad, Ben Y. Zhao, Anthony D. Joseph, Reiner Ludwig. A Markov-Based Channel Model Algorithm for Wireless Networks. Wireless Networks 9, 189-199, 2003.

[14] David D. Clark. Window and Acknowledgment Strategy in TCP. RFC 813, July 1982.

[15] V. Jacobson, R. Braden, D. Borman. TCP Extensions for High Performance. RFC 1323, May 1992.

[16] M. Mathis, J. Mahdavi, S. Floyd, A. Romanow. TCP Selective Acknowledgment Options. RFC 2018, October 1996.

[17] J. Postel. TCP Maximum Segment Size and Related Topics. RFC 879, November 1983.

[18] S. Floyd, T. Henderson. The NewReno Modification to TCP's Fast Recovery Algorithm. RFC 2582, April 1999.

[19] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. Laurence Berkeley National Laboratory.

[20] Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specification. ANSI/IEEE Std 802.11, 1999 Edition (R2003).

[21] Victor Shnayder, Mark Hempstead, Bor-rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the Power Consumption of a Large-Scale Sensor Network. Harvard University.