ComBench: A Benchmarking Framework for Publish/Subscribe Communication Protocols under Network Limitations

Stefan Herrnleben[1][0000−0001−7960−0823], Maximilian Leidinger[1], Veronika Lesch[1][0000−0001−7481−4099], Thomas Prantl[1], Johannes Grohmann[1][0000−0001−9643−6543], Christian Krupitzer[2][0000−0002−7275−0738], and Samuel Kounev[1]

[1] University of Wuerzburg, 97074 Wuerzburg, Germany
{firstname}.{lastname}@uni-wuerzburg.de
[2] University of Hohenheim, 70599 Stuttgart, Germany
{firstname}.{lastname}@uni-hohenheim.de

Abstract. Efficient and dependable communication is a highly relevant aspect for Internet of Things (IoT) systems in which tiny sensors, actuators, wearables, or other smart devices exchange messages. Various publish/subscribe protocols address the challenges of communication in IoT systems. The selection process of a suitable protocol should consider the communication behavior of the application, the protocol's performance, the resource requirements on the end device, and the network connection quality, as IoT environments often rely on wireless networks. Benchmarking is a common approach to evaluate and compare systems, considering the performance and aspects like dependability or security. In this paper, we present our IoT communication benchmarking framework *ComBench* for publish/subscribe protocols focusing on constrained networks with varying quality conditions. The benchmarking framework supports system designers, software engineers, and application developers to select and investigate the behavior of communication protocols. Our benchmarking framework contributes to (i) show the impact of fluctuating network quality on communication, (ii) compare multiple protocols, protocol features, and protocol implementations, and (iii) analyze scalability, robustness, and dependability of clients, networks, and brokers in different scenarios. Our case study demonstrates the applicability of our framework to support the decision for the best-suited protocol in various scenarios.

Keywords: IoT, publish/subscribe, benchmarking, load testing

1 Introduction

The road to success of the Internet of Things (IoT) leads to an enormous increase of devices exchanging data over the internet [27]. Many popular IoT communication protocols follow the publish/subscribe communication pattern [1], which

decouples space, time, and synchronization and is well-suited for upscaled distributed systems due to its loose coupling [11]. MQTT, AMQP, and CoAP are well-known examples of publish/subscribe protocols which exchange their messages via a central message broker [28,17,15,3,22]. All participating clients can be both publishers and subscribers. Whoever wants to receive a message subscribes to a specific topic at the broker. For sending a message, a client publishes the message with a specific topic to the broker, which delivers the message to all subscribers of the topic.

The argument to apply a publish/subscribe protocol in an IoT system is straightforward due to the characteristics of those systems [10,20,30]. Performance, scalability, and overhead of a protocol often play a significant role when looking for a suitable protocol for a particular use case [22,3,28,17]. Selecting the most suitable protocol and a performant implementation can be challenging due to the wide variety of existing protocols, each with different broker and client implementations [20,10].

A special challenge in IoT is the robustness of a communication protocol against the variation of network quality [5,6]. While data centers typically use wired connections, IoT devices often communicate via wireless networks, which are often faced with bandwidth limitations, high latencies, and packet loss [4,16]. In addition to WiFi and cellular networks, low-power wireless network protocols such as LoRaWAN can also be used. In addition to already challenging aspects of standardization, availability of implementations, and licensing, developers should also be aware of protocol performance even in stressed environments and under unstable connections [6].

Existing benchmarking tools can test load and scalability [13,7,9], but do not limit the network communication by, e.g., adding artificial packet loss or latency.

To support designers, developers, and operators of IoT systems, we present in this paper our benchmarking framework *ComBench* coupled with a methodology for evaluating and comparing communication-related characteristics like performance, dependability, and security [19]. The key attributes of ComBench can be summarized as follows:

- **Multi-protocol client** supporting the protocols MQTT, AMQP, and CoAP out of the box.
- **Customized load profiles** with configurable message size and frequency for each client or group of clients.
- **Virtually unlimited clients** for testing large-scaled environments.
- **Fine-grained communication configuration** per client or group of clients. Messages can be published after a fixed delay, following a statistical distribution, or as a (delayed) response to an incoming message.
- **Configurable network quality** in terms of bandwidth limit, transmission delay, and packet loss. Each can be configured per client or group of clients, statically or using a time series, e.g., for emulating a moving device.
- A **Benchmarking controller** serving as a single configuration point for managing the experiment, collecting all measurements, and making them available for visualization and export.

We demonstrate the wide range of ComBench's applicability in five exemplary use cases by varying protocols, configurations, workloads, network conditions, and setups. The case study shows how our benchmarking framework and measurement methodology analyzes and compares communication protocols in different IoT environments. ComBench allows developers and system designers to investigate the performance of protocols and the resource consumption on clients and networks to select an appropriate protocol or optimize their application. Artificial variation of network quality allows statements about the protocol's performance even under limited network connectivity.

The remainder of this paper is structured as follows. In Section 2, we describe the characteristics of our benchmark, its components, the supported metrics and give an insight into its usage. Section 3 presents the case study demonstrating its applicability to various evaluation objectives. In Section 4, we discuss weaknesses of our benchmarking framework that an operator should be aware of. Section 5 shows related work that deals with benchmarking and performance evaluation of publish/subscribe systems. Section 6 summarizes the paper and provides ideas for future work.

2 Benchmark

This section introduces our benchmarking framework, including the specified requirements, the measurement methodology, the captured metrics, and some design and implementation details. Section 2.1 presents the requirements from which the capabilities of the framework are derived. In Section 2.2, the metrics based on the measured values are formally defined. Section 2.3 describes the design and architecture of our ComBench, while Section 2.4 provides implementation and configuration details.

2.1 Requirements

For the development of a benchmarking framework for IoT communication protocols, a number of requirements arise. The general requirements for a benchmark [18] are supplemented by additional requirements due to the large heterogeneity of protocols, clients, and communication behavior, as well as the usually wireless connection [26,5]. This section presents the essential requirements to be met by our benchmarking framework for IoT communication protocols.

Support of Multiple Protocols Various publish/subscribe communication protocols are omnipresent in IoT systems. While some protocols are similar, other protocols are more suitable for specific communication scenarios and requirements. A benchmarking framework should address this heterogeneity and support various protocols to investigate scenarios under different protocols.

Variation of Network Conditions IoT devices often communicate over wireless links exposed to bandwidth fluctuations, transmission delays, and packet loss. A benchmarking framework for IoT systems should take such varying network conditions into account by artificially influencing the network quality. This

allows investigating and comparing protocol behavior and features under different workloads.

Heterogeneous Clients IoT systems often consist of clients with different message types (e.g., messages with different payload sizes), different communication behavior (e.g., rarely arising vs. high frequent), and different network quality (e.g., GSM vs. LTE). A benchmarking framework should reproduce this heterogeneity by configuring bandwidth limit, transmission delay, and packet loss per device or device group.

Protocol Features Protocols like MQTT support different QoS levels or security features like encryption via *Transport Layer Security (TLS)*. Such features often influence the performance and resource consumptions of the device as well as the network load. A benchmarking framework should support easy activation of these features to investigate the impact of such protocol features on CPU, RAM, and network.

Scalability IoT systems often scale over a massive amount of devices, and the number of IoT devices is continuously increasing [27]. The number of emulated IoT devices should not be limited to the hardware capabilities of a single node; instead, the experiment should be scaleable to multiple servers or a cloud. Scaling the number of clients, messages, and network load enable investigating brokers in stressed environments.

Flexibility at Broker Selection Various message broker implementations exist, which differ in the supported protocols, programming language, licensing, performance, stability, vendor, and how they are developed (e.g., community-driven, proprietary). Operators of IoT systems need to be aware of the broker's performance and reliability. Therefore, a benchmarking framework should not be fixed to a specific broker; instead, the broker, configuration, and deployment should be freely choosable.

Metrics Metrics are used in benchmarks to provide insights into the performance, reliability, or security of a tested system [19]. The supported metrics should not be limited to a specific evaluation objective. Instead, the metrics and measurements like client resource consumption, i.e., CPU and RAM utilization, network throughput, latency, packet loss, and protocol efficiency contribute to investigate and compare different objectives. Metrics are formally defined in the next section.

2.2 Metrics

Before defining the metrics, we introduce the meaning of the used sets and symbols. The set C containing all clients, the set T refers to all topics (for topic-based protocols like MQTT), and the set M contains all published messages by any client to any arbitrary topic. $P_c = \{m \mid m \in M$ and message m was published by client $c\}$ represents the set of messages which have been published by client c. The subscribed messages of each client c are described by $S_c = \{m \mid m \in M$ and topic t of message m was subscribed by client $c\}$. Since messages can be lost, $S'_c = \{m \mid m \in S_c$ and message m was received by client $c\}$ with $S'_c \subseteq S_c$

only considers messages for client c that were actually delivered. From these definitions and the introduced measurements, we derive the following metrics:

Message Loss Ratio This metric defines the ratio of lost messages to expected received messages at the application layer. This metric provides information about the reliability of a protocol and the success rate of packet loss compensation mechanisms, e.g., retransmissions. The message loss ratio is defined for each client c in Equation (1) as well as for all clients in Equation (2).

$$MsgLoss_c = 1 - \frac{|S'_c|}{|S_c|} \qquad (1) \qquad MsgLoss = 1 - \frac{\sum_{c \in C} |S'_c|}{\sum_{c \in C} |S_c|} \qquad (2)$$

In both cases, the numerator represents the number of messages subscribed and successfully received by a client. The denominator depicts the messages which have been subscribed by the client, regardless if they were received or not. It is essential to sum the messages per client for the total message loss. Considering only the number of total messages would include messages without a subscription and count messages with multiple subscriptions only once.

Latency of received messages This metric refers to the mean latency of received messages. Equation (3) defines the mean latency for each client, while Equation (4) defines it for all involved clients. $\sigma_{m,c} \in \mathbb{R}$ states the spent time for message m from sender to recipient c, including the broker processing time.

$$\overline{Lat}_c = \frac{\sum_{m \in S'_c} \sigma_{m,c}}{|S'_c|} \qquad (3) \qquad \overline{Lat} = \frac{\sum_{c \in C} \sum_{m \in S'_c} \sigma_{m,c}}{\sum_{c \in C} |S'_c|} \qquad (4)$$

In both definitions, the numerator sums up the latency of each received message. The denominator represents the total number of received messages per client, respectively, for all clients.

Protocol efficiency This metric indicates the ratio between the transferred payload and the number of bytes sent through the network interface. While the payload only considers the bytes of the message content, the bytes at the network interface include the complete protocol stack, i.e., payload, headers, checksums, and trailers. The protocol efficiency metric reflects the proportion of the payload to the transferred bytes. This metric further reflects additional bytes for retransmissions in case of packet loss. Moreover, the overhead for encryption or optional header fields are also included. Equation (5) shows the protocol efficiency for an individual client c while Equation (6) considers all clients. $\psi_m \in \mathbb{N}$ refers to the payload of message m, and Ψ_c denotes the total transferred bytes for client c.

$$PE_c = \frac{\sum_{m \in P_c} \psi_m}{\Psi_c} \qquad (5) \qquad PE = \frac{\sum_{c \in C} \sum_{m \in P_c} \psi_m}{\sum_{c \in C} \Psi_c} \qquad (6)$$

In both definitions, the numerator sums up the payload of each published message. The denominator depicts the amount of sent bytes, measured at the network interface, for one client, respectively, for all clients.

2.3 Harness Design

Section 2.1 defined the requirements for a benchmarking framework for IoT communication protocols. For the software design, additional requirements arise addressing correctness, ease of use, modularity, and extensibility. This section presents the design, the entities involved, and the process of a benchmark run. Figure 1 depicts the involved participants, i.e., the benchmark operator, the benchmark controller as central management instance, and the *system under test (SUT)* which consist of the multiple clients and the broker. The benchmark harness contains the controller and the clients, which emulate a freely configurable application. Any message broker, including cloud brokers that supports the desired protocol, can be used as a message broker.

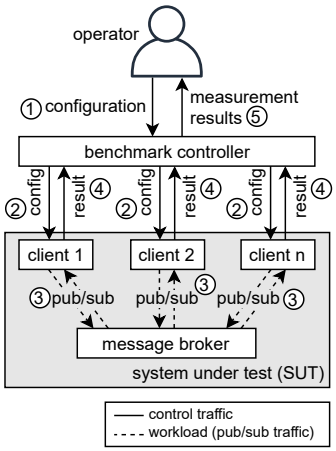

Fig. 1: Benchmark architecture with involved participants.

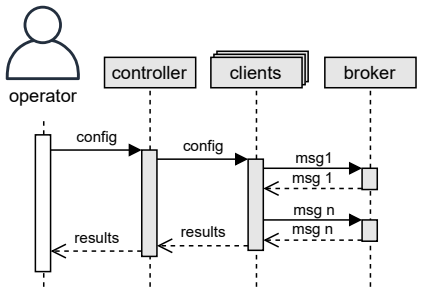

Fig. 2: Sequence diagram of a benchmark run.

The *benchmark operator* configures the benchmark by providing a configuration to the central controller. This configuration includes, amongst others, the start time and duration of the benchmark, some global settings like the selected publish/subscribe protocol, and the definition of the communication behavior as well as network constraints of the clients. After the experiment, the operator retrieves the measurement results from the central benchmark controller. Instead of a human operator, scripts or external software artifacts can use our API to configure and run the benchmark.

The *benchmark controller* is responsible for the overall control of the benchmark. It is instantiated as a separate software artifact in a container and must be deployed once. The benchmark controller receives the global configuration from the operator via a REST API, connects to the clients via an additional API, and configures them independently. The controller knows the benchmark timing and

waits for the duration of the experiment before collecting the raw measurement data from the clients, aggregating and evaluating them, and generating reports.

A *client* represents a communicating entity and acts as a workload generator using a publish/subscribe protocol. The client runs as a container, likewise the central controller. Clients can coexist on the same machine as long as their performance does not negatively affect each other. To scale the experiment or to isolate performance, the deployment can also be spread over multiple hosts. Our client implementation can handle the publish/subscribe protocols MQTT, AMQP, and CoAP via included protocol adapters. The selected protocol is one part of the individual configuration received by each client from the benchmark controller. Other parts of the configuration are the start time of the run, the duration, the subscriptions, the response behavior to incoming messages, scheduled publishing events, and network conditions. The client verifies time synchronization, applies the artificial network constraints on the container's network interface, and subscribes to the assigned topics, and starts publishing messages if configured. The central controller collects all measurements for further analysis after each run.

A central *message broker* is required by several publish/subscribe protocols like MQTT, AMQP, and CoAP. The message brokers manage the client subscriptions and deliver messages based on the message topics and subscriptions.

For an experiment, the benchmark operator provides a global configuration to the controller as shown in Figure 2. This configuration contains the global specifications and client configurations, including communication behavior, load profiles, and network constraints. The central controller sends an individual configuration file to each client, as depicted in Figure 1. After a client receives a configuration, it can operate autonomously, i.e., no further control traffic during the experiment is necessary. This time-decoupling allows using the same network connection for configuration and the workload. The clients configure themselves independently, start the experiment at a predefined time and collect measurement data. After a benchmark run, the controller collects and aggregates the measurement results from all clients, processes them, and generates summaries and graphs. This report is accessible from the controller, depicted in the last step in the sequence diagram of Figure 2.

2.4 Implementation

The implementation of ComBench consists of the artifacts controller and client, as introduced in Section 2.3. The controller manages the benchmark, configures the clients, and evaluates the measurements. The clients exchange messages with the chosen publish/subscribe protocol and conduct the measurements. This section describes the implementation of these artifacts, which are written in Python.

The *controller* provides a REST-API for configuration. The benchmark operator configures the benchmark via the API and retrieves the measurement results. The controller sends the preprocessed configuration again as an HTTP request to each client and collects the raw measurement results.

The benchmark *client* application currently includes three protocol adapters for MQTT (*asyncio-mqtt*[3] library), AMQP (*pika* library[4]), and CoAP (*aiocoap*[5] library). The REST API on the client receives the configuration commands and returns the measurements. To influence the network conditions and retrieve specific system parameters, the client has some dependencies on Linux tools and is, therefore, only executable on Linux.

Both artifacts, controller and client, are published as open-source under the Apache 2.0 license. The source code, a manual, and some evaluation examples are available at ComBench's GitHub repository [29]. The Docker images can be pulled from Docker Hub [6,7].

The setup of an experiment is specified in a single configuration file to facilitate repeatability and sharing of different scenarios since all configuration options are stored in one file. The configuration contains some global settings like the chosen communication protocol, the start time, the runtime of the benchmark, and the IP address/hostname of the broker given as a JSON file. Furthermore, the configuration defines the client roles, like group settings, subscriptions, the associated publishing events, and network conditions. The network conditions like packet loss rates, bandwidth limits, and transmission delays can be specified either statically or as a time series so that the values change during the experiment and thus, e.g., emulate moving devices. The last part of the configuration is the individual clients. Each client is assigned to a group, simplifying large-scale experiments.

To calculate the metrics defined in Section 2.2, measurements must be performed. On the client, our benchmark collects (i) message logs, (ii) system performance parameters, and (iii) network performance parameters. The message log records every published message with its unique message ID, the topic, and the transmission timestamp. Each received message also creates a log entry at the subscriber with the message ID and arrival timestamp. The topic of received messages is derived using the unique message ID.

For the system performance parameters, the client periodically queries the utilization of CPU (total and per core) and RAM as well as the numbers of sent packets, received packets, sent bytes, and received bytes from the operating system's network interface. This measurement is repeated every 1000 ms, and the measured values and timestamps are logged into a file. The measured values are stored on each client to avoid additional network traffic during each run. The benchmark controller collects all clients' measurements after the runs, as described in Section 2.3.

[3] https://pypi.org/project/asyncio-mqtt/
[4] https://pika.readthedocs.io/
[5] https://github.com/chrysn/aiocoap
[6] https://hub.docker.com/r/descartesresearch/iot-pubsub-benchmark-controller
[7] https://hub.docker.com/r/descartesresearch/iot-pubsub-benchmark-client

3 Case Study

ComBench addresses multiple evaluation objectives related to performance, scalability, reliability, and security, which can be investigated with different network conditions. The captured measurement data and derived metrics introduced in Section 2.1 enable a wide range of investigations and comparisons. This section presents the universal applicability of ComBench in a case study consisting of five exemplary IoT scenarios. We show how ComBench contributes to analyzing specific concerns for each of those scenarios, such as broker resilience, protocol efficiency, and the impact of network limitations on communication. Each scenario includes a short motivation, the associated study objectives (SO), a description of the benchmarking experiment, and a brief interpretation of results.We like to emphasize that the focus of this case study is to demonstrate the capabilities of ComBench and how it can be applied and not the discussion of the observations itself. Hence, we do not present a detailed description and interpretation of the concrete measurement results.

The experiments are executed on a node with a four-core Intel Core i7-4710HQ processor and 8 GB RAM using Ubuntu 18.04.2 LTS with Docker version 19.03.11. The benchmark configuration and the detailed measurement results are available at our GitHub repository [29] and on Zenodo [14].

3.1 Broker Resilience

Motivation Many publish/subscribe protocols like MQTT or AMQP use a central message broker that manages the subscribed topics and distributes the published messages. An increasing number of clients and messages can overload the broker, resulting in higher latencies or message loss. When selecting an appropriate broker or for configuration tuning the resilience can be a crucial criterion.

Study objectives A first study objective that can be analyzed using ComBench is the load level at which the broker start to delay messages (SO 1.1). As brokers are implemented in different programming languages and might operate less or more performant, their resilience may differ. Identifying performance variations between the brokers at different load levels is a further objective of our study (SO 1.2).

Scenario To compare the resilience of different brokers, we stress the broker by horizontal scaling, i.e., by increasing the number of clients and messages. We analyze the captured latency of the message transmission of all clients (ref. Section 2.1: Equation (4)) and compare different load levels of different broker implementations. We apply a supermarket company's supply chain load profile as introduced by the SPECjms2007 benchmark [25]. Due to space limitations, we refer to the SPECjms2007 documentation for a detailed workload description. We perform measurement runs of 60 seconds, each with three repetitions at the scaling factors one, five, ten, 15, 20, 30, 40, 45, 50, and 55.

Conclusions Figure 3 depcits the average latency per scaling factor for each broker . The x-axis shows the different scaling levels, while the y-axis represents

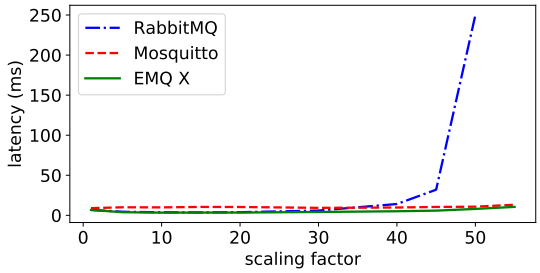

Fig. 3: Average latency for message transmission for different scaling factors and MQTT broker implementations.

the measured average latency in milliseconds. The analysis indicates no noticeable increase in latency until a scaling factor of 30 for RabbitMQ and up to 45 for Mosquitto and EMQ X (SO 1.1). All brokers perform similar until a scaling factor of 45, at which the latency of RabbitMQ increases rapidly (SO 1.1). EMQ X and Mosquitto show only a slight latency increase starting at a scaling factor of 50 (SO 1.2).

3.2 Protocol Efficiency

Motivation In addition to the message payload, the protocol headers of the protocol stack also affect the actual number of transmitted bytes at the network interface. Various publish/subscribe protocols generate different overheads due to their headers and the underlying protocols. Especially for low data rates, as often present in Wireless Sensor Networks (WSNs), low overhead is crucial. The protocol efficiency (ref. Section 2.2: Equation (5)) indicates the proportion of the payload, i.e., the message content, compared to the transferred bytes observed at the network interface.

Study objectives A first study analyzes and compares the protocol efficiency of AMQP, MQTT, and CoAP at different payload sizes (SO 2.1). The objective is to identify which of the protocols is best suited for low-bandwidth networks due to its efficiency (SO 2.2). Furthermore, the study compares the protocol efficiency of MQTT at the three quality of service (QoS) levels, which differs through additional control messages (SO 2.3).

Scenario To investigate protocol efficiency, we deploy a simple setup consisting of one publisher and one subscriber, without any configured restrictions related to the network conditions. The publisher sends ten messages per second over 60 seconds with a fixed payload via the broker to the subscriber. The measurements are repeated for payload sizes between 100 and 1000 bytes in step sizes of 100 bytes for the protocols MQTT, AMQP, and CoAP. To compare the protocol efficiency of MQTT at the QoS levels 0 (default), 1, and 2, we executed the measurement series with payload sizes between 200 and 10000 bytes in step sizes of 200 bytes.

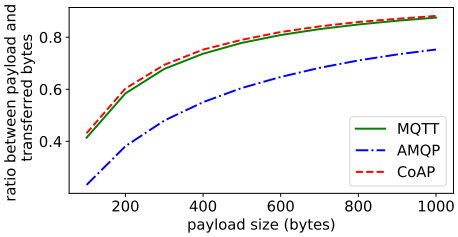

Fig. 4: Ratio between payload and transmitted bytes for different protocols.

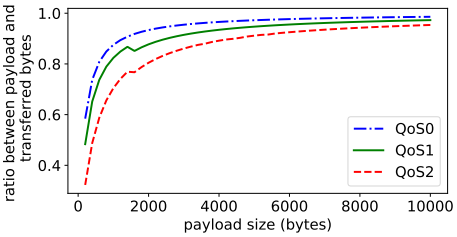

Fig. 5: Ratio between payload and transmitted bytes for different MQTT QoS levels.

Conclusions Figure 4 shows the protocol efficiency of the protocols MQTT, AMQP, and CoAP. The x-axis depicts the size of the payload in bytes, while the y-axis indicates the ratio between the payload and the transferred bytes observed at the network interface. The graph shows that AMQP has a lower efficiency than MQTT and CoAP, i.e., it has a higher overhead due to, e.g., protocol headers (SO 2.1). MQTT and CoAP have similar efficiency; therefore, both protocols are well suitable for low bandwidth networks from the protocol overhead perspective (SO 2.2). As expected, the efficiency increases with the payload size since the payload takes a higher proportion of the transferred bytes than the header. Figure 5 depicts the efficiency of MQTT at different QoS levels, with an identical axis interpretation to Figure 4. This measurement complies with the expectations that with an increasing QoS level, the efficiency decreases due to the additional control traffic (SO 2.3).

3.3 Effect of packet loss

Motivation Packets can get lost during data transmission, especially in unstable wireless networks. A few lost packets are usually compensated by the protocols or the underlying protocol layers through re-transmissions so that messages on application layer still arrive (usually delayed). For applications running on unstable networks, a protocol with adequate compensation mechanisms should be chosen. The message loss rate can be determined by Equation (1) (see Section 2.2).

Study objectives One objective of this study is to determine at which packet loss rate messages get lost (SO 3.1). Based on this, we test at which packet loss rate the communication is no longer possible (SO 3.2). For particularly lossy connections, a comparison of the message loss rates of protocols would support identifying particularly robust protocols (SO 3.3).

Scenario To investigate the packet loss, we again use a setup consisting of one publisher and one subscriber. The publisher sends ten messages per second with 20 repetitions. The network interface of the publisher is configured with a constant packet loss rate, increasing along with a measurement series from 0 to 100 percent in steps of 5 percent. Further traffic and network disturbances

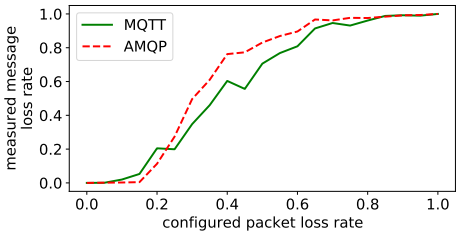

Fig. 6: Message loss rate at artificially configured packet loss.

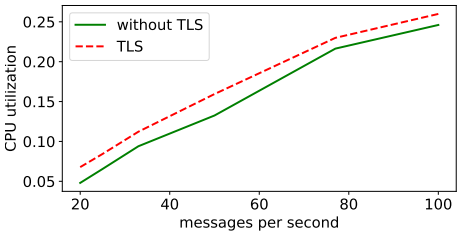

Fig. 7: Impact of TLS on client's CPU load on an increasing number of messages.

are not configured. Lost messages are determined by matching the IDs of the messages sent by the publisher and the messages received by the subscriber. This study is performed for MQTT and AMQP.

Conclusions Figure 6 depicts the message loss rate for different configured packet loss rates for the protocols MQTT and AMQP. The x-axis shows the measured message loss rate, as indicated by Equation (1). The y-axis depicts the configured packet loss rate at the subscriber's network interface. The results show that the first message losses occur at a packet loss rate of 10 percent for MQTT and 15 percent for AMQP (SO 3.1). Communication is impossible, i.e., almost all messages are lost, for both protocols at a packet loss rate of approximately 85 percent (SO 3.2). The two considered protocols behave similarly in terms of robustness against packet loss (SO 3.3). Further measurements are necessary to identify significant differences between these protocols related to the sensitivity to packet loss.

3.4 Impact of TLS on Client CPU Utilization

Motivation TLS enables encrypted communication for MQTT on the transport layer (Layer 4 on ISO OSI model). However, the handshake, encryption, and decryption of TLS may introduce additional load on the client's CPU. Especially for resource-constrained IoT devices, the influence of encryption mechanisms on the system load should be taken into account.

Study objectives The first objective of this study is to quantify the additional load on the CPU by enabling TLS (SO 4.1). Furthermore, the effect of an increased number of encrypted messages on the client's CPU utilization is analyzed (SO 4.2).

Scenario For this scenario, we use a setup consisting of one publisher and one subscriber. The publisher sends messages with a fixed rate for 60 seconds with a payload of 100 bytes, which the subscriber will receive. This message rate is increased at rates between 20 and 100 messages per second within a measurement series. No encryption is used for a first measurement series, while in a second series all measurements are repeated with TLS enabled.

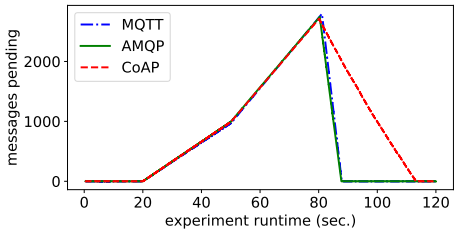

Fig. 8: Delayed messages during the execution of the experiment.

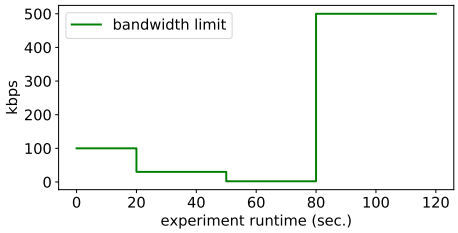

Fig. 9: Artificially configured bandwidth during the experiment.

Conclusions For the evaluation, we use the reported CPU utilization on the publisher. Figure 7 shows the impact of TLS on the client's CPU utilization. The x-axis depicts the number of sent messages, and the y-axis shows the ratio of utilized CPU. The graph shows that TLS introduces approximately 2 percent more CPU utilization compared to the scenario without encryption (SO 4.1). It can moreover be seen that the influence of TLS introduces an almost constant overhead regardless of the number of messages (SO 4.2).

3.5 Influence of Fluctuating Bandwidth

Motivation IoT applications in wireless networks often face fluctuating network quality, like limited bandwidths. Fluctuating bandwidths occur particularly often for moving devices. The communication behavior at such fluctuating bandwidths is of particular interest in mobile applications. Besides the behavior in case of a static bandwidth limitation, it is especially relevant how the protocol behaves at bandwidth variations.

Study objectives When considering varying bandwidths, it can be essential to know the bandwidth limit to which packets can be sent without any delay for the protocols MQTT, AMQP, and CoAP (SO 5.1). In contrast, it could further be relevant how fast queued packets are sent after bandwidth limitation (SO 5.2). It is also essential if packets were lost during this time (SO 5.3).

Scenario To analyze the influence of fluctuating bandwidth, we use a setup with one publisher and one subscriber. The publisher sends 67 messages per second for 120 seconds with a payload of 1000 bytes each. The bandwidth is not set with a static value. Instead, different bandwidth limits are specified as time series, which are applied automatically by the client. The bandwidth is set to 100 kbps initially, limited to 30 kbps from second 20, and reduced to 2 kbps from second 50 to imitate a decreasing signal quality, as depicted in Figure 9. From second 80, it is assumed that the mobile network is available again, and the bandwidth limitation is increased to 500 kbps. The experiment is performed for MQTT, AMQP, and CoAP.

Conclusion Figure 8 shows the number of delayed messages during the experiment for MQTT, AMQP, and CoAP. The x-axis depicts the runtime of the

experiment, while the y-axis indicates the number of messages that were not delivered at the respective time. The first delay for the three protocols occurs in second 20, when the bandwidth is limited to 30 kbps (ref. Figure 9), which means that transmission at the previous 100 kbps was possible without delay (SO 5.1). The graph follows the expectation that from second 50 on, more messages are delayed due to the further decreased bandwidth, which is similar for all used protocols. After pushing the bandwidth limit to 500 kbps in second 80, MQTT and AMQP send the delayed messages in about 8 seconds, while CoAP takes about 35 seconds (SO 5.2). Since no delayed messages remain at the end of the experiment, all messages were delivered without loss (SO 5.3).

4 Threats to Validity

The previous section has shown the wide variety of use cases in which ComBench can be used. Nevertheless, there are a few weaknesses that a benchmark operator should be aware of. This section discusses these vulnerabilities and provides some approaches to remedy or mitigate them.

ComBench offers several advantages in usability, verifiability, and metrics due to its unified, multi-protocol client and its instrumentation. However, the protocol adapters contained in the client create a small performance overhead that must be considered compared to a single-protocol implementation. Also, the client's instrumentation for logging the messages and recording CPU, RAM, and network utilization introduces additional load on the client. We assume that this overhead can be neglected; however, a detailed analysis of the overhead is part of our future work. If the pure client performance needs to be measured, a single-protocol client and another instrumentation should be used, which is out of this work's scope. By implementing the REST interface against our controller, alternative clients — also if necessary with alternative protocol implementations — can be integrated transparently.

Another possible vulnerability of our benchmarking framework is time synchronization. Latency measurements determine the delta between the timestamps from sending the message to receiving it. An accurate calculation requires exact time synchronization between all participants. Some previous work implements a request/reply pattern, i.e., the subscriber responds to the sender, to measure the round trip time [23]. However, we deliberately decided against this procedure because we do not consider request/reply very practical in publish/subscribe environments. Instead, we rely on the Precision-Time-Protocol (PTP), and the benchmark verifies before each run that the participants' times are synchronized. Studies show a deviation of less than 1 μs when using PTP [21], which we assume is satisfiable in practice. Please note that in small setups, where all clients are running on a single host, the issue of possible time deviation is irrelevant as all guest systems use the host's clock.

5 Related Work

Testing the performance of communication protocols is not a new field in academia. Therefore, tools dealing with load generation and measurements related to communication systems are especially important for this paper. Section 5.1 presents some work that deals with the comparison of IoT network protocols. While Section 5.2 presents tools that focus on load testing and measurements, Section 5.3 identifies benchmarks providing predefined load scenarios in addition to load generation and measurement instrumentation. Section 5.4 summarizes the related work by listing all the works with their primary focus and characteristics.

5.1 Protocol Comparisons

A large number of papers deal with the comparison of different communication protocols, which emphasizes the relevance of a benchmarking framework. In the following, we provide a possible categorization of related comparisons of IoT application layer protocols. We have assigned the studies to their primary focus, but some can also be well suited in multiple categories.

The first category is the theoretical comparison of protocols [10,20]. Header structure, payload size, and security features are discussed and compared. Furthermore, related protocol comparisons deal with scalability and analyzing the resource consumption of clients or brokers [28,17]. Another category of related studies focuses on network performance and network load of protocols [15,3,22]. More specialized studies in this area analyze and compare the performance of protocols under constraint networks [5,6,23] and different topologies [12].

The studies provide valuable insights into the protocols, some of them also with artificially restricted network communication [5,6,23]. Although some authors make their developed test tools publicly available, they are usually targeted to the specific test and do not allow free configuration as we expect from a benchmarking framework.

5.2 Load Testing Frameworks

Tools and benchmarking frameworks for load testing are most related to our work. These artifacts are characterized by their accessibility as a tool that can both generate loads and perform measurements. In the following, we present a selection of some well-known load testing frameworks.

JMeter [13] is a Java-based open-source application designed to load test functional behavior and measure performance. While it was originally designed to test web applications, extensions add other features and communication protocols such as MQTT. *LoadRunner* [31] is a commercial testing solution supporting a wide range of technologies and protocols in the industry with focus on testing applications and measuring system behaviour and performance under load. It supports the IoT protocols MQTT and CoAP and provides a IDE for scripting and running unit tests. *Gatling* [9] is an open-source load testing framework written in Scala for analyzing and measuring the performance of different

services, focusing on web applications. Community plugins can be used to add protocol support for, e.g., MQTT, and AMQP. *MZBench* [8] is a community-driven open-source benchmarking framework written in Erlang and focusing on testing software products. Among others the communication protocols XMPP, and AMQP are supported by MZBench, furthermore can be added as extension. *LOCUST* [7] is another community-driven open source load testing tool focusing on load testing of web applications. Although the common protocols for web applications (HTTP, Websocket, ...) are supported, there are currently no clients for IoT communication protocols. Although load test tools usually have a wide range of configuration options and measurement methods, to the best of our knowledge there are no tools that support comprehensive network connectivity constraints.

5.3 Benchmarks

In this section, we discuss existing IoT benchmarks using publish/subscribe protocols and show that already some research effort was made on a static analysis and comparison of publish/subscribe protocols. Benchmarks differ from load test frameworks primarily in that one or more predefined load profiles are provided in addition to the tooling.

Sachs et al. propose their *SPECjms2007* benchmark [25], focusing on evaluating the performance of message-oriented middleware (MOM), i.e., the Java message service (JMS). As a follow-up work, *Sachs et al.* proposed the *jms-2009-PS* benchmark [24], focusing on publish/subscribe patterns. Afterward, *Appel et al.* add another use case by changing the used protocol to AMQP in the jms2009-PS benchmark to analyze the MOM [2]. *Zhang et al.* presented in 2014 *PSBench*, a benchmark for content- and topic-based publish/subscribe systems [32]. Under the name *IoTBench* a research initiative pursues the vision of a generic IoT benchmark [4]. Their goal is to test and compare low-power wireless network protocols. This initiative collaborates on the vision of this benchmark and already published first steps toward a methodology in [16]. *RIoTBench* is a real-time IoT benchmark suite for distributed stream processing systems [26]. The covered performance metrics include latency, throughput, jitter, and CPU and memory utilization.

The presented benchmarks are a valuable contribution to the investigation of the performance of communication protocols including competitive workloads. However, we could not identify a fluctuating network quality in any of the benchmarks, which is an inherent issue for IoT systems.

5.4 Summary

As our review for related work shows, the evaluation of IoT communication protocols has a significant relevance in current research [10,20]. While individual evaluations usually focus on one or multiple specific aspects [22,3], load test frameworks or benchmarks typically offer a more universal applicability due to their configurable workload and included reporting [13,26]. Most evaluations and

Approach	Type	IoT Protocol Support	Client Perf. Measurement	Network Perf. Measurement	Influence Netw. Conditions	Released Tool inc. Reporting
Dizdarević et al. [10]	Survey	✓				
Naik et al. [20]	Survey	✓				
Talaminos-Barroso et al. [28]	Evaluation	✓	✓	✓		✓
Kayal et al. [17]	Evaluation	✓	✓			
Iglesias-Urkia et al. [15]	Evaluation	✓		✓		
Bansal et al. [3]	Evaluation	✓		✓	✓	
Pohl et al. [22]	Evaluation	✓		✓	✓	
Chen et al. [5]	Evaluation	✓		✓	✓	
Collina et al. [6]	Evaluation	✓		✓	✓	
Profanter et al. [23]	Evaluation	✓	✓	✓		
JMeter [13]	LT Tool	✓	✓	✓		✓
LoadRunner [31]	LT Tool	✓	✓	?		✓
Gatling [9]	LT Tool	✓	✓			✓
MZBench [8]	LT Tool	✓	✓			✓
LOCUST [7]	LT Tool	✓	✓			✓
SPECjms2007 [25]	Benchmark		✓	✓		✓
jms2000-PS [24]	Benchmark		✓	✓		✓
Appel et al. [2]	Evaluation	✓	✓			
PSBench [32]	Benchmark	✓	✓			
IoTBench [4,16]	Benchmark	✓				
RIoTBench [26]	Benchmark	✓	✓	✓		✓
ComBench	LT Tool	✓	✓	✓	✓	✓

Table 1: Matrix summarizing the scope and characteristics of related work compared to our ComBench benchmarking framework.

tools assume a lossless connectivity, while some work shows the demand to study the protocols also under connections with constrained network quality [5,23]. However, to the best of our knowledge, no benchmarking framework currently exists that focuses on the evaluation of IoT communication protocols under constrained network quality, such as packet loss or a temporary link failure.

Table 1 summarizes our considered related works. The first column names the approach, while the second column classifies it according to the category of its main purpose. We distinguish between *surveys* providing a theoretical comparison but no measurements, *evaluations* comparing different protocols through measurements, *load test tools* (*"LT tool"*) focusing particularly on the reuse of measurement tools, and *benchmarks* providing additional predefined competitive workload scenarios. The subsequent columns indicate by '✓' whether certain properties are met by the approach. An '?' indicates that we have been unable

to determine this property. Note that we only consider the primary focus of the approach, and for tools, we only check properties that can be enabled through official plugins and settings within the tool. IoT protocol support is fulfilled if at least one of the protocols MQTT, AMQP, XMQP, CoAP, ZeroMQ, DDS, or OPC UA is supported. Client performance measurements specifies whether the approach observes client resource consumption such as CPU or RAM, while network performance measurements targets aspects such as latency or bandwidth consumption. The aspect influence network conditions is fulfilled if at least one of the configurations packet loss, transmission delay or link failure is present or examined. If an official tool is provided as an artifact including a reporting, the requirement for the last column is met.

The summary shows that so far there is no specific load test tool or benchmarking framework for the evaluation of IoT communication protocols under network constraints, which motivates us for the developing ComBench.

6 Conclusion

In this paper, we presented ComBench, our publish/subscribe benchmarking framework for IoT systems. The framework is designed to analyze and compare different application layer protocols and includes three key features. Firstly, ComBench can be useful to investigate the effects of varying network quality on communication behavior. Second, due to its multi-protocol capability, ComBench can compare different protocols and their features. At third, our benchmarking framework supports designers, developers, and operators of IoT systems, analyzing the scalability, robustness, and reliability of clients, networks, and brokers. For all these areas, ComBench offers the appropriate instrumentation for collecting and analyzing measurement data. Section 3 presented some exemplary benchmarking scenarios and pointed out some briefly answered objectives during the case study. During the development of ComBench, we paid attention to high usability, which is especially characterized by the multi-protocol client, the central benchmark controller, the deployment in a containerized environment, and the included generation of reports. ComBench is published as open-source under the Apache License 2.0 on GitHub [29] and is accessible to other researchers, system designers, software engineers, and developers.

In the future, we want to continue developing ComBench and add additional technical features. One goal is to implement additional publish/subscribe protocol adapters for, e.g., ZeroMQ, DDS, and OPC UA pub/sub. Furthermore, we plan to add request/response patterns as used in REST applications. A graphical, interactive web interface can present the results in a more comfortable and user-friendly way, especially for users from the industry. Last, we aim to add representative workloads and establish a standardized benchmark from these.

Acknowledgement – This project is funded by the Bavarian State Ministry of Science and the Arts and coordinated by the Bavarian Research Institute for Digital Transformation (bidt).

References

1. Al-Fuqaha, A., Guizani, M., Mohammadi, M., Aledhari, M., Ayyash, M.: Internet of Things: A Survey on Enabling Technologies, Protocols, and Applications. IEEE Communications Surveys Tutorials **17**(4), 2347–2376 (2015). https://doi.org/10.1109/COMST.2015.2444095
2. Appel, S., Sachs, K., Buchmann, A.: Towards benchmarking of AMQP. In: Proceedings of the Fourth ACM International Conference on Distributed Event-Based Systems. pp. 99–100 (2010)
3. Bansal, S., Kumar, D.: IoT application layer protocols: performance analysis and significance in smart city. In: 2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT). pp. 1–6. IEEE (2019)
4. Boano, C.A., Duquennoy, S., Förster, A., Gnawali, O., Jacob, R., Kim, H.S., Landsiedel, O., Marfievici, R., Mottola, L., Picco, G.P., et al.: IoTBench: Towards a benchmark for low-power wireless networking. In: 2018 IEEE Workshop on Benchmarking Cyber-Physical Networks and Systems (CPSBench). IEEE (2018)
5. Chen, Y., Kunz, T.: Performance evaluation of IoT protocols under a constrained wireless access network. In: 2016 International Conference on Selected Topics in Mobile & Wireless Networking (MoWNeT). pp. 1–7. IEEE (2016)
6. Collina, M., Bartolucci, M., Vanelli-Coralli, A., Corazza, G.E.: Internet of Things application layer protocol analysis over error and delay prone links. In: 2014 7th Advanced Satellite Multimedia Systems Conference and the 13th Signal Processing for Space Communications Workshop (ASMS/SPSC). pp. 398–404. IEEE (2014)
7. Community: Locust Website. https://locust.io/ (2021), online; visited 2021-04-15
8. Community: MZBench Website. https://github.com/satori-com/mzbench (2021), online; visited 2021-04-13
9. Corp, G.: Gatling Website. https://gatling.io/ (2021), online; visited 2021-04-13
10. Dizdarević, J., Carpio, F., Jukan, A., Masip-Bruin, X.: A survey of communication protocols for internet of things and related challenges of fog and cloud computing integration. ACM Computing Surveys (CSUR) (2019)
11. Eugster, P.T., Felber, P.A., Guerraoui, R., Kermarrec, A.M.: The many faces of publish/subscribe. ACM computing surveys (CSUR) **35**(2), 114–131 (2003)
12. Gündoğran, C., Kietzmann, P., Lenders, M., Petersen, H., Schmidt, T.C., Wählisch, M.: NDN, CoAP, and MQTT: a comparative measurement study in the IoT. In: Proceedings of the 5th ACM Conference on Information-Centric Networking. pp. 159–171 (2018)
13. Halili, E.: Apache JMeter. Packt Publishing (2008)
14. Herrnleben, S., Leidinger, M., Lesch, V., Prantl, T., Grohmann, J., Krupitzer, C., Kounev, S.: Evaluation Results of ComBench as Open Data. Tech. rep., University of Wuerzburg (2021), online, https://doi.org/10.5281/zenodo.4723344; published 30. April 2021
15. Iglesias-Urkia, M., Orive, A., Barcelo, M., Moran, A., Bilbao, J., Urbieta, A.: Towards a lightweight protocol for Industry 4.0: An implementation based benchmark. In: 2017 IEEE International Workshop of Electronics, Control, Measurement, Signals and their Application to Mechatronics (ECMSM) (2017)
16. Jacob, R., Boano, C.A., Raza, U., Zimmerling, M., Thiele, L.: Towards a methodology for experimental evaluation in low-power wireless networking. In: Proceedings of the 2nd Workshop on Benchmarking Cyber-Physical Systems and Internet of Things. pp. 18–23 (2019)

17. Kayal, P., Perros, H.: A comparison of IoT application layer protocols through a smart parking implementation. In: 2017 20th Conference on Innovations in Clouds, Internet and Networks (ICIN). pp. 331–336. IEEE (2017)
18. von Kistowski, J., Arnold, J.A., Huppler, K., Lange, K.D., Henning, J.L., Cao, P.: How to Build a Benchmark. In: Proceedings of the 6th ACM/SPEC International Conference on Performance Engineering (ICPE 2015). ICPE '15, ACM, New York, NY, USA (February 2015)
19. Kounev, S., Lange, K.D., von Kistowski, J.: Systems Benchmarking. Springer International Publishing, 1 edn. (2020). https://doi.org/10.1007/978-3-030-41705-5
20. Naik, N.: Choice of effective messaging protocols for IoT systems: MQTT, CoAP, AMQP and HTTP. In: 2017 IEEE international systems engineering symposium (ISSE). pp. 1–7. IEEE (2017)
21. Neagoe, T., Cristea, V., Banica, L.: NTP versus PTP in Computer Networks Clock Synchronization. In: 2006 IEEE International Symposium on Industrial Electronics. vol. 1, pp. 317–362. IEEE (2006)
22. Pohl, M., Kubela, J., Bosse, S., Turowski, K.: Performance Evaluation of Application Layer Protocols for the Internet-of-Things. In: 2018 Sixth International Conference on Enterprise Systems (ES). pp. 180–187. IEEE (2018)
23. Profanter, S., Tekat, A., Dorofeev, K., Rickert, M., Knoll, A.: OPC UA versus ROS, DDS, and MQTT: Performance Evaluation of Industry 4.0 Protocols. In: Proceedings of the IEEE International Conference on Industrial Technology (ICIT) (2019)
24. Sachs, K., Appel, S., Kounev, S., Buchmann, A.: Benchmarking publish/subscribe-based messaging systems. In: International Conference on Database Systems for Advanced Applications. pp. 203–214. Springer (2010)
25. Sachs, K., Kounev, S., Bacon, J., Buchmann, A.: Performance evaluation of message-oriented middleware using the SPECjms2007 benchmark. Performance Evaluation **66**(8), 410–434 (2009)
26. Shukla, A., Chaturvedi, S., Simmhan, Y.: RIoTBench: An IoT Benchmark for Distributed Stream Processing Systems. Concurrency and Computation: Practice and Experience **29**(21), e4257 (2017)
27. Statista, IHS: Internet of Things - number of connected devices worldwide 2015-2025 (2018), https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/
28. Talaminos-Barroso, A., Estudillo-Valderrama, M.A., Roa, L.M., Reina-Tosina, J., Ortega-Ruiz, F.: A Machine-to-Machine protocol benchmark for eHealth applications – Use case: Respiratory rehabilitation. Computer methods and programs in biomedicine **129**, 1–11 (2016)
29. University of Wuerzburg, Institute of Computer Science, Germany, Chair of Software Engineering: Git repository of ComBench (2021), https://github.com/DescartesResearch/ComBench
30. Wirawan, I.M., Wahyono, I.D., Idfi, G., Kusumo, G.R.: Iot communication system using publish-subscribe. In: 2018 International Seminar on Application for Technology of Information and Communication. pp. 61–65. IEEE (2018)
31. Zhang, H.l., Zhang, S., Li, X.j., Zhang, P., Liu, S.b.: Research of load testing and result application based on LoadRunner. In: 2012 National Conference on Information Technology and Computer Science. Atlantis Press (2012)
32. Zhang, K., Rabl, T., Sun, Y.P., Kumar, R., Zen, N., Jacobsen, H.A.: PSBench: a benchmark for content-and topic-based publish/subscribe systems. In: Proceedings of the Posters & Demos Session, pp. 17–18. Association for Computing Machinery (2014)