BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INTELLIGENT SYSTEMS
ÚSTAV INTELIGENTNÍCH SYSTÉMŮ

MIDDLEWARE FOR TESTOS PLATFORM

MIDDLEWARE PRO PLATFORMU TESTOS

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **RADIM ČERVINKA**
AUTOR PRÁCE

SUPERVISOR **Ing. ALEŠ SMRČKA, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2020/2021

Master's Thesis Specification

Student:	**Červinka Radim, Bc.**
Programme:	Information Technology
Field of study:	Information Technology Security
Title:	**Middleware for Testos Framework**
Category:	Software analysis and testing

Assignment:

1. Get familiar with tools for communication of software systems. Study current implementations of Publish/Subscribe protocols for message passing (e.g. MQTT, AMQP, DDS).
2. Analyse requirements for communication of tools implemented in Testos framework. Design the solution for message passing between 2 and more communication nodes. Designed middleware should automatically adapt to current parameters of communication channel.
3. Implement the designed middleware. Implement adaptors for the middleware in C/C++ and Python languages.
4. Verify the basic functionality using automated tests. Measure the performance of the middleware.

Recommended literature:

- Tanabe K., Tanabe Y., Hagiya M. (2020) Model-Based Testing for MQTT Applications. In: Virvou M., Nakagawa H., C. Jain L. (eds) Knowledge-Based Software Engineering: 2020. JCKBSE 2020. Learning and Analytics in Intelligent Systems, vol 19. Springer, Cham. https://doi.org/10.1007/978-3-030-53949-8_5
- OASIS. MQTT Standard pro zasílání zpráv pro IoT. https://mqtt.org/
- Advanced Message Queuing Protocol. https://www.amqp.org/
- OMG. Data Distribution Service. https://www.omg.org/omg-dds-portal/

Requirements for the semestral defence:

- The first two steps.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Smrčka Aleš, Ing., Ph.D.**
Head of Department:	Hanáček Petr, doc. Dr. Ing.
Beginning of work:	November 1, 2020
Submission deadline:	May 19, 2021
Approval date:	January 15, 2021

Abstract

This goal of this thesis is to create a communication bus for the Testos platform, which enables the tools to communicate and utilize each other's services. The thesis consists of a research of current Publish-Subscribe protocols and solutions. It also outlines the requirements for a communication bus that fits the Testos platform's needs and proposes solutions that satisfy them.

As a part of the research, there were 3 message-oriented software solutions explored - MQTT, DDS and AMQP. The examination of each solution was focused on the communication model and main features. The MQTT protocol was chosen as the starting point of the bus implementation. The thesis also specifies how to extend the protocol in order to satisfy the requirements.

Main MQTT extensions proposed by this project include an introduction of a management of request life cycle on top of the MQTT message delivery and the request/response mechanism. The protocol was also expanded by ability to pack messages into a BULK packet to decrease the needed network resources.

The result is a Testos Bus, which is based on a modified and expanded version of MQTT, that includes a broker implementation as well as implementation of client libraries for Python and C++. Testos Bus satisfies all mandatory requirements, which is verified by automated tests.

Abstrakt

Cílem této práce je vytvořit komunikační sběrnici pro platformu Testos, což umožní nástrojům platformy spolu komunikovat a využívat navzájem svoje služby. V textu jsou prozkoum8na současná řešení a protokoly založené na modelu Publish-Subscribe. Dále také práce specifikuje požadavky na komunikační sběrnici vyhovující potřebám platformy Testos a také navrhuje řešení pro splnění daných požadavků.

V rámci výzkumu byly prozkoumány tři řešení - MQTT, DDS a AMQP. Průzkum každého řešení byl zaměřen na způsob komunikace a hlavní funkční prvky. Jako startovací bod implementace sběrnice byl vybrán protokol MQTT. Tato práce také specifikuje jak tento protokol rozšířit, aby byly splněny požadavky na sběrnici.

Mezi stěžejní rozšíření navrhnuté v rámci této práce patří management životního cyklu požadavků rozšiřující způsob doručování zpráv a mechanismus zasílání požadavků standardu MQTT. Protokol byl také rozšířen o možnost shlukování zpráv do BULK paketu za účelem snížení množství potřebných síťových zdrojů.

Výsledkem je Testos Bus, který je postavený na upraveném a rozšířeném protokolu MQTT, který zahrunuje implementaci brokeru a klientských knihoven pro Python a C++. Testos Bus naplňuje všechny povinné požadavky platformy, což ověřují automatické testy.

Keywords

Communication bus, communication middleware, messaging, MQTT protocol, publish-subscribe pattern, request management.

Klíčová slova

Komunikační middleware, komunikační sběrnice, management požadavků, MQTT protokol, Publish-Subscribe model, zasílání zpráv.

Reference

ČERVINKA, Radim. *Middleware for Testos platform.* Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Aleš Smrčka, Ph.D.

Rozšířený abstrakt

Platforma Testos se skládá z nástrojů napomáhající automatizaci testování, které může probíhat na různách úrovních, od jednotkového testování po testování uživatelského rozhraní a akceptační testy. V současnou chvíli jsou nástroje odkázáni na znalost lokace ostatních nástrojů, aby s nimi mohly komunikovat. Cílem této práce je vytvořit komunikační sběrnici, která by komunikaci zjednudušila. Její přínos je znatelný například v situaci, kdy generátor testovací sady potřebuje provést náročný matematický výpočet. Namísto nutnosti implementace této funkcionality v daném generátoru je možné využít již existující nástroj platformy, který je dělaný na dané výpočty a výsledku dosáhne mnohem rychleji. Generátoru testovací sady pak stačí se přes dotázat o výsledek problému kolegy a sám doručí výslednou sadu testovacích případů v kratším čase.

V rámci této práce bylo potřeba definovat jednotlivé požadavky platformy, mezi které například patří možnost zasílání požadavků a správa jejich životních cyklů včetně schopnosti doručení odpovědi, možnost odložení doručení zpráv v případě absence příjemce, možnost zrušit běžící požadavek, možnost označení komunikace jako prioritní anebo schopnosti propojit více instancí sběrnic a možnost komunikace s klientskými uzly na sousedních instancích sběrnice.

Na základě těchto požadavků byly prozkoumány některé existující protokoly a řešení používájící Publish-Subscribe protokol, který umožňuje komunikovat pomocí tzv. témat, ke kterým se jednotlivé služby přihlásí jako odběratelé zpráv a požadavků. Producentům různých zpráv a požadavků pak jen stačí poslat sběrnici zprávu se speicfikovaným tématem a sběrnice se sama postará o doručení dat správným odběratelům. V této práci věnuji pozornost třem existujícím řešením - MQTT, DDS a AMQP. U každého řešení bylo potřeba zjistit, jakým způsobem probíhá komunikace a jaké funkcionality nabízí, aby pak šlo následně posoudit, které požadavky platformy řešení splňuje a jak snadné je dané řešení upravit nebo rozšířit tak, aby splnilo požadavky, které nesplňuje samo o sobe. Z výše zmíněných řešení byl vybrán protokol MQTT, který byl nejméně komplexní, pokrýval výrazné množství požadavků a bylo velmi jednoduché jednotlivé funkcionality upravit, vypustit anebo přidat něco úplně nového.

Úpravy protokolu MQTT zahrnovaly zjednodušení nebo vypuštění některých kontrolních paketů, přidání některých polí (např. přepínač priority, přepínač pro odlišení požadavků a odpovědí od běžných zpráv), přidání nových typů zpráv, jako třeba typ BULK sloužící k zabalení více zpráv do jedné. Ke shlukování zpráv do BULKu dochází ve chvíli, kdy frekvence odchozích zpráv překročí určitou mez. Při překročení meze nedochází k odesílání zpráv, ale v rámci krátkého časového okna jsou zprávy seskupovány do jedné zprávy typu BULK, která je odeslána po uplynutí daného časového okna. Tento přístup slouží k úspoře síťových zdrojů na úkor prodloužení doby získání odpovědi maximálně o délku trvání časového okna. Dále broker získal schopnost uložit zprávy, požadavky a odpovědi v případě, že žádný příjemce není dostupný (prozatím je ukládá do hlavní paměti). Brokery je také možné propojit do plně propojené sítě. Sousedi si mezi sebou přeposílají komunikaci a zajišťují tím jednotlivým klientům možnost komunikovat s klienty a službami připojenými na sousední broker.

Výsledná sběrnice jménem Testos Bus zajišťuje všechny povinné požadavky sběrnice. Skládá se z brokeru implementovaného v jazyce C# a dvou klientských knihoven pro C++ a Python. Tyto dva jazyky byly vybrány, protože drtivá většina nástrojů platformy Testos jsou implementovány právě v těchto dvou jazycích. Implementace je doprovozena automatickými testy ověřující základní funkcionalitu a splnění povinných požadavků. Cílem výkonnostních experimentů bylo ozkoušet režijní záběr sběrnice na komunikaci jedné ko-

munikující dvojice rychle generující mnoho požadavků a na komunikaci mnoha souběžných konverzací, kde je generování 10 požadavků za vteřinu.

Experimenty ukázaly, že v případě 200 souběžných konverzací mezi klienty (10 požadavků za vteřinu) je latence získání odpovědi pod 15 ms na lokální síti. V případě jedné osamocené konverzace prokazovala sběrnice minimální dopad režie do rychlosti zhruba 64 zpráv za vteřinu (latence pod 5 ms), při překročení rychlosti 100 požadavků za vteřinu byl dopad režie značný (latence okolo 100 ms), ale dobrou zprávou bylo, že sběrnice komunikaci ustála bez selhání.

Závěr také zmiňuje mnoho možností pro budoucí vývoj sběrnice, jako například zavedení autentizace a autorizace, zpracovávání záznamů logovaných brokerem pomocí log serveru, dynamickou implementaci shlukování zpráv do BULKu pomocí měření odezvy komunikačního kanálu včetně dynamického nastavování parametrů shlukování nebo implementace detekce výskytu komunikujících klientů na stejném přístroji a následné předávání komunikace pomocí hlavní paměti.

Middleware for Testos platform

Declaration

I hereby declare that this master's thesis was prepared as an original work by the author under the supervision of Mr. Aleš Smrčka. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Radim Červinka
May 17, 2021

Acknowledgements

I would like to thank my supervisor, Mr. Aleš Smrčka, for the supervision, time and effort he invested in me when I was working on this thesis. This thesis would not have been finished without him.

Contents

1 Introduction **3**

2 Platform Description and Bus Requirements **4**
2.1 Mandatory Requirements 4
2.1.1 Messaging and Requests 4
2.1.2 Bus Instance Connection 5
2.1.3 Client Libraries 6
2.1.4 Logging 6
2.1.5 Specifiable Limit of Connected Clients to Bus Instance 6
2.1.6 Control of Memory Capacity Limit 6
2.2 Optional Requirements 7
2.2.1 Authentication and Authorization 7
2.2.2 Communication Integrity and Privacy 7
2.2.3 Service Load Balancing 7
2.2.4 Monitoring 7

3 Communication Basis for the Bus **8**
3.1 Publish-Subscribe Pattern 8
3.1.1 Message Filtering 8
3.1.2 Advantages of Publisher-Subscribe Pattern 9
3.2 Message-Oriented Middleware Solutions 9
3.2.1 MQTT 9
3.2.2 Data Distribution Service 12
3.2.3 Advanced Message Queuing Protocol v1.0 14

4 Possible Requirement Satisfaction Solutions **16**
4.1 Requests 16
4.2 Bulk Messaging 17
4.3 Bus Instance Bridging 18
4.4 Message Priority 19
4.5 Subscriber Load Balancing 19
4.5.1 Load Balancing Strategies 20
4.6 Logging 20
4.7 Authentication and Authorization 20

5 Implementation and Evaluation **21**
5.1 Testos Bus Features 21
5.1.1 Communication Model 21

5.1.2 Messaging . . . 22
5.1.3 Request/Response Pattern . . . 22
5.1.4 Bus Instance Connection . . . 23
5.1.5 Message Bulking . . . 23
5.2 Communication Protocol Messages . . . 23
5.2.1 Message Format . . . 24
5.2.2 Variable Length Integer and String Encoding . . . 24
5.2.3 CONNECT and CONNACK Messages . . . 24
5.2.4 PUBLISH Message . . . 25
5.2.5 MSGACK Message . . . 26
5.2.6 REQACK Message . . . 26
5.2.7 BULK Message . . . 27
5.2.8 SUBSCRIBE and SUBACK Messages . . . 27
5.2.9 UNSUBSCRIBE and UNSUBACK Messages . . . 28
5.2.10 PINGREQ and PINGRESP Messages . . . 28
5.2.11 DISCONNECT Message . . . 29
5.3 Broker . . . 30
5.3.1 Configurable Broker Parameters . . . 30
5.3.2 Bus Instance Interconnection . . . 31
5.3.3 Client Connection . . . 31
5.3.4 Subscription and Unsubscription . . . 31
5.3.5 Messaging . . . 32
5.3.6 Message Storage . . . 32
5.3.7 Logging . . . 33
5.4 Client Libraries . . . 33
5.4.1 Client Components . . . 33
5.4.2 Client Creation and Connection . . . 34
5.4.3 Subscribing and Unsubscribing . . . 35
5.4.4 Publishing Messages . . . 37
5.4.5 Receiving Responses to Requests . . . 38
5.4.6 Canceling Requests . . . 39
5.4.7 Disconnection . . . 39
5.5 Automated tests . . . 40
5.6 Performance testing . . . 40

6 Conclusion **43**

Bibliography **44**

Chapter 1

Introduction

This thesis explores the topic of using a software to facilitate easy communication among independent components of the Testos platform such as databases, data generators, servers, individual tools and services. The software would enable each component to use services provided by other components to finish its tasks faster. A good example is a test set generator that sometimes needs to solve some complicated math problems in order to generate the set. It could use the new communication tool to contact a platform component designed for solving given math problems very fast in order to get the result faster than computing it itself.

Current communication middleware solutions already offer superb performance, scalability and many features, configurations and options, how to utilize them, but some of them are relatively complex and take a significant amount of time to learn how to deploy or how to incorporate them into programs in order to make them communicate with each other. On the other hand, solutions with less complexity usually do not offer features that are crucial for specific use cases.

The goal is to create a lightweight communication bus that satisfies requirements and needs of the platform by adopting an existing approach and adding features that are not part of that particular solution or model. It is also substantial to offer client libraries that are easy to use in applications so that the development time is spent more on feature development rather than communication interface.

This thesis consists of five other chapters. Chapter 2 specifies and describes the bus requirements that emerged from the needs of the platform. The following chapter, Chapter 3 explores the Publish-Subscribe pattern and its applications as a communication basis for the Testos Bus. Chapter 4 explores adjustments of the MQTT protocol needed to perform to satisfy the requirements identified in Chapter 2 Chapter 5 describes the implemented broker and client libraries. Whole thesis is enclosed with Chapter 6 which contains a short conclusion.

Chapter 2

Platform Description and Bus Requirements

Testos (Test Tool Set) platform [11] supports the automation of software testing. Tools within the platform combine different levels of testing (from unit to acceptance testing) with various categories of testing, such as model-based testing, requirement-based testing, GUI testing, data-based testing, and execution-based testing with dynamic analysis.

The purpose of the communication bus is to enable easy access to available services for Testos tools and clients that would like to use them. The client setup and connection should not be complex because it is substantial not to waste developer's time on connecting to a platform that should help to automate and ease the testing process.

This chapter specifies requirements mandatory and optional for satisfaction of basic needs of the platform. They are defined in separate paragraphs and in most cases the description is accompanied by an explanation why the requirement is important or what is achieved by fulfilling it. Most of them are functional and connected to data transfer or security, the rest covers requirements essential e.g. for serviceability of the bus (Subsection 2.1.4 and 2.2.4).

2.1 Mandatory Requirements

Requirements in the following subsections are essential so that the Bus is able to satisfy basic needs of the Testos platform. Most of the requirements are functional apart from those specified in Subsections 2.1.3, 2.1.4, 2.1.5, and 2.1.6.

2.1.1 Messaging and Requests

The primary function of a communication bus is enabling exchange of messages between the connected clients. This should be implemented via the Publish-Subscribe pattern in order to mitigate the need of knowing the precise location of a service a client wants to use. There is a Section 3.1 explaining what the Publish-Subscribe pattern is and what are its advantages.

Request Management

Another fundamental requirement for the Testos bus is clients' ability to send requests as a specific message. The bus should offer a mechanism that ensures the request publisher

receives a response for its request from the request topic subscriber that received and processed the request. The bus should also inform the request publisher about an error that occurred during request handling, processing, or computation of a result.

Waiting for Topic Subscriber Availability

It should be possible to mark a message or request so that the bus waits with its delivery when there is no subscriber available at the moment instead of dropping the message or request.

Binary Priority of Messages

There should be a possibility to mark a message or request as priority communication and it should be handled prior to the non-priority messages or requests. Two levels of priority are sufficient - regular and priority messages and requests.

Request Timeout

It is important to be able to specify the timeout or expiration period for requests so the bus can cancel the request (and delete it if the request is waiting for an available subscriber) after expiration and inform the request publisher that the timeout occurred.

Request Cancel

Clients should be able to cancel their ongoing requests even before a timeout occurs (if any timeout was specified). This is very useful in situations when the response computation takes longer than a few seconds and the publisher does not require the result anymore (e.g. some kind of error occurred on the publisher's side, it got canceled by a person managing the publishing client, etc.).

Sending Bulks of Messages and Requests

It is also important for the platform that clients can publish messages and requests in bulk due to overload or performance parameters of the communication link. Sending a bulk of messages benefits from reduction of overhead needed for message transfer.

2.1.2 Bus Instance Connection

The bus instances will be able to accommodate a limited number of connected clients and it is important to be able to connect multiple bus instances in order to scale the number of clients connected to the bus and communicating with each other. It is also possible that different services will run on various bus instances.

The interconnection is very useful in the following situation: A research group develops a new tool that helps with automated testing. To incorporate and utilize it via other existing Testos bus instances, it would need to be instantiated on every single one of them. This way the researches create their own bus instance, connect their services, tools, and devices and then connect their bus instance to other existing ones (or a network of those), which enables everyone to use the newly developed tool and also the institution has access to other institutions' tools and services.

2.1.3 Client Libraries

The bus implementation should also offer client libraries so that it is possible to easily develop new tools and services that can use the features of the bus. The libraries will provide support for Python and C++. Python is very popular high-level language with programming beginners as well as scientists that work with artificial neural networks, image processing do data mining or need some automation scripts. The choice to support C++ was made because it is also very popular, but mainly because it facilitates creation of high-performance solutions. Another important factor was that most of current Testos tools are written in these two languages.

2.1.4 Logging

The bus instance needs to be serviceable in order to be usable in the real world. The base feature that helps with fault identification is logging. The instance should log various events such as:

- successful receiving, storing, restoring and delivering request response to request publisher,
- canceling request by its publisher,
- successful receiving, storing, restoring and delivering message to subscribers,
- error occurring during request handling or message delivery,
- successful/unsuccessful subscription and subscription cancel,
- successful/unsuccessful connection and disconnection,
- successful/unsuccessful authentication,
- unauthorized action performed,
- or discarding messages due to overload.

2.1.5 Specifiable Limit of Connected Clients to Bus Instance

As mentioned earlier, the bus instance is able to hold a limited amount of connected clients, that amount should be specifiable when running the bus instance, the administrator will be able to limit this threshold according to the computational power that will be available. After the limit is reached, the instance will decline new connections and if connected to other instance, it can provide such information so the client connect to the neighbor instance.

2.1.6 Control of Memory Capacity Limit

While running the service, it is possible that the memory capacity will be full and it will not be possible to store or process more messages or requests (e.g. when waiting for an available subscriber). This situation should be handled by the bus and also the bus should take in consideration priority messages and requests, those should not be dropped in case there is a possibility to drop non-prioritized messages or requests. Plain messages could be dropped prior to requests and responses. Memory capacity limit can be specified as a parameter that can be changed during runtime.

2.2 Optional Requirements

Following requirements describe features, that are important but not essential in order to use the Bus as a communication mediator in the Testos platform. These features might be satisfied in future development. Requirement described in Subsection 2.2.3 is function, those in Subsection 2.2.2 and 2.2.4 are non-functional.

2.2.1 Authentication and Authorization

In order to be able to check the identities of bus instances and individual clients, the bus requires a mechanism that clients will use to authenticate against the bus and also to authenticate the bus against the clients.

The clients' ability to use various bus features should be conditioned by each client's authorization to use such features or topics. This is important to restrict access to bus configuration, logs, or monitoring for most users, but also be able to enable access to such features or data for clients privileged by the bus administrator.

2.2.2 Communication Integrity and Privacy

Clients using Testos Bus for communication exchange and request handling might transmit sensitive data in the message payloads. Also receiving manipulated data is critical for the platform reliability. The Bus should provide mechanisms to ensure communication integrity and privacy.

2.2.3 Service Load Balancing

When offering a service, it is very beneficial to deploy multiple instances of the same service. The Bus should offer a mechanism that enables clients to publish messages and requests that arrive at only one instance, that is not currently busy with computation related to a different request.

2.2.4 Monitoring

When running an instance of the Bus, it is important for the instance operator to be able to monitor the state of the Bus. It should be possible to use some user interface to check following pieces of information:

- connected clients and how much traffic they transmit via the instance,
- what topics are being used and which clients are subscribed to them,
- how many messages and how much data was transferred via given topics
- contents of message storage,
- or instance up-time.

Chapter 3

Communication Basis for the Bus

This chapter explores the Publish-Subscribe pattern and message-oriented middleware solutions in order to find a foundation on top of which it is possible to build the Testos Bus. First we look at the Publish-Subscribe principals and why it is beneficial to use in the TBus. After that we look at modern and popular middleware solutions, describe their communication approach and also their pros and cons which play an important role in deciding which protocol to implement in the TBus.

3.1 Publish-Subscribe Pattern

The publish-subscribe pattern (*PubSub* pattern for short) [14] offers a mechanism which enables senders of messages to communicate without the need of knowing the message recipients. The messages are categorized into classes, the sender (called publisher) simply publishes a message and it is delivered to every receiver (called subscriber) that is interested in receiving given class of messages, the interest is shown by subscription to given class of messages.

3.1.1 Message Filtering

Messages are being filtered in order to determine which subscribers should receive them. This can be performed by two common filtering approaches - topic-based and content-based.

Topic-based Filtering

This approach uses classes called *topics*. A topic is specified by a string with a hierarchical structure. The publishers are responsible for topic creation and also for specification to which topic a published message belongs. The subscribers subscribe to topics in order to received messages published to given topic. Given the hierarchical structure it is usually possible to use some kind of a wildcard in order to specify a group of topics with a common trait.

Content-based Filtering

Content-based filtering uses attribute or message content constraints defined by the subscribers in order to determine, who is interested in a message. As we can see here, the roles of message classification switched, in this approach it is the subscriber who is responsible, whereas the publisher is responsible for message classification in the topic-based approach.

3.1.2 Advantages of Publisher-Subscribe Pattern

The PubSub pattern has two main advantages against the traditional client-server architecture [10]. One of them is decoupling in three dimensions:

- **space decoupling** - publisher and subscriber don't need to know each other's location (e.g. IP adress and port number) in order to exchange messages,
- **time decoupling** - publisher and subscriber don't need to run at the same time, if we want to be able to transit messages between them, we just need to implement some kind of storage on the wait so that the message can wait for the subscriber to come online,
- **synchronization decoupling** - operations on both sides don't need to be interrupted during publishing or receiving.

The second main advantage is scalability. The operations on the message broker can be parallelized, it is also benefitial to use message caching and intelligent message routing. Scaling up to millions of connections is still challenging but possible via broker node clustering and load balancing.

Testos platform benefits from this pattern because it is very simple to connect a new application and use all available services only based on knowledge of available topics and the message formats. It is very easy to add more services instances, the clients do not need to know which one to choose or how many are available, the topology can be very dynamic without noticeable impact on the clients.

3.2 Message-Oriented Middleware Solutions

This section describes three popular solutions of message-oriented middleware - MQTT, Data Distribution Service (DDS) and Advanced Message Queuing Protocol (AMQP). Every solution's subsection describes its basic principles and features, mainly focused on communitation. It also contains a subsection shortly describing advantages and disadvantages of using given solutions as a Testos Bus implementation foundation.

3.2.1 MQTT

The MQTT [3][12] is an OASIS [4] standard messaging protocol for the Internet of Things (IoT), it implements the PubSub pattern. It was designed to be lightweight, the aim was to make MQTT clients very small, require minimal resources and network bandwidth so it can be effectively used by a server as well as an IoT device. The communication between client devices and cloud is bi-directional, supports 3 QoS levels and also works over unreliable networks, because it supports storing session information (e.g. which topics is the client subscribed to) or last topic message, which eases the reconnection of the client, the client does not need to subscribe again to topics in which it is interested in and also does not miss the last important message for given topic (e.g. latest status update of a sensor). The standard also does not omit authentication, authorization and secure communication options, which the standard describes as non-normative. This subsection was adapted based on information from the MQTT standard [12].

Architecture and Communication

The architecture of MQTT network is very simple, it consists of a broker and clients. The broker is a central piece of the network, it accepts new connections, subscriptions, it processes published messages and sends them to clients that subscribed to the message's topic. Client simply connects to the broker and if that succeeds, it can use SUBSCRIBE message in order to subscribe to one or more topics. Clients can also publish messages using PUBLISH message, that is used to transfer the message from a publisher to the broker and also from the broker to all subscribers.

It is also possible to create a persistent session between a broker and a client. It can be set up via the *Clean Start* flag in the CONNECT message. If the flag is set to 1, both sides delete any existing session information and start a new one. If the flag is set to 0, then the broker stores all client's subscriptions and all new undelivered messages with QoS level 1 or 2. The client also stores messages undelivered to the broker with QoS level 1 or 2. Upon reconnection, the broker continues the previous session if there are any data stored (and the flag is set to 0). This feature eases the reconnection process which is beneficial for lightweight clients, because the client does not need to subscribe again to all the topics it is interested in. This approach also saves the bandwidth between the client and the broker.

Topics

Topics are UTF-8 encoded strings that create a hierarchy using the forward slash ('/' U+002F) as a level separator. Topic must contain at least 1 character, it is case sensitive and permits usage of spaces.

MQTT topics have an advantage that they don't need to be declared before you publish to them, which increases flexibility and enables easier usage of the wildcards. The wildcards are usable only when subscribing There is a single-level wildcard - the plus sign ('+' U+002B). It matches one topic level, such as `home/first_floor/+/temp` would match temperature topics of all rooms in the first floor. There is also a multi-level wildcard - the number sign ('#' U+0023). This one matches any number of topic levels, e.g. `home/first_floor/#` would match topics such as `home/first_floor/living_room/temp`, `home/first_floor/kitchen/smoke_detector` or `home/first_floor/motion_sensor`.

MQTT also offers a special group of topics that are excluded from the wildcard matching. Those topics start with the dollar sign - ('$' U+0024). The standard states that the server (broker) should prevent clients from using these topics to exchange messages with other clients. This gives opportunity to implementations to use such topics for other purposes, e.g. the `$SYS/` prefix is being widely adopted for topics for server-specific information exchange, control API etc.

Quality of Service

MQTT defines 3 QoS levels for message transfer - level 0, 1 and 2. Level 0 (also called *at most once delivery*) only offers a delivery according to the capabilities of the underlying network. There is no response from the receiver and no retry effort from the sender, on this level, the messages arrive either once or not at all.

Level 1 (also called *at least once delivery*) guarantees that the message arrives at the least once to the receiver. The messages get acknowledged with a PUBACK message. It is possible that the sender retries to send the message before it receives the acknowledgement, then it is possible that the receiver can obtain multiple copies of the same message. Duplicate

messages are recognizable by the DUP flag in the message header, the original has this flag set to 0, duplicates have it set to 1.

Level 2 (also called *exactly once delivery*) ensures that the message arrives exactly ones. It is the highest level of Quality of serviec and it is valuable in situations, when loss nor duplication is acceptable. It uses a 4-way mechanism using PUBLISH, PUBREC, PUBREL and PUBCOMP messages respectively. Only after these messages are exchanged, both sides can consider the message as delivered and the sender can discard all stored information.

Quality of Service is always applied between two communicating sides. Clients cannot define QoS for the whole delivery, they can only define QoS level for their communication with the broker. They define QoS of published messages on the way from the publisher to the broker and also QoS level of given subscriptions, the level with which the broker will deliver messages to them. It is not possible to set up a QoS level for the topic, once the message leaves the client and arrives at the broker, the QoS level of message delivery between the broker and topic subscribers is defined by their individual subscriptions. It is possible that one client publishes a message with the QoS level 2, but one subscriber is receiving messages from given topic with QoS level 0, another one with QoS level 1 and others with QoS level 2, all according to which level they defined during the subscription process.

Request-Response Mechanism

Request management is a very important requirement of the Testos Bus. In version 5, the MQTT offers a very simple mechanism to send a request and receive an answer for that request. Publisher can use the *Response Topic* field in the PUBLISH message in order to specify topic name that it listens to for a response. The message can also contain a *Correlation Data* field that helps with matching a response to its request. The client that received the request now knows to which to send the response, it is also important to add the correlation data if it was specified in the request. This mechanism is too simple to satisfy all request-related requirements of the Bus, but this concept is a good foundation to build on.

Retained message

MQTT offers a mechanism of a retained message. It is specified by a RETAIN flag in a normal PUBLISH message and what is does is that the broker stores this message as kind of a "last known good value" for the topic it is published to. The broker saves only the last one for given topic and the retained message is also sent to a new subscriber after a successful subscription. This feature is very useful in situation when the topic transmits messages that serve as a status update or status report since every new subscriber can immediately get the last known status of the publisher (e.g. a temperature sensor).

Shared subscriptions

The standard version 5 offers a new type of a subscription that can be associated with multiple connections. This type of subscription delivers the message only to one of the subscribers, not all of them, which performs the client load balancing. It differs from a regular subscription via the topic filter format, which is `$share/{ShareName}/{filter}`. The string always stars with `$share` indicating that this not a regular subscription. Then it is followed by *ShareName* that serves as kind of a group identifier, a message arrives only

to one member of the group. The ShareName must not contain the characters '/', '+' nor '#'. The last part of the shared subscription topic filter string is the *filter* part, it represents the topic filter that a client would use in order to subscribe to the same topic regularly. It is possible to use regular subscription as well as the shared one to subscribe to the same topic. The broker sends a copy of a published message to each regularly-subscribed client as well as to one client from each shared subscription group.

MQTT as a Testos Bus Foundation

MQTT is missing a lot of features that are required from the Testos Bus, on the other hand it is a very straight-forward implementation of the PubSub pattern and it is very simple to build additional features upon it. Centralized architecture based on a broker is instrumental in moving a lot of logic and computational requirements away from clients which makes a big contribution to building lightweight and easy-to-use client libraries that promise simplicity for the programmers developing clients and applications using the Bus. MQTT is also frequently a part of research for the past few years, which offers a base of research papers, articles and implementations from which it is possible to adapt ideas in order to extend the MQTT standard in order to satisfy the requirements.

3.2.2 Data Distribution Service

The Data Distribution Service (DDS) is a middleware protocol and API standard [7] from the Object Management Group (OMG) [5]. It is capable of connecting system components of businesses as well as mission-critical Internet of Things applications. It uses a Data-Centric Publish-Subscribe (DCPS) model which implements the PubSub pattern using messages that also include the contextual information a receiver needs in order to understand the received data. When using a more traditional approach that is message-centric, the programmer writes code that sends messages, but when using the data-centric middleware, the code is written to specify how and when to share the data and the DDS directly implements data sharing that is controlled, managed and secure. It also offers a discovery protocol to help the programmers find other communication participants and ease the development with this plug-and-play feature.

Communication

Every communication is conceptually restricted by *domain*. Components can only communicate with other components within the same domain which is identified by a unique integer ID. To communicate within a domain, the application has to create a *DomainParticipant*. It is possible to create multiple DomainParticipants within a single application in order to create components to communicate across multiple domains. DDS uses *Topics* as a communication medium for message exchange. A Topic has a unique identifier, quality of service setting and a type which defines what kind of data is being sent. An application has to create a *Publishers* and *Subscribers* that are connected to one DomainParticipant. Publisher and Subscriber are connected to *DataWriters* and *DataReaders*, which are always dependent on a single Topic and are used for sending and receiving data. The class association is shown in Figure 3.1. This subsection was adapted from [9].

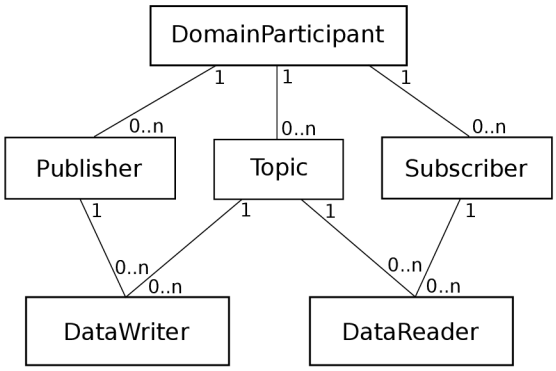

Figure 3.1: DDS communication classes association.

Global Data Space

This subsection was adapted from [6].DDS has a special approach to store and access data, it has a global data space. For the client application, the data space seems to be a local memory that is accessed via API. In reality, reading and writing to the storage sends appropriate messages to update the correct store on remote nodes. This illusion of a global data storage gives freedom to programmers of one global storage, but on the inside utilizes the benefits of decentralized storage such as lower impact of a node on the whole infrastructure or performance of individual nodes.

Discovery protocol

DDS defines a discovery protocol that helps to find relevant Participants and Endpoints, DDS also relies on this mechanism in order to establish communication between according DataWriters and DataReaders. This functionality is described and adapted from [13]. The protocol splits into two independent ones - Participant Discovery Protocol (PDP) and Endpoint Discovery Protocol (EDP). The first one (PDP) specifies how the Participants discover each other in the network. Once they do discover each other, they use EDP to exchange information about the Endpoint they contain. The discovery information is accessible to the user through build-in topics, basically what happens is there are few pre-defined Topics with build-in DataWriters and DataReaders, which are used to announce and consume the presence and assigned quality of service of the local DDS Participant and other entities such as DataWriters and DataReaders.

DDS as a Testos Bus Foundation

One of the main advantage of DDS is the data-centricity and the fact that the topics and messages have pre-defined scheme of transferred data. This would help to keep the format correct, the resource would not be wasted on rejection of transferred data. On the other hand, a topic needs to be declared before the communication can occur, which limits the flexibility of communication, it also prevents the clients from using subscriptions with wildcards. Also DDS offers quite complex features that are not necessary in order to satisfy the Testos Bus requirements. It would also be more complicated in comparison to MQTT to build new features upon the DDS such as request management or waiting for subscriber availability.

3.2.3 Advanced Message Queuing Protocol v1.0

The Advanced Message Queuing Protocol (AMQP) [1] is an OASIS [4] open standard application layer protocol [8]. This section describes the version 1.0, which is the latest, previous version of AMQP significantly differ from this version, version 1.0 offers a different communication approach. AMQP is a wire-level protocol, which means that it describes the format of data sent across the network as a stream of bytes. It was designed as a general-purpose messaging standard. It provides a control flow for the message-oriented communication, it also provides message delivery guarantees *at most once*, *at least once* and *exactly once* as well as authentication and encryption based on TLS and SASL. It is dependent on a reliable transport layer protocol such as TCP. This section was adapted from [8].

Architecture and Communication

The architecture of an AMQP network is quite straight-forward, it consists of *nodes* that are encapsulated within a *container*, containers can hold multiple nodes. An example of a container could be a client application or a broker, within these we could find nodes such as producers, consumers and queues.

In the AMQP network, there are two basic types of data units that travel through the infrastructure - *frames* and *messages*. Frames travel between containers and their purpose is to establish, end and support communication (such as parameter negotiation) and to transfer messages. Frames consist of three parts - a fixed length frame header (8 bytes), a variable length extended header and a variable length frame body. Messages carry application data, they travel between nodes and are encapsulated in frames with the *Transfer* performative. One such frame may encapsulate multiple messages as long as the frame size does not exceed the negotiated maximum frame size. An annotated message consists of header, footer, annotations and a bare message, which consists of standard properties, application properties and opaque binary application data. Nodes are responsible for safe storage and delivery of messages, this responsibility is transferred between the nodes as the message travels through the network.

Two containers are connected with an AMQP *connection*, which is a full-duplex ordered sequence of frames. This connection is divided into *sessions*, the number of sessions is negotiated during connection establishment. Session consists of two *channels*, one is outgoing and the second one is incoming. Each frame contains a channel number which makes it possible to multiplex multiple sessions into one sequence of frames that is transferred over the connection between containers. *Links* are unidirectional and they serve as a communication medium for messages between individual nodes. They are attached to a node at a terminus, which can be a source or a target. A message can travel over a link only if they meet the entry criteria at the source terminus, where filtering happens. Links are attached to sessions in order to communicate with nodes outside of the source node container.

AMQP as a Testos Bus Foundation

The AMQP advantage is that it is general-purpose and it fits a wide variety of use cases, adaptations and extensions. Another benefit it could bring as a building stone of the Testos bus is using the message queue paradigm, because every message is consumed by just one receiver. On the other hand this potential benefit limits the possibility to deliver messages to multiple subscribers as the PubSub pattern does, this feature would have to be built upon the AMQP protocol. There are more disadvantages to using AMQP as a foundation stone

of the Testos bus, the queues and links have to be established prior to the message transfer. PubSub pattern is much more flexible, for example MQTT topics don't need any special prior declaration and it is possible to use wildcards when subscribing to a topic, which covers multiple even yet unknown topic names. Message queues also store messages until they are consumed, which blocks other messages from the queue to be delivered. The Testos bus should be able to postpone message delivery when the receiver is not available and deliver other messages instead. And finally, AMQP communication links and connections are more complex than for example MQTT and it would be much harder to build new things upon that as well as it could be more complex to use the bus for the client developers in order to properly use the AMQP architecture advantages that its complexity brings.

Chapter 4

Possible Requirement Satisfaction Solutions

In this chapter it is elaborated how to satisfy particular Bus requirements. These ideas and solutions build upon the MQTT, which was chosen because it offers a centralized architecture enabling a creation of easy-to-use and lightweight client libraries and a very straight-forward implementation of the PubSub pattern that is easy to build upon and does not contain extra unnecessary features that could not be cut out.

4.1 Requests

Probably the biggest and most important requirement for the Testos bus is the request management. As described earlier, MQTT offers a very simple request/response mechanism in its control message PUBLISH, the client is able to add a field with a topic name, to which the request receiver should send the response. This mechanism alone does not provide any way to wait for a subscriber to join the request topic in case when there is no subscriber subscribed (the only way to somehow store a message/request is to use the retain flag), the request has no timeout and it is not possible to cancel the request by the client that send the request (e.g. when the result is no longer needed). The MQTT broker does not provide any additional request management, it does not know the request state and treats messages with specified *Response topic* field the same way as it would treat any other message.

In order to be able to satisfy such requirements, the broker needs to some kind of request life cycle in order to know, what is the state of a request and what actions can it perform in that state. The life cycle should include request's transfer to the request topic subscriber, response transfer from the request receiver to the response sender and should also cover states when the request is stored in order to wait for any client to subscribe to the request topic.

Following life cycle pictured in Figure 4.1 represents proposed request life cycle that should cover all request related requirements. The middle horizontal line of states (from the *new* state to the *answer delivered* state) shows the most straightforward use case of a request. A client publishes to the bus, the bus delivers the request to a service (a client subscribed to the request topic), the service creates a response, which it published to the response topic and the bus delivers the response to the client, which published the request.

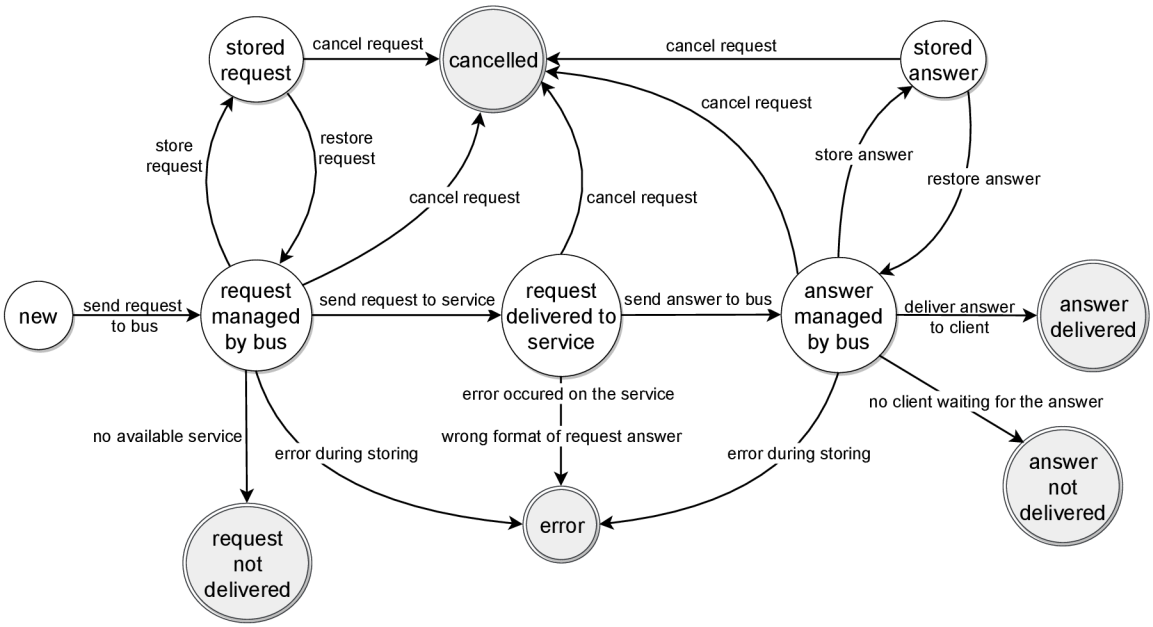

Figure 4.1: Finite state machine representing the request life cycle.

A request or a response for that request can be stored when the given message (a request or a response) was received by the bus, it can also be restored from the storage when the bus is able to deliver the message to its recipient. A request can also end up in a *request not delivered* or a *answer not delivered* state which can occur in case that the request does not specify to be stored in case there is no client able to receive and respond to that request. A request can be canceled until the life cycle has ended. The model also counts with the possible of an error occurrence.

4.2 Bulk Messaging

Sending messages in bulks can be accomplished by introducing a new type of a message, let's call it BULK. The BULK message would be very simple, it just needs a unique message type value and an information how long the whole message is, for which the fixed MQTT header can be used. There is a 4-bit header field for specifying the message type and there are 15 types already specified in the standard, which means that there is still one free value to assign to the BULK message. The payload will be made of individual encoded messages. Included message do not need to be changed, their type and length can again be read after parsing its fixed header.

It is possible to base the bulking strategy on a premise that sending a single big message means a lower load on the network resources than multiple smaller messages transferring the same amount of application data. This is because TCP acknowledges every successfully transferred packet, sending less messages containing more data each result in less acknowledgements sent over the network. The resource load difference is expressed by inequation

$$\frac{C_{raw} + C_o}{t_s} > \frac{N * C_{raw} + C_o}{N * t_s} \tag{4.1}$$

where:

- C_{raw} is the capacity of transmitted raw application,
- C_o is the capacity of overhead data added in order to transfer the application data,
- t_s is the duration of data transfer,
- and N is the number of transmitted messages in a bulk.

The left side of the inequation represents the bit rate needed to transfer single smaller message. The right side represents the bit rate needed to transfer a bulk message containing N smaller messages in the payload. In our case the extra data overhead (C_o) is TCP ACK packet size, which is 66 B. Consider following variable values as an example:

- $C_{raw} = 100$ B,
- $C_o = 66$ B, which is the TCP ACK packet size,
- $t_s = 10$ ms,
- and $N = 3$.

Transferring messages of size 100 B packed in a bulk of 3 messages results in

$$\frac{100 \text{ B} + 66 \text{ B}}{10 \text{ ms}} > \frac{3 * 100 \text{ B} + 66 \text{ B}}{3 * 10 \text{ ms}} \tag{4.2}$$

which results to 16600 B/s > 12200 B/s, where bulking proves to use less network resources. The difference would be even greater if we counted the data packet headers' size into the overhead (C_o).

The disadvantage of bulking is that the messages are not sent right away when they are ready, but they are gathered during a time period - a bulk window. This means that those messages are delivered with a delay, therefore the bulk window should be reasonable small.

The proposed method measures how much traffic is generated on the output. When a certain threshold is reached, the bulking mode is enabled, which means that all outgoing messages are not sent right away, but gathered and concatenated into a BULK message during a bulk window. When the time period passes, gathered BULK message is sent. The threshold triggering the bulking mode and the length of the bulking window can be static values, which results in a communication adapting to rate of outgoing messages. It is possible to use time periods when the rate of outgoing messages is lower and utilize them to measure RTT of a PING message and calculate the threshold and the bulking window length in order to achieve a configurable amount of network load resource reduction. The Testos Bus implements the bulking approach using static values for the threshold and the bulking window length.

4.3 Bus Instance Bridging

Architecture of the MQTT specified by the standard is based on a broker, to which all the clients are connected. The standard does not describe any way of connecting brokers in a network in order to create decentralized bus instance network. The instances would need to share what subscriptions they manage. There are several possible architectures of such network of instances. There are research papers on this topic such as [15] that experiments with linear and star topology. The linear topology is a series of connected brokers, each

broker has 2 neighbors, only the first and the last one in the series have just one. The star topology has 1 broker in the middle and all the other brokers are connected to the middle one. Both topologies need to know about all broker locations in case their neighbor is not available so they can reconnect. In case of the linear topology that would mean trying to connect to the next broker in the series (neighbor of the unavailable neighbor), in case of the star topology that would mean choosing new middle broker. In both cases the locations of the brokers can be ordered so that it is possible to choose the replacement for the missing one. The linear topology has the advantage that the nodes have similar flow rate (the middle node in the star topology way more burdened than the other nodes) and it requires less reconnections after node failure.

Another possible approach would be creating a Full-Mesh network where every instance is connected to each other. Instances would offer their subscriptions as well as a list of their neighbors, the list will be used in order to find new neighbors a new instances is not connected to, yet. This architecture would grant the best resistance to node failure, on the other hand each broker would need to check its own subscription sets as well as each neighbor's with every published message, because it would need to determine, if the message should travel to neighbors or not. A variation could be that all published messages are automatically send to neighbors, this would mean generating more traffic, but message processing would be faster.

An important question arises about the message and request management - who will be responsible? The most straight-forward proposal would be that messages and requests will be managed and stored by the broker which received the message directly from the client and the other brokers will only care about forwarding them.

4.4 Message Priority

Binary message and request priority can be simply specified by a flag in the variable header of the PUBLISH message.

4.5 Subscriber Load Balancing

The idea behind the load balancing requirement is that you can have multiple services that can perform certain task, let's use a simple example: you can have multiple optimized SMT solvers that provide it's services for the platform, they all subscribe to the same topic (e.g. `org/testos/smt_solving`), in that situation your client connects to the bus and publishes an SMT formula to the `org/testos/smt_solving` topic, it does not care who performs the calculations, it only cares about the result. The load balancing here would provide the possibility to choose one of the subscribers that is not busy at the moment (because there would be multiple publishing clients using the SMT solving services of the platform) and sends the request only to that one particular client, which will compute a result and send back the response.

This functionality usage would require special type of topic, which the bus treats in way that it only forwards published messages or request only to one of the subscribers (which is different than the general publish-subscribe approach), or a message type or field that indicates a load balanced message that should be delivered only to one of the subscribers regardless of the topic. In both cases it is bus' responsibility to determine the receiver based on an implemented scheduling strategy, the request receiver does not need to check with

other subscribers whether it is the only one who received the request, because that would be against the very basic idea and advantage of the Publish-Subscribe pattern.

MQTT already offers *Shared subscriptions* in version 5 that satisfy this requirement, however it supports combinations of regular and shared subscriptions to the same topic as well as multiple shared subscription groups within a same topic. However the standard does not specify which load balancing strategy to use.

4.5.1 Load Balancing Strategies

The selection of a subscriber to deliver the request to could be done in a few different ways. The most basic one would be having a list of them (e.g. ordered by the time they subscribed) and send the request to the subscriber, which joined the given topic first and is not busy. This approach would burden clients at the top of the list much more often than the ones at the bottom.

The unbalanced usage of subscribers from the previous method could be mitigated by using again a list of subscribers, but this time it would cycle through the list and always choose the next subscribed client, that is not busy. The client to receive the request could also been chosen randomly from a pool of subscribers (again that are not busy).

A little more interesting approach would be keeping a record of an average response time of each subscriber. This would allow us to choose the fastest client, that is not busy. The advantage would be potential decrease of response time for the client that published the request, but tracking an average response time of subscribers does not take into account that different requests could take completely different time to compute on the same client (e.g. formula satisfiability). This method also requires more computation to be done by the bus.

4.6 Logging

Logging is understandably not part of the MQTT standard since this topic is part of development areas such as serviceability of a product. It is possible to store and maintain the logs by the server which is logging its activity, but this creates more load for the server. Better approach is to send all logs to a log server. The log server is responsible for collecting and storing the logs and is also able to offer additional features such as searching and filtering. There are several open-source solutions, one of them is Graylog [2]. Graylog is a log management tool that consists of a log processor which collects the logs, a web UI for users to be able to search for logs, a MongoDB to store configurations and other metadata, and an ElasticSearch which stores all the logs processes queries.

4.7 Authentication and Authorization

These two requirements can be satisfied with a modern approach of a web token. Client authenticates once against a server and receives a token that is from now on included in requests that the client makes against the MQTT broker without the need of using credentials. The token can be sent in the CONNECT message in the *password* field.

Chapter 5

Implementation and Evaluation

This chapter contains a description of the Testos Bus solution. This is the best chapter to get information about how various quirks and features of this middleware were actually implemented. The first section contains a general description and explanation of how individual TBus features work. The next section consists of a declaration of format for every message type used in the TBus. The chapter than continues with an elaboration of broker and client implementations. The chapter is ended with information on how the TBus was tested and what were the results of performance experiments.

5.1 Testos Bus Features

This section brings a high-level overview and explanation of individual TBus features, which provides context for the following sections that discuss specific parts of the solution such as the broker or the client libraries. The section unfolds the communication model, it explains how the messaging and the request/response pattern works, how to interconnect TBus instances or what specific strategy does the TBus use to bundle messages together in a BULK.

5.1.1 Communication Model

The communication is implemented very similarly to the MQTT protocol. The basic architecture consists of a central component called broker. The broker provides all the bus functionality such as subscription, communication forwarding or storing undelivered messages. Other communication participants are the clients. A more complex architecture can be set up when connecting brokers into a network, which is in more detail described in Subsection 5.1.4.

All communicating client connect to the broker via TCP. Right after the TCP connection is established, the client should send a CONNECT message and receive a CONNACK message (see 5.2.3). This message exchange confirms the connection and helps establish the *keepAlive* interval, which is a time period during which any data need to be sent via the connection (it is being kept alive by PINGREQ and PINGRESP messages, see 5.2.10), otherwise it is considered inactive and is ended with a DISCONNECT message (see 5.2.11). A connection is expected to be gracefully ended with a DISCONNECT message in both cases of normal and abnormal disconnection.

After successful CONNECT/CONNACK exchange, it is possible to send or receive PUBLISH messages. In order to receive messages, it is mandatory to subscribe to a topic

using a topic filter. The topic hierarchy is the same as MQTT's, it is possible to divide each level by the forward slash '/', so it is possible to create structured topics such as `org/testos/solvers/smt_solving`. The topic filter used during subscription can contain the MQTT's multi-level '#' wildcard at its end. This wildcard is matching the rest of the topic name. For example, the topic filter `org/testos/solvers/#` matches following topics:

- `org/testos/solvers/smt_solver`,
- `org/testos/solvers/sat_solver/next_combination`
- or `org/testos/solvers/`,

but does not match topics like:

- `org/company/solvers/smt_solver`
- or `org/testos/data_generators/db_gen`.

The MQTT single-level wildcard (specified with a '+' sign) was not adopted, because there is no need for such topic subscriptions in the Testos platform.

In order to stop receiving messages with a previously subscribed topic filter, clients can unsubscribe from that topic filter via the UNSUBSCRIBE message (see 5.2.9).

5.1.2 Messaging

As previously stated, sending messages to other connected clients can be performed via the PUBLISH message (see 5.2.4). Every PUBLISH message can be sent as a priority or non-priority (normal) communication. The priority messages are being process and sent by the broker prior to non-priority ones. Messages are always accompanied by a topic name, which cannot contain the '#' wildcard and is used to identify subscribed clients that should receive the message. This receiver resolution is performed by the broker based on stored topic filters associated with particular client connections.

The broker is able to store the message in case it is not able to deliver it to any subscriber. The message is either sent to someone upon their subscription to a topic filter, that matches message's topic, or is deleted when message's specified timeout period ends.

5.1.3 Request/Response Pattern

It is possible to utilize PUBLISH messages to send requests and receive responses for these messages. The request life cycle can be seen in Figure 4.1. As all publish messages, the request can have a timeout (which means it can also be stored) and it could be marked as a priority communication. The broker manages the request based on a *packetId* property of the PUBLISH message and a client's ID. It adds a *responseDestination* property when delivering it to subscribers. The subscriber computes and returns a response that uses properties from the received PUBLISH message such as priority or response destination.

The request publisher can cancel a request before the response is delivered by using the REQACK message. Stored requests are also canceled when the broker receives a CONNECT message with a set *CleanStart* flag. The cancel is delivered all the way to the request receiver, which does not send a response when the response computation is completed and it received a cancel method.

5.1.4 Bus Instance Connection

It is possible to connect multiple brokers into a network in order to use services of a client connect to a different broker. This connection happens when starting the broker and using `--neighbor-ip` and `--neighbor-port` arguments. The starting broker attempts to create a connection with the neighbor via CONNECT and CONNACK messages. If it was successful, the CONNACK message might use the *serverReference* property to share other broker locations in the network. The starting broker then attempts to connect to all other neighbors in the list in order to create a full-mesh network.

Being connected to at least one neighbor means a change in messaging behaviour, because now the connected clients are able to communicate with client connected to a different broker. When a normal message arrives at a broker, it tries to deliver it to its subscribers and also forwards it to all neighbors. If it fails and the message has a timeout specified, the message is stored. If a neighbor succeeds delivering the message to at least one subscriber, it notifies the broker with a MSGACK (see 5.2.5), which makes the broker delete the message from its storage.

When it comes to request management, the management responsibilities lies on the broker that received the request from one of its clients. When a broker receives a request, first it tries to deliver it to one of its other clients. If that is not possible, it forwards the request to all neighbors. Neighbor uses a REQACK message (see 5.2.6) in order to notify the broker that the request was delivered to at least one subscriber. The broker managing the request changed the request state and waits for a response. The response is routed back via the *responseDestination* property in the PUBLISH message, because it was prefixed by the broker's ID (which is a `<ip address>:<port>` string).

Subscribing to a broker in a broker network results in the SUBSCRIBE message forwarded to every broker in the network. When a broker receives a SUBSCRIBE message from a neighbor, it looks up all messages with a topic that matches the subscription. If it finds any, it sends it to the broker.

5.1.5 Message Bulking

Implemented message bulking takes place on the way from client to the broker and also between brokers. The Testos Bus implements a bulking strategy described in 4.2 using a fixed threshold and fixed bulking window width. Bulking mode is enabled when average time period of 10 last outgoing messages is lower than the fixed predefined threshold of 10 milliseconds. When the bulking mode is enabled, outgoing messages are not sent right away, but they are encoded and concatenated during a fixed bulking window, which lasts 50 milliseconds. After the window ends, concatenated messages are packed into a BULK message (see 5.2.7) and sent to the broker).

5.2 Communication Protocol Messages

Testos Bus communication protocol (TBus protocol for short) is heavily based on MQTT v5.0, but it was customized and simplified to be even less complex. Protocol for the Testos Bus and MQTT are not compatible, since the message format slightly changed. This section introduces individual protocol messages, that the protocol offers.

5.2.1 Message Format

Similarly to the MQTT control packets, the Testos Bus messages consist of a fixed header, variable header, properties and a payload (in this order). Every message starts with the fixed header which is the only mandatory part. It is comprised of a Message Type (4 most significant bits of the first byte), Flags field (4 least significant bits of the first byte) and Remaining Length field. Remaining Length states how many bytes after the fixed header are part of the current message. The method of encoding such values is described in Subsection 5.2.2. The fixed header is visualized in Listing 5.1.

```
              7   6   5   4   3   2   1   0
-----------+---+---+---+---+---+---+---+---+
byte 1     |  Message Type |     Flags     |
-----------+---+---+---+---+---+---+---+---+
byte 2...  |       Remaining Length        |
-----------+---+---+---+---+---+---+---+---+
```

Listing 5.1: Fixed Header

The other three parts of the message are optional and their presence and content depend on the message type. Variable header usually contains some mandatory field which is specific for given type. Properties contain predefined property pairs (type and value). Each message defines which property presence is valid or not. At the beginning of properties there is a byte length of the properties part encoded as a variable length integer.

5.2.2 Variable Length Integer and String Encoding

Variable length integer is encoded as a *variable length integer* - this format uses 7 least significant bits of a byte to encode the value, most significant bit is used as indicator, if we should include the next byte while decoding, those 7 least significant bits from all participating bytes are concatenated in order to get the original value.

Strings are utf-8 encoded in a way that they start with its byte length encoded as a variable length integer, followed by the encoded string value. This is a slight difference from the way MQTT encodes its strings, MQTT restricts the length of the encoded string value to 65535 bytes in order to be able to encode the length into 2 bytes. TBus protocol way of encoding does not restrict the length of the string value, which means that this could be also used as a way of encoding PUBLISH payloads if authors of the communicating clients agree on using these string encode/decode functions provided by the client libraries.

5.2.3 CONNECT and CONNACK Messages

These messages serve as a medium to establish the connection and exchenge the connection parameters. The CONNECT message has a Message Type of value 1. It uses two flags - *CleanStart* (bit 0) and *BrokerConnect* (bit 1). When set, the flag *CleanStart* indicates, that in case the broker has any responses stored, the client does not wish to receive them. The broker deletes stored responses and marks corresponding requests as canceled. The second flag is used when the broker is connecting to its neighbor. The neighbor then knows that the incoming connection does not belong to a client and informs the other broker about its neighbors, so that the connection initiating broker can connect to other participants in the broker network in order to create a full-mesh network.

The CONNECT message fixed header is followed by a variable header with a *keepAlive* value, which is a 2 byte unsigned short integer, which declares a period of time, in which the broker needs to receive any kind of TBus message (can be filled in with PINGREQ messages, see Subsection 5.2.10), otherwise it sends a DISCONNECT message with code 141 = *KeepAliveTimeout* and ends the connection. MQTT CONNECT properties were omitted, because they specify functionality that is not used in TBus protocol. Good example is the Maximum Packet Size, which is much more needed in systems working over UDP (such as real-time systems) or those having connected client with limited resources. Neither of those are relevant in Testos Bus, which is operating over TCP and it is not expected to work with client with limited resources (such as IoT devices). The payload is an encoded string which contains the client ID. The MQTT CONNECT feature called Last Will was also omitted, because it does not add any valuable information for the Testos Bus participants.

The broker expect a CONNECT message right after the TCP connection is established. If it is not sent in a short time period, the broker ends the connection.

The CONNACK message uses a Message Type of value 2 and its flags are not unused. The variable header contains a byte representing the *ConnectReasonCode* (individual values are described in Table 5.1). CONNACK properties can provide more debugging information in case of unsuccessful connection via the *reason string* property. Also when the CONNACK reacts to a CONNECT message from a neighbor (the message has set the *BrokerConnect* flag), the *serverReference* property is used to forward a list of other neighbors in the broker network. The CONNACK message has no payload.

Value	Code Name	Description
0	Success	States a successful subscription.
1-127		Reserved
128	Unspecified Error	An unspecified error occurred while processing of subscription request.
129	Malformed Packet	Received CONNECT message did not have correct format.
130		Reserved
131	Client Limit Reached	The broker has reached a configured limit or concurrent client connections.
132	Connection ID Already In Use	Active connection with the same client ID already exists.

Table 5.1: Connect Reason Codes

5.2.4 PUBLISH Message

This message's purpose is to exchange data between clients. It offers a possibility to send a regular one-way message, a request, for which it expects a response, which is the third kind of this type of messages. The PUBLISH messages utilize a Message Type value of 3. It uses 3 following flags:

- a Request Flag (bit 0), which marks the message as a request, which means that the broker will manage it as one,
- a Priority Flag (bit 1) - marks a priority communication that should be handled and stored prior to non-priority communication

- and a Response Flag (bit 2), this flag marks it as a response to an existing request (based on the packet ID), setting both the Request Flag and a Response Flag is a protocol error.

In case the PUBLISH message is not a response, the variable header contains a topic name (without the wildcard) that is used to identify which subscribers should receive the message. The PUBLISH properties contain:

- a 4-byte unsigned integer *Timeout* which specifies for how long the message is stored in case it could not be delivered to a single subscriber,
- a string *ResponseDestination* that accompanies the request and response messages in order to be able to route the response to the request publisher (it contains its identifier added by the broker),
- a 2-byte unsigned short integer *PacketId* which identifies the request and response together in connection with the *ResponseDestination* that is the client's identifier.

The payload contains raw data that is being delivered.

In comparison to the MQTT's PUBLISH control packet, it does not use MQTT's flags *DUP*, *QoS level* nor *Retain*. Since it does not use QoS levels, there are no duplicates being sent (and also no reason to specify any QoS level). It also omits the MQTT's *Retain* feature, it is not expected that it would be valuable, because the primary aim of the Testos Bus is usage of Request/Response for which it offers better request management.

5.2.5 MSGACK Message

The MSGACK message is a new message type that has no equivalent in the MQTT protocol. Its purpose is to acknowledge message delivery between neighbors. It is used in a situation, when a broker receives a normal message from a client and it is not able to deliver to any of its other clients. It then stores the message based on the *Timeout* property. After that, it adds a *PacketId* to the message and sends it to all of its neighbors. If a neighbor was able to deliver the message to at least one client, it sends back a MSGACK in order to communicate, that the first broker can delete the message from its storage, because there was someone, that received the message. This message uses a Message Type equal to 4, it has no variable header nor payload. It uses two properties in order to identify given message:

- a 2-byte unsigned integer *MessageId*, which is provided by the first broker
- and a string containing a topic name.

5.2.6 REQACK Message

The REQACK message is an another one newly introduced message type. It is utilized to update a request status. The neighbors use it in order to notify a broker managing a request that a request was delivered or canceled. The broker can also notifies the request publisher that their request was removed from the storage (in case of a full storage and the request being selected as a victim). The client uses this message type This message utilizes a Message Type value of 5, it has no flags nor payload. The properties contain:

- a 2-byte unsigned short integer *PacketId*,

- a byte *RequestState* code, that determines the new state of the request (used values can be seen in Table 5.2),
- a string *ResponseDestination* that contains the request publisher's identifier,
- and a string containing a topic name, which is used when delivering a cancel to the subscribers.

Value	Code Name	Description
0-1		Reserved
2	Request Delivered	Inform neighbor that the request was delivered to at least one subscriber.
3-5		Reserved
6	Canceled	Inform the message receiver that the request was canceled by its publisher.
7-9		Reserved
10	Deleted From Storage	Inform the request publisher that the request was selected as a victim and delete when the storage was filled up.

Table 5.2: Request State Codes used in REQACK messages

5.2.7 BULK Message

The BULK message uses a Message Type value 6 and the flags are not used. The Remaining Length value is followed by individual encoded messages which is this message's payload. It does not use any properties or variable header, which makes the BULK very simple do encode and decode. This message type cannot be found in the MQTT protocol, it adds a new functionality to the TBus protocol, enabling it to pack more messages together in order to save network resources.

5.2.8 SUBSCRIBE and SUBACK Messages

The SUBSCRIBE message is used to subscribe with a topic filter in order to receive messages with a topic that corresponds to given topic filter. It uses a Message Type value 8. Flags field in the fixed header are not used. The variable header contains a 2-byte packet identifier. It does not use any properties, which were omitted from the MQTT SUBSCRIBE control packet, because MQTT's subscribe property *Subscribe Identifier* belongs to a feature which was not included in the TBus protocol, because it was not considered as bringing value in comparison with the complexity increase when including such feature.

TBus protocol also does not use MQTT's *Subscription Options*, because they are used to specify features omitted in the TBus protocol, such as QoS. TBus protocol does not acknowledge received PUBLISH messages (as MQTT does in different way on QoS level 1 and 2) which is an equivalent to MQTT's QoS 0. This decision was made based on two main reasons - the first one is that the TBus protocol operates over TCP which provides reliable delivery and the second one is a future extension of communication capabilities by adding a communication acceleration via the main memory, which would be used in situation when the communicating clients are running on the same machine, where extra acknowledging would be slowing the communication down. The payload contains a topic filter encoded as a string with a method described in Subsection 5.2.2.

The SUBACK message is a response to a previous SUBSCRIBE message in order to communicate whether the subscription was successful or not (and potentially what went wrong). It uses a Message Type value 9, the flags are not being used. The variable header consists of a packet identifier, its value is the same as the packet identifier in the SUBSCRIBE in order to identify which subscription attempt it acknowledges. There is a possibility to use a property *Reason string* in order to provide more specific description of an issue and help diagnose the problem in case of unsuccessful subscription. SUBACK payload contains a *SubscriptionReasonCode*, 2-byte value that determines if the subscription was successful or not. Reason codes adopted from MQTT can be found in Table 5.3.

Value	Code Name	Description
0	Success	States a successful subscription.
0-127		Reserved
128	Unspecified Error	An unspecified error occurred while processing of subscription request.
129-142		Reserved
143	Invalid Topic Filter	The topic format was not correct - it contained the # wildcard at an incorrect position.

Table 5.3: Subscription Reason Codes

5.2.9 UNSUBSCRIBE and UNSUBACK Messages

The UNSUBSCRIBE and UNSUBACK message formats are the same as SUBSCRIBE and SUBACK messages, their purpose is to unsubscribe the client from given topic filter. UNSUBSCRIBE message uses the Message Type value 10 and UNSUBACK uses Message Type value 11. The topic filter specified in the SUBSCRIBE message payload needs to match exactly the topic filter, that the client subscribed to earlier. The UNSUBACK reason codes can be found in Table 5.4.

Value	Code Name	Description
0	Success	States a successful subscription.
1-16		Reserved
17	No Subscription Existed	Returned when the client tries to unsubscribe from a topic filter that it wasn't subscribed to.
18-127		Reserved
128	Unspecified Error	An unspecified error occurred while processing of subscription request.

Table 5.4: Unsubscription Reason Codes

5.2.10 PINGREQ and PINGRESP Messages

Both PINGREQ (= ping request, shown in Listing 5.2) and PINGRESP (= ping response, shown in Listing 5.3) messages consist of just a fixed header, that states the Message Type (PINGREQ uses value 12 and PINGRESP uses value 13). The flags are empty and the Remaining Length is set to 0 in both of them. These messages are used to notify and check whether the connection participants are responsive. The broker ends a connection if

there is no incoming message (PINGREQ or) from the side, that initiated the connection, for a period of time, which was agreed on during connection establishment (the *keepAlive* value). The PINGRESP is expected to be sent immediately, so the connection initiator (a client or a neighbor broker) ends the connection if it does not receive a PINGRESP messages in a small predefined time period of 5 seconds. The connection initiator sends the PINGREQ message with a period equal to a half of the agreed *keepAlive* interval.

```
              7   6   5   4   3   2   1   0
-----------+---+---+---+---+---+---+---+---+
byte 1     | 1   1   0   0 |       0       |
-----------+---+---+---+---+---+---+---+---+
byte 2     |               0               |
-----------+---+---+---+---+---+---+---+---+
```

Listing 5.2: PINGREQ message

```
              7   6   5   4   3   2   1   0
-----------+---+---+---+---+---+---+---+---+
byte 1     | 1   1   0   1 |       0       |
-----------+---+---+---+---+---+---+---+---+
byte 2     |               0               |
-----------+---+---+---+---+---+---+---+---+
```

Listing 5.3: PINGRESP message

5.2.11 DISCONNECT Message

The DISCONNECT message is used in order to properly close the connection. It uses a Message Type of value 14, it does not use any flags nor payload. The variable header contains a DisconnectReasonCode - a 2-byte value (individual codes can be found in Table 5.5. The properties can contains a string *Reason*, which can provide more information in case of a abnormal disconnection.

Value	Code Name	Description
0	Normal Disconnection	Sent when a client wants to disconnect without an occurrence of any outstanding situation.
1-16		Reserved
17	No Subscription Existed	Returned when the client tries to unsubscribe from a topic filter that it wasn't subscribed to.
18-127		Reserved
128	Unspecified Error	An unspecified error occurred.
129-140		Reserved
141	KeepAliveTimeout	Sent when one of the communicating sides did not respond during an agreed time period.

Table 5.5: Disconnection Reason Codes

5.3 Broker

The broker was implemented in the C# language. It is a multi-threaded application that uses asynchronous callbacks to accept new connections or receive and process incoming data. It manages connected clients and their subscriptions, it also forwards incoming messages to clients that subscribed for given topics. It is able to store PUBLISH messages in case there is no available client willing to receive them. The broker also manages ongoing requests life cycle. It is possible to connect brokers into a full-mesh network providing the possibility to communicate with clients that are connected to a different broker.

5.3.1 Configurable Broker Parameters

The broker offers a small amount of configuration possibilities:

- parameters `--ip` and `--port` specifying the broker's IP address and port number (default values are `127.0.0.1` for the address and `5035` for the port number),
- parameters `--neighbor-ip` and `--neighbor-port`, that can be used to connect to a neighbor broker (needs a value for both or none),
- parameter `--client-limit`, which limits the maximum number of client that can concurrently connect to the broker instance,
- and a parameter `--log-level`, which determines the minimal level of log entries that are being logged.

The log levels can be seen in Table 5.6 from the most verbose to no logs at all. The client limit does not count in connections to neighbors.

Value	Description
ALL	CLI value to see all types of entries. It is the same as using the TRACE level.
TRACE	Provides very detailed information that could be useful when debugging. Selecting this output level is the same as selecting the ALL level.
DEBUG	Contains a slightly more detailed information than the INFO level, useful for developer when debugging or working on the TBus.
INFO	This level gives a generally useful information of what is going on the broker, the entries should be understandable for everyone understanding how the bus should work from a product perspective.
WARN	A level for entries signaling a potential issue. No harm was done and the TBus works without any limitation.
ERROR	Used when an error occurred, some functionality might be not available or working correctly.
FATAL	Level suitable for entries logged while the TBus stops working completely.
NONE	Does not output any log entries.

Table 5.6: Logging Levels

5.3.2 Bus Instance Interconnection

A broker connects to a neighbor based on the `--neighbor-ip` and `--neighbor-port` CLI arguments. In order to differ the connection from a client connection, it sets the *BrokerConnect* flag in the CONNECT message. Every time a neighbor accepts a connection from a broker, it sends back CONNACK with a *serverReference* property. The value is a string of concatenated `<broker ip>:<port>` pairs that are delimited with a semicolon ';'. The broker initiating the connection then parses this list of broker addresses and ports and attempts to connect to each of them.

Every connection with a neighbor is associated with a separate output queue of messages. This queue is operated by a separate thread, that takes care of dequeueing messages and sending them to the neighbor. When sending the messages, bulking can occur. Bulking is performed the same way as described in Subsection 5.1.5, measuring time periods between last 10 enqueued messages and packing them together during a 50 millisecond window into a BULK message.

5.3.3 Client Connection

When the broker is done connecting to the broker network, it is ready to accept a new connections. It does that via asynchronous function *BeginAccept* (from *System.Net.Sockets*) that uses a provided callback function *NewConnectionCallback* to take care of the new incoming connection. The *NewConnectionCallback* function receives the CONNECT message and attempts to create a new *Connection* in the *ConnectionManager*. The result is then sent back. If the creation failed, the TCP connection is ended. It it succeeded, the broker accepts data via asynchronous function *BeginReceive* (again from *System.Net.Sockets*). Accepting a connection initiated by other broker has the same approach, but the *Connection* object contains a *isNeighbor* variable set to *true* in order to differentiate it from a client connection when processing incoming messages.

The *ProcessIncomingMessage* function is used as a callback when accepting data from a connection. First, this function reads a fixed header from a socket associated with the connection - it reads two bytes, which help determine type of the message, and then read byte after byte until it read the whole *RemainingLength* part of the fixed header. If the remaining length of the header is greater than 0, the rest of the message is read from the socket. The end of reading the whole message is notified via a *MessageEvent* to allow another message to be read. When the whole message content is received, the raw messages is parsed and then processed.

5.3.4 Subscription and Unsubscription

To process a SUBSCRIBE message, the broker tries to add the topic filter and client pair into the *SubscriptionManager*. Result of this operation determines the reason code in the SUBACK message sent back to the subscribing client. The broker also looks up all stored messages (regular messages and requests), whose topic matches the newly subscribed topic filter. These messages are sent to the client.

If the broker has at least one neighbor, the SUBSCRIBE message is also forwarded to all neighbors. Receiving a SUBSCRIBE message from a neighbor is processed differently, no new subscription is registered, but the broker responds to the neighbor with stored messages (regular ones and requests)

UNSUBSCRIBE messages are handled similarly, the broker attempts to remove the topic filter from the *SubscriptionManager*. Result of this operation determines the reason code in the UNSUBACK message. UNSUBSCRIBE messages are not forwarded to any neighbor.

5.3.5 Messaging

When a PUBLISH message is received, it is not processed right way, but pushed into a *Publish Message Queue* that is responsible for processing all incoming PUBLISH messages. The broker instance manages a separate thread that is responsible for removing PUBLISH messages one by one from the queue and processing them. The *Publish Message Queue* internally consists of two separate queues, one is a priority queue and the second is a regular queue. Messages enqueued to the priority queue (based on the *PriorityFlag* in the PUBLISH message fixed header) are processed before all regular communication.

All other types of messages are processed right away without any queue or prioritization. The PUBLISH message is the only one being queued and prioritized, because it is expected to be a dominating majority of traffic. Other types of messages are expected to be a very small fraction of incoming traffic that does not require any further management or ordering.

When processing a PUBLISH message, the broker uses the topic name specified in the messages to identify subscribers, that should receive it. In situation when there is no subscriber to deliver the message to, the messages with specified timeout are stored via the *StorageManager*. Those messages are removed from storage and sent when a client subscribes with a topic filter matching the topic name associated with the stored message.

If the PUBLISH message is a request, the broker also creates a *Request* object, stores and manages request information via the *RequestManager*. The request is uniquely identified by publisher's client ID and message packet ID values. Requests are stored (in case of specified timeout and unsuccessful first delivery) and delivered upon other client's subscription to a matching topic same way a regular PUBLISH message is with a small difference of updating the *Request* status in the *RequestManager*.

If the PUBLISH message is a response, it uses *responseDestination* property in order to deliver it to a specific client based on the client ID. If the client is not available, the response is stored. Stored responses are not removed from storage and delivered upon subscription, but upon connection of a client with a client ID that matched the *responseDestination* value.

When a message arrives from a neighbor, the broker uses MSGACK and REQACK messages according to 5.1.4 in order to share the fact that the message was delivered. All requests are managed by a broker where the request originated.

5.3.6 Message Storage

The message storage is implemented to store messages as objects in the main memory. Default maximum limit of stored messages is 500. This limit can be changed during runtime by connecting according to the TBus Protocol and sending a PUBLISH message with a topic `$memoryLimit` and payload containing the new limit value. This value should be a string (e.g. "1024") that is encoded according to the algorithm in Subsection 5.2.2, it is possible to do this with the *EncodeString* function available as a part of client libraries.

5.3.7 Logging

Logging is printed into the standard output via the *Logger* class. It is divided into levels in order to offer different granularity. These levels are described in Table 5.6. It is possible to specify a level with which the broker outputs log entries via the `--log-level` CLI argument described in Subsection 5.3.1.

5.4 Client Libraries

As stated in the Chapter 2 that describes the requirements, this solution aims at implementing client libraries for C++ and Python languages. The libraries are implemented to be as simple as possible, resulting in few precise functionalities that are can be used very similarly to each other compared between the two language implementations. This section starts with a description of the client architecture which is followed by an explanation of how each functionality, that is offered to the user, works and how to use it. Because the behavior is the same, each offered functionality will be described together for both languages. The description will also contain a demonstration of usage in both C++ and Python.

5.4.1 Client Components

All the ability to communicate via Teston Bus is offered the user via single *TBusClient* class. An instatiated client utilizes multiple threads, each has individual background functionality to perform - listening for incoming messages, sending out outgoing PUBLISH messages, pinging the broker and processing incoming requests. There is also a set of functions to encode and decode data. These functions are heavily used by the client to parse or encode transmitted data, but they can also be used by the user to ease the payload creation or processing. More about user usage of these functions can be found in Subsection 5.4.4.

The *listener* thread simply receives a message by receiving fixed header first and the rest of the message second based on the remaining length specified in the header. After that it attempts to parse the data and performs the action based on the received message. Receiving a CONNACK, REQACK, SUBACK or UNSUBACK results in noting that the acknowledgement arrived (if it was expected), hand over message contents and notifying a condition variable to communicate that the acknowledgement was successfully received. PUBLISH messages are processed based on the type. Requests are put to a queue, that is consumed and processed by the *request processor* thread. Responses are handed over to the client (in case it expects such response) and a condition variable is notified. Normal messages are just used as an argument for a callback that was provided by the user when subscribing to given topic, value returned by the callback is ignored since not being a request means that message publisher does not expect a response.

The *sender* thread consumes a queue of PUBLISH messages. The reason why this thread only operates PUBLISH messages is that they are expected to be an overwhelming majority of outgoing traffic. Other message types are sent directly from client methods without going through a queue. Based on a client's attribute *bulkMode* it either just sends a dequeued message or starts bulking. Bulking consists of concatenating dequeued messages during the bulking time period. When the period ends, concatenated messages are completed with a fixed header containing the BULK message type and the *remaining length* value. The completed BULK message is sent afterwards. The decision whether to bulk or not is made when a message is enqueued. Call client's special publish enqueue function measures

time periods since the last message was enqueued. These time periods are kept for 10 last messages. Every enqueue also includes averaging last 10 periods, which is then compared to a threshold of 10 milliseconds. If the average is lower then the threshold, bulking mode is set until the average increases over the threshold value.

Another important thread is the *request processor* thread which consumes a queue of incoming requests. These requests are enqueued to this particular queue by the *sender* thread while processing PUBLISH messages. After the *request processor* dequeues a request, it uses the request payload as an argument for a callback, that was registered by the user during topic subscription. When the callback returns, the thread checks the request status in case the client received a request cancel from another client that sent that given request, which would lead to simply ignoring the request result and moving on to the next request. If not, the value returned by the callback is sent back as a response using metadata from the request message.

There is also a thread responsible for pinging the broker (via PINGREQ message) in order to make sure that the broker does not close the connection as a result of inactivity. Secondary effect of this thread is also that the client makes sure the broker is responding based on received PINGRESP. The interval used for the pinging is a half of the *keepAlive* interval specified when connecting to the broker.

Lastly, threads are also used when working with timers, which are used for example for tracking whether a sent request timed out. Python offers a built-in *threading.Timer* whereas the C++ library incorporates a simple timer implementation, that was written by Shalitha Suranga [16] and was just slightly adjusted. This timer creates a detached thread that sleeps for given time interval. After the thread wakes up, it check whether the timer was canceled, if it wasn't, it uses a callback provided during timer instantiation. The callback modifies request state to reflect that it timed out.

5.4.2 Client Creation and Connection

The client can be instantiated with three arguments - a string containing client identifier, a string containing IP address of the broker the user will be interacting with, and a port number, on which the broker accepts new connections. There are two optional boolean parameters. First one controls whether the client should print output log messages. The default is that logging is not printed out. Error messages are printed to *stderr* regardless the parameter value. The second optional parameter enables or disables the bulking feature. The default value is that the bulking is enabled.

The connection can be done with a method *connectToBroker()* that accepts two arguments, the first one is a *cleanStart* boolean (with a default value set to *true*) and the second one is an unsigned integer *keepAlive* (with a default value equal to 60). Setting *cleanStart* makes the client forget any previously sent requests, which means you cannot retrieve their status or response. The *keepAlive* specifies a period (in seconds) in which both sides need to receive any message (or a PINGREQ / PINGRESP) in order to not end the connection due to unavailability. The client sends a PINGREQ message every time a half of this time interval passes.

The *connectToBroker()* method performs a connection to the broker, it also awaits a CONNACK for a small predefined time period (5 seconds). It also starts the threads that are responsible for receiving, sending and processing messages. After a successful connection, it starts up the thread responsible for periodic pinging the broker. In case of any

unsuccessful event (unsuccessful socket creation or TCP connection, timed out CONNACK or not receiving a *Success* reason code), the function raises a appropriate exception.

Python Example

```
# client creation with disabled logging and enabled bulking
client = TBusClient("python_client_id", "192.168.0.173", 5035)
client.connect_to_broker(clean_start=True, keep_alive=180)

# client creation with enabled logging and disabled bulking
client = TBusClient("python_client_id", "192.168.0.173", 5035, True, False)
client.connect_to_broker(clean_start=True, keep_alive=180)
```

C++ Example

```
// client creation with disabled logging and enabled bulking
TBusClient client ("cpp_client_id", "192.168.0.173", 5035);
client.connectToBroker(true, 20);

// client creation with enabled logging and disabled bulking
TBusClient client ("cpp_client_id", "192.168.0.173", 5035, true, false);
client.connectToBroker(true, 20);
```

5.4.3 Subscribing and Unsubscribing

For subscription the user needs two things - a string containing a topic filter, that will identify messages that the client is interested in, and a callback function that the client will use to process incoming requests and normal messages. The callback receives an unchanged payload from the request message. If the user expects to process requests on given topic, the callback must return bytes representing a response payload, which will be appended to the response message without any further change by the client library. The user is expected to handle message payload decoding and response encoding on his own. However, the library provides following encode and decode functions that could be used if you are sure that the other side uses them too:

- `encode2Bytes()` / `decode2Bytes()` - encodes/decodes a ushort in 2 bytes,
- `encode4Bytes()` / `decode4Bytes()` - encodes/decodes a uint in 4 bytes,
- `encodeVarLenInt()` / `decodeVarLenInt()` - encodes/decodes a variable length integer in variable number of bytes (described in Subsection 5.2.2)
- and `encodeString()` / `decodeString()` - encodes/decodes a string (also described in Subsection 5.2.2).

The most useful functions are probably the ones operating on a string. Usage of these function is not mandatory, but it could help when transferring data that can be easily represented e.g. as a string or an integer.

In the Python library, the callback is a function that will receive PUBLISH message payload as *bytes*. In case of requests, the callback should also return *bytes*. All encode and decode functions can be found in the *messaging.py* module.

In the C++ library, the callback type must be `void (*callback)(ReceivedMessage*)` - they receive an argument, which is a pointer to a simple structure *ReceivedMessage*, which offers following methods:

- `vector<unsigned char> getPayload()` retrieves the payload of the incoming PUBLISH message,
- `string getTopic()` retrieves the topic which was a part of the PUBLISH message, which could make it easier to create a single callback function processing message payloads from multiple topics,
- `void setResponse(vector<unsigned char> data)` - sets a payload, that will be included by the client in the response PUBLISH message
- and `vector<unsigned char> getResponse()` which is called after the callback returns if the incoming PUBLISH message was a request.

```
class ReceivedMessage {
        std::vector<unsigned char> payload;
        std::string topic;
        std::vector<unsigned char> response;
  public:
        ReceivedMessage(std::vector<unsigned char>, std::string);
        std::vector<unsigned char> getPayload();
        std::string getTopic();
        void setResponse(std::vector<unsigned char> data);
        std::vector<unsigned char> getResponse();
};
```

If a SUBACK with a code indicating success is received, topic and callback are saved among active subscriptions. If the broker does not send a response within a certain period of time (coded as 5 seconds) or the subscription was not successful, an exception is raised.

Python Subscription Example

```
def cb(data):
        number_as_str, byte_len = decode_string(data)
        number = int(number_as_str)
        return encode_string(number*3)

def main():
        client = TBusClient("python_client_id", "192.168.0.173", 5035)
        client.connect_to_broker()
        client.subscribe("tripleValue", cb)
```

C++ Subscription Example

```
void Cb(ReceivedMessage *msg) {
        vector<unsigned char> payload = msg->getPayload();
       int len;
       string data~= decodeString(payload, &len);
       int res = stoi(data) * 3; // calculate result
       msg->setResponse(encodeString(to_string(res)));
}

int main() {
       TBusClient client ("cpp_client", "192.168.0.173", 5035);
       client.connectToBroker();
       client.subscribe("tripleValue", *Cb);
}
```

Unsubscription requires only a topic filter, upon broker timeout or obtaining an unsuccessful code an exception is raised.

Python Unsubscription Example

```
client.unsubscribe("tripleValue")
```

C++ Unsubscription Example

```
client.unsubscribe("tripleValue");
```

5.4.4 Publishing Messages

For sending messages, the client offers two methods to perform this tasks for the user - `publishMessage()` and `publishRequest()`. Both methods take two mandatory arguments - a string with a topic name (it cannot contain the '#' wildcard) and a payload (expecting *bytes* in Python and *vector<unsigned char> in C++*). The methods also take two optional arguments - a boolean priority flag and a timeout. Default value for the priority flag is *False* and for timeout it is 0. Not specifying any timeout value for a normal message means that the broker will try to deliver the message to the subscribers, if there aren't any, the message will not be stored in the broker storage. Not specifying any timeout for a request results in the request being sent with a very low predefined timeout value (same as when waiting for a CONNACK or SUBACK, which is 5 seconds). The user should always add some timeout to requests in order to prevent unnecessary request timeout due to longer processing time on the other client which computes the response. The `publishRequest()` also notes down an ongoing request and starts a timer that takes care of the situation, when the given request times out. There is also a difference in return values, method `publishMessage()` does not return anything whereas `publishRequest()` returns request's packet ID.

Python Publish Example

```
payload = encode_string("test_payload")

# publish a~message
client.publish_message("test_topic", payload, False, 20)
# publish a~request
request_id = client.publish_request("test_topic", payload, True, 60)
```

C++ Publish Example

```
vector<unsigned char> payload = encodeString("42");

// publish a~message
client.publishMessage("test_topic", payload, False, 20);
// publish a~request
ushort requestId = client.publishRequest("test_topic", payload, True, 60);
```

5.4.5 Receiving Responses to Requests

In order to get the result of a request, the user must use the `awaitAndProcessResponse()` method. This method takes two arguments, one of them is the packet ID, which identifies a specific request, and the second one is a callback. The method returns whatever is returned from the callback, since it is used to process the response payload. If the request timed out or was deleted from the broker storage (as a victim when the storage capacity was reached), the method raises an according exception. In Python, the callback should be a function, that takes one argument, which is the raw response payload as *bytes*. In C++, the function callback argument is again a raw response payload as a *vector<unsigned char>*.

Python Receive Response Example

```
def cb_req_pub(data):
        numstr, len = decode_string(data)
        return numstr

def main():
        client = TBusClient("python_client_id", "192.168.0.173", 5035)
        client.connect_to_broker()
        payload = encode_string("test_payload")
        # publish a~request
        request_id = client.publish_request("test_topic", payload, True, 60)
        # await response
        request_result = client.await_and_process_response(request_id, cb_req_pub)
```

C++ Receive Response Example

```
int CallbackPub(vector<unsigned char> response) {
        int len;
        string data~= decodeString(response, &len);
        return stoi(data);
}

int main() {
        TBusClient client ("cpp_client", "192.168.0.173", 5035);
        client.connectToBroker();
        vector<unsigned char> payload = encodeString("42");
        // publish a~request
        ushort requestId = client.publishRequest("test_topic", payload, True, 60);
        // await response
        int requestResult = client.awaitAndProcessResponse(requestId, *CallbackPub);

}
```

5.4.6 Canceling Requests

The user is able to cancel requests which it sent out, but did not get a response for, yet. Basic use case could be that the program using Testos Bus sent a request that takes longer to process and the program does not need the response anymore. The client method only requires a packet ID of a request that was sent out. The method takes care of sending a REQACK message with proper reason code to the broker and marking the request as *canceled*.

Python Cancel Example

```
client.cancel_request(request_id)
```

C++ Cancel Example

```
client.cancelRequest(requestId);
```

5.4.7 Disconnection

Disconnection si performed by method that takes two optional argument, a reason code and a string containing a reason for the disconnection. This reason might help people identify what happened when going through the logs. Default usage for the user is calling the method without specifying arguments, which results in normal peaceful disconnection. The user also can specify the code and reason in order to identify disconnection after an error occurs. Disconnection results in all client threads being stopped and joined. The client also looses all subscriptions.

Python Disconnection Example

```
# normal disconnection
client.disconnect()

# abnormal disconnection
client.disconnect(DisconnectReasonCode.IMPLEMENTATION_SPECIFIC_ERROR,
                  "failed connection to database")
```

C++ Disconnection Example

```
// normal disconnection
client.disconnect();

// abnormal disconnection
client.disconnect(DisconnectReasonCode::IMPLEMENTATION_SPECIFIC_ERROR,
                  "failed connection to database")
```

5.5 Automated tests

The solution also contains automated end-to-end tests validating satisfaction of the basic mandatory requirement specified in the Section 2.1. The tests make sure it is possible to perform basic operations via client libraries and a running broker instance. The Table 5.7 shows a list of requirements and which test cases cover them. The tests check that it is possible to use a feature in a happy path scenario or that an incorrect usage is detected and not permitted. They also check that it is possible to communicate with clients connected to different brokers in the broker network. Single test runs on a separate broker instance so that the test result is not affected by previous tests. Each test scenario is performed with a C++ publisher-subscriber pair and also a Python publisher-subscriber pair.

5.6 Performance testing

Performance testing was aimed to explore the influence of the overhead of a single TBus instance on the communication speed and response time. The testing was conducted on a single computer running the broker and all of the clients, which limits the computation capabilities, but reduces network transfer delay. In order to focus purely on the middleware's overhead and for example not on request payload computation when sending requests via TBus, the experiments were conducted using the simplest publisher and subscriber pairs possible. The subscriber just returns whatever payload it receives in the request. The publisher sends out requests with an encoded string `test` in certain intervals and also has a thread that is checking if the response returned and how long it took to receive it.

There were two experiments conducted. The first was about exploring how the TBus behaves when you have a single publisher-subscriber pair that generates traffic in very small intervals. It was performed by sending a block of messages, after which the speed increased.

Requirement	Corresponding Test Case
Messaging from 2.1.1	messaging/happy_path_two_subs.sh messaging/happy_path_wildcard_sub.sh messaging/multiple_subs_on_client.sh subscription/invalid_filter_sub.sh subscription/invalid_filter_unsub.sh subscription/sub_twice.sh subscription/sub_unsub_happy_path.sh subscription/unsub_without_previous_sub.sh
Request Management from 2.1.1	requests/happy_path.sh requests/wildcard_sub.sh
Waiting for Message Receiver from 2.1.1	request_storing/store_request.sh request_storing/store_response.sh
Request Timeout from 2.1.1	requests/timeout.sh
Request Cancel from 2.1.1	requests/request_cancel.sh
Message Priority from 2.1.1	requests/request_priority.sh
Bulking from 2.1.1	requests/bulking.sh
Bus Instance Connection from 2.1.2	messaging/subs_on_different_nodes.sh messaging/wildcard_sub_different_node.sh requests/happy_path_sub_on_different_node.sh requests/wildcard_sub_on_different_node.sh request_storing/store_request_sub_on_different_node.sh request_storing/store_response_sub_on_different_node.sh
Limit of Connected Clients from 2.1.5	connection/limit_of_connected_clients.sh
Control of Memory Capacity from 2.1.6	request_storing/storage_capacity.sh

Table 5.7: Mapping of requirements to automated test cases.

The Table 5.8 shows that the latency significantly increases when the requests are produced faster than 64 messages per second. The bulking explains the delay when it was enabled, because when collecting messages into a bulk during a 50 ms window increases the latency in order to save network resources. We can also see that disabling bulking when using the C++ client also suffers from a significant latency increase when passing the speed of 64 messages per second.

The curious column is the one containing values for a case when the Python clients were used in combination with disabled bulking, because we don't see any significant latency during the whole process. I believe that the numbers were affected by Python's overall performance and I believe that the client was not able to produce requests that fast. All other data could suffer from client's inability to produce requests fast enough, but that particular case I believe it had a serious impact on the results. The good thing is that the TBus was able to withstand such traffic generated in a single publisher-subscriber pair.

The second experiment was meant to explore what happens when multiple publisher-subscriber pairs communicate via the TBus at the same time. Each publisher used separate topic to communicate with exactly one separate subscriber. Also each publisher was producing 10 messages per second.

First attempt at connecting multiple subscribers and publishers showed a flaw that taking care of a new connection has a significant influence on messages being delayed

client library used	C++		Python	
bulking	enabled	disabled	enabled	disabled
2 messages/sec	2.8157 ms	2.2712 ms	3.8853 ms	3.0776 ms
4 messages/sec	0.7799 ms	0.9591 ms	3.0137 ms	2.4576 ms
8 messages/sec	0.9335 ms	0.9290 ms	3.0141 ms	1.9542 ms
16 messages/sec	0.9564 ms	1.0919 ms	3.0136 ms	2.0971 ms
32 messages/sec	1.0638 ms	1.0347 ms	2.9905 ms	2.0929 ms
64 messages/sec	0.8916 ms	0.9536 ms	2.9046 ms	2.0126 ms
128 messages/sec	52.0214 ms	38.374 ms	29.9026 ms	1.8388 ms
256 messages/sec	77.5221 ms	35.173 ms	75.7054 ms	1.6265 ms
512 messages/sec	77.6337 ms	30.991 ms	68.5393 ms	2.0159 ms
1024 messages/sec	75.1017 ms	34.660 ms	58.6613 ms	2.1736 ms
~2048 messages/sec	27.6548 ms	66.6631 ms	62.5354 ms	3.7976 ms
~4096 messages/sec	61.9949 ms	53.5614 ms	96.3299 ms	2.5598 ms
~8192 messages/sec	98.5921 ms	39.0048 ms	92.87 ms	2.8644 ms
~16384 messages/sec	114.7219 ms	28.0954 ms	90.72 ms	2.8456 ms
~33333 messages/sec	24.7726 ms	31.6303 ms	82.7191 ms	1.3552 ms
~66666 messages/sec	161.1337 ms	61.9033 ms	94.6853 ms	1.2160 ms
~142900 messages/sec	155.0267 ms	94.4913 ms	103.5409 ms	1.2013 ms
~333333 messages/sec	148.9574 ms	120.6753 ms	134.5862 ms	1.2749 ms
~1000000 messages/sec	138.0919 ms	122.5201 ms	96.44 ms	2.2854 ms

Table 5.8: The average latency of requests produced at certain speeds.

and overall costs the operation is not easy for the bus to cope with in case there are multiple new clients connecting at the same time. The second attempt included slower client connection with certain time periods between each connection. The number of concurrently communicating pairs was gradually increased up to a number of roughly 250 pairs, when the timeouts started to occur more frequently. As a result of this behavior, it was more reasonable to try 200 concurrently communicating pairs. The period between a request message being sent and receiving the result payload was measured. The results in Table 5.9 show that in every case the TBus was able to provide a response for a request faster than speed in which the requests were produced.

client library used	C++		Python	
bulking	enabled	disabled	enabled	disabled
average latency	4.4007 ms	8.032 ms	12.0666 ms	6.1445 ms

Table 5.9: The average latency of request while 200 pub-sub pairs concurrently communicate.

Chapter 6

Conclusion

This thesis's goal was to create a middleware that would serve as a communication bus for the Testos platform. I needed to define requirements and needs of the platform, then I explored few popular message-oriented middleware solutions and chose MQTT as a basis for the Testos Bus. The MQTT protocol was simplified, modified and extended in order to fulfill the needs of the platform. Than I implemented the modified version of the protocol including two client libraries for Python and C++.

After the implementation was finished, the Testos Bus was tested via automated tests and experimented with in order to discover performance abilities and limitations of the solution, which showed that the Testos Bus is able to withstand few thousand requests at once and its overhead does not significantly prolong the communication when a single publisher produces around 60 requests per second. The middleware overhead also does not significantly delay the communication when there are 200 publisher-subscriber pairs exchanging a small number of requests per second.

There is a lot of possibilities for future development. First of all, it is possible to implement optional requirements of the Testos platform such as authentication and authorization, service load balancing or monitoring. It would be very useful to hook up the broker to a log server in order to make storing and searching for log information much easier. Second of all, the bulking strategy implementation can be more dynamic, using periods of time without high traffic to measure the network connection status in order to select more appropriate threshold and bulking window with in order to save the resources and also reduce the delay of obtaining a response caused by collection of messages into a bulk. It is also possible to modify client libraries so that they can have separate requests queues for each individual subscription. Lastly, there is an opportunity to add an ability to detect a subscriber situated on the same machine in order to exchange messages with it via the main memory to increase the communication speed.

Bibliography

[1] *AMQP* [online]. [cit. 2021-01-06]. Available at: https://amqp.org/.

[2] *Industry Leading Log Management / Graylog* [online]. [cit. 2021-01-16]. Available at: https://www.graylog.org/.

[3] *MQTT - The Standard for IoT Messaging* [online]. [cit. 2020-12-15]. Available at: https://mqtt.org/.

[4] *OASIS Open* [online]. [cit. 2019-12-15]. Available at: https://www.oasis-open.org/.

[5] *OMG / Object Management Group* [online]. [cit. 2021-01-09]. Available at: https://www.omg.org/.

[6] *What is DDS?* [online]. [cit. 2021-01-09]. Available at: https://www.dds-foundation.org/what-is-dds-3/.

[7] *What's in the DDS Standard?* [online]. [cit. 2021-01-09]. Available at: https://www.dds-foundation.org/omg-dds-standard/.

[8] *OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0* [online], 29. october 2012 [cit. 2021-01-06]. Available at: http://docs.oasis-open.org/amqp/core/v1.0/os/amqp-core-complete-v1.0-os.pdf.

[9] *Data Distribution Service (DDS): Version 1.4* [online], 10. april 2015 [cit. 2021-01-09]. Available at: https://www.omg.org/spec/DDS/1.4/PDF.

[10] *Publish & Subscribe - MQTT Essentials: Part 2* [online]. January 2015 [cit. 2020-12-28]. Available at: https://www.hivemq.com/blog/mqtt-essentials-part2-publish-subscribe/.

[11] *Testos Platform* [online]. 2018 [cit. 2020-12-29]. Available at: http://www.testos.org.

[12] *MQTT Version 5.0* [online], 7. march 2019 [cit. 2020-12-13]. Available at: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf.

[13] *The Real-time Publish-Subscribe Protocol DDS Interoperability Wire Protocol (DDSI-RTPS) Specification* [online], 3. april 2019 [cit. 2021-01-09]. Available at: https://www.omg.org/spec/DDSI-RTPS/2.3/PDF.

[14] *Publish–subscribe pattern - Wikipedia* [online]. December 2020 [cit. 2020-12-28]. Available at: https://en.wikipedia.org/wiki/Publish%E2%80%93subscribe_pattern.

[15] Schmitt, A., Carlier, F. and Renault, V. Data Exchange with the MQTT Protocol: Dynamic Bridge Approach. In: *2019 IEEE 89th Vehicular Technology Conference (VTC2019-Spring)*. 2019, p. 1–5. DOI: 10.1109/VTCSpring.2019.8746333.

[16] Suranga, S. *Timercpp*. Available at: https://github.com/99x/timercpp.