How Fast is MQTT?

Statistical Model Checking and Testing of IoT Protocols

Bernhard K. Aichernig and Richard Schumi

Institute of Software Technology, Graz University of Technology, Austria
{aichernig, rschumi}@ist.tugraz.at

Abstract. MQTT is one of the major messaging protocols in the Internet of things (IoT). In this work, we investigate the expected performance of MQTT implementations in various settings. We present a model-based performance testing approach that allows a fast simulation of specific usage scenarios in order to perform a quantitative analysis of the latency. Out of automatically generated log-data, we learn the distributions of latencies and apply statistical model checking to analyse the functional and timing behaviour. The result is a novel testing and verification technique for analysing the performance of IoT protocols. Two well-known open source MQTT implementations are evaluated and compared.

Keywords: statistical model checking, model-based testing, performance, latency, Internet of things, MQTT, Mosquitto, emqtt.

1 Introduction

With the growing popularity of the Internet of Things (IoT), the quality of its underlying infrastructure moves into focus. In particular, the software needs special attention, since it is often exempted from any warranty. In this work, we are investigating implementations of the Message Queuing Telemetry Transport (MQTT) protocol, one of the major machine-to-machine messaging protocols of the IoT. MQTT follows a publish-subscribe pattern and allows clients, e.g., sensors in a smart home, to distribute messages via a central server, called the *broker* [7]. Recently, we have found 18 protocol violations in four open-source MQTT brokers [30]. In this work, we concentrate on performance.

In contrast to previous performance studies [13,17,20,31], we present a statistical model checking (SMC) approach that is able to (1) predict the expected performance on a model, and (2) to verify the prediction on a real system. SMC [21] is a verification method that can answer both, quantitative and qualitative questions. The questions are expressed as properties of a stochastic model which are checked by analysing simulations of this model.

Our method is realised with a property-based testing (PBT) tool that performs the data generation for learning the distributions of broker latencies, the model simulation, and the verification of the system. PBT is a random testing technique that tries to falsify a given property. Properties describe the expected

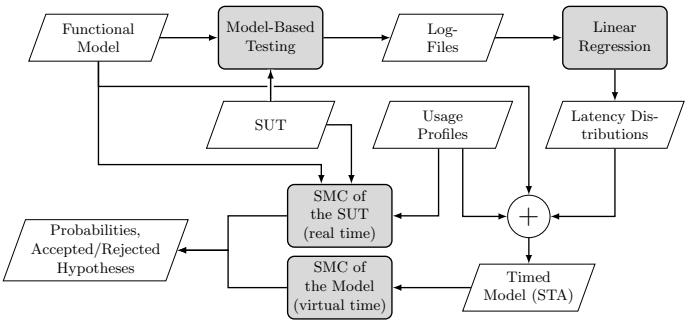

Fig. 1: Overview of the data flow of our method.

behaviour of a system-under-test (SUT) and may be algebraic or model-based. A PBT tool generates inputs and checks if the property holds.

Previously, we had integrated SMC into a PBT tool [3] in order to check stochastic models as well as implementations. With this technique, we checked the expected response-time of an industrial web application [29]. Based on this previous work, we present a method that statistically verifies the latencies of MQTT brokers from a client's perspective.

Figure 1 illustrates our method: (1) we automatically test a broker from multiple clients and record its latencies in log-files. For every client, we run a model-based testing process concurrently and generate test cases from a functional model. (2) We derive latency distributions via linear regression. Since the latency is influenced by the parallel activity on the server, the distributions are parametrised by the number of active clients. These latency distributions are added to the functional model resulting in a stochastic timed automata (STA) [6] model. (3) For simulating the behaviour of real MQTT clients, we add usage profiles, containing probabilities and waiting times related to messages. (4) We perform SMC on the resulting stochastic model in order to answer the question "What is the probability that the message latency is under a certain threshold?". This process can be accelerated by using a virtual time scale, i.e., a fraction of real time. (5) We check if the predicted performance hypothesis holds on the SUT. This is achieved via a statistical hypothesis test (another SMC algorithm). For this final test, less samples than for forming the hypothesis are needed, and hence, this SMC step scales well, although carried out under real time with real delays.

Related Work. In contrast to our work, classical load testing methods analyse the performance directly on the SUT. Models are mostly used for test-case generation [5] and for modelling user populations [14,28]. Others focus solely on simulation on the model-level [8,9,11,23]. In our work, we exploit the models for both, testing a system as well as simulating the performance on the model.

The most related tool is UPPAAL SMC [10], because it supports SMC and test-case generation. However, our use of PBT facilitates the definition of specialized generators for complex test-data, which is important for load testing. Furthermore, modelling in a programming language may be more acceptable to programmers and testers.

The performance of MQTT implementations has been tested in the past [13,20,31], but without constructing a performance model for simulating MQTT under different usage scenarios. The most similar work to ours [17] modelled MQTT with probabilistic timed automata and checked performance with SMC. However, they did not validate their model against real implementations, and hence, it did not include real timing behaviour.

To the best of our knowledge our work is novel: we are the first who apply SMC to the performance analysis of MQTT brokers with learned latency distributions, and who check the results from the model against real MQTT brokers by performing hypothesis testing.

Contributions. This research builds upon our previous work [29], where we introduced our method and applied it to an industrial web-service application. Here, we present the following novel contributions: (1) the evaluation of our method for another application domain, namely for protocol testing. This demonstrates the generality of our method. (2) We present a comparative evaluation of two MQTT implementations. This shows that our method is also able to compare the performance of different systems and helps to choose the right one, depending on a specific usage scenario. (3) This is the first SMC approach for MQTT that also supports a direct verification of the results by testing real MQTT brokers. (4) We release the source of our tool in order to make our method available to the public and to facilitate the reproduction of our results.[1]

Structure. First, Sect. 2 introduces the background of SMC and PBT based on our previous work [3]. Next, in Sect. 3 we give an example and demonstrate our method. Then, Sect. 4 presents an evaluation with two open-source MQTT implementations. Finally, we conclude in Sect. 5.

2 Background

2.1 Statistical Model Checking (SMC)

SMC is a verification method for checking qualitative and quantitative properties of a stochastic model. These properties are usually defined with (temporal) logics. In order to answer questions, like "What is the probability that the model satisfies a property?" or "Is the probability that the model satisfies a property above or below a certain threshold?", a statistical model checker produces samples, i.e. random walks on the stochastic model and checks whether the property holds for these samples. Various SMC algorithms are applied in order to compute the total number of samples needed to find an answer for a specific question, or to compute a stopping criterion. This criterion determines when we can stop sampling, because we have found an answer with a required certainty. In this work, we focus on the following algorithms common in the SMC literature [21,22].

Monte Carlo simulation with Chernoff-Hoeffding bound. The algorithm computes the required number of simulations n in order to estimate the probability γ that a stochastic model satisfies a Boolean property. The procedure

[1] `https://github.com/schumi42/mqttCheck`

is based on the Chernoff-Hoeffding bound [16] that provides a lower limit for the probability that the estimation error is below a value ϵ. Assuming a confidence $1-\delta$ the required number of simulations is $n \geq 1/(2\epsilon^2)\ln(2/\delta)$.

The n simulations represent Bernoulli random variables $X_1, \ldots, X_n$ with outcome $x_i = 1$ if the property holds for the i-th simulation run and $x_i = 0$ otherwise. Let the estimated probability be $\bar{\gamma}_n = (\sum_{i=1}^{n} x_i)/n$, then the probability that the estimation error is below ϵ is greater than our required confidence. Formally we have: $Pr(|\bar{\gamma}_n - \gamma| \leq \epsilon) \geq 1-\delta$. After the calculation of the number of required samples n, a standard Monte Carlo simulation is performed [22].

Sequential Probability Ratio Test (SPRT). This sequential method [32] is a form of hypothesis testing that can answer qualitative questions. Given a random variable X with a probability density function $f(x,\theta)$, we want to decide, whether a null hypothesis $H_0 : \theta = \theta_0$ or an alternative hypothesis $H_1 : \theta = \theta_1$ is true for desired type I and II errors (α, β). In order to make the decision, we start sampling and calculate the log-likelihood ratio after each observation of x_i:

$$\log \Lambda_m = \log \frac{p_1^m}{p_0^m} = \log \frac{\prod_{i=1}^{m} f(x_i,\theta_1)}{\prod_{i=1}^{m} f(x_i,\theta_0)} = \sum_{i=1}^{m} \log \frac{f(x_i,\theta_1)}{f(x_i,\theta_0)}$$

We continue sampling as long as the ratio is inside the indifference region $\log \frac{\beta}{1-\alpha} < \log \Lambda_m < \log \frac{1-\beta}{\alpha}$. H_1 is accepted when $\log \Lambda_m \geq \log \frac{1-\beta}{\alpha}$, and H_0 when $\log \Lambda_m \leq \log \frac{\beta}{1-\alpha}$ [15].

In this work, we form a hypothesis about the expected latencies with the Monte Carlo method on the model. Then, we check with SPRT if this hypothesis holds on the SUT. This is faster than running Monte Carlo directly on the SUT.

2.2 Property-Based Testing (PBT)

PBT is a random-testing technique that aims to check the correctness of properties. A property is a high-level specification of the expected behaviour of a function- or system-under-test that should always hold. With PBT, inputs can be generated automatically by applying data generators, e.g., a random list generator. The inputs are fed to the function or system-under-test and the property is evaluated. If it holds, then this indicates that the function or system works as expected, otherwise a counterexample is produced.

One of the key features of PBT is its support for model-based testing. Models encoded as extended finite state machines (EFSMs) [19] can serve as source for state-machine properties. An EFSM is a 6-tuple (S, s_0, V, I, O, T). S is a finite set of states, $s_0 \in S$ is the initial state, V is a finite set of variables, I is a finite set of inputs, O is a finite set of outputs, T is a finite set of transitions. A transition $t \in T$ can be described as a 6-tuple (s_s, i, g, op, o, s_t), s_s is the source state, i is an input, g is a guard, op is a sequence of assignment operations, o is an output, s_t is the target state [19].

In order to derive a state-machine property from an EFSM, we have to write a specification comprising the initial state, commands and a generator for the

next transition given the current state of the model. Commands encapsulate (1) preconditions that define the permitted transition sequences, (2) postconditions that specify the expected behaviour and (3) execution semantics of transitions for the model and the SUT. A state-machine property states that for all permitted transition sequences, the postcondition must hold after the execution of each command [18,25]. Simplified, such properties can be defined as follows:

$$cmd.runModel, cmd.runActual : S \times I \to S \times O$$
$$cmd.pre : I \times S \to Boolean, cmd.post : (S \times O) \times (S \times O) \to Boolean$$
$$\forall s \in S, i \in I, cmd \in Cmds :$$
$$cmd.pre(i, s) \implies cmd.post(cmd.runModel(i, s), cmd.runActual(i, s))$$

We have two functions to execute a command on the model and on the SUT: *cmd.runModel* and *cmd.runActual*. The precondition *cmd.pre* defines the valid inputs for a command. The postcondition *cmd.post* compares the outputs and states of the model and the SUT after the execution of a command.

PBT is a powerful testing technique that allows a flexible definition of generators and properties via inheritance or composition. The first implementation of PBT was QuickCheck for Haskell [12]. Numerous reimplementations followed for other programming languages. We use FsCheck[2] for C#.

2.3 Stochastic Timed Automata

Several probabilistic extensions of timed automata [4] have been proposed. Here, we follow the definition of stochastic timed automata (STA) by Ballarini et al. [6]: an STA is a tuple $(L, l_0, A, C, I, E, F, W)$ comprising a classical timed automaton (L, l_0, A, C, I, E), probability density functions (PDFs) $F = (f_l)_{l \in L}$ for the sojourn time, and natural weights $W = (w_e)_{e \in E}$ for the edges. L is a finite set of locations, $l_0 \in L$ is the initial location, A is a finite set of actions, C is a finite set of clocks with valuations $u(c) \in \mathbb{R}_{>0}$, $I : L \mapsto \mathcal{B}(C)$ is a finite set of invariants for the locations and $E \subseteq L \times A \times \mathcal{B}(C) \times 2^C \times L$ is a finite set of edges between locations, with an action, a guard and a set of clock resets.

The transition relation can be described as follows. For a state given by the pair (l, u), where l is a location and u a clock valuation $u \in C \to \mathbb{R}_{\geq 0}$, the PDF f_l is used to choose the sojourn time d, which changes the state to $(l, u + d)$, where we lift the plus operator to the clock valuation as follows: $u + d =_{def} \{c \mapsto u(c) + d \,|\, c \in C\}$. After this change, an edge e is selected out of the set of enabled edges $E(l, u + d)$ with the probability $w_e / \sum_{h \in E(l,u+d)} w_h$. Then, a transition to the target location l' of e and $u' = u + d$ is performed. For our models the underlying stochastic process is a semi-Markov process, since the clocks are reset at every transition, but we do not assume exponential waiting times, and therefore, the process is not a standard continuous-time Markov chain.

2.4 Integration of SMC into PBT

Recently, we have demonstrated that SMC can be integrated into a PBT tool in order to perform SMC of PBT properties [3]. With this approach, we can

[2] `https://fscheck.github.io/FsCheck`

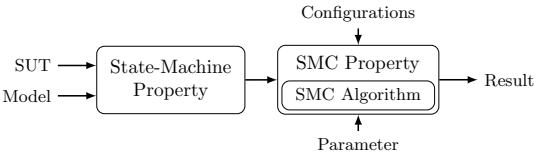

Fig. 2: Data flow diagram of an SMC property.

verify stochastic models, like in classical SMC, as well as stochastic implementations. For the integration, we introduced our own new SMC properties that take a PBT property, configurations for the PBT execution, and parameters for the specific SMC algorithm as input. Then, our properties perform an SMC algorithm by utilizing the PBT tool as simulation environment and they return either a quantitative or qualitative result, depending on the algorithm. Figure 2 illustrates how we evaluate a PBT state-machine property within an SMC property.

Algorithm 1 shows pseudo code of an SMC property for the SPRT (see Sect. 2.1). The inputs of this algorithm are a PBT property, configurations for PBT, probabilities for H_0/H_1 and the type I and type II error parameters α, β. The algorithm produces samples (Line 3) and calculates the log likelihood ratio (Line 4 & 6) repeatedly, until we are outside the indifference region that is defined by α and β (Line 7). Finally, when we are outside the indifference region, we return H_1 as result, when the ratio is below the lower bound and H_0 otherwise. Our integration method can, e.g., be applied for a statistical conformance analysis by comparing an ideal model to a stochastic faulty implementation or it can also simulate a stochastic model. In this work, we apply it for a performance analysis with the model and for the verification of real brokers.

3 Method

In this section, we show how we derive timed models from logs and how we can apply these models to simulate stochastic usage profiles. The description follows the steps from the overview in Fig. 1.

Model-Based Testing. Our SUT is an MQTT broker that allows clients to connect/disconnect, subscribe/unsubscribe to topics and publish messages for such topics. Each of these actions can be performed with a corresponding control message, which is defined by the MQTT standard [7]. We treat the broker as a black box and test it from a client's perspective.

Algorithm 1 Pseudo code of a SPRTProperty.

Input: *prop*: PBT property for producing a sample, *config*: configuration for checking the property with PBT, p_0, p_1: probabilities for H_0 and H_1 α, β: type I and type II error parameters

1: $ratio \leftarrow 0$
2: **do**
3: **if** $prop.Check(config)$ **then** ▷ produces sample and checks result of PBT property
4: $ratio \leftarrow ratio + log(\frac{p_1}{p_0})$ ▷ calculate the log likelihood ratio
5: **else**
6: $ratio \leftarrow ratio + log(\frac{1-p_1}{1-p_0})$ ▷ calculate the log-likelihood ratio
7: **while** $\log \frac{\beta}{1-\alpha} < ratio \wedge ratio < \log \frac{1-\beta}{\alpha}$ ▷ stop when threshold was reached
8: **if** $ratio \geq \log \frac{1-\beta}{\alpha}$ **then**
9: **return** H_1 ▷ H_1 is accepted
10: **else**
11: **return** H_0 ▷ H_0 is accepted

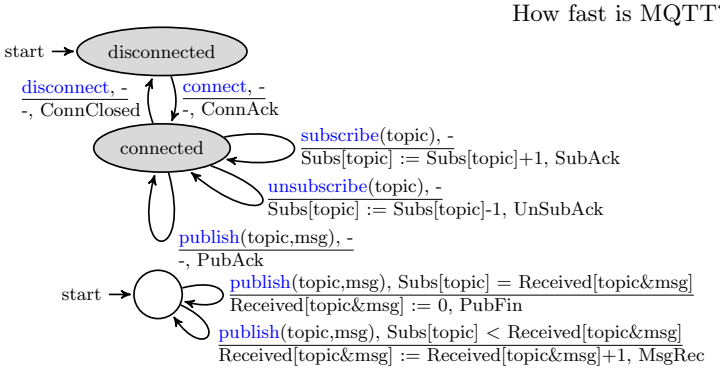

Fig. 3: Functional model for an MQTT client.

The upper state machine in Fig. 3 represents the messages that we test. We run multiple of these state machines concurrently, in order to produce log-data that includes latencies for simultaneous messages of several clients. Each transition of the state machine is labelled with an input i, an optional guard g / assignment operations op, and an output o. Some transition inputs are parametrised with generated data, e.g., a topic for subscribe. We apply PBT generators in order to produce inputs and their required data. Previously, we have demonstrated the data generation for such functional models and also model-based testing [1,2]. To keep it simple, we assume that a client can only subscribe to topics that it did not subscribe to before (the same for unsubscribe).

In order to manage the subscriptions, we have a global map *Subs* that stores the subscription numbers for each topic. This map is needed when publishing, because we want to check if the number of received messages corresponds to the number of subscribed clients. In order to perform this check, we have a second state machine (Fig. 3 bottom) that represents the message receivers. This machine stores the number of received messages in a map *Received* that takes the topic concatenated with the message *(topic&msg)* as key. The map is updated for each message receiver, and when all messages were delivered, then a *PubFin* output is produced. For simplicity, we omit some assignment operations, e.g., for a subscriptions set.

Based on this functional model, we perform model-based testing with a PBT tool, which generates random test cases that are executed on an MQTT broker. During this testing phase, we capture the latencies of messages in a log-file. Note that the latency is the duration that a client must wait until it receives a response to a sent message from the broker or until the message is delivered to all receivers in case of a publish.

A simplified log excerpt from the MQTT implementation Mosquitto is presented in Table 1. It shows that we record the message type (*Msg*), the number of active clients resp. open message exchanges (*#ActiveMsgs*), the total number of

Table 1: Example log-data of one client for Mosquitto.

Msg	**#ActiveMsgs**	**#TotalSubs**	**TopicSize**	**MsgSize**	**#Subs**	**#Receivers**	**Latency[ms]**
connect	47	266	0	-	-	-	110.82
subscribe	47	270	14	-	-	-	2.45
publish	47	270	14	52	7	7	32.72
unsubscribe	45	12	14	-	-	-	1.25
publish	46	272	14	74	1	1	2.13

subscriptions (*#TotalSubs*), the size of the topic (*TopicSize*) and message string (*MsgSize*), the number of subscribers for a topic when a *publish* occurs (*#Subs*), the number of receivers of a published message (*#Receivers*), and the latency. For this initial logging phase, the available transitions in the current state of the functional model (Fig. 3 top) are chosen with a uniform distribution. In the *disconnected* state, the only choice is a *connect* message and in the *connected* state all other messages are selected with equal frequency. We do not apply any sojourn times in this phase, since we want to capture the latencies for many concurrent messages.

Linear Multiple Regression. In previous work, we showed that linear regression can be applied for learning response-time distributions of a web application [29]. Now, we learn latency distributions with this method.

Linear regression produces a regression model that describes the relationships of the log-variables (or features) with the target variable and can be applied for the prediction of the target variable. The quality of the regression model can be measured with the coefficient of determination (R^2-score) [24], which defines how well a prediction model for regression fits given data.

First, we checked if we can find any bias in our logs, e.g., a bias might be caused by log-data generation that is not random enough. In this case, we could obtain an artificial correlation between features. Another problem might be that the log-data generation might be unintentionally set up in a way, where relevant scenarios for the prediction were not tested frequently enough. Both these issues can result in a regression model that has a good R^2-score, but it would not produce reliable predictions for our simulation with SMC. In order to reduce the risk of such biases, it is helpful to carefully analyse the data with visualisations, like scatter plots, histograms or correlation matrices. For example, if a correlation matrix shows correlations that should not be there, then this might indicate a problem during the initial test-case generation.

In the next step, the *data cleaning*, these data visualisations also helped to find issues with the data. Here, log-entries with disproportionately long latencies, i.e. outliers, are removed. We consider the top 5% of the entries per message type as outlier. Moreover, we flag and remove entries where exceptions were raised, e.g., due to time-outs or connection failures, since they are rare and we are primarily interested in latencies of successful message exchanges.

Next, comes the *feature selection*, where we select variables that have a significant influence on the target variable. We can also apply the correlation matrices and look for features that are correlated with the target variable. The correlation can be measured with a correlation coefficient r, e.g., a common one was introduced by Pearson [26] and gives us a value $r \in [-1, 1]$, where 1 is a total positive correlation and -1 a negative correlation. Features that have a medium or strong correlation $r \geq 0.3$ are most important for the regression, but sometimes also features with a weak correlation $0.1 < r \leq 0.3$ can help to improve the regression model. In addition, most regression tools show what features are relevant for the regression, which we will see later. Note, we should avoid features that have a high correlation among each other, since they might be redundant. For

example, the number of subscribers to a topic of a published message is highly correlated with the number of message receivers. Hence, we only select one of these. Additionally, we checked if certain features only have an effect on specific message types, which can, e.g., be resolved by setting these features to zero for this message. The selected features can be seen in our regression formula:

$$Latency \sim Msg + \#ActiveMsgs + \#TotalSubs + \#Subs$$

We performed the regression in R with the standard *lm* function.[3] It was performed with log-data from Mosquitto, which contained 100 test cases with a random number of clients (3–100) and a length of 50 messages. This produced log-files with about 300,000 entries. The required number of test cases was determined by stepwise increasing the dataset and by executing the regression, until there was no more increase in the R^2-score.

Listing 1.1 shows the results of the linear multiple regression. We are mainly interested in the first three columns of this listing. The first column shows the intercept and the regression coefficients. The intercept is the mean of the latency, when all features are zero and the coefficients come from the features. For categorical variables, e.g., the message type *Msg*, we have multiple coefficients. In the second column, there is the estimate of the mean and the third column shows the standard error that gives the average variation of the estimate from the actual average value. Note that the *** at the end of each line, shows that the variables are all highly significant. For more details, see [27].

```
1                 Estimate   Std.Error   t value   Pr(>|t|)
2  (Intercept)   -8.009707   0.1106356  -72.397   < 2e-16 ***
3  Msgdisconnect  8.084679   0.1234019   65.515   < 2e-16 ***
4  Msgpublish     9.066681   0.1395017   64.993   < 2e-16 ***
5  Msgsubscribe   8.771242   0.1419899   61.774   < 2e-16 ***
6  Msgunsubscribe 9.294850   0.1294843   71.784   < 2e-16 ***
7  #ActiveMsgs    1.358794   0.0033433  406.417   < 2e-16 ***
8  #TotalSubs     0.002503   0.0002084   12.011   < 2e-16 ***
9  #Subs          0.294270   0.0307663    9.565   < 2e-16 ***
```

Listing 1.1: Excerpt of the linear regression output.

In order to apply this regression model in our method, we encode it in a *delay* function that takes the message type, the number of active messages, the total number of subscribers, and the number of subscribers for the currently published message as input and returns the parameters μ and σ of the normal distribution as result:

$$delay : Msg \times \mathbb{N}_{>0} \times \mathbb{N}_{\geq 0} \times \mathbb{N}_{\geq 0} \to \mathbb{R} \times \mathbb{R}$$

In this function, we perform a linear combination of the distributions given by the estimates and standard errors of the associated regression coefficients for the inputs. This gives us a combined normal distribution that depends on the inputs of this function. For example, for a *subscribe* message that happens when 15 other messages are active and when there are zero subscribers, the linear combination works as follows. The associated regression coefficients of Listing 1.1 (Lines 2, 5 & 7) are combined in order to obtain parameters for a normal distribution $\mu = -8.010 + 8.771 + 15 \times 1.359$ and $\sigma = \sqrt{0.111^2 + 0.142^2 + (15 \times 0.003)^2}$. In the next phase, we integrate the *delay* function that calculates these parameters into the functional model.

[3] `https://www.r-project.org/`

```
MinTimeBetwMsg: 0, MaxTimeBetwMsg: 500,
MsgWeights:{connect: 1, disconnect: 1, publish: 5, subscribe: 3, unsubscribe: 2}
```

Listing 1.2: Usage profile UP1 with time bounds and weights for messages.

Statistical Model Checking. For the evaluation of the model, we introduce usage profiles that describe the behaviour of an MQTT client, i.e. how long it should wait between sending messages, and with what probabilities it should send certain messages. An example usage profile (UP1) is shown in Listing 1.2. The time between messages is selected uniformly inside the bounds [MinTimeBetwMsg, MaxTimeBetwMsg] and we have weights that define the message frequency. (In specific settings, it may make sense to create different usage profiles for certain types of components, e.g., a sensor might only publish messages.)

This usage profile is added to the functional model and also the learned latency distributions (expressed in the *delay* function) are integrated. This gives us a combined timed model in the form of a stochastic timed automaton, as explained in Sect. 2.3 and illustrated in Fig. 4. In this model, all locations have probability density functions f_l for the sojourn time. The *connected* and *disconnected* locations have a uniform distribution given by an upper and lower bound $[a, b]$. These bounds come from our usage profile. All other locations have a normal distribution for the sojourn time. The parameters for this distribution are computed by the delay function. In contrast to the functional model, these additional locations apply the message latencies. The locations have one incoming edge that represents sending a message and an outgoing edge for the response. Moreover, the weights w_e from our usage profile are added to the transitions for sending messages. Note that we have omitted the parameters of the delay function and also some assignments that are necessary for these parameters, in order to keep the figure more readable.

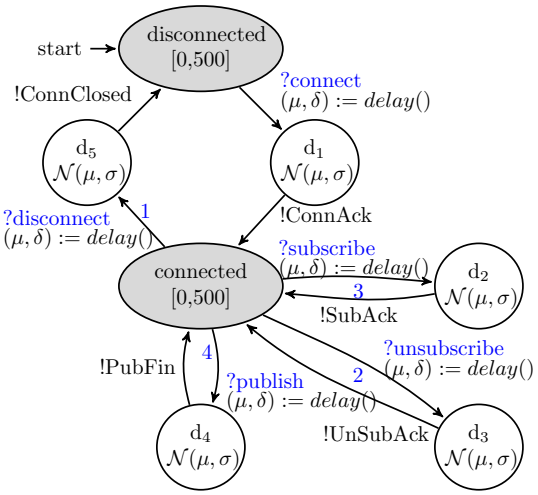

Fig. 4: Stochastic timed automaton for the timing behaviour of an MQTT client.

With this model, we can evaluate a usage profile by simulating the expected latencies. Moreover, we can simulate a complete MQTT setup by running multiple models concurrently. A run of the model can be defined as: $(l_0, u_0) \xrightarrow{d_1,a_1} (l_1, u_1) \xrightarrow{d_2,a_2} \ldots$ and it produces a timed trace in the form $(d_1, a_1), (d_2, a_2), \ldots$, where d_i is a delay and $a_i \in A$. An example trace may look like this:

(97,connect),(9,ConAck),(344,subscribe)(24,SubAck),(58,subscribe),(64,Suback)

While we execute the model, we can check properties to answer questions, like "What is the probability that the latency of each interaction of a client within a given MQTT setup is under a certain threshold?". In order to estimate the probability of such properties, we perform a Monte Carlo simulation with Chernoff-Hoeffding bound. This evaluation requires too many samples to be efficiently executed on the SUT, and hence, we only run it on the model. For example, checking the probability that the latency threshold of 50 ms is satisfied for each client of an MQTT setup with 130 clients with parameters $\epsilon = 0.05$ and $\delta = 0.01$, requires 1060 samples and returns a probability of 0.84, when a test-case length of ten is considered.

Fortunately, the SPRT requires fewer samples, and is therefore, better suited for the evaluation of the SUT. The probability that was computed on the model serves as a hypothesis to be checked on the model, i.e. we check if the SUT is at least as good as predicted. We consider the predicted probability 0.84 as alternative hypothesis and select a probability of 0.74 as null hypothesis, which is 0.1 smaller, because we want to be able to reject the hypothesis that the SUT has a smaller probability. (We select a difference of 0.1, because for this difference our model prediction was close enough to the actual probability of the SUT in most cases, and a smaller difference would need too many samples for an efficient evaluation of the SUT.) By running the SPRT (with 0.01 as type I and II error parameters) for each client, we can check these hypotheses. The alternative hypothesis (probability 0.84) was accepted for all clients and on average 41.15 samples (test cases) were needed for the decision.

Implementation. Our method was implemented in a similar way, as described in our previous work [29], where we illustrated how timed models can be executed with PBT. Previously, we introduced custom generators for the simulation of response times, which work in a similar way for latencies. Moreover, we demonstrated the application of user profiles that work in the same way as our usage profiles, and we presented a test-case generation algorithm for PBT that can perform the initial model-based testing phase as well as the execution of our timed model. For brevity, we omit the details of the implementation and refer to our previously mentioned published source code.

4 Evaluation

We evaluated our method by applying it to two open-source MQTT implementations: Mosquitto 1.4.15 and emqtt 2.3.5, running with quality of service level one and with the default configurations. We analyse the needed number of samples and the run times. MQTT implementations typically have various settings, e.g., the length of the in-flight message queue or an option to group together TCP packets (Nagle's algorithm). The influence of such settings might be a potential threat to the validity of our comparison. We worked with the default settings as this is commonly done and we also tried to adapt the mentioned settings to face this threat. A comparison of the regression models and response-time visualisations did not show a difference for the adapted settings. Note that Nagle's

```
MinTimeBetwMsg: 50, MaxTimeBetwMsg: 250,
MsgWeights:{connect: 1, disconnect: 1, publish: 7, subscribe: 1, unsubscribe: 1}
```

Listing 1.3: Usage profile UP2 with more frequent publish messages.

algorithm has no effect, because it only groups messages if acknowledgements are pending. This situation does not occur, since our tests are synchronous, i.e. we always wait for an acknowledgement before sending a new message.

The evaluation was performed on a Windows server (version 2008 R2) with a 2.1 GHz Intel Xeon E5-2620 v4 CPU with 8 Cores and 32 GB RAM. This machine was running the clients and the broker in order to avoid an influence of the network. However, a possible influence of the client processes on the broker might cause a threat to validity of our evaluation. To face this issue, we measured the CPU load, to make sure that it is no bottle neck. During the evaluation, the CPU load was below 60% most of the time, and there were only some rare peaks, where the CPU was over 90%. We also tried to increase the priority of the broker process, but this showed no difference. The RAM usage of the brokers was insignificant since the total RAM of the servers was more than enough.

We applied Visual Studio 2012 with .NET framework 4.5, NUnit 2.64, and FsCheck 2.92 in order to run the tests and for SMC. The library M2Mqtt[4] served as a client interface to facilitate the interaction with the brokers.

We follow the method of Sect. 3, in order to answer the question "What is the probability that the message latency is under a certain threshold?". Hence, we check the probability that all messages within a sequence of ten messages for all clients of a selected MQTT setup have a latency under this threshold. We perform the analysis as shown in Sect. 3, with the difference that we test Mosquitto and emqtt, and we check various thresholds and different numbers of clients. We apply the same usage profile as before and the regression model for emqtt was similar to the one shown for Mosquitto in Listing 1.1. Additionally, we evaluate another usage profile (UP2), as shown in Listing 1.3 that has a higher weight for *publish* messages and different bounds for the time between messages.

As shown before, we apply a Monte Carlo simulation with Chernoff-Hoeffding bound with parameters $\epsilon = 0.05$ and $\delta = 0.01$, which requires 1060 samples per data point, to evaluate the timed model. The results for Mosquitto and emqtt for both user profiles are shown in Fig. 5 and Fig. 6. Table 2 shows the average time needed for these evaluations.

As expected, a decrease in the probability can be observed, when the number of clients increases, and a higher threshold causes a higher probability. The advantage of applying SMC on a model is that it runs much faster than on the SUT. With a virtual time of 1/10 of the actual time, we can perform evaluations that would take hours on the SUT within minutes.

It is also important to check the probabilities that we received through SMC of the timed model, on the SUT. This was done as explained in Sect. 3 with the SPRT.

Table 2: Average time [min:s] for the Monte Carlo simulation of the model.

Number of Clients	50	70	90	110	130	150
UP1 Mosquitto	4:27	4:48	4:54	5:00	5:09	5:25
UP1 emqtt	4:28	4:49	4:57	5:05	5:15	5:23
UP2 Mosquitto	2:39	3:03	3:16	3:22	3:40	3:51
UP2 emqtt	2:39	3:02	3:18	3:27	3:41	3:55

[4] `https://m2mqtt.wordpress.com`

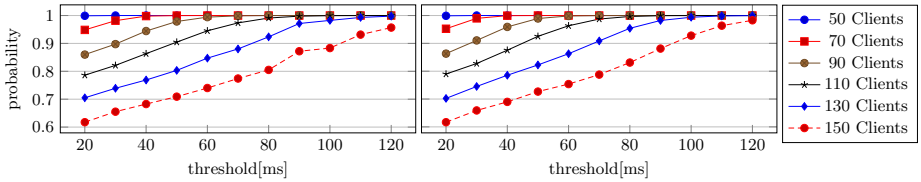

Fig. 5: UP1 Monte Carlo simulation results for Mosquitto (left) & emqtt (right).

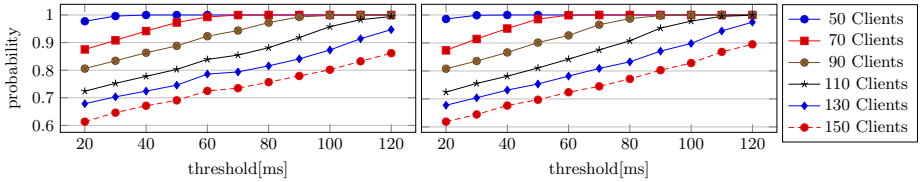

Fig. 6: UP2 Monte Carlo simulation results for Mosquitto (left) & emqtt (right).

Table 3 and Table 4 show the results for both usage profiles and brokers. We focused on some of the more interesting data points for the evaluation. The tables show hypotheses, test results, the needed number of samples and execution times for different numbers of clients and thresholds. Note that in order to obtain an average number of needed samples, we run the SPRT concurrently for each client and calculate the average of these runs.

In most cases, hypothesis H_1 was accepted for almost all clients, which means that the probability of the SUT was at least as high, as the predicted one from the model. However, the prediction was not always accurate. H_0 was also sometimes accepted and in some cases H_1 was only accepted by a fraction of the clients that tested this hypothesis, e.g., for Mosquitto with threshold 30 ms and 90 clients, only 60% of the clients accepted H_1 for UP1. The prediction was sometimes inaccurate for small latency thresholds. The reason might be that we mainly learned the latency distributions under conditions with high load, and hence, our model might not be completely accurate for small latencies.

Moreover, the prediction performed rather poorly for high numbers of clients ($\geq$ 130), especially for UP2. This might be caused by the fact that the initial testing phase for log-data had only a maximum of 100 clients and the higher number of clients might be too far away from this initial test phase. However, H_1 was still accepted for most data points, which means that the model was good enough in these cases. Furthermore, it is apparent that the SPRT can be performed with fewer samples, i.e. we need mostly about 50 samples (except for some outliers), compared to the 1060 for the Monte Carlo simulation.

By comparing the results of Mosquitto and emqtt, it can be seen that predicted probabilities are too similar to make a clear distinction. However, the evaluation of the SUT with hypothesis testing was able to find some differences, i.e. in some cases emqtt showed a slightly better performance. For example, the second data row of Table 3 shows that Mosquitto was not able to accept H_1, where emqtt accepted it, although the same hypotheses were tested. This means that emqtt had a better performance in this case. For UP1, this was the case especially for small thresholds, for UP2 the performance was more similar for

Table 3: Results of the evaluation of the SUT with the SPRT for UP1.

Threshold	#Clients	Mosquitto					emqtt				
		H_0	H_1	Result	#Samples	Time[min:s]	H_0	H_1	Result	#Samples	Time[min:s]
30	50	0.9	1	H_1	44	2:31	0.9	1	H_1	44	2:28
30	70	0.88	0.98	H_0	22.47	5:43	0.88	0.98	H_1	44.14	2:51
30	90	0.79	0.89	60% H_1	276.31	39:12	0.8	0.9	H_1	41.02	2:56
30	110	0.74	0.84	H_1	73.26	7:22	0.72	0.82	H_1	42.55	3:40
30	130	0.68	0.78	H_0	46.68	11:33	0.64	0.74	H_1	77.92	9:21
50	50	0.9	1	H_1	44	2:10	0.9	1	H_1	44	2:06
50	70	0.9	1	73% H_1	43.53	10:01	0.9	1	H_1	44	2:09
50	90	0.88	0.98	H_1	50.47	4:18	0.88	0.98	H_1	43	2:30
50	110	0.8	0.9	H_1	41.35	3:19	0.84	0.94	H_1	41.25	2:50
50	130	0.74	0.84	H_1	41.15	3:12	0.75	0.85	H_1	38.41	2:37
70	50	0.9	1	H_1	44	2:04	0.9	1	H_1	44	2:33
70	70	0.9	1	H_1	44	2:10	0.9	1	H_1	44	2:08
70	90	0.9	1	H_1	44	2:37	0.9	1	H_1	44	2:29
70	110	0.88	0.98	H_1	43.16	2:57	0.89	0.99	H_1	44.38	3:14
70	130	0.78	0.88	H_1	39.32	3:00	0.83	0.93	H_1	41.21	2:37

both implementations and there is also a case where Mosquitto showed better performance. (Row 13 of Table 4, shows that only 90% of the clients accepted H_1 for emqtt, but all clients for Mosquitto.)

We analysed the execution times of the different phases of our method. The initial testing phase took about 5–8 min and the linear regression about 10-12 s. Note that these two phases have to be performed only once, and the resulting model can then be applied for various evaluations.

A Monte Carlo simulation of the model required about 3–5 min for 1060 samples as shown in Table 2. The evaluation of the SUT with hypothesis testing took most of the time 2–4 min, in some cases about 10 min and in only in one case 39 min. Hence, most of our predictions could be tested efficiently in about the same time that was needed to make the prediction with the timed model.

Running a Monte Carlo simulation with 1060 samples directly on the SUT would take approximately 2–3 hours. Performing this simulation becomes quickly impractical when various data points should be analysed. Therefore, our model-based approach makes sense, because it can be executed faster.

Table 4: Results of the evaluation of the SUT with the SPRT for UP2.

Threshold	#Clients	Mosquitto					emqtt				
		H_0	H_1	Result	#Samples	Time[min:s]	H_0	H_1	Result	#Samples	Time[min:s]
30	50	0.9	1	96% H_1	42.88	1:18	0.9	1	96% H_1	43.42	1:16
30	70	0.88	0.98	H_0	17.6	1:15	0.88	0.98	H_1	46.4	2:07
30	90	0.8	0.9	H_0	17.18	3:55	0.8	0.9	H_1	44.98	1:42
30	110	0.72	0.82	H_0	13.43	1:39	0.72	0.82	H_0	14.61	1:14
30	130	0.65	0.75	H_0	14.55	0:56	0.7	0.8	H_0	12.68	0:24
50	50	0.9	1	H_1	44	1:17	0.9	1	H_1	44	1:19
50	70	0.9	1	67% H_1	37.94	3:48	0.9	1	H_1	44	1:44
50	90	0.88	0.98	H_1	51.36	3:01	0.89	0.99	H_1	46.2	1:54
50	110	0.79	0.89	H_1	41.42	1:46	0.81	0.91	87% H_1	152.34	8:34
50	130	0.72	0.82	H_0	16.46	0:58	0.76	0.86	H_0	9.51	0:23
70	50	0.9	1	H_1	44	2:07	0.9	1	H_1	44	1:16
70	70	0.9	1	H_1	44	2:11	0.9	1	H_1	44	1:18
70	90	0.9	1	H_1	46.04	2:41	0.9	1	90% H_1	52.81	2:47
70	110	0.87	0.97	H_1	65.55	4:39	0.88	0.98	H_1	43.82	1:58
70	130	0.79	0.89	H_0	48.18	5:28	0.81	0.91	H_0	9.58	0:34

5 Conclusion

We have shown, how to apply SMC in order to predict the performance of MQTT implementations under various usage scenarios. Moreover, we showed how such predictions can be verified by testing real implementations with the SPRT.

First, we collected log-data by running model-based testing with a functional model. Then, we applied linear regression to learn latency distributions that we integrated into our model. Additionally, we combined this model with usage profiles. The resulting model is a stochastic timed automaton that was simulated to predict the expected latencies of different MQTT implementations. Finally, we verified our prediction with hypothesis testing of the implementations.

A big advantage of our method is that we can predict the performance for various usage scenarios with a fast model simulation and we can efficiently test the prediction on the SUT with the SPRT. The prediction can be accelerated by applying a virtual time that is a fraction of real time, and the test of the SUT is efficient, because it needs fewer samples. Another benefit is that we can do both, SMC and testing of models and SUTs, inside a PBT tool. This enables an easy verification of the model prediction inside the same tool and it facilitates the model and property definition in a high-level programming language.

We have evaluated our method by applying it to well-known open-source implementations of MQTT: Mosquitto and emqtt, and the results were promising. We analysed various numbers of clients and checked the probability that the latency is within certain thresholds. Moreover, we demonstrated that the predicted probability was accurate in most cases and we showed that emqtt has better performance in some cases.

In the future, we plan to evaluate different learning methods for latency distributions and we envisage to test various types of usage profiles.

Acknowledgements. This work was supported by the TU Graz LEAD project "Dependable Internet of Things in Adverse Environments". We are grateful to our colleague Martin Tappler and to the anonymous reviewers for their excellent feedback that helped in improving the quality of the paper.

References

1. Aichernig, B.K., Schumi, R.: Property-based testing with FsCheck by deriving properties from business rule models. In: ICSTW. pp. 219–228. IEEE (2016)
2. Aichernig, B.K., Schumi, R.: Property-based testing of web services by deriving properties from business-rule models. Software & Systems Modeling (Dec 2017)
3. Aichernig, B.K., Schumi, R.: Statistical model checking meets property-based testing. In: ICST. pp. 390–400. IEEE (2017)
4. Alur, R., Dill, D.L.: A theory of timed automata. Theor. Comput. Sci. 126(2), 183–235 (1994)
5. Arts, T.: On shrinking randomly generated load tests. In: Erlang'14. pp. 25–31. ACM (2014)
6. Ballarini, P., Bertrand, N., Horváth, A., Paolieri, M., Vicario, E.: Transient analysis of networks of stochastic timed automata using stochastic state classes. In: QEST. LNCS, vol. 8054, pp. 355–371. Springer (2013)
7. Banks, A., Gupta, R.: MQTT version 3.1.1. OASIS Standard (Dec 2014)

8. Becker, S., Koziolek, H., Reussner, R.H.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software 82(1), 3–22 (2009)
9. Book, M., Gruhn, V., Hülder, M., Köhler, A., Kriegel, A.: Cost and response time simulation for web-based applications on mobile channels. In: QSIC. pp. 83–90. IEEE (2005)
10. Bulychev, P.E., David, A., Larsen, K.G., Mikucionis, M., Poulsen, D.B., Legay, A., Wang, Z.: UPPAAL-SMC: statistical model checking for priced timed automata. In: QAPL. EPTCS, vol. 85, pp. 1–16. Open Publishing Association (2012)
11. Chen, X., Mohapatra, P., Chen, H.: An admission control scheme for predictable server response time for web accesses. In: WWW. pp. 545–554. ACM (2001)
12. Claessen, K., Hughes, J.: QuickCheck: A lightweight tool for random testing of Haskell programs. In: ICFP. pp. 268–279. ACM (2000)
13. Collina, M., Corazza, G.E., Vanelli-Coralli, A.: Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST. In: PIMRC. pp. 36–41. IEEE (2012)
14. Draheim, D., Grundy, J.C., Hosking, J.G., Lutteroth, C., Weber, G.: Realistic load testing of web applications. In: CSMR. pp. 57–70. IEEE (2006)
15. Govindarajulu, Z.: Sequential statistics. World Scientific (2004)
16. Hoeffding, W.: Probability inequalities for sums of bounded random variables. Journal of the American Statistical Association 58(301), 13–30 (1963)
17. Houimli, M., Kahloul, L., Benaoun, S.: Formal specification, verification and evaluation of the MQTT protocol in the Internet of Things. In: ICMIT. pp. 214–221. IEEE (Dec 2017)
18. Hughes, J.: QuickCheck testing for fun and profit. In: PADL. LNCS, vol. 4354, pp. 1–32. Springer (2007)
19. Kalaji, A.S., Hierons, R.M., Swift, S.: Generating feasible transition paths for testing from an extended finite state machine. In: ICST. pp. 230–239. IEEE (2009)
20. Lee, S., Kim, H., Hong, D., Ju, H.: Correlation analysis of MQTT loss and delay according to QoS level. In: ICOIN. pp. 714–717. IEEE (2013)
21. Legay, A., Delahaye, B., Bensalem, S.: Statistical model checking: An overview. In: RV. LNCS, vol. 6418, pp. 122–135. Springer (2010)
22. Legay, A., Sedwards, S.: On statistical model checking with PLASMA. In: TASE. pp. 139–145. IEEE (2014)
23. Lu, Y., Nolte, T., Bate, I., Cucu-Grosjean, L.: A statistical response-time analysis of real-time embedded systems. In: RTSS. pp. 351–362. IEEE (2012)
24. Nagelkerke, N.J.: A note on a general definition of the coefficient of determination. Biometrika 78(3), 691–692 (1991)
25. Papadakis, M., Sagonas, K.: A PropEr integration of types and function specifications with property-based testing. In: Erlang'11. pp. 39–50. ACM (2011)
26. Pearson, K.: Note on regression and inheritance in the case of two parents. Proceedings of the Royal Society of London 58, 240–242 (1895)
27. Rencher, A., Christensen, W.: Methods of Multivariate Analysis. Wiley (2012)
28. Rina, Tyagi, S.: A comparative study of performance testing tools. Intern. Journal of Adv. Research in Comp. Sci. and SW Eng., IJARCSSE 3(5), 1300–1307 (2013)
29. Schumi, R., Lang, P., Aichernig, B.K., Krenn, W., Schlick, R.: Checking response-time properties of web-service applications under stochastic user profiles. In: ICTSS. LNCS, vol. 10533, pp. 293–310. Springer (2017)
30. Tappler, M., Aichernig, B.K., Bloem, R.: Model-based testing IoT communication via active automata learning. In: ICST. pp. 276–287. IEEE (2017)
31. Thangavel, D., Ma, X., Valera, A.C., Tan, H., Tan, C.K.: Performance evaluation of MQTT and CoAP via a common middleware. In: ISSNIP. pp. 1–6. IEEE (2014)
32. Wald, A.: Sequential analysis. Courier Corporation (1973)