UNIVERSITÀ DI PARMA

UNIVERSITA' DEGLI STUDI DI PARMA

DOTTORATO DI RICERCA IN
"Tecnologie dell'Informazione"

CICLO XXXII

Security in the Internet of Things

Coordinatore:
Chiar.mo Prof. Marco Locatelli

Tutore:
Chiar.mo Prof. Luca Veltri

Dottorando: Yanina Protskaya

Anni 2016/2019

Contents

List of Figures

Introduction

The Internet of Things (IoT) interconnects billions of heterogeneous devices in an Internet-like structure extending the current Internet and enabling new forms of interactions between objects. When it comes to integration of IoT systems in such essential fields of life as medicine and healthcare, transportation and manufacturing control, computer-controlled devices in vehicles, security becomes a crucial aspect and it is essential to protect communication between IoT devices and the data they exchange. The constraints of some IoT devices, such as low processing powers, lack of memory and storage space and limited energy sources, make it difficult to apply the already existing solutions used in the standard Internet and to implement strong cryptographic algorithms. In such a scenario, new algorithms and security mechanisms should be designed and the existing well-known methods must be optimized taking into account the particularities of the IoT environment.

This thesis focuses on the study of security mechanisms for the IoT environments. In particular, we propose a new authentication and authorization architecture for MQTT-based IoT systems and novel anonymity solutions working at network and application levels. The thesis is organized as follows.

In *Chapter 1*, an overview of the Internet of Things and its most popular protocols (CoAP and MQTT) is provided. The most required security aspects, such as end-to-end authentication and authorization, data confidentiality and integrity, and anonymity, are described, together with the existing solutions aiming at providing security features taking into consideration the constrained and heterogeneous nature of the Internet of Things.

In *Chapter 2*, end-to-end authentication and authorization in the IoT is studied. We introduce dynamic broker bridging – a novel mechanism which extends standard publish/subscribe systems to multi-hop architectures. Three possible approaches of securing the proposed mechanism through end-to-end authentication and authorization capabilities are considered and analyzed. A use case of application of the proposed dynamic broker bridging mechanism to an Industrial IoT scenario is discussed. The proof-of-concept implementation of the proposed dynamic broker bridging mechanism is described in detail.

Chapter 3 focuses on providing anonymity to IoT systems. In this chapter, we present a novel anonymization protocol for datagram-based communication. An overview of the protocol's design is provided, and the mechanisms used for anonymity path establishment and data exchange processes are explained. The protocol has been specially designed to satisfy constrained scenarios typical for the Internet of Things. The design of the protocol also implies confidentiality, thus eliminating the necessity to use any secure communication protocol. The proposed anonymization protocol has been implemented and tested in order to demonstrate its feasibility.

Chapter 4 describes the results of studying anonymity provision at application level. In particular, we propose a novel solution for anonymizing publish/subscribe-based networks. The design of the solution is based on the dynamic broker bridging mechanism proposed in Chapter 2. According to this anonymization mechanism, a client creates an anonymity path lying through several brokers and uses this path for anonymizing messages sent to a remote broker, thus preventing any participant of the communication process from knowing which topic it, client, is interested in. The proof-of-concept implementation of the proposed mechanism is described in detail.

Chapter 1

Security in the Internet of Things

If you think technology can solve your security problems,
then you don't understand the problems and
you don't understand the technology.

– Bruce Schneier

The Internet of Things (IoT) interconnects billions of heterogeneous devices, denoted as "smart objects", in an Internet-like structure, which extends the current Internet, enabling new forms of interactions between objects – Machine-to-Machine (M2M). The IoT smart objects are embedded with electronics and software. They may also have one or more sensors and/or actuators. The devices collect various data, interconnect with each other and exchange the data. Smart objects are often constraint, mainly in terms of memory capacity, processing powers and limited energy sources.

Nowadays, IoT systems are integrated in almost every field of life. The IoT is related to various applications [1] addressing diverse needs of the society such as smart city and smart buildings, smart health and smart living, smart transport and smart energy, etc. Figure 1.1 represents the distribution of the IoT technologies according to GrowthEnabler analysis[1].

[1] https://growthenabler.com/flipbook/pdf/IOT Report.pdf

Figure 1.1: Distribution of IoT systems.

1.1 IoT protocols

IoT communication protocols are the way the devices communicate and exchange data, which sometimes may include also some security services. IoT protocols can be classified into two separate categories:

- network and transport protocols which are used for connecting devices over the network (e.g., HTTP [2], LoRaWAN [3]);
- application protocols which are used for communication at application level between low power IoT devices (e.g., CoAP [4], MQTT [5], Websocket [6]).

The most preferred application protocols in the IoT applications are Constrained Application Protocol (CoAP) and Message Queuing Telemetry Transport (MQTT).

1.1.1 Constrained Application Protocol

The Constrained Application Protocol (CoAP) [4] is a specialized web transfer protocol for use with constrained nodes and constrained networks in the Internet of Things.

The protocol is designed for M2M applications such as smart energy and building automation.

For simplified integration with the web, CoAP has been designed to be easily translated to/from HTTP, while also meeting some specific requirements of the IoT: multicast support, very low overhead and overall simplicity. The following features of CoAP have been proposed by the Constrained RESTful Environments (CoRE) IETF[2] group that has designed the protocol with these main requirements:

- RESTful [7] protocol design which minimizes the complexity of mapping to HTTP,
- URI and content-type support,
- low header overhead,
- low parsing complexity,
- discovery of resources provided by known CoAP services,
- simple subscription for a resource and resulting push notifications,
- simple caching capability.

Similarly to HTTP protocol, CoAP is based on the REST (Representational State Transfer) model [7], according to which servers make resources available under a URL, and clients access the resources using appropriate methods such as GET, PUT, POST, and DELETE.

CoAP has been designed to use minimal resources in terms of both the devices and the network. Instead of a complex transport stack, it runs over UDP on IP. A 4-byte fixed header and a compact encoding of options enables small messages which lead to no or little fragmentation on the link layer.

The interaction model of CoAP is similar to the HTTP client/server model. However, machine-to-machine interactions usually result in a CoAP implementation acting both as a client and a server. A CoAP request is similar to that of HTTP and is

[2]Internet Engineering Task Force. https://www.ietf.org/

sent by a client to a server in order to request an action (using a Method Code to specify the action) on a resourceidentified by a URI. The server then replies with a response containing a Response Code; this response may also include the resource representation. However, unlike HTTP protocol, CoAP deals with those interchanges over a datagram-oriented transport such as UDP in asynchronous way.

In Figure 1.2 the format of CoAP messages is represented. Both for requests and responses CoAP utilizes a short fixed-length (4 bytes) binary header which may be followed by a variable-length (0..8 bytes) Token value, followed by a sequence of zero or more CoAP Options in Type-Length-Value (TLV) format, optionally followed by a payload that possesses the rest of the datagram.

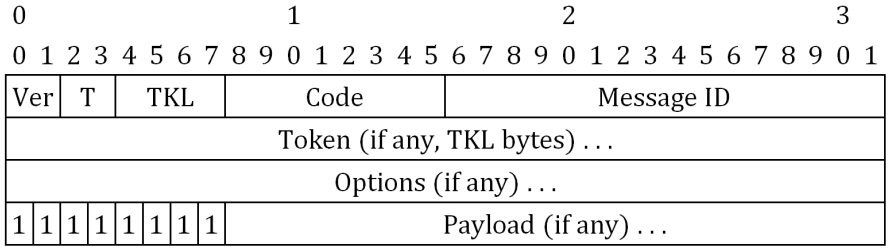

Figure 1.2: CoAP message format.

The header fields are the following:

- Ver (Version): Indicates the CoAP version number (2-bit unsigned integer).
- T (Type): Indicates whether the message is Confirmable (0), Non-confirmable (1), Acknowledgement (2), or Reset (3).
- TKL (Token Length): Indicates the length of the variable-length Token field (0..8 bytes are allowed).
- Code: the first 3 bits represent the message class, the rest 5 bits detail the class.
- Message ID: Is used to match Acknowledgement or Reset messages to Confirmable and Non-confirmable messages and to detect message duplication.

The header is followed by the Token value the length of which varies from 0 to 8 bytes according to the Token Length field. The Token value is used to correlate requests and responses.

Header and Token value are followed by zero or more Options. Each Option can be followed (i) by another Option, (ii) by the Payload Marker and the payload or (iii) by the end of the message.

Following the header, token and options comes the optional payload. If it is present and of non-zero length, it is prefixed by a one-byte Payload Marker (0xFF). The payload data extends from after the payload marker to the end of the UDP datagram, i.e., the Payload Length varies based on the size of the datagram and the token and options if any.

A CoAP message, appropriately encapsulated, should fit within an IP packet (in order to avoid IP fragmentation) and (by fitting into a single UDP payload) needs to fit within a single IP datagram.

1.1.2 Message Queuing Telemetry Transport

MQTT (Message Queuing Telemetry Transport) [5] is a messaging protocol based on publish/subscribe pattern that works in IPv4 and IPv6 networks on top of TCP.

A simplified scheme of MQTT communication is depicted in Figure 1.3. A standard MQTT system consists of clients communicating with a server, usually called a "broker". A client may be either a publisher of information or a subscriber, and in some cases both of them. Subscribers are clients that want to receive data on a specific topic. In order to do this, they send a subscription request (*SUBSCRIBE*) to the broker indicating the desired topic name/names. Publishers are clients that publish information on particular topics. When a publisher has some data to distribute, it sends a message (*PUBLISH* request) including those data and the topic name to the broker. The broker then forwards the information to all clients that have subscribed to the topic.

In MQTT data are sent in plain text format, even such sensitive information as username and password. For this reason, the standard proposes to use the Transport

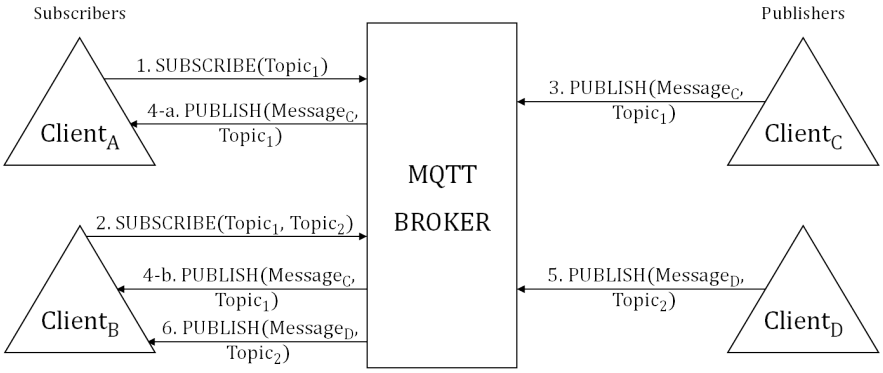

Figure 1.3: An example of MQTT communication.

Layer Security (TLS) protocol [8] as a mean of protecting message exchanges providing authenticity and confidentiality of data exchanged within the system.

MQTT-SN

MQTT For Sensor Networks (MQTT-SN) [9] is a version of MQTT developed for use in wireless sensor networks (WSN). It has been specially designed for communication between constrained devices with their low processing powers, small storage and, sometimes, energy sources limits. The restrictions of wireless communication environments, such as high link failures, low bandwidth, or shorter message length have been also taken into account.

One of the important differences from the original MQTT protocol is that in MQTT-SN specific support for sleeping clients has been added, which allows the devices to send a message and then go in sleep mode without waiting for an acknowledgement, thus saving energy.

While in MQTT data confidentiality and integrity protection can be provided by means of TLS, as it is specified in the official documentation, security aspect has not been deeply described in the specification of the MQTT-SN protocol, so it is a responsibility of a developer to provide it if needed. Communication in wireless sensor networks over MQTT-SN, which, in its turn, uses UDP transport, needs other tech-

niques for making it secure. One possible solution can be use of Datagram Transport Layer Security (DTLS) protocol [10] that provides security for datagram-based applications and prevents eavesdropping, tampering, and message forgery, similarly to TLS. However, using DTLS may cause unwanted outcomes, such as increase of data, packets and handshake overhead. Another way can be taking care of security at the application level. This could be done by protecting MQTT topic names and payloads using proper techniques, possibly based on light-weight cryptography.

Since MQTT-SN is designed for use on very constrained devices, their limitations must be taken into account while designing new or applying standard security solutions for this protocol.

MQTT-SN topic name registration

An important novelty in the MQTT-SN protocol is the *topic name registration* procedure. Due to the limited message size and bandwidth of wireless sensor networks, in the systems working over MQTT-SN protocol, when a *SUBSCRIBE* or *PUBLISH* request is sent, it does not contain the original topic name value. A special topic name registration procedure has been introduced to substitute long topic name strings (those longer than 2 bytes) with a shorter topic id.

An example of a successful topic name registration scenario is depicted in Figure 1.4.

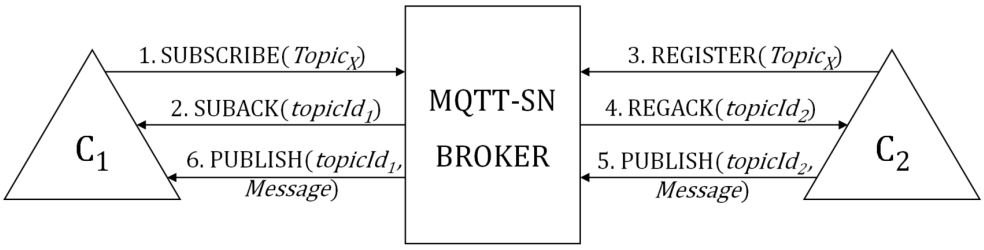

Figure 1.4: An example of topic name registration in MQTT-SN.

In the provided example, a client C_1 subscribes to the topic $Topic_X$ on an MQTT-SN broker and a client C_2 publishes a message in that topic. The following actions

take place between the clients and the broker:

1. The client C_1 sends a *SUBSCRIBE* message to the broker.

Length (octet 0)	MsgType (1)	Flags (2)	MsgId (3-4)	TopicName or TopicId (5:n) or (5-6)

Figure 1.5: SUBSCRIBE message format.

The structure of the message is depicted in Figure 1.5, where:

- *Length* field contains the size of the message in bytes,
- *MsgType* is set to value 0x12, which corresponds to the *SUBSCRIBE* message type,
- the value of the $TopicIdType\ flag$ is equal to 0x0b00, which means the use of a normal topic name,
- *MsgId* is the unique identifier of the message,
- *TopicName* is the topic name to which the client wants to subscribe ($Topic_X$ in the provided example).

2. The broker assigns to the received topic name string a unique (for the communication between the broker and the client C_1) identifier $topicId_1$, stores the mapping record internally, and responds to C_1 with a *SUBACK* message of the structure depicted in Figure 1.6.

Length (octet 0)	MsgType (1)	Flags (2)	TopicId (3,4)	MsgId (5,6)	ReturnCode (7)

Figure 1.6: SUBACK message format.

where:

- *Length* field is the length of the message,
- *MsgType* is set to 0x13, which means subscription acknowledgement,

- *TopicId* contains $topicId_1$ value,
- *MsgId* has the same value that C_1's request,
- *ReturnCode* is equal to 0x00 ("Accepted").

3. Before publishing a message in the topic $Topic_X$, client C_2 sends to the broker a *REGISTER* request with the structure shown in Figure 1.7.

Length (octet 0)	MsgType (1)	TopicId (2,3)	MsgId (4:5)	TopicName (6:n)

Figure 1.7: REGISTER message format.

where:

- *Length* field contains the size of the message in bytes,
- *MsgType* is set to 0x0A, which corresponds to the *REGISTER* message type,
- *TopicId* value is set to 0x0000,
- *MsgId* is the unique identifier of the message,
- *TopicName* is the topic name that is to be registered ($Topic_X$).

4. If the registration of the provided topic name can be performed, the broker assigns an identifier ($topicId_2$) to the topic, stores the association internally, and sends to the client C_2 a *REGACK* message of the following structure:

Length (octet 0)	MsgType (1)	TopicId (2,3)	MsgId (4,5)	ReturnCode (6)

Figure 1.8: REGACK message format.

where:

- *Length* is the size of the message in bytes,

- *MsgType* is set to 0x0B, which implies a *REGACK* message,
- *TopicId* contains $topicId_2$ value, assigned to the requested topic name,
- *MsgId* equals to the *MsgId* value sent by the client C_2 in its *REGISTER* request,
- *ReturnCode* is set to 0x00 ("Accepted").

Note that although both the clients are interested in $Topic_X$, the broker can assign different ids to the same topic name for different clients.

5. The client C_2 sends a *PUBLISH* message to the broker, using the obtained $topicId_2$ value instead of the original topic name string.

6. Based on the received from the client C_2 topic id value $topicId_2$, the broker obtains the original topic name $Topic_X$. It then retrieves the client C_1 from the subscribers database, maps the $Topic_X$ string to the $topicId_1$ value used during the C_1's subscription, and forwards the *PUBLISH* message using $topicId_1$ as the topic id.

If topic name registration procedure failed, the broker returns a rejection code. Depending on the value of that code, a client may try to request it again later, e.g. in case of "rejected: ingestion" error, the client mush wait for a certain interval of time before repeating the topic name registration attempt.

The MQTT-SN protocol also allows using predefined topic ids, which are known to client's and broker's applications in advance.

MQTT broker bridging

Some implementations of MQTT protocol allow configuring a broker as a "bridge", when it can also act like a client: it connects to other brokers and subscribes and/or publishes to some, or all, topics on those brokers. An example of an MQTT system with a bridged broker is depicted in Figure 1.9.

In the provided example broker B_1 is configured as a bridge to the broker B_2 for topics T_X and T_Y. Once set up has been done, the initial subscription is performed:

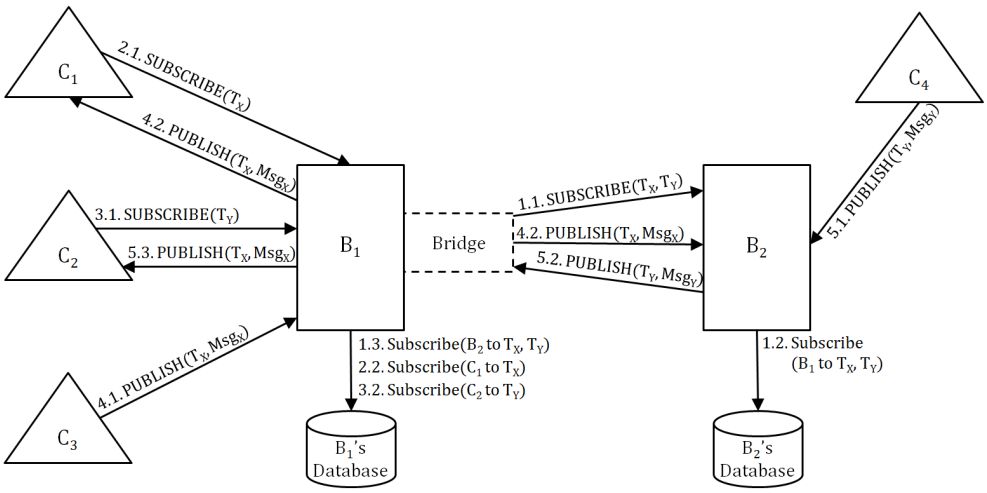

Figure 1.9: An example of MQTT broker bridging.

1. B_1 subscribes as a client to topics T_X and T_Y on B_2.

2. B_2 processes this request in a standard way, storing the subscription in its database ($B_2's$ *Database*).

3. B_1 adds a subscription of the broker B_2 to topics T_X and T_Y in its own database ($B_1's$ *Database*).

As a result of these operations, whenever B_1 or B_2 receive a publication request (*PUBLISH* packet) from their clients, they will forward this request to each other. In the provided example, clients C_1 and C_2 are the subscribers on B_1, while clients C_3 and C_4 are publishers connected to brokers B_1 and B_2 respectively. In particular:

4. Client C_1 subscribes to topic T_X on broker B_1. B_1 adds the subscription in its database.

5. Client C_2 subscribes to topic T_Y on B_1. B_1 stores this information in its database as well.

When client C_3 wants to publish some data (message Msg_X) to topic T_X on the broker B_1, the following steps occur:

6. C_3 sends a PUBLISH request containing topic name T_X and application message Msg_X to B_1.

7. B_1 receives the packet and relays it accordingly to T_X subscription information from its database: to C_1 and to B_2. Then B_2 relays the message to the subscribers (except the source B_1): none in this case.

When client C_4 has some data (message Msg_Y) to publish to topic T_Y on the broker B_2, the following steps occur:

8. C_4 sends a *PUBLISH* request to B_2.

9. B_2 forwards the request to B_1, accordingly to T_Y subscription information from its database.

10. B_1, in turn, forwards the *PUBLISH* request further to its subscribers: C_2 in this case.

This bridging mechanism allows for sharing a topic among different brokers. This, in turn, enables:

- creation of clusters of brokers, thus letting clients connected to those brokers share the same topic;

- creation of MQTT proxying mechanism, in which clients subscribe to a topic by means of a broker different from the one to which publishers send messages.

- connection to an external network. It this case a broker that belongs to a given network plays the role of a gateway processing and filtering outgoing and incoming messages.

1.2 Security in the IoT

The Internet of Things has now become an integral part of our life. The speed with which the number of IoT devices grows increases from year to year. In Figure 1.10 the graph of the yearly growth in the Internet of Things devices[3] is shown.

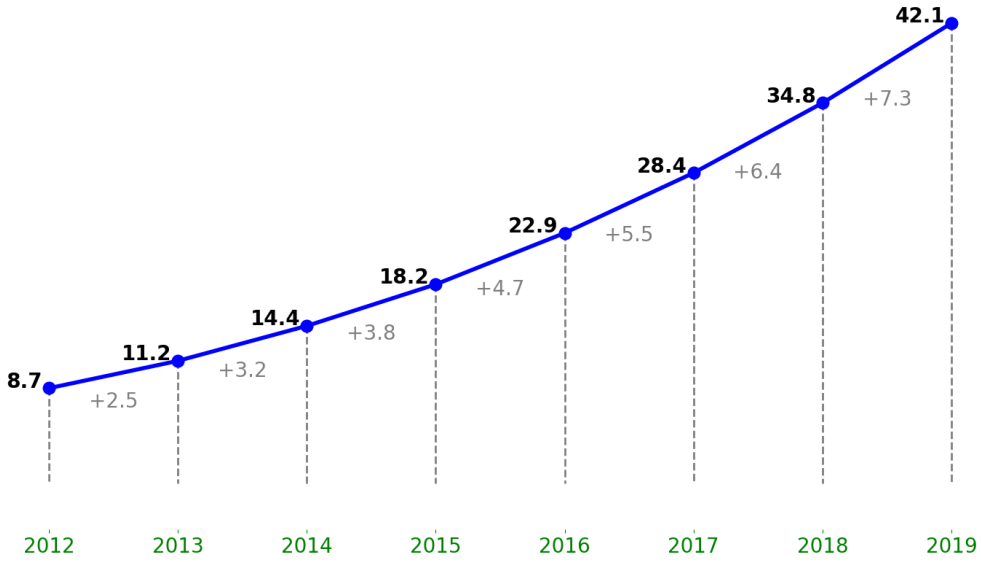

Figure 1.10: The number of IoT devices (in billions).

Nowadays, IoT devices and technologies are used in nearly every sphere of life. As increases the amount of IoT devices, so does the variety of domains where they are applied. When it comes to integration of IoT systems in such essential fields as medicine and healthcare, transportation and manufacturing control, computer-controlled devices in vehicles, security becomes a crucial aspect and it is essential to protect communication between IoT devices and the data they exchange.

IoT systems are highly vulnerable for several reasons:

- many IoT nodes are constrained in terms of energy resources and computing capabilities, which makes them unable to support complex security schemes;

[3] According to NCTA – The Internet & Television Association

- most of the communications in IoT systems are wireless, and this makes them vulnerable for eavesdropping;

- the majority of IoT devices are usually left unattended, which makes it easy to attack them physically.

Security is a serious requirement for all systems that exchange possibly important data, and standard mechanisms have been defined and are currently used in classical IP networks and the Internet. However, providing security services in the Internet of Things is a challenging task. The most important security services that may be required are:

- end-to-end authentication and authorization,

- data confidentiality,

- data integrity,

- anonymity.

The main constraints of the IoT smart objects, such as low processing powers, lack of memory and storage space and limited energy sources make it difficult to apply the solutions that already exist and are widely used in the standard Internet and to implement strong cryptographic algorithms for protecting data confidentiality. The majority of exiting solutions providing security for the IoT do not satisfy security needs completely and/or have large computation overhead. Moreover, new network architectures and communication protocols appear, which leads to the need of designing novel approaches for providing security in the IoT systems. In such a scenario, new algorithms and security mechanisms should be designed and the existing well-known methods must be optimized taking into account the particularities of the IoT environment.

1.3 End-to-end authentication and authorization

Authentication is the process of verifying the identity, which is usually required in order to allow access to confidential data or systems. Authorization is the function of specifying access rights/privileges to resources.

End-to-end authentication and authorization within IoT systems are one of the biggest security concerns. It is difficult to use the well-known mechanisms which usually require certain authentication infrastructures and/or servers exchanging messages with the nodes in order to authenticate/authorize them. In the IoT environment those approaches may be not feasible since some constrained nodes are not capable of exchanging too many messages and/or to store necessary authentication data (e.g., certificates).

In literature there are various proposals trying to secure IoT network scenarios.

An architecture for secure communication between constrained IoT devices has been described in [11]. The proposed solution uses Datagram Transport Layer Security (DTLS) based on certificates with mutual authentication. A new device called IoT security support provider (IoTSSP) has been introduces. IoTSSP is responsible for managing and analyzing the certificates of the devices and for authentication and session establishment between the devices. The study has also introduced two new mechanisms: (i) optional handshaking delegation – for communication with a constrained device a client performs the handshaking process with the IoTSSP in order to be authenticated; and (ii) transfer of session – an extension of DTLS that transfers a secure communication session to the constrained device.

The authors of [12] have proposed an architecture of an IoT-OAuth-based authorization service – IoT-OAS – that targets HTTP/CoAP services to provide an authorization framework, which can be integrated by invoking an external OAS. Due to the authorization functionality delegation, IoT-OAS achieves benefits consisting in (i) lower processing load comparing with the solutions in which access control mechanisms are implemented on the smart object; (ii) fine-grained remote customization of access policies; and (iii) higher scalability, since there is no need to perform operations directly on the device.

In [13], an IoT heterogeneous identity-based authentication scheme has been proposed. The authors apply the concept of software-defined networking (SDN) on IoT devices. Every set of devices communicates with a gateway which supports authentication of the things. The gateways are connected to a central controller that has access to the central data storage. In order to give access to the things, the authentication process goes through the gateway and after through the central controller: (i) the gateway obtains an authentication certificate from the controller, (ii) things register on the gateway, and (iii) the devices send authentication request to the gateway.

The authors of [14] propose an authorization and authentication framework for the Internet of Things that exploits the security model of OAuth 1.0a using the lightweight building blocks of ACE. Due to the self-securing tokens the framework security does not depend on the security of the network stack. This is achieved by using signatures on the new tokens obtained from the authorization server and HMACs on the tokens that have been updated. The authors have used basic PKI functionalities for bootstrapping a chain-of-trust between the devices, thus simplifying future exchanges of a token. In this work, an alternate key establishment scheme has been proposed for those use cases in which devices are not able to directly communicate. The proposed framework has been implemented and tested, and tests results have shown that although the initial verification of the fresh tokens is expensive, the framework offers a strong level of security for IoT devices.

In [15] a light-weight user authentication technique for IoT systems, called LiGAT (Lightweight Gait Authentication Technique), has been proposed. This technique exploits various information that has been collected from IoT devices (such as subconscious level of user activities) in order to effectively authenticate users while reducing the resource consumption. According to the proposed scheme, users authentication is performed by extracting and identifying their walking patterns (gait). The solution has been implemented by the authors on an Android platform to collection and analysis of accelerometer data of different users. The authentication is done using various machine learning classifiers. The experiment results have shown that the proposed technique successfully authenticates users with the accuracy of 96.69%.

A lightweight and privacy-preserving two-factor authentication scheme for IoT

devices has been presented in [16]. In addition to a password or a shared secret key as the first authentication factor, as the second authorization factor, the authors have considered physically uncloneable functions (PUFs). PUFs are the result of the manufacturing process of Integrated Circuits (ICs) which leads to various random physical variations of the micro-structure of an IC, making it unique. Those variations are difficult to predict and almost impossible to clone. According to this, PUFs use their internal structure for provide a one-way function which cannot be duplicated. Security and performance analysis has been provided and shows that the proposed scheme is robust against several attacks. The proposed solution is efficient in terms of computational requirements.

The authors of [17] propose a design and analysis for an authentication hardware module which allows the use of two-factor authentication based on smart cards in the IoT. The proposed solution takes into consideration the limited processing power and energy resources of many IoT nodes. The authors have proposed to exploit ECDH as the authentication and key exchange scheme. The scheme has been analyzed in terms of security issues and evaluated under different types of attacks. The proposed authentication scheme has been implemented and simulated.

The IETF Authentication and Authorization for Constrained Environments (ACE) working group aims to produce a standardized solution for authentication and authorization required for the access to resources hosted on a resource server in constrained environments. They mainly assume that access to resources takes place using CoAP [18] and is protected by DTLS [19]. An MQTT-TLS profile of ACE [20] has been specified to enable authorization in MQTT-based publish/subscribe messaging systems. The proposed schemes use a trusted third party – authentication server – which is in charge of releasing access tokens to the clients. The clients then use those tokens for communicating with other entities.

A general scheme of the basic protocol flow of the security mechanism, proposed by ACE working group, is depicted in Figure 1.11. For the sake of clarity of the mechanism, the scheme considers a case of successful authorization and message exchange.

When a client wants to subscribe/publish to a topic on a broker, the following

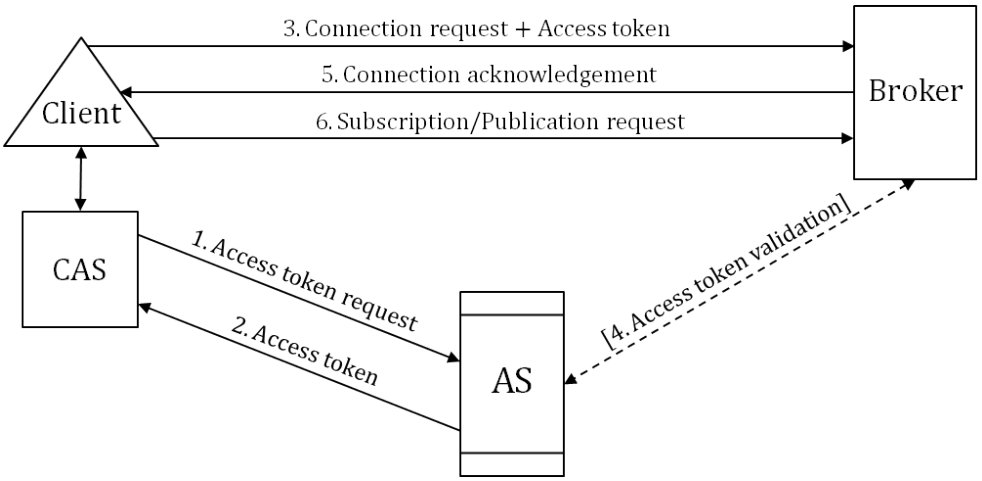

Figure 1.11: MQTT-TLS profile of ACE general scheme.

events occur:

1. The client authorization server (CAS) requests an access token from the authorization server (AS) providing necessary credentials (e.g., client ID and password). If the client node has enough resources, these steps can be performed by a client directly.

2. If the AS successfully verifies the access token request and authorizes the client for performing requested actions on a specified broker, the AS issues an access token, which may be a reference, a JSON Web token (JWT) [21] or a CBOR token [22].

3. When the client obtains the token, after the TLS handshake, it transports the token to the broker via a *CONNECT* request with the *Username* field set to the access token and the *Password* field set to the keyed message digest (MAC) or signature associated with the token.

4. When the broker receives the connection request, it verifies the validity of the token. This may be done locally or by communicating with the AS for introspection.

5. Based on the validation result, the broker sends a connection acknowledgement (*CONNACK*) message to the client. If the broker accepts the connection, it stores the client's access token.

6. On receiving a *SUBSCRIBE* or *PUBLISH* message from a client, the broker uses the request type and the topic name to compare against the cached token. If the client is allowed to subscribe/publish to the topic, the broker stores the subscription/publishes the client's message to all valid subscribers of the topic.

A secure communication scheme named MQTT-Auth has been proposed in [23]. The solution is based on the AugPAKE algorithm for setting up a session key and on authentication and authorization tokens which are used to publish on a certain topic and to grant access to a specific topic respectively.

In [24] the authors design and implement secure versions of MQTT and MQTT-SN protocols (SMQTT and SMQTT-SN respectively) in which security is based on Key/Ciphertext Policy-Attribute Based Encryption (KP/CP-ABE) and lightweight Elliptic Curve Cryptography is used. ABE supports broadcast encryption that allows to deliver a message to multiple intended users performing only one encryption.

In [25] an access control system to manage publisher (device node and gateway) authentication using authentication server for communication over MQTT protocol has been described. The assumed network architecture is based on fog computing. Publishers obtain tokens from the authentication server via HTTPS. Moreover, X.509 certificates are used to secure the communication between publishers and brokers.

This thesis contributes to the research of end-to-end authentication and authorization solutions for the IoT, introducing the design of a novel secure broker bridging mechanism for the systems based on publish/subscribe paradigm. Chapter 2 describes the proposed dynamic broker bridging mechanism and its implementation in detail and provides the analysis of possible end-to-end authentication/authorization approaches. As an example of use case, the proposed mechanism has been applied to an Industrial IoT scenario.

1.4 Data confidentiality

Data confidentiality – is the property that information is not available or disclosed to unauthorized individuals, entities, or processes [26]. In other words, data confidentiality service must ensure that only authorized users have access to the data.

Confidentiality of the data is usually ensured by encryption of those data. However, conventional cryptography is not always applicable to IoT systems due to the devices' limitations not allowing them to satisfy the storage and/or computing requirements of strong cryptographic algorithms. Since the processing abilities of constrained devices are poor, symmetric key algorithms are usually preferred in the IoT environment. The most important issue related to the use of symmetric cryptography is the way the secret keys are shared by the (possibly constrained) devices involved in the communication.

1.4.1 Shared key establishment

The most known approach of exchanging a shared symmetric key over an insecure channel is Diffie-Hellman key exchange (DH) [27]. A general scheme of this approach is depicted in Figure 1.12.

When two communicating entities want to share a symmetric key, they perform the following actions:

1. The entities agree on the key exchange scheme and its parameters.
2. Each entity generates a secret value (also referred as private key).
3. Based on the secret value, each entity computes a public value (public key).
4. The entities communicate to each other their public values.
5. The entities compute a shared secret (symmetric key) based on their own private value and the obtained public value. This secret is then used in order to encrypt and decrypt the messages exchanged between them.

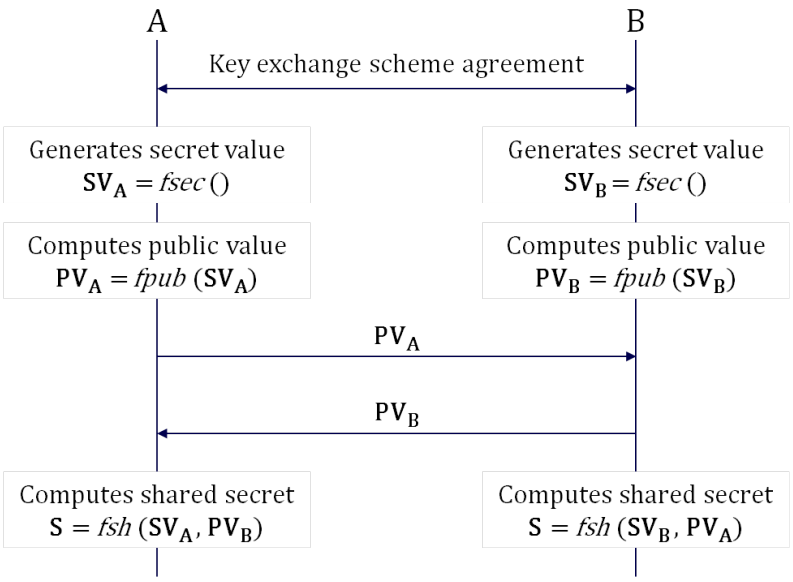

Figure 1.12: Shared symmetric key exchange.

This approach allows the communicating entities to exchange all data necessary for protection of their messages even through a communication channel which is not protected and can be eavesdropped.

1.4.2 Private/public values computation

Up to recent time, the mostly used public-key cryptographic scheme was RSA. However, the discovery of particular characteristics of elliptic curves introduced Elliptic-curve cryptography (ECC), which is based on the algebraic structure of elliptic curves over finite fields.

The most significant advantage of ECC comparing to RSA is that it requires smaller keys in order to provide equivalent level of security. In Figure 1.13 are present the sizes of public values needed for computing a symmetric key using RSA scheme and ECC scheme.

Based on the data of the table, a graph of Figure 1.14 is built and shows the great

Symmetric key (bits)	RSA public key (bits)	ECC public key (bits)
56	512	112
80	1024	160
112	2048	224
128	3072	256
192	7680	384
256	15360	521

Figure 1.13: Public key lengths.

difference between the ECC and the RSA public key sizes. In the provided figure, symmetric key length of 112 bits is defined by National Institute of Standards and Technology (NIST) as the smallest allowed for key agreement in order to provide sufficient data protection until 2030.

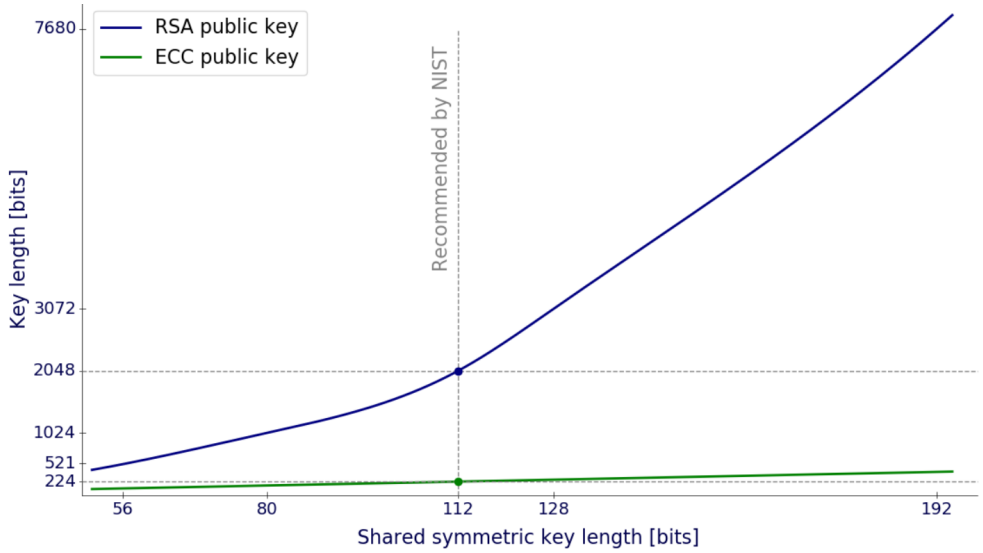

Figure 1.14: Required public key length: ECC vs RSA.

To use elliptic-curve cryptography, the communicating parties must agree on all

the elements that define the elliptic curve – the domain parameters of the scheme. Since this involves computing the number of points on a curve, which is a time consuming process and is troublesome to implement, the domain parameters generation is not usually done by the participants of the communication process. Several standard bodies have computed and published elliptic curves domain parameters for several common field sizes. Those parameters are known as "standard curves" and are defined in the standard documents: Recommended Elliptic Curves for Government Use[4], SEC 2: Recommended Elliptic Curve Domain Parameters[5] and ECC Brainpool Standard Curves and Curve Generation[6].

In order to exchange public ECC values and compute a secret symmetric key, Elliptic-curve Diffie-Hellman (ECDH) key agreement scheme can be used in place of the traditional DH. ECDH allows the entities establish a shared secret communicating over an insecure channel [28]. The obtained key is then used for encrypting the communication using a symmetric-key cipher.

In this work, we have applied ECDH key agreement scheme for encrypting messages in the implementation of the anonymization mechanism for communication over MQTT protocol proposed in Cahpter 4.

1.5 Anonymity

Anonymity is a security service allowing entities to communicate with each other in such a way that no third party knows that they are the participants of a certain message exchange. Hiding the fact that communication is taking place between certain source and destination entities is the biggest challenge of preserving privacy. In a general scenario, even if the data payload are protected (encrypted), the packet header is usually transmitted in clear in order to correctly route the packet to the destination entity. The information included in the header (the source and the destination addresses and port numbers) discloses which entities are communicating.

[4]https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf
[5]http://www.secg.org/sec2-v2.pdf
[6]http://www.ecc-brainpool.org/download/Domain-parameters.pdf

Although anonymity is not the essential security service and is not always required, it is an important topic for research nowadays. Anonymization of communications may be needed in high-risk environments, e.g., in the military context it could obscure the real roles of communicating units, their location or their position in the command chain. Another application of anonymity is individuals' privacy protection. Many research efforts have been conducted in order to provide the Internet communication with anonymity, but little has been done in the context of IoT. Hereinafter, some solutions for providing anonymity for communications in the Internet of Things, which differs from the standard Internet in terms of both communicating nodes capabilities and communication system architecture characteristics, that have been already proposed are considered.

Secure and Trust Anonymous Communication protocol for IoT (STAC) is presented in [29]. The protocol is based on lightweight key agreement protocol, the Identity Based Encryption (IBE) and Pseudonym Based Encryption (PBC). It has been applied in clustered network to achieve anonymity and trusted communication for sender, receiver and link between them, thus guaranteeing the privacy of nodes identities and the secrecy of data. The protocol works in several steps. First, a Private Key Generator (PKG) – a trusted central authority – chooses the parameters for private keys generation. Then, the Base Station (BS) broadcasts those necessary parameters to all nodes in the network. When a node obtains the parameters, it chooses a random number x and computes private and public key. Then, the node sends to the PKG its real identity (in secure way), its virtual identity (Pseudonym Based Cryptography (PBC) [30] is used) and the public key. After that, neighbour nodes establish shared session keys between each other and with the BS. Then, after the network setup and association table generation on each node, the entities can start communicating according to the specific algorithm which includes diverse steps of encryption/decryption of data while forwarding them towards the receiver. The proposed solution has been analyzed in terms of its robustness and is claimed to be secure against passive, public and replay attacks. However, it requires a complicated (regarding low-cost IoT devices) process of key establishment and many cryptographic operations, which makes it hard to apply to the systems consisting of constraint de-

vices. Moreover, the PKG possesses the information of which pseudonym belongs to which entity, so once it is compromised, the adversary will be able to disclose all the communications within the system.

An architecture based on per-packet One Time Address (OTA) – an address used by the host for sending or receiving exactly one packet – has been described in [31]. The architecture eliminates from packet headers the flow information (e.g., flow identifier, source and destination address). OTA is based on Autonomous Systems (ASs) connected to the Internet. Each AS contains a Border Router (BR), a Core Router and an Access Router (AR). The OTA address is the result of encryption of the combination of the Host IDentifier (HID) and the Flow IDentifier (FID) performed by the AR. s. Before communication between a source and a destination nodes (belonging to different ASs) can start, authentication of hosts and negotiation of encryption keys between ASs and between the components of each AS must happen. After that, OTA Address-pool generation process, which involves the source host, the destination host and the ARs of both the ASs, takes place. In order to send a packet, the source host picks an address from the pools and sends the packet to the AR of its AS. When AR receives the packet, it generates an OTA and sends the packet to the BR which decrypts OTA destination to obtain the AS destination and sends the packet to the BR of receivers' AS. The receiver BR decrypts the message in order to obtain the corresponding AR and forwards OTA destination to the next hop CR, which in turn continue forwarding the packet to other CRs until it reaches the AR, which extracts destination host by decrypting the destination OTA and sends the packet to destination host. The authors of [32] exploit the One Time Address approach in order to disguise the communication between a source and a destination entities in IPv6 Low Power Wireless Personal Area Network (6LoWPAN). Their solution is similar to the OTA approach proposed for the traditional Internet, but it takes care of adopting that approach to 6LoWPAN in terms of complexity, memory footprint, and energy consumption. The proposed mechanism prevents a possible adversary from detecting between which source and destination entities communication takes place by masking the identifying fields of 6LoWPAN packet headers. While this solution makes it impossible to obtain the flow information from the header, it requires introduction of

various third parties can be applied only to the systems with the proper architecture. The data exchange between the entities and the necessity to encrypt/decrypt those data on each step significantly increase the time needed for message delivery.

A TOR-based anonymous communication approach for smart home has been proposed in [33]. The focus of the work is on the anonymization of communication between smart home devices that are connected to a gateway which is located between the smart home network and the Internet. The system uses advantages of TOR and The Amnesic Incognito Live System (Tails) – a live Debian-based operating system. The authors set up Tails as a central control gateway which takes care of anonymization of data packets before they go out to the public Internet environment: the gateway acts like a TOR client which chooses a random path through the TOR nodes towards the destination. Although this solution is based on a well-known and widely used one, it also shares the same drawbacks that the original TOR has. For instance, TOR has been shown to have vulnerabilities against some flow-correlation timing analysis based attacks ([34], [35]).

A lightweight anonymous authentication protocol for constraint sensor nodes is presented in [36]. The architecture which the authors have proposed includes four necessary components: an authenticated cloud server (ACS), a Cluster Head (CH), home IoT server (HIoTS) and edge devices like sensor nodes (SNs). Before starting message exchange, HIoTS sends security credential to SNs through a secure channel. Then HIoTS generates a random number and computes a key Ksh and generates a random track sequence number (Trseq) which it stores internally and sends to the SN. Then the authentication process takes place: (i) the source SN generates a random one-time-alias identity (AID) which it sends to the receiver SN, which generates its AID and shares it with the CH. (ii) The CH, in its turn, forwards the data to the HIoTS. (iii) The HIoTS performs the verification of the data and responds to the CH, which then responds to the receiver SN, which responds to the sender SN. Although the proposed solution claims to be anonymous, it relies of the pseudonyms (AIDs in this case), while the real identifiers are known to the third parties and, moreover, even if the receiving SN does not know the real ID of the sender, it knows where the message comes from.

The contribution of this thesis is providing the solution for anonymizing communications in the Internet of Things environment. Among the requirements set for the design of the solutions the most important are: (i) they must be light-weight for being applicable to constraint IoT nodes and (ii) the anonymization approach should be decentralized. As the result, we propose two anonymization mechanisms working on different levels (network level and application level):

- In Chapter 3 we have designed an anonymization protocol specifically targeted for IoT applications. In the proposed solution, nodes are organized in an Onion Routing [37] anonymity network which is completely based on datagram transport. The protocol solves the issues related to the use of an onion routing-based anonymity system in a constrained UDP-based network scenario. Additionally, the anonymity level is increased by letting the anonymity path be chosen on a per-packet basis.

- In Chapter 4 we present an anonymization mechanism for providing MQTT and MQTT-SN networks with anonymous communication. The design of the proposed anonymization solution is based on the novel dynamic broker bridging mechanism (described in Section 2.1) and allows clients to subscribe and to publish data to a topic remaining incognito.

Chapter 2

Authentication and authorization

How many IoT devices exist?
How many others have access to the data?
No one really knows. We just don't know.

– Rebecca Herold

Authentication and authorization are two interrelated concepts used to make communication between entities secure, exchanging data only with the right/authorized entities. While during authentication process the identity of a user is verified based on the credentials it provides, authorization grants authenticated users access to the data based on the rules (policies) of the authorization service. End-to-end authentication and authorization is usually referred as application level security.

Although a lot of work has been done on providing security mechanism for the Internet of Things in general, one of the newest paradigms – publish/subscribe – still requires optimal solutions. This chapter is focused on providing end-to-end authentication and authorization in networks working using the MQTT protocol, which has become standard only in December, 2015. MQTT protocol provides support for basic authentication based on *user name* and *password* fields of the payload of *CONNECT* messages in MQTT v3.1.1 [38] and on *AUTH* packets in MQTT v5.0 [5]. These methods permit only authenticated and authorized clients to subscribe/publish

to given topics.

The client-to-broker authentication and authorization scheme is currently being extended by the IETF ACE working group[1]. The ACE group is working on a standard solution to enable authorized access to resources in constrained environments based on third party authentication/authorization entities. Although the work is mainly focused on CoAP-based REST applications, an MQTT-TLS profile of ACE [20] has been also proposed in order to extend the authentication and authorization scheme to MQTT-based systems (as described in Section 1.3). The scheme exploits a separate trusted entity – authentication server – which releases access tokens to the clients. Confidentiality and integrity in the proposed architecture are provided by TLS. Considering the constraints of IoT nodes, exploiting a trusted third party allows significant decrease of the load of nodes.

This chapter is organized as follows. Before discussing end-to-end authentication and authorization mechanisms, we introduce a novel mechanism – dynamic broker bridging – extending a standard MQTT system to a multi-hop architecture in Section 2.1. Various approaches of how the proposed mechanism can be secured through authentication and authorization capabilities are considered in Section 2.2. In Section 2.3 proof-of-concept implementation of the dynamic broker bridging mechanism is described, together with the test proving feasibility of the proposed system. Section 2.4 describes how the dynamic broker bridging approach can be applied to an Industrial IoT (IIoT) scenario. A description of the organization of IIoT production systems is provided, together with a modified, MQTT-based, architecture. Finally, in Section 2.5 conclusions on the work are drawn.

2.1 Dynamic broker bridging

Some MQTT-based communications scenarios may benefit from a multistage broker relay system implemented through broker bridging mechanism (described in Section 1.1.2).

Broker bridging mechanism can be used for:

[1] https://datatracker.ietf.org/wg/ace/

- security reasons, e.g., when only selected brokers are allowed to forward messages in and out of a network domain,
- inter-domain routing, in order to create a hierarchy of brokers capable to interconnect two or more different network domains,
- scalability, with the aim of creating a larger and scalable publish/subscribe distribution architecture.

A significant limitation of the standard MQTT bridging mechanism is that it is based on static configuration of the brokers and requires platform-specific manual setup. Moreover, some of the implementations of MQTT protocol does not provide this feature at all. Hereafter we propose a way for transforming the static bridging into dynamic and flexible multi-broker routing. The idea behind the solution is to let a client subscribe and/or publish messages to the topics which are handled by brokers different from the actual broker to which the client is connected.

A dynamic broker bridging mechanism can be useful to dynamically let the clients route messages through one or more intermediate brokers without requiring any static pre-configuration of those brokers. In order to let the client specify a remote broker which is in charge of handling the topic in which the client is interested, different approaches can be used:

a) definition of a new MQTT header field that carries this information;

b) use of current MQTT protocol at application level, possibly overloading some header fields.

Although the former approach seems simpler, the latter has the advantage of not requiring any modification of the MQTT protocol specification and implementations. For this reason we have considered application of the second approach. In particular, we used the *Topic Name* string to encode both the desired topic name and the identifier of a remote broker. As a result, a new *Topic Name* string is transformed into a URL-like *topic-at-broker* field. The "@" character has been used as a separator between the topic name and the ID of the broker that is charge of this topic. The resulting MQTT *Topic Name* is the following:

$$Topic_{new} = Topic_{actual} \ @ \ Broker,$$

where $Topic_{actual}$ is the name of the topic in which the client is interested and *Broker* is the ID (e.g., IP address and port number) of the broker which is in charge for this topic.

The corresponding architecture is depicted in Figure 2.1, where forwarding of a *SUBSCRIBE* request from a client is considered.

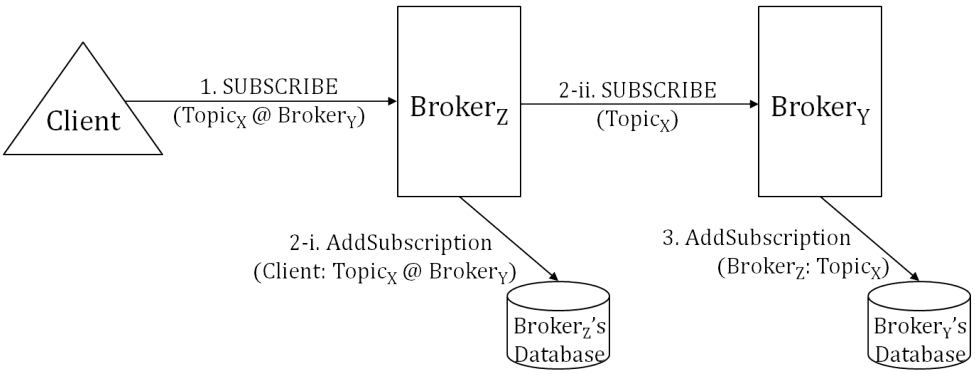

Figure 2.1: An example of dynamic MQTT broker bridging subscription.

When a broker receives a *SUBSCRIBE* or *UNSUBSCRIBE* request in which *Topic Name* value (or one of topic names in case a list of topics has been received) corresponds to a pattern $Topic_X$ @ $Broker_Y$, the broker's actions must be the following:

i) Subscribe/unsubscribe the sender of the request to/from the topic $Topic_X$.

ii) Send a subscription/subscription cancellation request with *Topic Name* value $Topic_X$ to the broker $Broker_Y$. In case the broker has already subscribed to (unsubscribed from) the $Topic_X$ on the $Broker_Y$ this action is not needed.

When a broker receives a publication request in which *Topic Name* value corresponds to a pattern $Topic_X$ @ $Broker_Y$, it forwards the received message to $Broker_Y$ setting *Topic Name* value to $Topic_X$. The described mechanism is depicted in Figure 2.2.

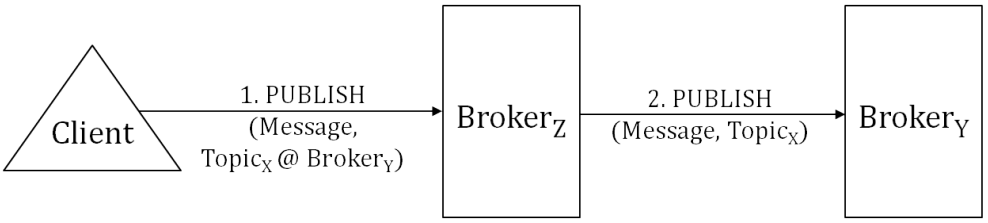

Figure 2.2: An example of dynamic MQTT broker bridging publication.

This proposed mechanism can be further extended by allowing multi-broker topics, like $Topic_X$ @ $Broker_n$ @ $Broker_{n-1}$ @ ... @ $Broker_1$, which will allow clients to forward the request by means of $n-1$ intermediate brokers to the destination broker $Broker_n$.

An example of such multi-broker request forwarding is depicted in Figure 2.3.

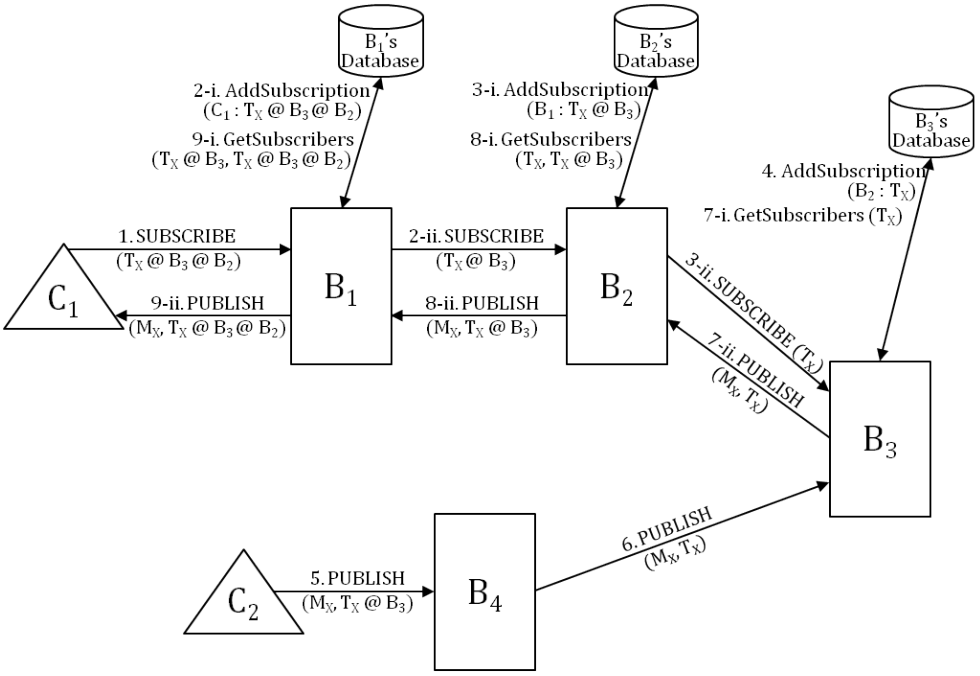

Figure 2.3: Multi-broker dynamic bridging.

In the provided example, client C_1 subscribes to the topic T_X on the broker B_3. However, rather than sending the request directly to this broker, it uses the sequence of intermediate brokers (B_1 and B_2). Then the client C_2 publishes a message M_X in the topic T_X on the broker B_3 using B_4 as an intermediate broker towards the broker B_3. Hereafter the steps of these subscription and publication procedures are described.

1. Client C_1 starts by sending to broker B_1 a *SUBSCRIBE* request with the *Topic Name* value equal to T_X @ B_3 @ B_2.

2. When broker B_1 receives the request, it does the following steps:

   i) it saves the subscription of C_1 to the topic T_X @ B_3 @ B_2 in its database,

   ii) it forwards the request to B_2 changing the *Topic Name* value to T_X @ B_3.

3. When broker B_2 receives this request, it:

   i) stores in its database a record that B_1 has subscribed to topic T_X @ B_3,

   ii) sends a request to the broker B_3 to subscribe to the topic T_X.

4. Broker B_3 receives the new request and processes it as a standard MQTT subscription request.

5. When client C_2 wants to publish a message M_X to topic T_X on broker B_3, it sends a *PUBLISH* request to broker B_4, setting *Topic Name* value to T_X @ B_3.

6. Broker B_4 receives the request and forwards it to broker B_3, changing the *Topic Name* value to T_X.

7. B_3 receives the request and processes it as a standard MQTT publication request:

   i) it retrieves the list of subscribers to topic T_X; in this example the list includes (only) B_2,

   ii) it sends message M_X to the subscribers (B_2).

8. When B_2 receives the *PUBLISH* message, it (i) retrieves the list of entities subscribed to the obtained topic (T_X) together with the original *Topic Name* values, and (ii) sends the new request to those subscribers using the proper *Topic Name*. In the provided example, B_2 sends the *PUBLISH* message to B_1 with topic name equal to T_X @ B_3.

9. Similarly, when broker B_1 receives the request, it (i) retrieves the list of subscribers and *Topic Name* values, and (ii) forwards the message accordingly. In the example it sends message M_X to client C_1 with *Topic Name* equal to T_X @ B_3 @ B_2.

Not only the proposed system allows to communicate with remote brokers, but it also enhances the MQTT protocol, allowing a client forward the requests destined to several brokers in a single message, while the standard approach supports only requests for different topic names but managed by the same broker.

2.2 Secure dynamic broker bridging

Security is important for any IoT system. While security has various objectives, authorized access is the most important aspect when it comes to MQTT communication. An entity that does not have a permission to publish or to receive information on a particular topic must be not able to publish or to subscribe to that topic.

In this section, we have extended the authentication and authorization scheme of [20] by considering the scenario of multiple brokers described in the previous section, and a new third party-based authentication and authorization mechanism is defined. In the proposed mechanism, the presence of a single trusted Authentication and Authorization server (AS) is considered. In all described cases, the AS is in charge of the client authentication and generation of authorization tokens (access tokens), used by the clients in order to prove the rights to subscribe to a topic or to publish a message in a topic on a given broker (or a set of brokers). Depending on the type of the scheme used for authorization, the access token can be a simple opaque token (explicitly verified by the broker by interacting with the AS) or a fully

self-contained cryptographic token that allows the broker to directly verify the access grants.

Taking into consideration the constraints that IoT nodes may have, exploiting a trusted third party helps to decrease the load of other nodes. Such an approach also eliminates the devices' need to be "acquainted" with all the other participants of the network.

When dealing with authentication/authorization in presence of multiple MQTT brokers, different cases could be considered:

- *Client-to-all-brokers authorization, one token for all brokers* – Authorization server, after successful authentication of the client, releases to the client an access token which is then used for accessing each broker on the way.
- *Client-to-all-brokers authorization, a token for each broker* – The AS releases to the client different access tokens, one token for each broker through which the client wants to route the request. The client includes all these tokens in the request that is sent to the first broker. The tokens are then used, verified and removed from the request hop-by-hop by all brokers the client request is routed through.
- *Hop-by-hop authorization* – The AS releases to the client an access token valid only for the first broker the client will communicate with. All other brokers relaying the request have to explicitly communicate with the AS in order to obtain a new token to send the message to the next broker.

According to the first and second approaches, the access is directly granted from the AS to the client and other brokers, and it has the advantages of minimizing the interactions with the AS and the number of exchanged messages.

Conversely, in the second approach, the grant is provided only for single interaction (client-to-broker and broker-to-broker). In this case, the intermediate brokers request from the AS an access token for forwarding the message to the next broker, thus implementing a sort of an authorization delegation chain. Although according to this approach the number of interactions with the AS significantly increases, it has

the advantage of reducing the amount of authorization data that has to be processed by each hop and included in each request, resulting in a smaller packet size.

Hereafter, the proposed approaches are described in detail.

2.2.1 Client-to-all-brokers authorization: one token for all brokers

The first approach states that a client, after authenticating with the AS, obtains a single access token which it then uses in order to prove the right of the access to a given sequence of brokers.

In Figure 2.4 an example of this scenario is depicted. In the proposed scenario, the client C_1 wants to subscribe to a topic T_X on broker B_n or to publish a message to topic T_X on broker B_n. In the provided example, client C_1 wants to forward its request through the sequence of intermediate brokers $B_1, B_2, \ldots, B_{n\text{-}1}$ towards the destination broker B_n.

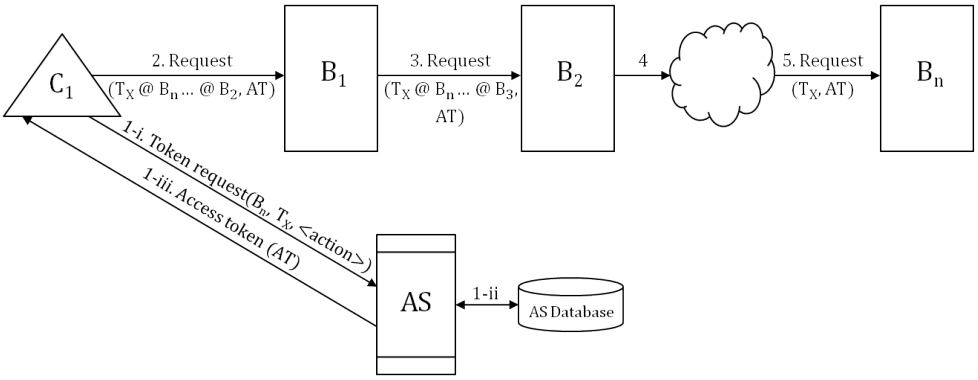

Figure 2.4: A client obtains a single token for all brokers.

The authentication and authorization process will include the following steps.

1. Before sending the request to the first broker, C_1 authenticates with the *AS* and obtains a valid access token (AT) for the destination broker, granting to the client the access (subscribing and/or publishing) to the topic T_X on that broker. The authentication request includes the credentials of C_1 (e.g., username and

password), or implements a different authentication scheme. After successful authentication:

i) C_1 sends an authorization request containing the identifier of the destination broker (B_n in the provided example), topic name or a list of topics (in the described example topic T_X), and the action that the client wants to perform (subscription and/or publication);

ii) the *AS* matches the C_1's request with the authorization policy stored in the policy database;

iii) if C_1 is authorized to perform the actions claimed in the request, the AS returns an access token (*AT*) intended for the requested broker B_n.

2. C_1 sends the request to B_1 with T_X @ B_n @ $B_{n\text{-}1}$ @ ... @ B_2 as the *Topic Name* value (as described in Section 2.1). The request contains also the access token *AT* obtained from the Authorization server.

3. B_1 receives and processes the request according to the procedure described in Section 2.1.

4. The same operations are performed by all other intermediate brokers (B_2, B_3, ..., $B_{n\text{-}1}$.

5. When the request reaches the destination broker B_n, the broker checks whether the request contains an access token *AT*. If the token is present and is valid, the broker B_n processes the request in the standard MQTT way.

Although this option leads to the minimal load of all the participants of the communication process, it should be applied only in constrained environments where the following two approaches are not feasible.

2.2.2 Client-to-all-brokers authorization: a token for each broker

According to the second approach, a client, after authenticating with the AS, obtains a set of access tokens granting it the access to a given sequence of brokers.

In Figure 2.5 an example of this scenario is depicted, where the client C_1 wants to subscribe or to publish a message to topic T_X on broker B_n. In the provided example, client C_1 wants to forward its request through the sequence of intermediate brokers $B_1, B_2, \ldots, B_{n\text{-}1}$ towards the destination broker B_n.

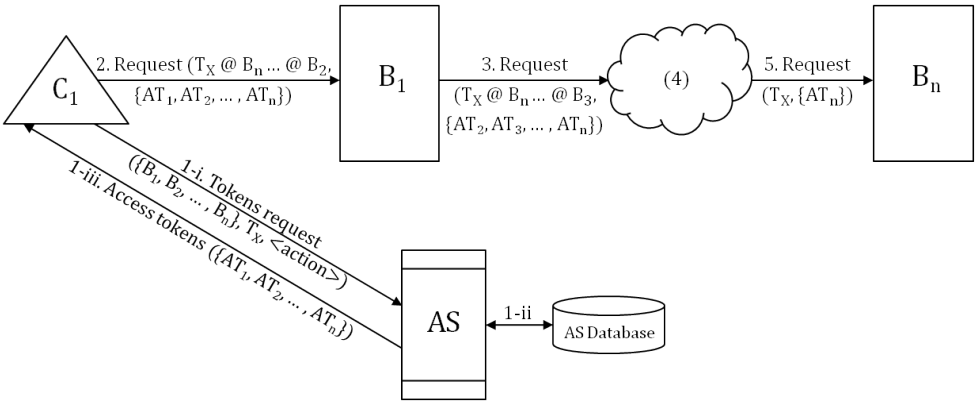

Figure 2.5: A client obtains tokens for each broker.

The complete authentication and authorization procedure consists of the following steps.

1. Before sending the request to the first broker, C_1 authenticates with the *AS* and obtains a set of valid access tokens $\{AT_1, AT_2, \ldots, AT_n\}$ for the corresponding sequence of brokers, granting the access (subscribing and/or publishing) to the topic T_X. The authentication request includes the credentials of C_1 or implements a different authentication scheme. After successful authentication:

   i) C_1 sends an authorization request containing the list of brokers, topic name or a list of topics (in the described example topic T_X), and the action that the client wants to perform (subscription and/or publication);

   ii) the *AS* matches the C_1's request with the authorization policy stored in the policy database;

   iii) if C_1 is authorized to perform the actions it claimed in its request, the AS returns a list of access tokens $\{AT_1, At_2, \ldots, AT_n\}$ for the requested

brokers $B_1, B_2, \ldots, B_n$

2. C_1 sends the request to B_1 putting T_X @ B_n @ $B_{n\text{-}1}$ @ ... @ B_2 to the *Topic Name* field (as described in Section 2.1). The request contains also the list of access tokens obtained from the AS.

3. B_1 receives and processes the request. It checks whether the request contains a token AT_1 for this broker. If the token is present and is valid, the broker removes it from the list. The request message is then processed according to the procedure described in Section 2.1.

4. The same operations are performed by all other intermediate brokers (B_2, B_3, ..., $B_{n\text{-}1}$.

5. When the request reaches the destination broker B_n, the broker verifies the last access token AT_n and finally processes the request in the standard MQTT way.

The described approach is suitable for scenarios where clients do not run on very constrained nodes and are able to transmit bigger amount of data.

2.2.3 Hop-by-hop authorization

In the third approach, access tokens are provided by the AS only for communication with next hop of the forwarding path. In Figure 2.6 an example of this scenario is depicted. Like in the previous case, client C_1 wants to subscribe or to publish a message to topic T_X on the broker B_n. In order to reach the destination broker B_n, the client C_1 routes the request through a sequence of intermediate brokers $B_1, B_2, \ldots, B_{n\text{-}1}$.

Differently from the previously described cases, the client obtains an access token valid only for the first broker (B_1). Broker B_1 is then in charge of requesting a new access token for relaying the request message to the next broker (B_2). The same rule applies to B_2 and next brokers until the penultimate broker $B_{n\text{-}1}$, as hereafter described.

1. In order to send the request to the first broker, client C_1 authenticates with the *AS*. After being successfully authenticated, C_1 sends an authorization request

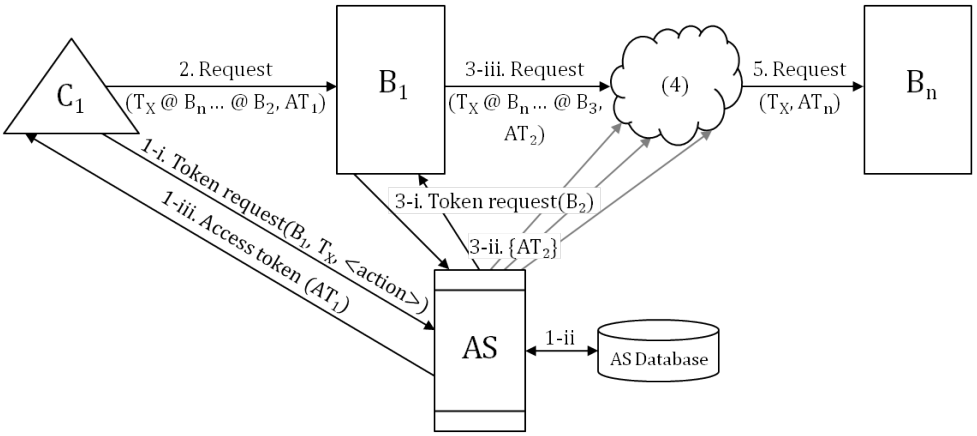

Figure 2.6: Each entity obtains a token for next hop.

containing the identifier of broker B_1, a topic name or a list of topics (only topic T_X in this example), and the action that it wants to perform (subscription and/or publication). If the C_1's authorization request matches the policies stored in the policy database of the AS, the requested token AT_1 is returned to C_1.

2. Setting *Topic Name* value to T_X @ B_n @ $B_{n\text{-}1}$ @ ... @ B_2 (as described in Section 2.1), C_1 sends the request to B_1. The request also contains the access token AT_1 obtained from the AS.

3. B_1 receives and processes the request. It checks whether the request contains a token AT_1 for this broker. If the token is present and is valid, B_1 removes it from the C_1's request and then authenticates with the AS and requests an access token AT_2 for relaying the client's request to B_2. After receiving the token AT_2, B_1 includes this token in the C_1's request message and forwards this message to B_2 according to the procedure described in Section 2.1.

4. The same operations are performed by all other brokers up to and including $B_{n\text{-}1}$.

5. When the request arrives at the destination broker B_n, the broker verifies the

access token AT_n and processes the message as a standard MQTT request.

Although this approach increases the number of connections to the AS, it significantly unloads the client both in terms of data storage and processing.

2.3 Implementation

The proposed dynamic broker bridging mechanism has been implemented in Python programming language. Python has been chosen due to the code compactness and flexibility. An extension pack of bridge functions (decomposition and validation of topic name structure, various data processing, requests forwarding, etc.) has been implemented in order to provide brokers and clients with ability to use dynamic bridging mechanism. The *HBMQTT*[2] package has been used as basis for the broker implementation, while dynamic bridges and clients instances are based on *Paho Python Client*[3] class.

The original database and database-related functions of *HBMQTT* broker implementation have been modified in order to store new data, such as the topic aliases. Additionally, a new storage containing the data about the bridges established for communicating with other brokers has been designed for controlling active connections and existing subscriptions on those brokers.

In Figure 2.7 a flowchart of subscription request processing, modified according to the dynamic broker bridging mechanism, is presented. The initial *topic_name* field contains the *Topic name* value received from the sender (which can be either a client node or another broker) identified by its *client_id* field.

The variables *fwd_topic_name* and *fwd_broker_id* contain the parts of the *Topic name* value before and after symbol "@" respectively. In the provided implementation, *fwd_broker_id* is expected to contain the next-hop (to which the request has to be forwarded) broker's IP address and port number according to "*IP address:port number*" pattern.

[2] https://hbmqtt.readthedocs.io/

[3] https://www.eclipse.org/paho/clients/python/

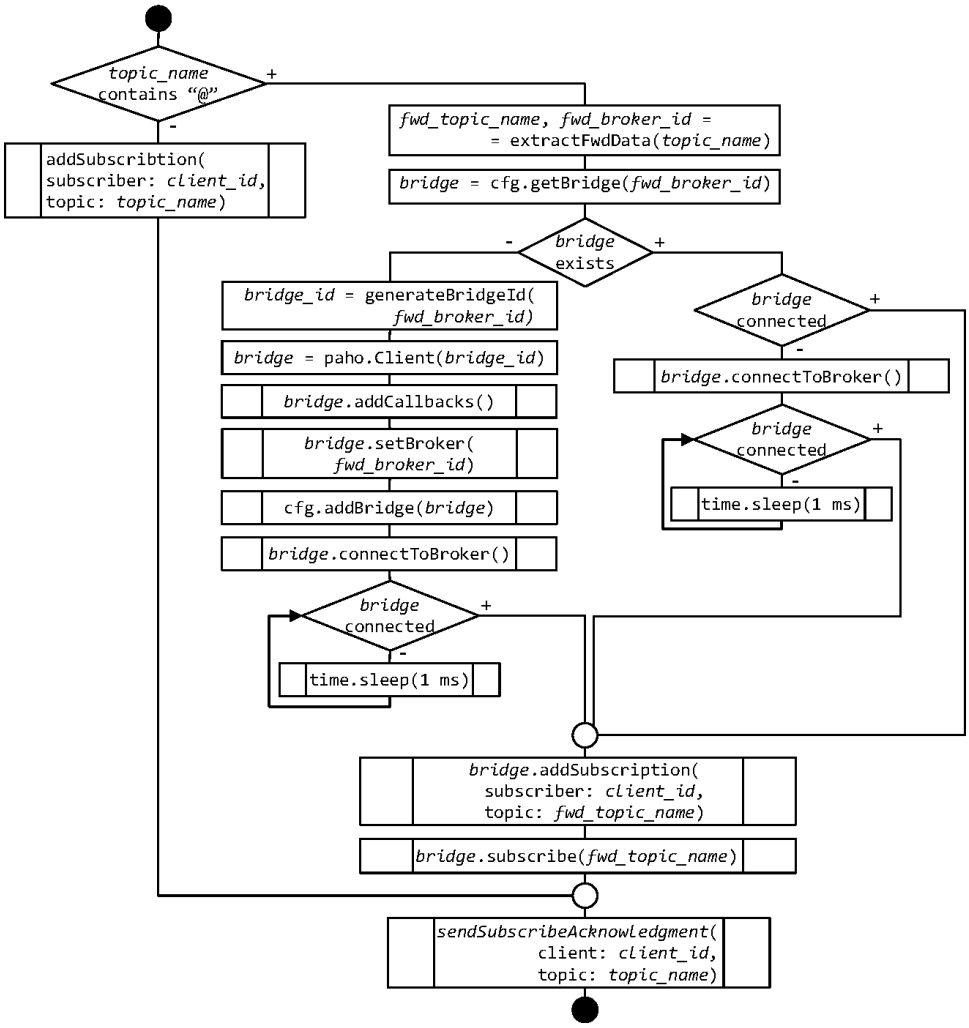

Figure 2.7: Subscription processing.

Method *cfg.getBridge(fwd_broker_id)* returns the pointer to the bridge which has been already setup for communication with the next-hop broker if such a bridge exists. Otherwise, a new bridge must be created according to the following algorithm:

1. A string value *bridge_id*, which will be further used as client ID when the

bridge sends requests to the other broker, is equal to *"client_"+<current broker's identifier>+"_"+<next-hop broker's identifier>*. Generation of such a string instead of generating a random value significantly reduces the probability that other client has the same identifier. Moreover, this value simplifies search among existing bridges on a broker.

2. A new instance of *Paho Python Client* class is created. This instance is the bridge that will connect to the other broker.

3. A predefined set of callback methods, modified in order to provide necessary functionality (e.g., incoming messages processing, subscription management), is added to the bridge.

4. The bridge instance is added to the broker's database of bridges.

5. The bridge sends a *CONNECT* packet to the next-hop broker and waits for the *CONNACK* (connection acknowledgement) response.

Once the connection between the bridge and the next broker is established, the following events occur:

1. A new subscription record, containing the sender's identifier *client_id* and *fwd_topic_name* values, is added to the bridge's database of subscriptions.

2. The bridge sends a *SUBSCRIBE* request to the next-hop broker, setting *Topic name* to *fwd_topic_name*.

3. When the subscription has been accepted by that broker, the current broker replies to the sender with an acknowledgement *SUBACK* message.

Handling of the publication requests that must be forwarded to another broker is similar to the subscription process described above, with the only difference that if a new bridge has been created specially in order to forward the *PUBLISH* request, then, after receiving an acknowledgement from the next-hop broker, this bridge should be disconnected and deleted.

When a broker receives an *UNSUBSCRIBE* request, it has to perform the actions depicted in the flowchart in Figure 2.8.

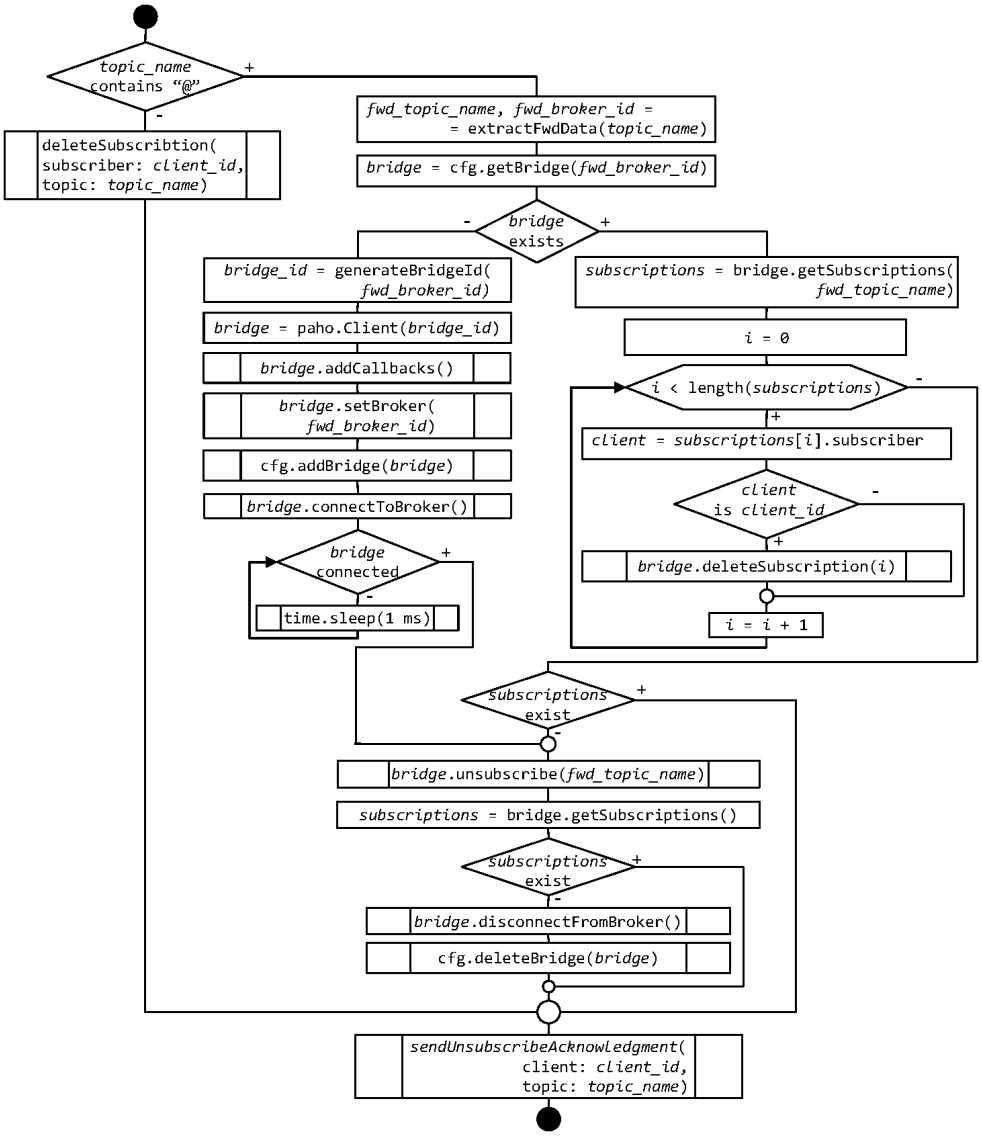

Figure 2.8: Unsubscription processing.

In the provided scheme, the *topic_name* value is the one obtain with the *UNSUBSCRIBE* request and *client_id* is the request sender's identifier.

First, the broker has to check whether the *SUBSCRIBE* request needs to be forwarded to another broker. In the provided implementation it is done by checking for the presence of the symbol "@" in the *Topic name* string. If the request does not have to be forwarded, then the broker processes it in a standard way. Otherwise, the variables *fwd_topic_name* and *fwd_broker_id* are set with the parts of the *Topic name* value before and after symbol "@" respectively.

If a bridge to the broker with *fwd_broker_id* has not been established yet, then a new bridge in a similar way to the subscription processing. Once a bridge is created, it connects to the next-hop broker and sends an *UNSUBSCRIBE* request with the *Topic name* value set to *fwd_topic_name*. As soon as the unsubscription acknowledgment arrives from that broker, the bridge closes the connection and is deleted.

Otherwise, if there is already a bridge established to communicate with the next-hop broker, the following events occur:

1. The bridge performs a look-up for all subscriptions to the topic *fwd_topic_name* in its database of subscriptions. The list of those subscriptions is returned to the *subscriptions* variable.

2. The subscriptions of the client *client_id* to the topic *fwd_topic_name* are deleted from the database.

3. If after the previous step no other subscription to the topic *fwd_topic_name* on the broker *fwd_broker_id* is left in the bridge's database, then:

   i) the bridge sends an *UNSUBSCRIBE* request to that broker;

   ii) if no subscription to any other topic on the broker *fwd_broker_id* is left, then the bridge disconnects from the broker and is deleted.

Finally, the current broker sends an acknowledgement (*UNSUBACK*) to the request sender informing it about successful cancellation of the subscription to the topic *topic_name*.

The implementation has been performed in Python 3.7 and tests have been made in Windows 10 and Linux environments: diverse combinations of brokers and clients instances were manually run and exchanged messages. The tests have demonstrated feasibility of the proposed solution.

With the aim to perform a big amount of tests automatically, an emulator has been specially designed and implemented. The emulator runs a given amount of client and broker instances, which then intercommunicate with accordance to the above described dynamic bridging mechanism.

The testing process can be divided into the following steps:

1. The main batch file is used to execute a set of commands in order to run all necessary broker and client instances. The flowchart of the algorithm of this file is present in Figure 2.9.

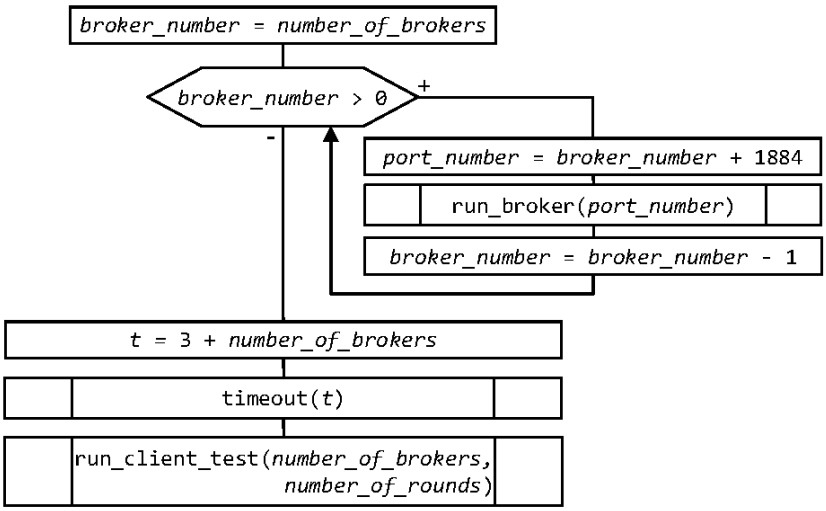

Figure 2.9: Test setup algorithm.

In the provided scheme, *number_of_brokers* and *number_of_rounds* are input parameters set by the user. The variable *number_of_brokers* is the number of brokers participating in the test, and *number_of_rounds* is the number of requests that will be performed by the client to a broker: the higher this value is,

the more accurate test results will be.

As the first step, the batch file runs all modified broker instances using different port numbers. After that, the appropriate client testing script is run. Since it takes time for a broker to start, the batch file waits for t seconds before running the clients in order to avoid the error when a client tries to connect to a broker that is not active yet. Value t has been calculated based on the time the brokers needed to start working.

2. The test client script performs requests to the brokers and registers the execution time of each request. The flowchart of the algorithm of this file is present in Figure 2.10.

   For an amount of brokers (represented by the variable *path_length*) which varies from 1 to the number of brokers set by the user (*number_of_brokers*) the client performs *number_of_rounds* rounds of the following actions:

   i) current system time is stored in the array *time_diffs*;

   ii) the client generates a topic name and stores it in the *topic_name* variable;

   iii) the client encapsulates this topic name for each broker on the way;

   iv) when the final topic name value is ready, the client sends the request to the first broker of the path and waits for the acknowledgement;

   v) current system time is stored in the array *time_diffs* in order to calculate the time spent for the round later.

   After performing all the requests to each amount of brokers, the time spent for each round is computed and the data needs to be processed: (i) the noise (e.g., some very high values of execution time caused by unexpected CPU load) should be removed, (ii) the average execution time has to be calculated and stored in the text file.

   In order to remove the noise, a special algorithm has been designed and implemented. The flowchart of this algorithm is depicted in Figure 2.11.

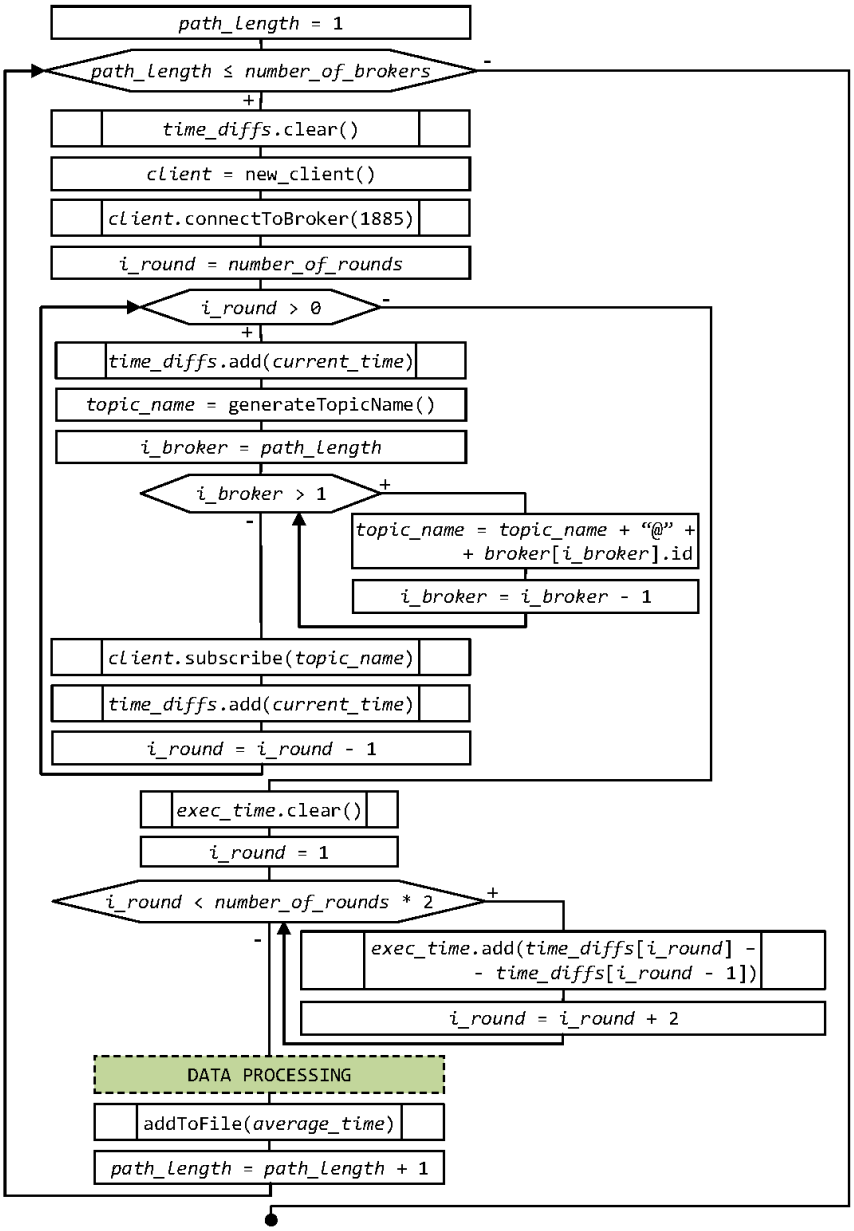

Figure 2.10: Test client algorithm.

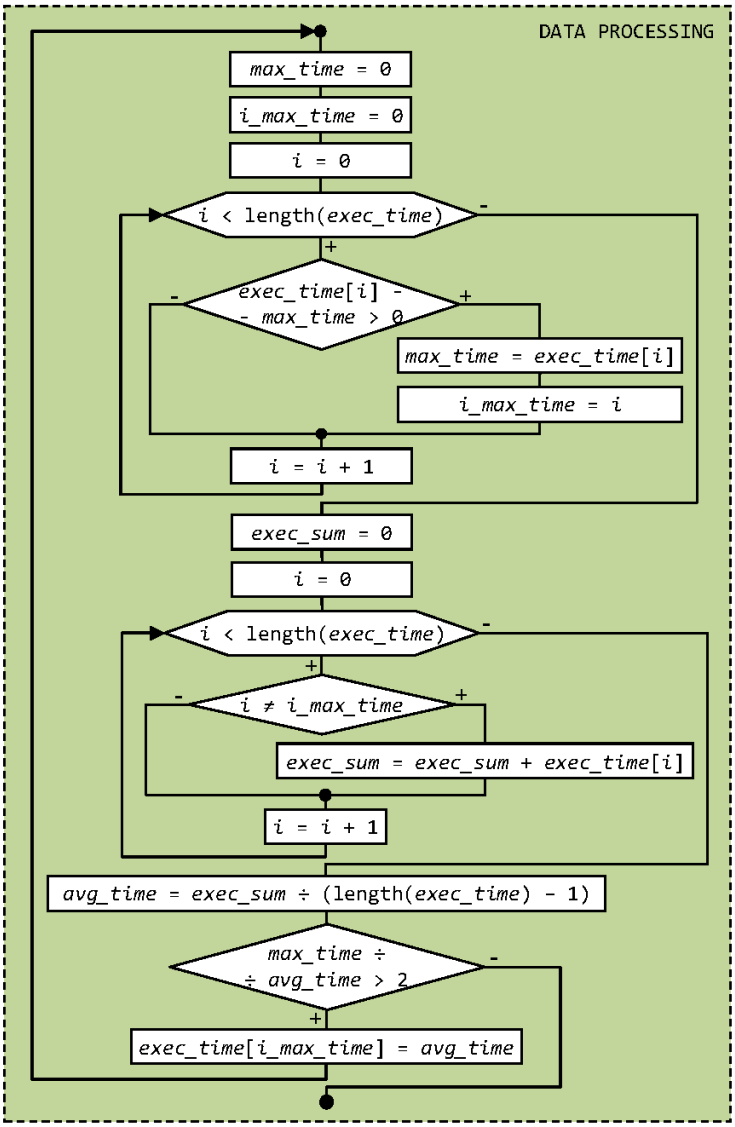

Figure 2.11: Noise removal and average value calculation.

The algorithm searches for the maximum value of the data (*max_time*), calculates the average value of all data (*avg_time*) except for the maximum one,

and, if the maximum value is at least twice greater than the average value, then the algorithm substitutes the value of the maximum element (with the index *i_max_time*) with the average value. Once the noise is removed, the algorithm returns the average value of the data.

3. Based on the results, obtained on the previous step, plots are built. A separate Python script has been implemented to process the results. It obtains the data from the text file/files, filled by the client test script, and builds one ore more plots. The *scipy.interpolate*[4] tools have been used to perform interpolation of the data in order to make the resulting plots look smoother, and the MATLAB-like plotting framework *matplotlib.pyplot*[5] has been used for building the plots.

In Figure 2.12 the results of testing the implemented dynamic broker bridging system in Linux environment are represented. An UP board (Intel® Atom™ x5-Z8350 CPU, 4GB DDR3L-1600 RAM, 64 GB eMMC storage) with Ubuntu 18.04 operating system has been used for testing the implementation performance.

In the provided tests, a client sends a *SUBSCRIBE* request to a broker, exploiting a sequence of intermediate brokers to forward the request. The number of brokers in the path varies from 1 (the request is sent directly to the destination broker) to 20 (the request is forwarded by 19 intermediate brokers).

The green plots demonstrate the execution time for the case in which a path to the destination broker is already created, i.e. MQTT/TCP connections between adjacent brokers are already established. During these tests, first, the client sends a request to the broker B_{20}, thus creating an anonymity path through all the brokers ($B_1, ..., B_{20}$), making them interconnect dynamically by establishing bridges. Then, when the client sends a request to the broker B_{19} (and any of the other brokers $B_{18}, ..., B_1$), the path between them already exists.

The blue plots represent the execution time for the system, in which the path is built hop-by-hop for each request, i.e. the client sends a request to the broker B_1 directly, then to the broker B_2 using B_1 as intermediate broker, after that it sends a

[4]https://pypi.org/project/scipy/

[5]https://pypi.org/project/matplotlib

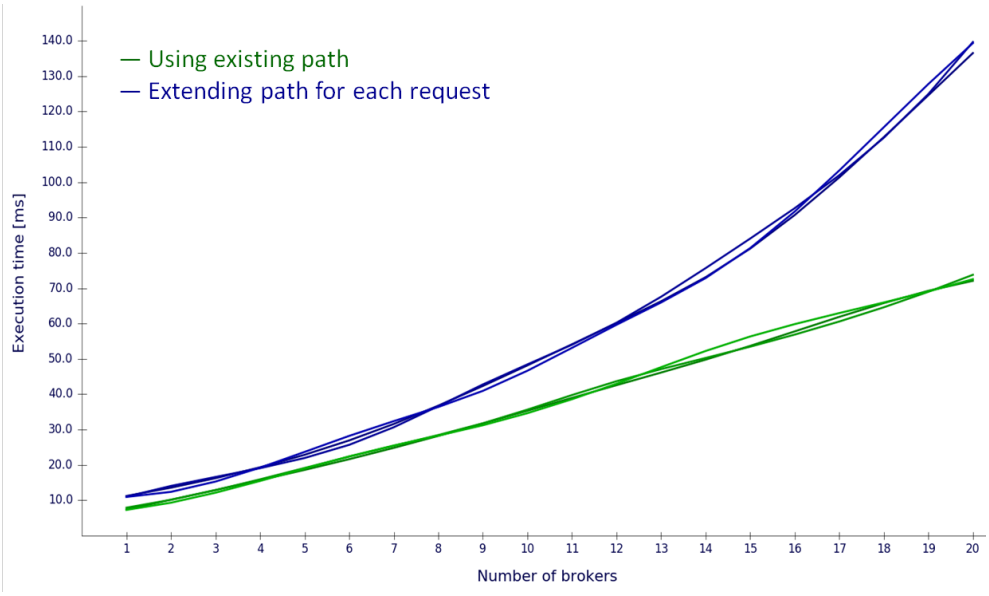

Figure 2.12: Implementation tests: Execution time.

request to the broker B_3 with $\{B_1, B_2\}$ as forwarding path, etc.

The performed tests have shown feasibility of the architecture of the proposed dynamic broker bridging mechanism. According to the tests results, once the multi-broker path is established, the dependence of execution time from the number of intermediate nodes is linear.

2.4 Example of use case: MQTT-based Industrial IoT

Nowadays, IoT devices and technologies are used in nearly every sphere of life. Industrial Internet of Things (IIoT) is a subset of IoT that regards the application of traditional IoT principles to the manufacturing industry. An IIoT system comprises networked smart objects, cyber-physical assets, and optional cloud or edge computing platforms, to enable real-time and autonomous access, collection, analysis and exchange of process, product and service information, within the industrial environ-

ment [39].

The IIoT focuses on the integration of operational technology with information technology, on machine-to-machine (M2M) communications between plants, on a very high number of interconnected devices, and with critical requirements, thus resulting into a peculiar distributed system [40]. The IIoT concept has emerged recently and has a series of challenges and issues to address to guarantee secure and reliable communications.

In an IIoT scenario M2M communication involves sensors, actuators, and controllers can exchange information autonomously. However, many current implemented communication architectures do not provide dynamic interoperability and security. Moreover, the constrained nature of some IoT devices makes it difficult to apply well-known standard security solutions to IIoT and to implement strong cryptographic algorithms for protecting data.

In this section, a dynamic broker bridging mechanism, described in Section 2.1, is applied to an IIoT scenario in order to make the architecture more flexible and scalable, transforming it into a new multi-stage system, where several stages of brokers are used to allow the clients to access different groups of devices.

2.4.1 MQTT-based IIoT production systems

A common organization of production resources in a manufacturing company is depicted in Figure 2.13. A company usually has a headquarter and one or more production sites. Each site can include one or more production lines which are formed by different machines.

PLCs, SCADAs and various distributed sensing systems, formed of IoT devices and organized as Wireless Sensor Networks, are used for monitoring and controlling manufacturing machines. Such systems are connected to per-line and per-site remote controllers. The resulting system can be also connected to the headquarter site and/or to an external Cloud system, in order to enable cross-site monitoring and control.

All mentioned devices and nodes form a complex IIoT network. That network, in turn, in some cases needs to be connected to other external entities (e.g., machine manufacturers, third-party maintenance companies) for different purposes, e.g. for

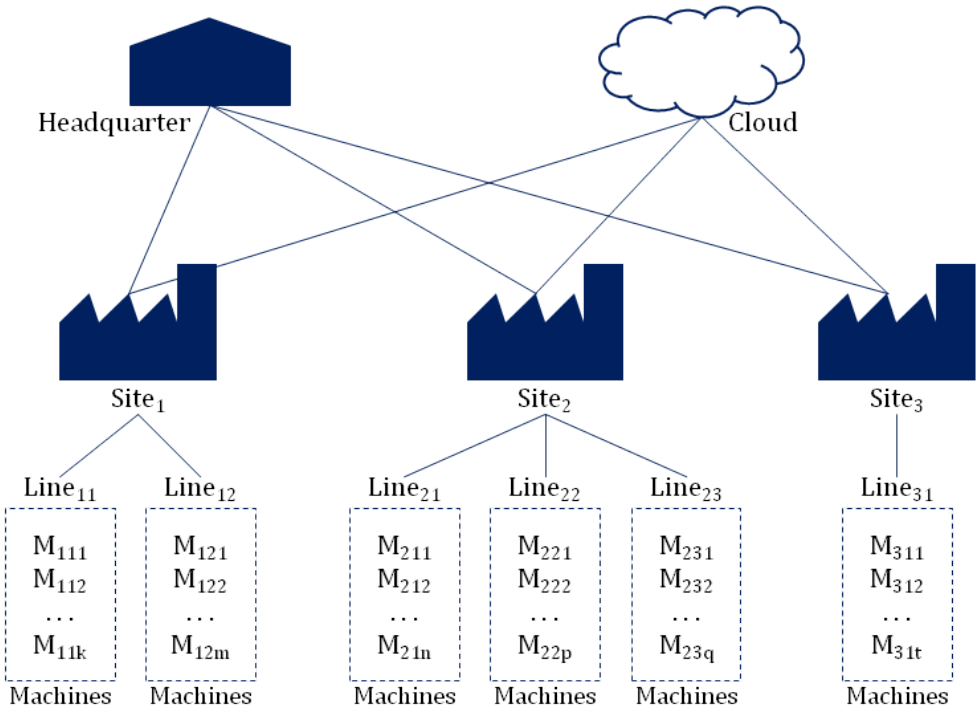

Figure 2.13: A general organization of production resources.

external monitoring.

According to the communication technology, standard protocols and proprietary ad-hoc solutions are currently used. The use of standard protocols is a better option in terms of interoperability, scalability and long-term maintenance. In the provided scheme, the most suitable communication paradigm is publish/subscribe, according to which the data are published by producers (sources) to a server and then are relayed to various consumers (receivers). In an industrial scenario, this mechanism can be used both for sending commands from controllers to target devices and for collecting data from different data sources (e.g., sensors and PLCs).

The publish/subscribe mechanism is usually implemented by means of a central node that receives and publishes data and relays them to the proper subscribers. According to the MQTT notation, this node is denoted as *broker*.

Every production line may be formed of various machines from different manufacturers. In general, it is supposed that machines produced by the same manufacturer can share the same communication system, which is based on a single broker. Accordingly, there must be at least one broker per line and different brokers for different lines within a single site. The resulting architecture is shown in Figure 2.14.

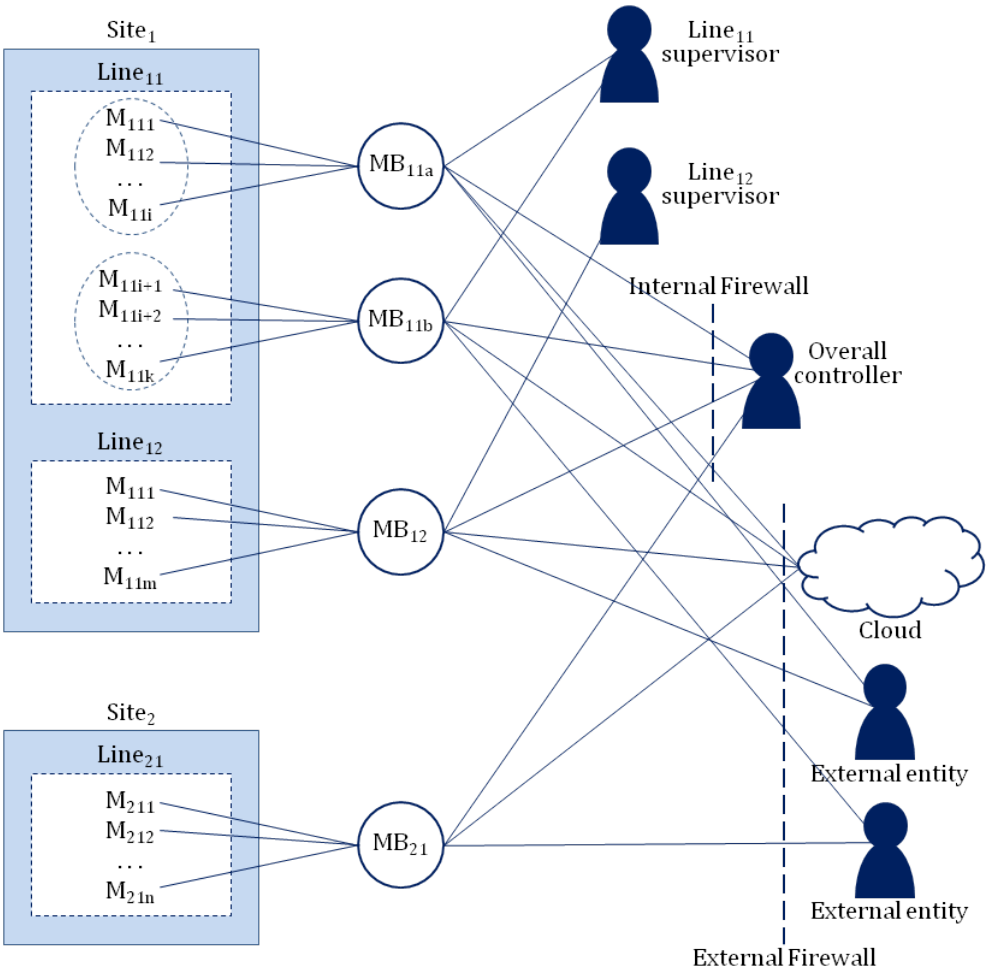

Figure 2.14: An MQTT-based organization of production resources.

In the left part of the figure are production lines with their machines and corre-

sponding brokers (*MB*). In the right part are entities that can interact with the production lines by means of those brokers. In general, at least one per-line control system must be present. The architecture can be further extended to a per-site control system and an overall control system. Data may be also collected on an external Cloud system or by external third parties.

2.4.2 MQTT-based multi-stage IIoT production systems

To provide the appropriate level of security, a proper authentication and authorization system must be designed and implemented. The task gets more complicated due to the high number of possible M2M interactions, which have to be considered separately. Moreover, on the lower lever these interactions have to be explicitly configured on the intermediate network nodes, such as NATs and firewalls. The resulting architecture of the network may be very complex, with a lack of scalability and difficult to configure and administrate.

For this reason, the new mechanism described in Section 2.1 and 2.2 can be used. The resulting system is fully based on publish/subscribe paradigm, properly extended with the novel dynamic multi-broker solution and authentication/authorization mechanism.

The idea behind the proposed solution is to transform the architecture depicted in Figure 2.14: several client-to-broker interactions are grouped and mapped to a new multi-stage architecture, in which different stages of brokers are used to allow the clients to access different groups of devices separately.

The new architecture is presented in Figure 2.15. In the left part are the production machines grouped by line. Assuming that machines of different machine manufacturers are connected to different manufacturer brokers (regardless the use of the same pub/sub protocol), the machines are grouped by manufacture as well.

In the right part of the figure are data consumers and controllers. Instead of connecting them directly to the brokers that manage the topics/devices in which they are interested, one more level of brokers has been introduced. These second-level brokers group machines with accordance to diverse access classes. For instance, the division can be the following:

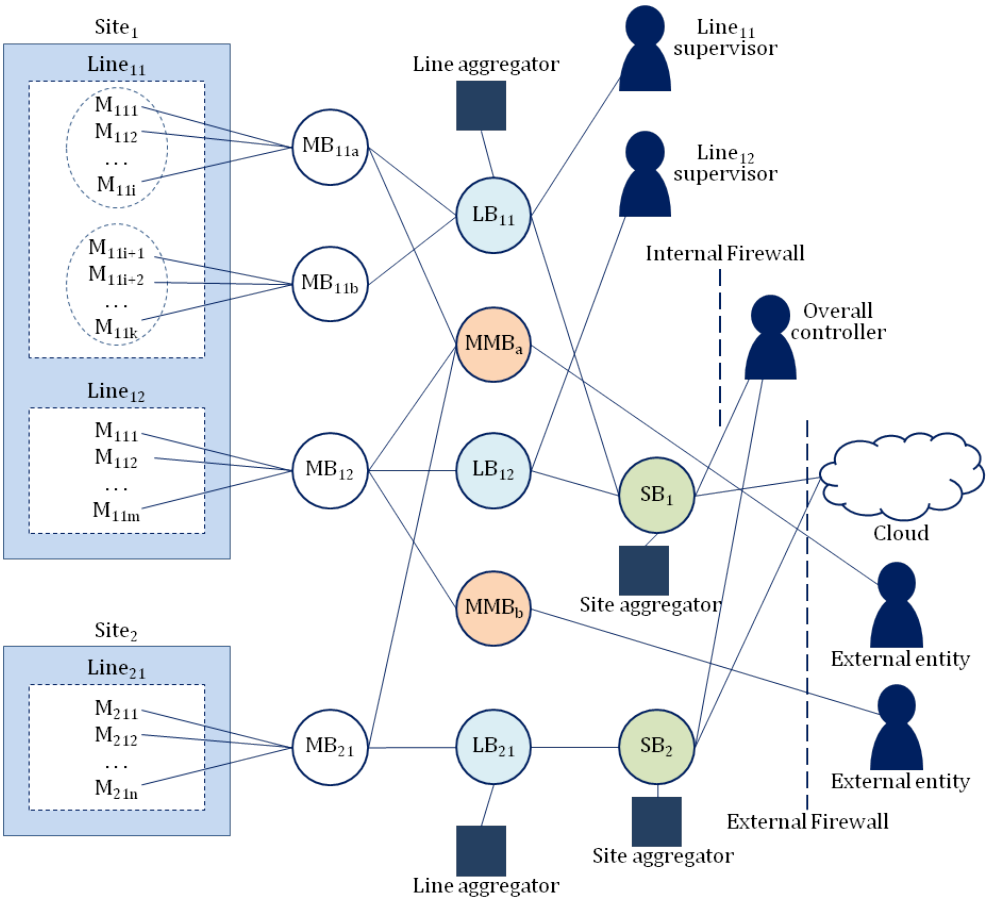

Figure 2.15: A multi-stage MQTT-based organization of production resources.

- brokers associated to different production lines, with one broker for each line (brokers LB_{ij} in Figure 2.15);

- brokers associated to different machine manufacturers, with one broker for each manufacturer (brokers MMB_i in Figure 2.15).

Other types of division into classes with corresponding brokers can be introduced. After collecting data and before re-publishing it, the brokers may perform

data aggregation operations according to consumer profiles. For instance, per-line data can be aggregated in order to provide overall measurements.

In a similar way, another another stage of brokers can be introduced with the aim of further grouping of different types of M2M interactions and of providing enhanced aggregation functions. For example, in case of a multiple-site manufactory, per-site brokers can be introduced (brokers SB_i in Figure 2.15).

The proposed architecture has the following advantages:

- due to grouping M2M interactions, it significantly simplifies client-to-broker relations, which leads to simplification of designing and performing authentication and authorization;
- the number of network level relations is reduced, which has to be considered in case of NAT and firewall configurations, since it simplifies the network administration, and drastically reduces the possibility of vulnerabilities caused by firewall miss-configuration or wrong-configuration;
- the new architecture is more scalable, since it reduces the total number of flows that each broker has to manage;
- it simplifies the design and implementation of new data processing functions fully integrated in the multistage publish/subscribe architecture.

2.5 Conclusions

In this chapter, a novel dynamic broker bridging mechanism has been presented. The design of the dynamic broker bridging is based on the standard broker bridging concept, however, it does not require a manual set-up of each broker – the feature that makes it able to dynamically adapt to different scalability requirements. Three authorization and authentication schemes have been designed for the proposed mechanisms: all requests to brokers contain authorization tokens released by an authorization server, so only authorized entities can subscribe or publish to a particular topic. The proposed scheme has been applied to the Industrial IoT scenario with the aim

of relaying M2M communications, based on publish/subscribe paradigm. The proof-of-concept implementation of the dynamic broker bridging mechanism has been described in detail. A simulator has been specially designed in order to allow for testing the proposed system automatically. The algorithm of the testing module has been described together with the tests results which prove the feasibility of the proposed architecture.

Chapter 3

Network Level Anonymity

Man is least himself when he talks in his own person.
Give him a mask, and he will tell you the truth.

– Oscar Wilde

Designing IoT systems with security features is a challenging task. Proper mechanisms should be added taking into considerations devices constraints, which means that they should not introduce excessive processing load or protocol overhead. While such security aspects as authentication, confidentiality and data integrity have been widely studied, end-to-end anonymity still remain an open issue for IoT.

This chapter focuses on providing anonymity to IoT systems. In particular, we introduce an anonymization protocol for datagram-based communication. The scheme has been specially designed for constrained scenarios that are typical for the Internet of Things. According to the proposed protocol, IoT entities form onion routing (OR) [37] anonymity networks and then create anonymity paths through intermediate nodes. The design also implies confidentiality, thus eliminating the necessity to use any secure communication protocol.

This chapter is organized as follows. In Section 3.1 the motivation of the work is explained, an overview of the anonymization protocol design is provided, and the explanations of the mechanisms used for anonymity path establishment and data ex-

change processes are given. Section 3.2 describes novel features of the proposed system, which allow to increase the level of anonymity. In Section 3.3 a proof-of-concept implementation and testbed are described. Finally, in Section 3.4 conclusions are drawn.

3.1 Datagram-based OR anonymity

Up to now, the problem of communication anonymization has been mostly studied for Human-to-Human or Human-to-Machine interactions in the traditional Internet, and the main mechanism that is currently used is Tor [41]. Tor is a low-latency anonymity system based on onion routing that allows the set-up of anonymous communications between a user node (called onion proxy) and a remote anonymity-unaware node (called corresponding node). Anonymization in Tor systems is provided by means of an anonymity circuit, which is setup between the source onion proxy and an exit node (EN). The onion proxy builds the anonymity circuit hop-by-hop in a telescopic manner.

However, Tor is designed for connection-oriented Internet applications and it cannot be directly applied for securing typical for IoT Machine-to-Machine (M2M) interactions through connectionless protocols run on non-persistent constrained nodes. The main problems of adopting Tor in the IoT context are the following:

- Tor has been designed for anonymizing TCP-based applications; IoT applications, on the contrary, are mainly based on UDP;
- Tor provides connection-oriented transport service, which may add undesirable delay in data exchange process;
- due to the TCP complexity in terms of both computations and exchanged data size, some ultra-constrained nodes may not support the protocol;
- in order to protect some control fields, Tor creates an additional hop-by-hop security layer of TLS connection between adjacent nodes; however, this extra encryption is useless for the end-to-end data and can be avoided.

In general, when designing anonymization mechanisms, a particular attention should be paid to the issues and challenges of the transport protocol used for end-to-end communication.

One key aspect related to transport protocols is the possible presence of a congestion control mechanism, an error control, and/or re-ordering mechanism. Since TCP possesses all these services, it is widely used in applications. However, a packet loss in TCP causes additional delay of packet delivery, due to both congestion control and packet retransmission. Congestion control is triggered by TCP segments loss, and this means that a failure in segment delivery is considered an indication of network congestion. Due to error and congestion control, loss of a single packet affects also the delivery of successive packets. The problem becomes worse in case multiple TCP connections share the same overlay link: in such a scenario the loss of a packet may affect other TCP connections. Therefore, mapping multiple circuits on one TCP connection should be avoided or carefully thought through.

An anonymization protocol described in this chapter targets IoT applications and is entirely based on datagram transport. It implements onion routing anonymity technique [42], trying to maintain the advantages of already existing onion routing-based anonymity systems and significantly reducing or eliminating the problems that appear when those systems are used in scenarios of constrained datagram-based networks.

Moreover, the design of the protocol implies confidential communication, so no additional security protocol, such as DTLS or IPSec, is needed.

The choice of datagram as transport mechanism has been made due to the following main reasons:

- Many applications for IoT use UDP as transport protocol. Besides, some very constrained devices may not support TCP at all.
- Datagram-based architecture allows to reduce the consumption of resources (processing power, memory, etc.) of the nodes, which is a fundamental goal in constrained environments.

The design of the proposed anonymization protocol is absolutely independent

from the implementation layer, which makes it easy to integrate the protocol in IoT nodes both as a security layer on top of existing UDP or as an extension of IP stack.

3.1.1 System overview

The design of the proposed anonymity system is based on a network of IoT nodes, hereinafter referred as IoT *onion routers* (*IORs*) that create anonymity paths which are then used for end-to-end application communication anonymization.

An anonymity path creation is initiated by a source IOR node (*SN*). The path is composed of several adjacent IORs selected from a larger set of all available IORs that form the anonymity overlay and ends at a target node, that can be either the actual corresponding node (*CN*) with which *SN* wants to communicate or an exit node (*EN*) which is used by *SN* as an exit point for forwarding datagrams to/from the actual *CN*. In the proposed protocol term "source IOR" refers to the first (source) node of the anonymity path and does not correlate with the direction of packets.

Two anonymity modes have been defined for the anonymization protocol:

- Two-way anonymity path mode, in which a bi-directional path is setup from *SN* to a target node (*EN* or *CN*). This path is further used for both sending and receiving datagrams.
- One-way anonymity path mode, in which a mono-directional path is setup from *SN* directly to *CN*. When *CN* wants to send a datagram to the *SN*, it has to establish a new anonymity path.

An example of both modes is depicted in Figure 3.1.

Figure 3.1 (a) represents the two-way anonymity path mode, when, in order to anonymously communicate with the corresponding node Y, a source node X sets up a path lying through IORs Z_1 and Z_2 toward the target node Z_3. Each packet that X sends to Y is encapsulated and encrypted 3 times with the keys K_3, K_2 and K_1 (shared by X and Z_3, Z_2 and Z_1 respectively) in sequence. When an IOR receives the packet, it decrypts it using the shared key, thus removing a layer of encryption, and forwards the packet to the next-hop node. After complete decryption, Z_3 forwards the data to

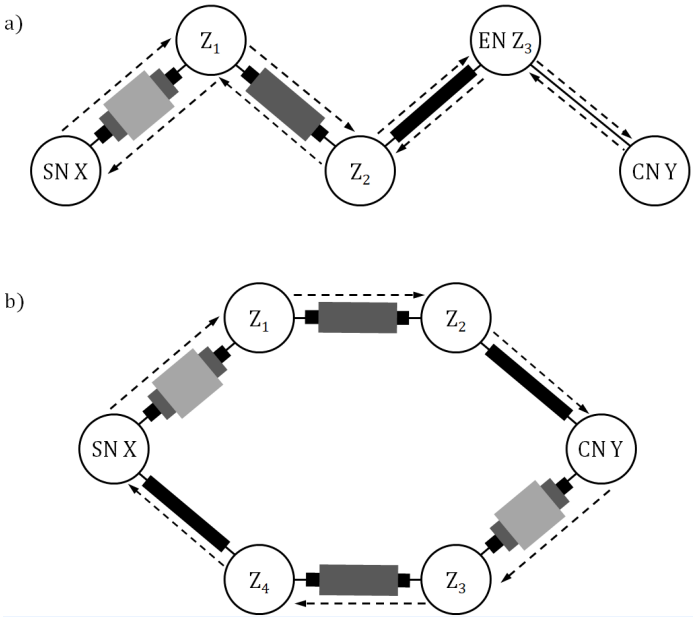

Figure 3.1: (a) Two-way and (b) one-way anonymity paths.

Y. When Y wants to reply to X, it sends the data to Z_3 which forwards them to X through the same path but reversed (Z_3, Z_2, Z_1). Each IOR adds an encryption layer using the key shared with X. One X receives the reply packet, it uses the keys (K_1, K_2, K_3) in a row in order to obtain the original data that Y sent.

Figure 3.1 (b) depicts the one-way anonymity path mode, in which nodes X and Y exchange packets using two different anonymity paths: source node X sets up a path X, Z_1, Z_2, Y, and Y sets up a path Y, Z_3, Z_4, X. Before forwarding packets, X and Y encrypt them using secret keys shared with the IORs of their own paths in reverse order (the key shared with the last IOR of the path is used first). Each intermediate IOR decrypts the packets using the proper key and forwards them to the next IOR until the destination node is reached. When the destination node receives a packet, it removes the last encryption layer and gets the original data.

One of the main differences from other anonymity systems is that the proposed anonymization protocol allows to choose anonymity paths on per-packet basis, thus

significantly increasing the level of anonymity.

3.1.2 Packet structure

When using the proposed anonymization protocol, all packets are sent independently as *anonymity packets*. Each anonymity packet consists of a short header and a payload.

Figure 3.2: Packet header structure.

The header structure is presented in Figure 3.2. The header of each packet contains *packet type*, *path identifier* (*PID*) and *rand* fields:

- The *packet type* field may have the following values: $CREATE$, $CREATED$, $RELAY$, and $PADDING$.
- The *path identifier* (*PID*) is used to associate a given packet with a corresponding anonymity path. Path identifiers are changed at each IOR of the path. Every IOR maintains per-hop state information, such as source address, path identifier, next hop address, next hop path identifier, secret key.
- The *rand* value is used as initialization vector (IV) for hop-by-hop header encryption/decryption and for the various encryption layers of $RELAY$ packets.

Both *packet type* and *path identifier* fields are hop-by-hop protected with a *link key* (*LK*) shared between the two adjacent IORs.

The payload of the packet depends on the packet type:

- In case of $CREATE$ packets, it contains the initiator's (SN's) part of the key exchange (KE_i) encrypted with the public key (K^+) of the target IOR.
- $CREATED$ packets contain the respondent's part of the key exchange (KE_r) and a hash of a created shared key ($H(K)$).

- *RELAY* packets are used for sending commands across the anonymity path. The payload of these packets contains *command code* and a command-dependent payload. The types of *RELAY* packets are the following:
  - $EXTEND$: In case of an $EXTEND$ command code, the payload includes the address of the target IOR and KE_i which is protected with the public key of that IOR.
  - $EXTENDED$: The payload of $EXTENDED$ command comprises a KE_r and a hash of a created shared key ($H(K)$)
  - *DATA*: The command contains the CN's address and the data which SN wants to send.
  - $TRUNCATE$: This command is used when a part of the circuit has to be torn down and contains the address of the *OR* which should remain the last one in the path.
  - $TRUNCATED$: This type of *RELAY* packets is used in order to send an acknowledgement when a $TRUNCATE$ command has been successfully executed.

  The payload of every *RELAY* packet is consequentially encrypted by the SN with secret keys established between this node and each IOR of the chosen anonymity path.
- *PADDING* packets are used for link padding (in order to frustrate potential passive observers counting packets).

3.1.3 Two-way anonymity path establishment

In the proposed protocol, the creation of anonymity paths relies on asymmetric cryptography, i.e. each IOR possesses a pair of keys (public and private), and the public key of each IOR is known to the other nodes.

In case of two-way anonymity mode, when a source node wants to send a datagram to a remote corresponding node, it has to setup a new anonymity path or to

chose an already established one. The anonymity path may end either directly at the CN or at some other IOR node, which in this case acts as an ending node. If a new anonymity path is to be established, it is performed in a telescopic way.

Figure 3.3 shows the process of an anonymity path creation. In the provided example, a SN X creates a path leading to an EN Z_3 and lying through IORs Z_1 and Z_2. The following notation has been used:

- *H: Data* stands for packet header, where *Data* is the contents of the header;
- *P: Data* stands for the payload of a packet, where *Data* represents payload contents;
- $K\{Data\}$ means encryption of *Data* using secret key K.

The procedure of setting up a new path comprises the following steps:

1. SN node X establishes with the first IOR (X_1) a shared secret *link key* LK_1 which will be used between X and Z_1 to protect packet headers.
2. X sends to Z_1 a $CREATE$ packet which contains a new path identifier PID_1 chosen for this hop in its header, and X's part of the KE (KE_{i1}) encrypted with the public key of Z_1 ($K^+_{Z_1}$) in its payload.
3. The Z_1's part of the KE (KE_{r1}) is returned by Z_1 together with the hash of the new key K_1 within a $CREATED$ packet.
4. In order to add an IOR node Z_2 to the anonymity path, X sends to Z_1 a *RELAY/EXTEND* command which contains the address of Z_2 and X's part of the KE (KE_{i2}) protected with Z_2's public key $K^+_{Z_2}$.
5. After receiving the $RELAY$ packet, Z_1 decrypts it and extracts the $EXTEND$ command.
6. After establishment (if not already done) of a link key LK_2 with Z_2, Z_1 sends to it a $CREATE$ packet with a new PID_2 in its header and X's part of the KE still encrypted with $K^+_{Z_2}$.

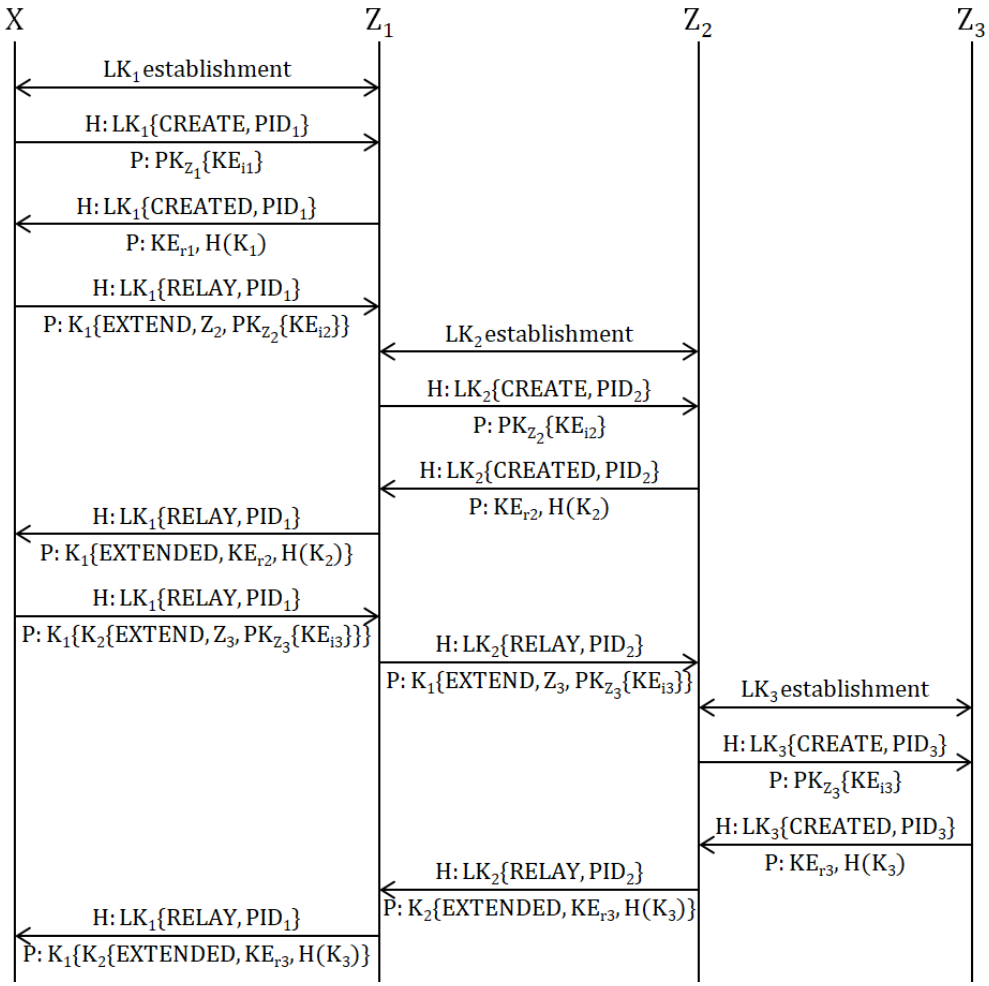

Figure 3.3: An anonymity path setup.

7. When Z_2 gets that *CREATE* packet, it has to complete key exchange and then to send to Z_1 its $KE_{r\ 2}$ and a hash of K_2 within a *CREATED* packet.

8. Z_1 forms an *EXTENDED* command containing the obtained $KE_{r\ 2}$ and $H(K_2)$ values and encrypted with K_1, and sends it to X within a *RELAY* packet.

9. In order to add Z_3 to the anonymity path, X creates an *EXTEND* command

encrypted with the keys K_2 and K_1 for Z_2, encapsulates it into a *RELAY* packet and sends it to Z_1.

10. Each IOR along the path that receives a *RELAY* packet, removes one layer of payload encryption using the secret key associated to the packet's PID, forms a new *RELAY* packet with a next-hop PID, and forwards it to the next-hop IOR.

11. When making the reverse path, every IOR adds a layer of encryption to the payload using the proper secret key and forwards the *RELAY* packet to the previous IOR.

Using this procedure, new hops can be added to the path and/or several anonymity paths can be set-up in parallel.

3.1.4 Data exchange

When a source node X needs to send a datagram/packet to a remote corresponding node Y, X passes the packet to the underlying anonymity layer and the following steps are performed:

1. The anonymity layer creates a new anonymity path $Z_1, Z_2, ..., Z_n$ or selects one of the already established.

2. Each packet is processed in the following way:

    2.1. A new *RELAY* packet containing the *PID* of the selected anonymity path is created.

    2.2. The payload of the *RELAY* packet contains *DATA* command code, which comprises the address of Y and the data payload.

    2.3. The payload of the packet is consistently encrypted with the keys established between X and each IOR of the selected path (K_n, $K_{n\text{-}1}$, ..., K_1).

    2.4. The header of the *RELAY* packet is encrypted with the link key LK_1, and the packet is sent to the first IOR node Z_1.

3. Each IOR along the path performs the steps below:
   3.1. Decrypt the header of the packet using the link key established with the previous IOR.
   3.2. Update the path identifier.
   3.3. Decrypt the packet payload using the shared key established when creating the path, thus removing one "layer" of encryption.
   3.4. Encrypt the header using the link key shared with the next-hop IOR, and forward the packet to that IOR.
4. The last IOR Z_n extracts the Y's IP address and port number and the data payload, forms a new UDP packet, and sends it to Y.

If the corresponding node Y wants to reply to the source node X, the following actions are performed:

1. Y sends the data to the end node (Z_n).
2. EN creates a *RELAY* packet with the path identifier PID_n. In the payload it puts *DATA* command code, in which Y's address and received data payload is encapsulated. The payload of the *RELAY* packet is encrypted with the corresponding shared key K_n, while the header is encrypted with the LK shared with the previous IOR of the anonymity path.
3. EN forwards the *RELAY* packet to the previous IOR node $Z_{n\text{-}1}$.
4. When receiving that *RELAY* packet, each IOR node Z_i ($i = n\text{-}1, n\text{-}2, ..., 1$) performs the following:
   4.1. Decrypt the header of the packet using the link key LK_{i+1}.
   4.2. Substitute the path identifier PID_{i+1} with PID_i.
   4.3. Encrypt the payload of the packet with the key K_i.
   4.4. Encrypt the header of the packet with the link key LK_i shared with the previous node of the anonymity path.

4.5. Forward the packet to the previous node $Z_{i\text{-}1}$ or to the SN in case of being Z_1.

5. The source node X decrypts all the encryption layers using corresponding keys (K_1, K_2, ..., K_n) consequentially and extract the sender's (Y's) address and the data payload.

An example of UDP data exchange between a source node X and a corresponding node Y is shown in Figure 3.4. In the provided example IOR node Z_3 is used as EN.

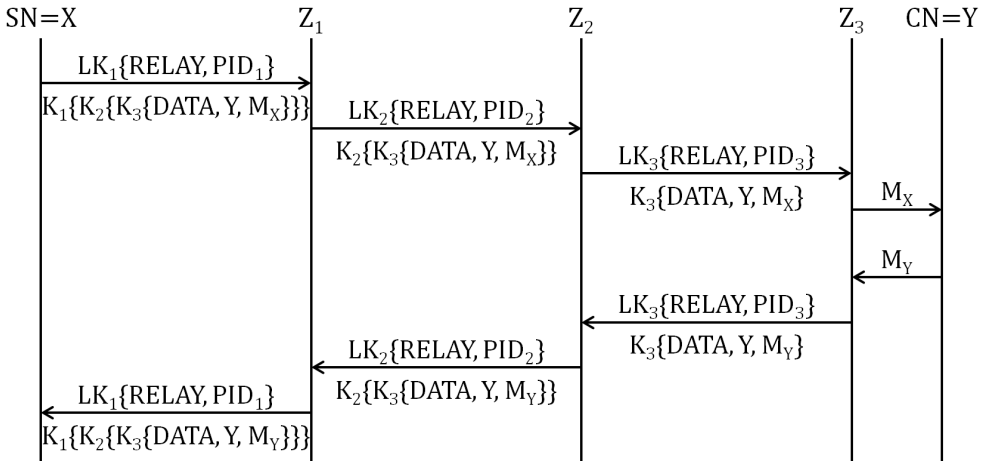

Figure 3.4: An example of anonymous UDP data exchange.

3.1.5 One-way anonymity path mode

The proposed anonymization protocol can also work in one-way anonymity mode, whose main difference from two-way mode is that when two endpoints want to communicate, they independently create anonymity paths towards each other for sending *RELAY* packets. Exchanging datagrams using different anonymity paths helps to increase the overall level of anonymity.

The process of anonymity path setting-up is similar to that of two-way anonymity path mode with two differences: (i) IORs along the chosen anonymity path do not

need to maintain information about backward path, and (ii) *RELAY* packets with *DATA* command code contain both the source and destination addresses.

3.2 Extensions

The proposed protocol provides communicating parties with anonymity. However, it can be further extended in order to increase the overall level of anonymization.

3.2.1 Dynamic anonymity path association

One of the features of the described anonymization protocol is its ability to use different anonymity paths for datagrams that belong to the same communication.

When working in one-way anonymity mode, this can be performed by creating a new anonymity path or by selecting an already established one every time a new datagram has to be transmitted to a remote IOR.

For two-way anonymity mode, similarly to one-way mode, for every packet that has to be sent from source node to the corresponding node an anonymity path is selected separately. However, the packets sent in the backward direction are forwarded by the CN to the previous IOR. In such a scenario, in order to maximize compatibility with anonymity-unaware corresponding nodes, the responsible for the backward per-packet anonymity path association should be the EN. However, according to the proposed anonymization mechanism, non of the IORs knows the source of the data, which means that EN does not know to which SN a circuit belongs.

This issue can be solved by modification of the anonymization protocol by introducing a new field (e.g., *flow_id*) in the payload of the packet destined to the EN, in order to let the EN merge different (k) circuits, i.e. associate diverse anonymity paths with one communication flow. In this case, when a CN sends a response to the SN, the EN will randomly choose one of those k circuits for the backward direction for each packet.

3.2.2 One-to-many communication

The protocol can be extended in order to create a one-to-many anonymity path mode in which an anonymity tree can be built. In such a scenario, at each intermediate IOR, one incoming PID can be associated to a set of outgoing PIDs defining the next-hop node. This feature can be used in different scenarios: (i) in unicast applications, the anonymity path can be dynamically chosen by intermediate nodes, thus increasing the randomness of the routing and the overall level of anonymity; (ii) in case of multicast applications, the anonymity tree can be used to deliver the same UDP packet to multiple destinations. An example of one-to-many anonymity path is depicted in Figure 3.5.

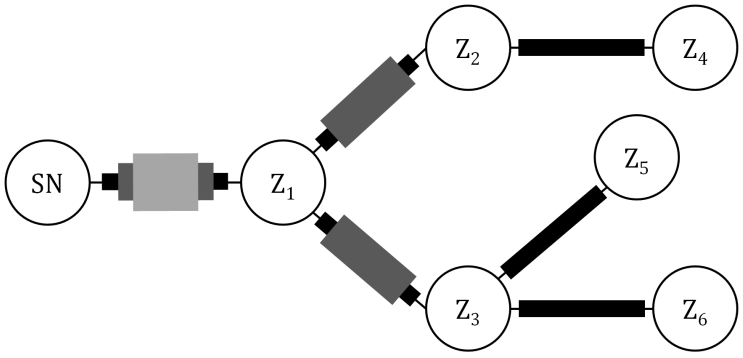

Figure 3.5: An example of one-to-many anonymity path.

3.3 Implementation

The proposed anonymity protocol has been implemented as anonymity user datagram layer that extends the standard UDP layer. The implementation has been done in Java and can be used by simply replacing the standard Java *DatagramSocket* UDP-layer. This allows a simple and easy integration in any UDP-based applications by simply replacing the standard UDP layer with the new anonymity datagram layer.

In order to demonstrate feasibility of the proposed solution, the implementation

has been tested within a testbed [43]. In particular, 100 virtual IoT nodes have been run on 2 Raspberry Pi devices (50 nodes per device). Each node runs both a CoAP client application, bounded to an underlying IOR layer, and a CoAP server application. Periodically CoAP clients send a CoAP GET request to a CoAP server requesting a resource status. All CoAP exchanges are anonymized by using an anonymity path randomly selected by the anonymity layer of the client among a set of already established 10 different paths, each composed by 6 IORs randomly chosen among the complete set of IORs.

3.4 Conclusions

In this chapter, a novel anonymization protocol specifically designed for use in the IoT M2M communication has been described. The proposed mechanism is based on the onion routing concept, but, differently from other known OR-based schemes, it is completely datagram-oriented. Two modes for anonymity path setup have been designed: (i) two-way anonymity path mode in which communicating nodes use the same sequence of IORs in order to send datagrams; and (ii) one-way mode in which a new anonymity path is built when the responding node needs to send data, thus making it harder to identify the communicating parties. Another novel and important feature of the protocol is the possibility to choose a new anonymity path for each packet, which significantly increases the overall level of anonymity. The solution described in this chapter can be considered lightweight, due to relatively small cryptographic and protocol overhead. The proposed anonymization protocol has been implemented and tested in order to demonstrate its feasibility.

Chapter 4

Application Level Anonymity

No one cared who I was until I put on the mask.

– Christopher Nolan, The Dark Knight Rises

The anonymization mechanism described in Chapter 3 has the advantage that it works at network or transport layer and can be used for different application protocols. However, the main disadvantage is that it requires modification of the network or transport layer. In some cases it is necessary to provide a security scheme which does not depend on the underlying layers. In such circumstances, the solution has to be designed at the application layer.

In this chapter, we present a mechanism providing anonymous communication for systems based on MQTT and MQTT-SN protocols. The design of the proposed solution is based on the novel dynamic broker bridging, described in Section 2.1. In the scenario proposed below, brokers act similarly to onion routers [37], and a client can create an anonymity path lying through these brokers, which it can then use for anonymizing messages sent to or received from a remote broker, thus making it impossible for any participant of the communication process to know which topic it is interested in.

The structure of this chapter is following. In Section 4.1 a scheme for anonymous communication is introduced and provided with in-depth examples of subscription

and publication processes. Section 4.2 describes the proof-of-concept implementation of the proposed mechanism. In Section 4.3 security aspects arisen while designing and/or implementing the scheme are considered. Finally, in Section 4.4 the conclusions on the performed work are drawn.

4.1 Publish/Subscribe-based Anonymity

Anonymity is a security service which allows clients to keep their identity unknown to any participant of a communication system. In publish/subscribe networks it should allow clients to subscribe and/or publish data to topics remaining incognito. Although anonymity is an important security service, it is rarely considered. For example, regarding the MQTT (and MQTT-SN) protocol, while some research has been done on various security aspects, anonymity remains an open issue.

Some solutions exist for so-called pseudo-anonymity. They allow clients to subscribe and/or publish to a topic without indicating their username or any other credentials. However, the messages sent by those clients can be traced back to where they come from, thus revealing communicating entities.

At the time of writing, in the public domain there was no work on providing complete anonymity for communication over MQTT protocol.

In a standard MQTT scenario the protocol itself provides some sort of "light" anonymity: it is only the broker who has information on which client is interested in which topic and who publishes the updates. However, this can work only if (i) the topic names and payload of the exchanged messages are not visible from outside (i.e. they are encrypted) and (ii) as long as and the broker is trusted and not compromised.

While the confidentiality of communication between clients and brokers can be protected by means of TLS in MQTT-based systems and by using DTLS and lightweight cryptography in MQTT-SN networks, reliability of a single broker is a matter of trust and a potential adversary's skills. A possible solution for increasing the level of confidentiality can be implementing a sort of onion routing through a sequence of brokers, i.e. by applying dynamic broker bridging technology to a set of brokers and making them relay encapsulated encrypted messages from a client until the destina-

tion broker.

In this section a description of such anonymity system is provided. In the following examples it is assumed that Elliptic Curve Integrated Encryption Scheme (ECIES) is used for message encryption, and Elliptic-curve Diffie-Hellman (ECDH) key agreement protocol based on static public keys is used for obtaining shared symmetric keys, since these mechanisms use relatively short symmetric keys and faster/lighter computations (more detailed description is given in Section 1.4). Just for the sake of the description of the anonymization scheme and corresponding examples, it is supposed that all data exchanged between the entities are valid and correct, therefore possible failures and mechanisms to detect and process them are not considered.

4.1.1 Subscription

When a client wants to anonymously subscribe to one or more topics on a particular broker, it chooses a set of n–1 brokers ($B_1, B_2, ..., B_{n-1}$) in order to lay through them an anonymity path to the destination broker (B_n). The client then forms a subscription message that iteratively contains its ECDH ephemeral public value (KS_i) and the topic name (the list of topic names in case of subscription to several topics) encrypted with the DH symmetric key (K_i^S) computed using the broker' public ECDH value. After encapsulation of obtained values according to the dynamic broker bridging scheme, described in Section 2.1, the resulting *Topic Name* will be:

$$KS_1 \,||\, \Big\{ ... \; KS_{n\text{-}1} \,||\, \big\{ KS_n \,||\, \{T_x\}_{K_n^S} @\; B_n \big\}_{K_{n-1}^S} ... \; @\; B_2 \Big\}_{K_1^S},$$

where:

- $\{x\}_K$ means encryption of x using the key K;
- T_x is a topic or a list of topics on the destination broker B_n to which the client wants to subscribe;

- K_i^S, $i=1...n$ are symmetric keys that are computed using the broker's public value and the client's private ephemeral value and are used for encrypting topic names and, in case of publication, messages between the client and broker B_i;

- KS_i, $i=1...n$ are the client's public values used for the ECDH symmetric key K_i^S computation.

When a broker receives a subscription request with the topic name corresponding to the pattern $KS_i \mid\mid \{T_i\}_{K_i^S}$, it performs the following actions:

1. Using KS_i as key entry, it performs a lookup in its internal database for the corresponding symmetric key.

2. If no symmetric key has been found, the broker computes the key according to ECDH scheme, using the received client's public value KS_i and its own private value. After the symmetric key K_i^S is obtained, it is stored together with the corresponding public value KS_i. Otherwise the previously stored key is retrieved from the database.

3. The broker decrypts the $\{T_i\}_{K_i^S}$ value using the key K_i^S. After this step, the value T_i is obtained, which is the real topic name destined for the current broker.

4. If T_i value corresponds to the $KS_j \mid\mid T_j$ @ B_j pattern, i.e. has to be forwarded, then the broker does the following:

   4.1. Checks whether B_j is a valid reference to another broker (e.g., contains a valid IP address and port number);

   4.2. Adds to its own database a subscription of the sender to topic T_j on broker B_j;

   4.3. If the broker has not subscribed to the topic T_j at the broker B_j yet, then it must send to the broker B_j a request to subscribe it to the topic T_j. The value KS_j is forwarded as well.

5. If the obtained value does not contain symbol "@" and T_i is a valid topic name, then the broker adds to its subscribers database a subscription of the sender to the obtained topic.

In Figure 4.1 an example of an anonymous multi-broker subscription process is depicted. In the provided example a client (C_1) wants to subscribe to the topic T_x on the broker B_3 and to the topic T_y on the broker B_4.

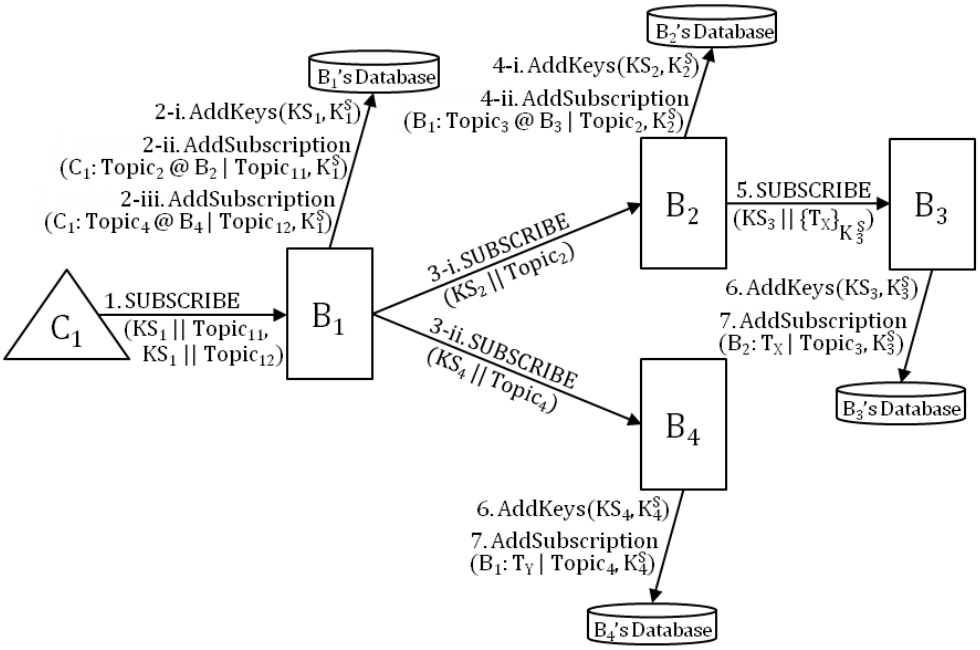

Figure 4.1: An example of subscription process.

The process of subscription will be the following:

1. The client C_1 chooses an anonymity path towards each destination broker, through which it will send the subscription request. In the example in Figure 4.1 the anonymity path goes through brokers B_1 and B_2 to a destination broker B_3 and through broker B_1 to another destination broker B_4. Using symmetric keys calculated based on the brokers' public values, the client encapsu-

lates its own public values, encrypted topic names and brokers' identifiers in such a way that all brokers along the anonymity paths will be aware only about previous (and next, if any) nodes of the path, and only the destination brokers B_3 and B_4 will know the real topic names, to which C_1 is subscribing, however, having no idea who the real subscriber is. Then the client sends to the broker B_1 a SUBSCRIBE request with a list of topic names T_{11} and T_{12}:

$$T_{11} = KS_1||\Big\{KS_2||\big\{KS_3||\{T_x\}_{K_3^S}@\ B_3\big\}_{K_2^S}@\ B_2\Big\}_{K_1^S} = KS_1||Topic_{11},$$

$$T_{12} = KS_1||\big\{KS_4||\{T_y\}_{K_4^S}@\ B_4\big\}_{K_1^S} = KS_1||Topic_{12},$$

which for a potential adversary will look like opaque topic names.

C_1 saves the association between chosen anonymity paths ($B_1 - B_2 - B_3$ and $B_1 - B_4$) and corresponding topic names it used (T_{11} and T_{12} in the provided example).

2. Broker B_1 receives the subscription request from the client C_1, computes symmetric key K_1^S based on the received public value KS_1, and stores both values in the repository of keys. Then it decrypts the received topic name values using the symmetric key KS_1. The obtained values will respectively be:

$$KS_2||\big\{KS_3||\{T_x\}_{K_3^S}@B_3\big\}_{K_2^S}@B_2 = KS_2||Topic_2@B_2,$$

$$KS_4||\big\{T_y\big\}_{K_4^S}@B_4 = KS_4||Topic_4@B_4.$$

Broker B_1 adds to its subscribers database records that the sender (C_1) has subscribed to topics $Topic_2@B_2$ and $Topic_4@B_4$ mapped to $Topic_{11}$ and $Topic_{12}$ respectively, and symmetric key K_1^S which it will use in case of sending a *PUBLISH* packet to the client.

3. Broker B_1 sends SUBSCRIBE packets to the brokers (i) B_2 and (ii) B_4 with topic name values T_2 and T_4 respectively:

$$T_2 = KS_2 || \{KS_3 || \{T_x\}_{K_3^S} @\ B_3\}_{K_2^S} = KS_2 || Topic_2,$$

$$T_4 = KS_4 || \{T_y\}_{K_4^S} = KS_4 || Topic_4.$$

4. When broker B_2 receives the subscription request from B_1, it computes the value of the symmetric key K_2^S based on the received public value KS_2 and stores this pair of values in the database of keys. Then the broker decrypts the topic name string using the symmetric key KS_2. The obtained value will be

$$KS_3 \ ||\ \{T_x\}_{K_3^S} @\ B_3 = KS_3 \ ||\ Topic_3\ @\ B_3.$$

   Broker B_2 adds to its database of subscribers a record that broker B_1 has subscribed to the topic $Topic_3@B_3$ mapped to a topic name $Topic_2$ and the symmetric key K_2^S.

5. Broker B_2 then sends to the broker B_3 (which is the destination one in this path) a SUBSCRIBE packet with the following topic name T_3:

$$T_3 = KS_3 \ ||\ \{T_x\}_{K_3^S},$$

   which for a potential adversary looks like a subscription request to an opaque topic name $Topic_3$.

6. Broker B_3 receives the SUBSCRIBE packet, uses the obtained public key KS_3 for computing a shared symmetric key K_3^S, stores the values of both keys, and decrypts the received topic name string using the symmetric key K_3^S. The obtained value will contain the real topic name T_x.

7. B_3 then adds to its subscribers database a record that broker B_2 has subscribed to the topic T_x mapped to $Topic_3$. The key K_3^S is also stored for this subscription to be used in case of incoming publications in this topic in the future.

8. Similarly, when broker B_4 receives the subscription request forwarded by the broker B_1, it computes a symmetric key K_4^S using the received public value KS_4, stores the keys, and decrypts the topic name using the computed symmetric key K_4^S. The result will contain the real topic name T_y.

9. Broker B_4 adds to its database of subscribers a record that broker B_1 has subscribed to the topic T_y mapped to $Topic_4$, and saves the symmetric key within this record.

4.1.2 Publication

According to the subscription procedure described in Section 4.1.1, a client that anonymously subscribed to topic T_x will receive messages published in this topic remaining unknown. In addition, also the publisher can perform publishing procedure anonymously.

When a client wants to publish a message in a topic on a particular broker in an anonymous way, it can choose a set of m brokers $(B_1, B_2, ..., B_{m\text{-}1})$ forming an anonymity path through them to the desired broker (B_m). The client then creates a PUBLISH request with the following topic name (similarly to the subscription procedure):

$$KP_1 \,||\, \Big\{ ... \; KP_{m\text{-}1} \,||\, \big\{ KP_m \,||\, \{T_x\}_{K_m^P} @ \; B_m \big\}_{K_{m\text{-}1}^P} ... \; @ \; B_2 \Big\}_{K_1^P}$$

and the following message as payload:

$$\Big\{ ... \big\{ \{Msg_x\}_{K_m^P} \big\}_{K_{m-1}^P} ... \Big\}_{K_1^P},$$

where:

- $\{x\}_K$ means encryption of x using the key K;
- T_x is a topic on the destination broker B_m in which the client wants to publish a message;
- Msg_x is the message that the client wants to publish to the topic T_x;

- K_i^P, $i=1...m$ are symmetric keys shared between the client and broker B_i and used for encrypting topic names and messages;
- KP_i, $i=1...m$ are the client's public values used for corresponding symmetric key K_i^P computation.

When a broker B_i receives a publication request from a client or another broker that has topic name value corresponding to the pattern $KP_i \mid\mid \{T_i\}_{K_i^P}$ and application message value $\{Msg_i\}_{K_i^P}$, it has to perform the following steps:

1. Using KP_i as key entry, it performs a lookup for the corresponding symmetric key in the internal database of keys.
2. If no symmetric key has been found, the key is computed according to the ECDH scheme. The obtained key K_i^P is then stored together with the corresponding public value KP_i in the keys database.
3. The broker decrypts received topic name and application message values using the obtained symmetric key K_i^P, thus obtaining topic name T_i and application message Msg_i values.
4. If T_i value corresponds to the $KP_j \mid\mid T_j$ @ B_j pattern, then the broker does the following:
   4.1. Checks whether B_j is a valid reference to another broker.
   4.2. Sends a PUBLISH packet to the broker B_j with topic name value set to T_j and application message set to Msg_i, including the key KP_j value.
5. If the value T_i does not contain "@" symbol, i.e. does not have to be forwarded to another broker, and is a valid topic name string, then the PUBLISH request is actually intended for the current broker. In this case, the broker selects from its subscribers database those entities subscribed to the topic T_i and sends them a PUBLISH packet applying the mapped topic name, if the latter was used for subscription, and encrypting the message with the symmetric key corresponding to the subscription.

In Figure 4.2 an example of publication process is depicted. In the provided example a client (C_2) wants to anonymously publish a message (Msg_x) in a topic T_x on a broker B_3, using brokers B_4 and B_5 as intermediate nodes of the anonymity path. The example assumes that another client (C_1) has already subscribed to the topic T_x on the broker B_3 following the steps from the example in Section 4.1.1.

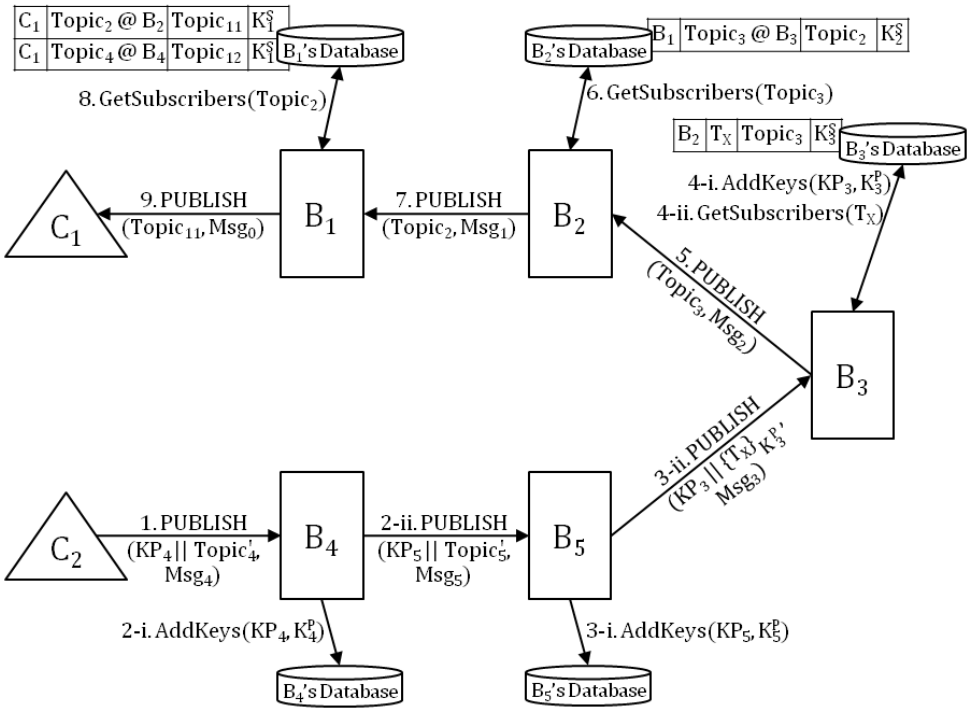

Figure 4.2: An example of anonymous publication.

The following events occur in the depicted example:

1. The publisher C_2 creates an anonymity path through brokers B_4 and B_5 to the destination broker B_3. Using symmetric keys of the three brokers calculated based on their public values, C_2 encapsulates its own public values (KP_3, KP_2, KP_1), encrypted topic name, application message values and next node identifiers in such a way that brokers B_3, B_4 and B_5 will be aware only about previous

(and next, if any) nodes of the anonymity path, and only the destination broker B_3 will know the real topic name and application message, however, without having information about the real publisher.

Then client C_2 sends to the broker B_4 a PUBLISH packet with topic name $T_4^{'}$ and application message Msg_4:

$$T_4^{'}=KP_4 \,||\, \Big\{KP_5 \,||\, \big\{KP_3 \,||\, \{T_x\}_{K_3^P} @\, B_3\big\}_{K_5^P} @\, B_5\Big\}_{K_4^P} = KP_4 \,||\, Topic_4^{'},$$

$$Msg_4=\big\{\{\{Msg_x\}_{K_3^P}\}_{K_5^P}\big\}_{K_4^P}.$$

For a potential adversary the request will look like a request to publish a message Msg_4 in a topic $Topic_4^{'}$.

2. Broker B_4 receives the PUBLISH request, computes symmetric key K_4^P based on the received public value KP_4, and stores both the values. Then it decrypts the topic name and application message values with the obtained symmetric key K_4^P. The values of topic name and message after decryption will respectively be:

$$KP_5||\big\{KP_3||\{T_x\}_{K_3^P}@B_3\big\}_{K_5^P}@B_5 = KP_5||Topic_5^{'}\ @B_5,$$

$$\{\{Msg_x\}_{K_3^P}\}_{K_5^P}.$$

Broker B_4 then forwards the PUBLISH request to the broker B_5 with topic name and application message values respectively:

$$KP_5 \,||\, Topic_5^{'} = KP_5 \,||\, \big\{KP_3 \,||\, \{T_x\}_{K_3^P} @\, B_3\big\}_{K_5^P},$$

$$Msg_5 = \{\{Msg_x\}_{K_3^P}\}_{K_5^P}.$$

3. When broker B_5 receives the request, it computes the value of the symmetric key K_5^P based on the received public value KP_5 and stores this pair of values in its database of keys. Then it decrypts the topic name and application message values of the received request using the symmetric key K_5^P. The obtained values will respectively be:

$$KP_3 \mid\mid \{T_x\}_{K_3^P} @\ B_3,$$

$$\{Msg_x\}_{K_3^P}.$$

Broker B_5 sends to the broker B_3 (which is the destination broker in this scenario) a PUBLISH packet with a topic name $T_3^{'}$ and an application message Msg_3:

$$T_3^{'} = KP_3 \mid\mid \{T_x\}_{K_3^P},$$

$$Msg_3 = \{Msg_x\}_{K_3^P}.$$

4. When B_3 receives the request, it uses the obtained public value KP_3 for computing a symmetric key K_3^P, stores both keys values, and, using the key K_3^P, decrypts the received topic name and application message values. After these operations, B_3 obtains the real topic name T_x and message Msg_x. B_3 then checks for the entities subscribed to the topic T_x and finds a record that broker B_2 has subscribed to the topic T_x mapped to $Topic_3$, and that symmetric key K_3^S must be used for message encryption.

5. B_3 encrypts the original message with the symmetric key K_3^S and sends to the broker B_2 a PUBLISH packet with topic name value set to $Topic_3$ and application message $Msg_2 = \{Msg_x\}_{K_3^S}$.

6. After receiving the publication request, B_2 checks its subscribers database for the entities that subscribed to the topic $Topic_3$ and finds a record that broker B_1 has subscribed to the topic $Topic_3$ mapped to $Topic_2$.

7. B_2 encrypts the received message Msg_2 with the symmetric key K_1^S stored for this subscription record, and sends a PUBLISH packet with topic name set to $Topic_2$ and application message set to $Msg_1 = \{Msg_2\}_{K_2^S}$ to the broker B_1.

8. When broker B_1 receives the request, it checks for the subscribers of the topic $Topic_2$ in its database of subscriptions and finds a record that client C_1 has subscribed to the topic $Topic_2$ mapped to the topic name $Topic_{11}$. It also retrieves the symmetric key K_1^S corresponding to this subscription.

9. Broker B_1 encrypts the received message Msg_1 with the symmetric key K_1^S and sends to the client C_1 a PUBLISH packet with topic name value set to $Topic_{11}$ and an application message $Msg_0 = \{Msg_1\}_{K_1^S}$.

   When C_1 receives the publication request, it associates topic name $Topic_{11}$ with the anonymity path $B_1 - B_2 - B_3$ created for subscription and the real topic name T_x that it has subscribed to. The subscriber consistently applies the symmetric keys shared with the brokers of the anonymity path in order to obtain the original message Msg_x from the received application message Msg_0:

$$Msg_0 = \left\{ \left\{ \{Msg_x\}_{K_3^S} \right\}_{K_2^S} \right\}_{K_1^S}.$$

4.1.3 Unsubscription

When a client wants to unsubscribe from a topic to which it has previously subscribed in an anonymous way, it has to perform the operations similar to those needed for subscription using the same anonymity path $(B_1, B_2, ..., B_{n-1})$ it used to subscribe to the topic that became unwanted, thus the *Topic Name* value for unsubscription has to be the same:

$$KS_1 \,||\, \left\{ ... \; KS_{n\text{-}1} \,||\, \left\{ KS_n \,||\, \{T_x\}_{K_n^S} @ \; B_n \right\}_{K_{n-1}^S} ... \; @ \; B_2 \right\}_{K_1^S},$$

where:

- $\{x\}_K$ means encryption of x using the key K;

- T_x is a topic or a list of topics on the destination broker B_n from which the client wants to unsubscribe;
- K_i^S, $i=1...n$ are symmetric keys that are computed using the broker's public value and the client's private ephemeral value and are used for encrypting topic names;
- KS_i, $i=1...n$ are the client's public values used for the ECDH symmetric key K_i^S computation.

When a broker receives an *UNSUBSCRIBE* request with a topic name corresponding to the pattern $KS_i \mid\mid \{T_i\}_{K_i^S}$, it performs the following actions:

1. Using KS_i as key entry, it performs a lookup in its internal database for the corresponding symmetric key.
2. If no symmetric key has been found, the broker computes the key according to ECDH scheme, using the received client's public value KS_i and its own private value. After the symmetric key K_i^S is obtained, it is stored together with the corresponding public value KS_i. Otherwise the previously stored key is retrieved from the database.
3. The broker decrypts the $\{T_i\}_{K_i^S}$ value using the key K_i^S. After this step, the value T_i is obtained, which is the real topic name destined for the current broker.
4. The broker deletes the record of sender's subscription to the topic T_i from its subscribers database.
5. If T_i value corresponds to the $KS_j \mid\mid T_j$ @ B_j pattern, i.e. the *UNSUBSCRIBE* request has to be forwarded, then the broker does the following:
   5.1. Checks whether B_j is a valid reference to another broker (e.g., contains a valid IP address and port number);
   5.2. Checks whether there are subscribers to the topic T_j on the broker B_j in the database of subscriptions;

5.3. If no record is found, then the current broker must send to the broker B_j a request to unsubscribe it from the topic T_j. The value KS_j is forwarded as well. Otherwise, no further action is needed.

An example of the subscription cancellation process is depicted in Figure 4.3. The proposed example assumes that some subscriptions took place before: the corresponding records are displayed in the tables next to the brokers's databases. In this example, a client C_1 unsubscribes from the topic T_X managed by the broker B_3, on which it has previously subscribed according to the procedure described in Section 4.1.1.

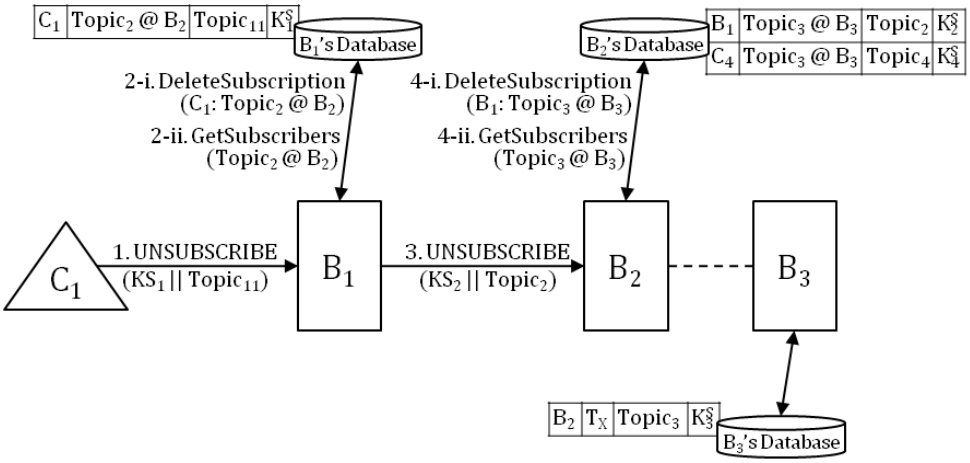

Figure 4.3: An example of subscription process.

The following steps occur during the unsubscription process:

1. The client C_1 selects from its internal storage the anonymity path it used in order to perform subscription to the topic T_X on the broker B_3: $\{B_1, B_2, B_3\}$. The client retrieves previously stored topic name :

$$T_{11} = KS_1||\Big\{KS_2||\big\{KS_3||\{T_x\}_{K_3^S}@\ B_3\big\}_{K_2^S}@\ B_2\Big\}_{K_1^S} = KS_1||Topic_{11}.$$

Client C_1 sends an *UNSUBSCRIBE* request containing topic name T_{11} to the broker B_1.

2. Broker B_1 receives the request, it uses the symmetric key K_1^S corresponding to the received public value KS_1 in order to decrypt the topic name, which will result in

$$KS_2||\left\{KS_3||\{T_x\}_{K_3^S}@B_3\right\}_{K_2^S}@B_2 = KS_2||Topic_2@B_2.$$

Then B_1's actions are the following:

i) it removes C_1's subscription from the database of subscribes;

ii) since the original subscription has been forwarded to another broker (B_2 in this case), B_1 checks for other subscribers of the topic $Topic_2$ on the broker B_2.

3. Since no subscribers to the topic $Topic_2$ on the broker B_2 are left in the B_1' database, broker B_1 sends an *UNSUBSCRIBE* request to the broker B_2 with the topic name value T_2:

$$T_2 = KS_2||\left\{KS_3||\{T_x\}_{K_3^S}@B_3\right\}_{K_2^S} = KS_2||Topic_2.$$

4. When broker B_2 receives the *UNSUBSCRIBE* request from B_1, it retrieves the symmetric key K_2^S corresponding to the received public value KS_2 and decrypts the received topic name. The obtained value will be

$$KS_3 \;||\; \{T_x\}_{K_3^S}@\; B_3 = KS_3 \;||\; Topic_3 \;@\; B_3.$$

Then B_2 performs the following steps:

i) it removes B_1's subscription to the topic $Topic_3$ @ B_3 from the database of subscribes;

ii) it checks for other subscribers of the topic $Topic_3$ on the broker B_3.

Since in the database there is another subscription the topic $Topic_3$ on B_3 (by some client C_4), the current broker B_2 does not have to forward the unsubscription request and can send an acknowledgement that the subscription has been removed ($UNSUBACK$) to the broker B_1.

4.2 Implementation

The proposed anonymization mechanism has been implemented as an extension of the dynamic broker bridging implementation described is Section 2.3. A library of necessary functions (e.g., cryptographic data processing) has been implemented in order to provide the communication via dynamic bridging mechanism with anonymization features. The original database and functions to work with it have been modified in order to store new data, such as key associations and new subscriptions data (e.g., corresponding shared symmetric keys). The support of data encryption is based on Elliptic Curve cryptography: *coincurve*[1] library is utilized to perform operations on curve *secp256k1*, and AES implementation of *PyCryptodome*[2] is used for symmetric encryption of the data.

In current implementation, in order to avoid memory overload of a client, brokers' public values are obtained before forming a request, by subscribing to a predefined topic (*get_public_key*). The broker's behaviour has been modified in the following way. Before a broker starts, a pair of private and public *ECDH* values is generated. The public value is then added as a retain message to (*get_public_key*) topic. When a broker receives a *SUBSCRIBE* request for the topic *get_public_key*, it replies with a *PUBLISH* packet containing its public value. The attempts of clients to publish in this topic are discarded.

In Figure 4.4 a flowchart of client's actions performed in order to anonymously subscribe to a topic is demonstrated. In the provided scheme, the topic name is obtained from the method $getTopicName()$ which may read the string value from the command line, or the topic name value may be hard-coded in the application, or

[1] https://pypi.org/project/coincurve/

[2] https://pypi.org/project/pycryptodome/

another approach can be applied.

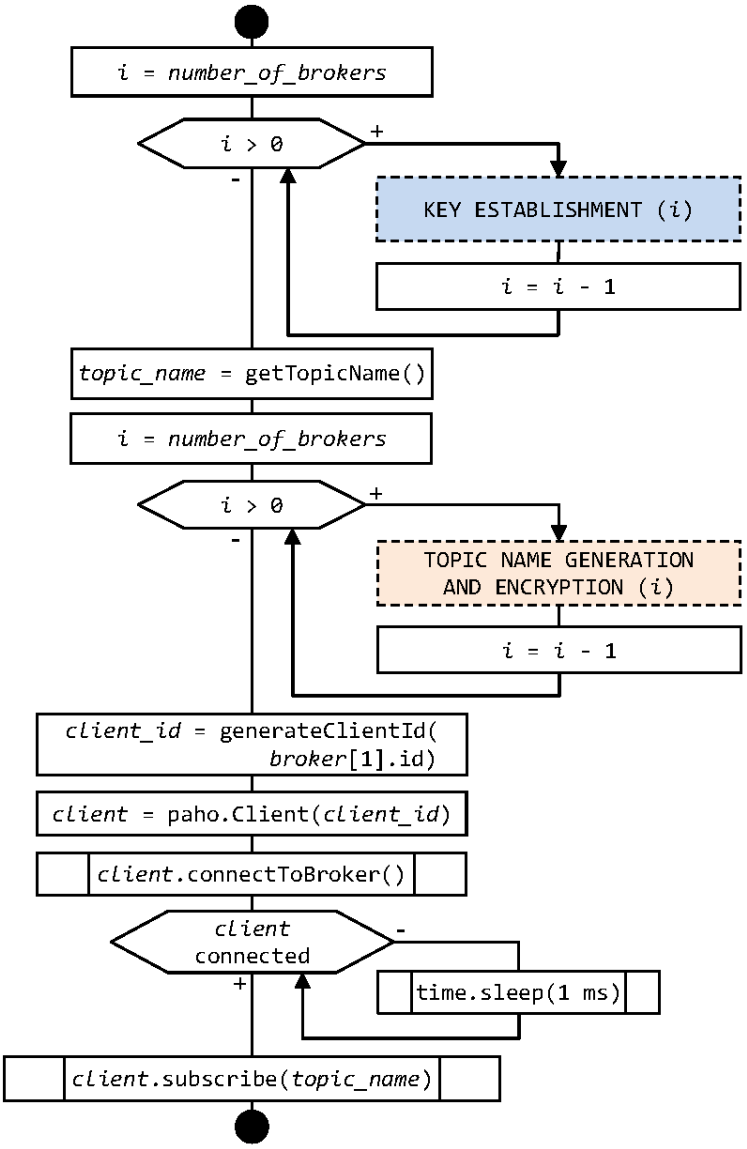

Figure 4.4: Anonymous subscription.

As the first step, the client has to obtain the public keys of each broker of the

anonymity path and to generate corresponding symmetric keys which will then be shared. The flowchart of the key establishment procedure is depicted in Figure 4.5.

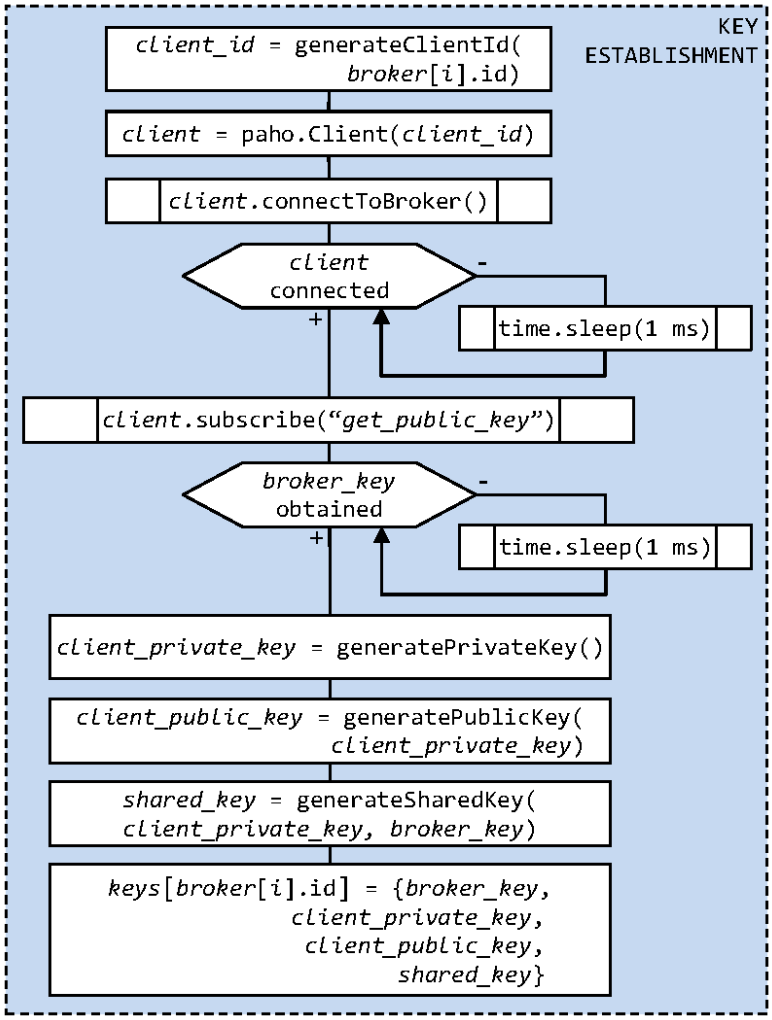

Figure 4.5: Client's key establishment.

According to the provided flowchart, the client generates a new id in order to communicate with each broker. The public key of a broker is obtained by subscrib-

ing to a predefined topic "*get_public_key*", in which a broker stores a retain message containing its public key. Once the broker's public value is received, the client generates a pair of private and public ECDH values (keys). Then, using its own private key and the broker's public key, the client computes a symmetric key, which will be then used for encryption and decryption of the data shared between the client and the broker. All the keys are then saved in the keys storage.

After successfully obtaining all the necessary keys, the client has to encapsulate its public keys, the topic name value and the brokers' identifiers, consequently (starting from the destination broker and going backwards to the first broker of the anonymity path) encrypting all the data with corresponding shared keys. The flowchart of the topic generation procedure is provided in Figure 4.6.

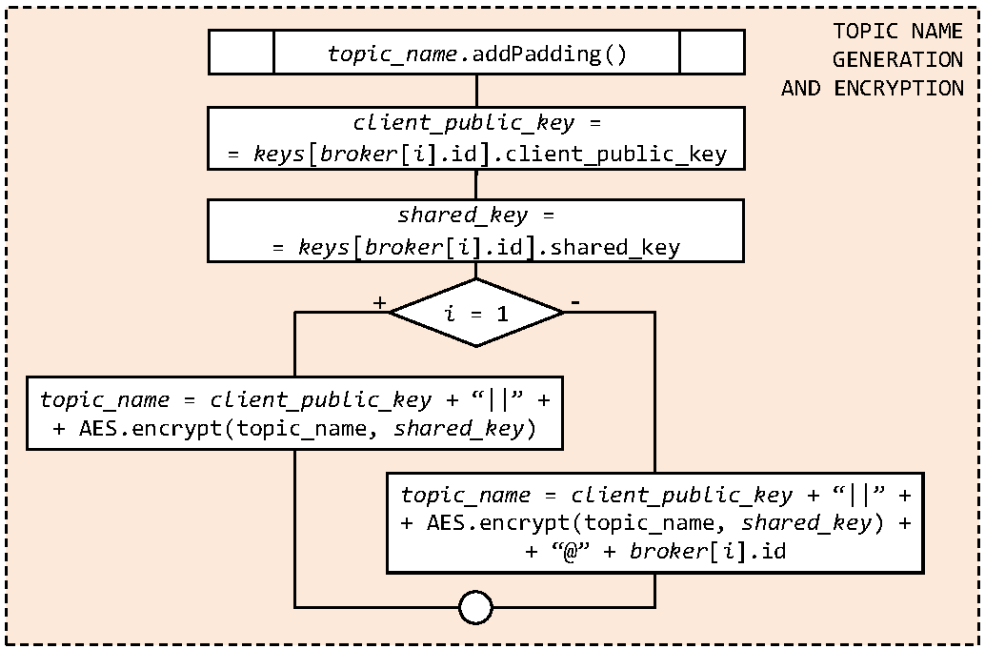

Figure 4.6: Topic name generation.

Before encrypting the topic name, the value has to be padded in order to apply AES encryption. In the provided implementation, PKCS#7 [44] padding method is

used, according to which the value of each byte of the padding is equal to the number of added bytes, i.e. to the padding length. After performing padding operations, the topic name is encrypted using AES symmetric encryption in Cipher Block Chaining (CBC) mode. After the encryption, the client concatenates the public key generated for the broker with the encrypted topic name and, if the broker is not the one, to which client will send the *SUBSCRIBE* request, i.e. not the first one on the way, then the client also concatenates the obtained value with the broker's identifier.

When the topic name is generated, the client connects to the first broker of the anonymity path and sends to it a *SUBSCRIBE* message with the encrypted topic name.

The client's actions necessary for performing an *UNSUBSCRIBE* request are similar to those for subscription, with the exception that it does not need to obtain the brokers' public keys and generate a new topic name. Instead, the same *Topic name* value used for the subscription must be used in order to unsubscribe from the topic.

When a client wants to publish a message in a topic in an anonymous way, it performs the same operations needed for subscription, in order to establish keys and generate a topic name. The *Application message* has to be encrypted as well. However, no additional value has to be concatenated to the encrypted message during the encapsulation process.

When an anonymity-aware broker receives a *SUBSCRIBE*, *UNSUBSCRIBE* or *PUBLISH* request, it processes the received *Topic name* value according to the flowchart presented in Figure 4.7, where *topic_name* is the topic name value received from the sender.

After performing the decryption operations, the topic is processed with accordance to the mechanism described in Section 2.3, with the exception that the *encrypted_topic_name* and *shared_key* values are stored in the subscriptions database as well. In case of an incoming *PUBLISH* request, the broker has also to decrypt the *Application message* value using the same *shared_key* it used for the topic name decryption.

If a *PUBLISH* message arrives to a bridge running on the current broker, i.e. the broker connected to the bridge has data to publish in the topic to which the bridge has

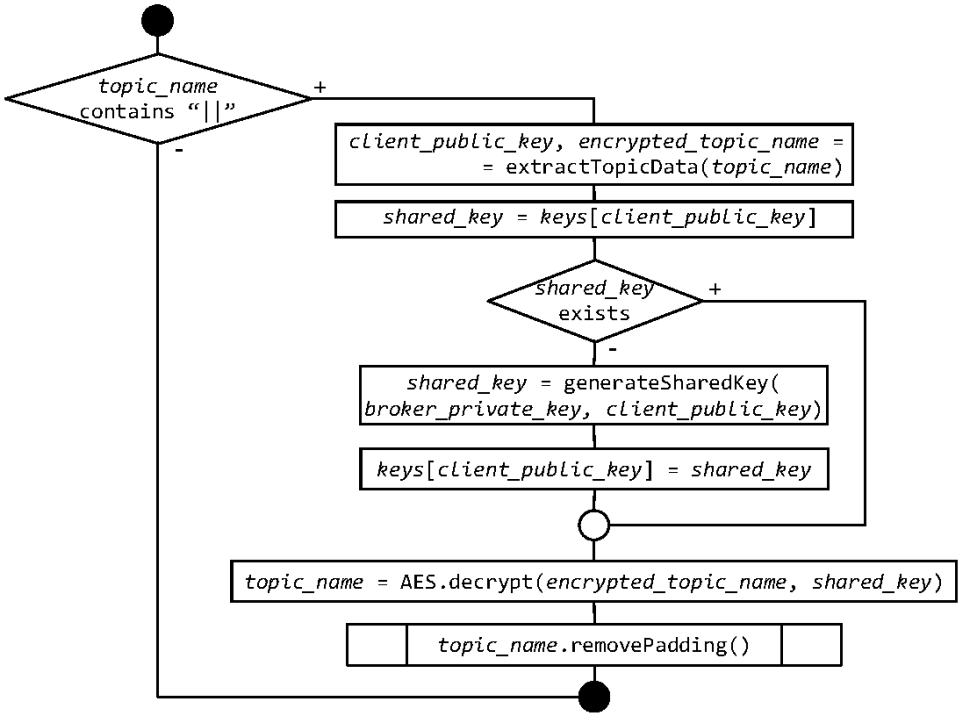

Figure 4.7: Topic name processing.

subscribed, the bridge performs the actions depicted in the flowchart of Figure 4.8. In the provided flowchart, the *topic_name* value is the received *Topic name*, the *message* contains the *Application message* received withing the request, and *broker_id* is the identifier of the broker that has sent the *PUBLISH* request to the bridge.

As the first step, the bridge performs a look-up in its subscribers database for the subscriptions to the topic *topic_name*. If one or more records are found, the list is saved in the variable *subscriptions*. Then the bridge generates a new topic name *topic_name* which will be further used in order to distribute the message to the subscribers. After that, for each record in *subscriptions* list the bridge performs the following actions:

1. It checks for the presence of a *shared_key* in the subscription data.

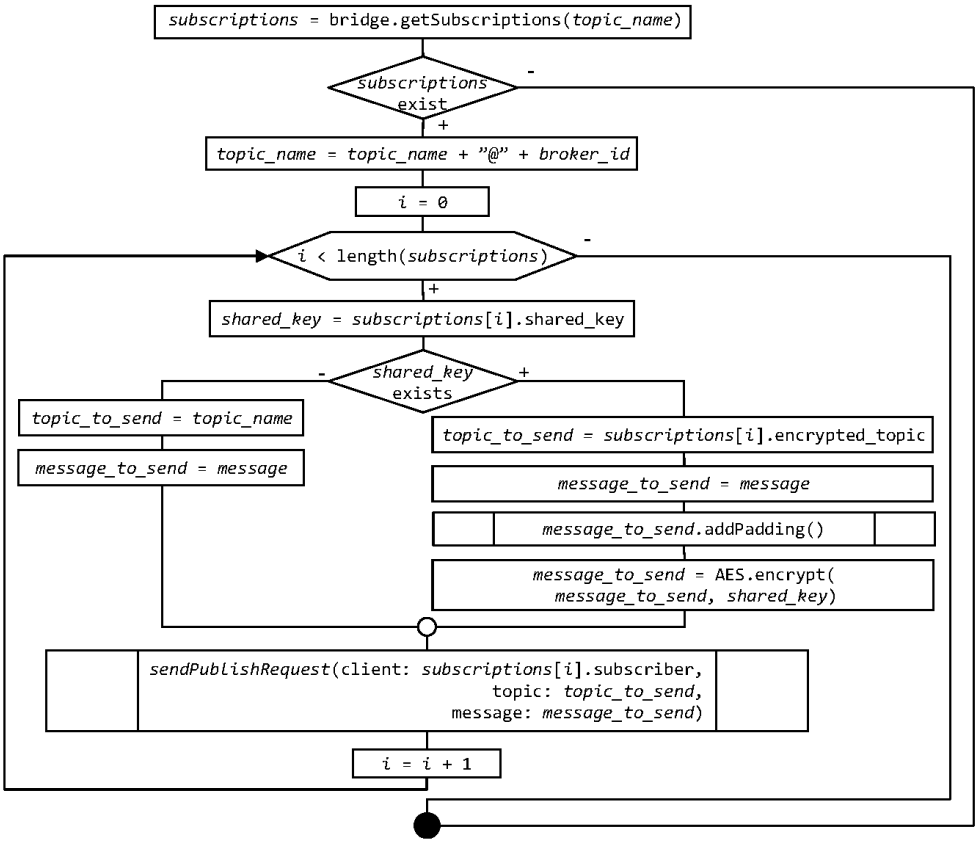

Figure 4.8: Publication request processing.

2. If the shared key is not present, i.e. the subscription request was not anonymous, the bridge sends a *PUBLISH* message to the client, using the *topic_name* value as *Topic name* and the received *message* as *Application message* payload.

3. Otherwise, if the shared key exists, the bridge's steps are the following:

   i) retrieve the encrypted topic name value which the subscriber used when sending a *SUBSCRIBE* request;

   ii) encrypt the *message* value using the *shared_key*;

iii) send a *PUBLISH* message to the subscriber, using the encrypted topic name and the encrypted message as *Topic name* and *Application message* values respectively.

The implementation has been tested in Windows 10 and Raspbian Stretch environments: diverse combinations of brokers and clients instances were run and exchanged messages. The main goal of the tests was to validate the correctness of the proposed architecture and to discover which possible issues can arise. The tests have demonstrated feasibility of the proposed solution and revealed the aspects, listed in Section 4.3, which should be taken into consideration for the future improvement.

4.3 Security Aspects

In order to cover up communication between the entities and to prevent a potential adversary from analyzing the traffic within the system, flushing anonymity technique [45] may be used while sending packets. One way of flushing can be defining a parameter N and sending packets only after N messages are collected in a broker's buffer [46]. Another option is to set a timer and periodically send dummy packets to each or several random connected nodes [47]. Other standard flushing techniques already defined for mix anonymity networks can also be used.

Another measure of preventing possible attacks based on traffic analysis is to ensure that all outgoing messages have the same size. This can be achieved by using padding after encryption of topic names and messages. Brokers in the path can pad the messages to a fixed length before forwarding them. In this case, the receiving party has to be aware of how to remove the padding in an unambiguous manner before decrypting the received data, so the communicating parties must use the same padding algorithm.

Since in MQTT-SN the topic name value is limited with length, it is assumed that the topic name registration procedure has been already performed prior to sending a SUBSCRIBE, UNSUBSCRIBE or PUBLISH packet, with accordance to the standard documentation. Although this procedure is not standard for the MQTT protocol,

it can be applied in MQTT-based system as well, in order to decrease the size of exchanged data. The descriptions of the standard topic name registration procedure is provided in Section 1.1.2.

It is essential that the clients correctly obtain public values of the brokers. One possibility is by using entity identifiers that contain the entity's public value. Another approach, which has been used in the proof-of-concept implementation described in Section 4.2, is to store brokers' public values as retain messages in a specified topic. Once a client needs to obtain the value, it has to send a *SUBSCRIBE* request indicating that topic name, and the broker will respond with a *PUBLISH* message containing its public value. Clearly, any attempt to publish in that topic must be rejected.

In the proposed anonymization scheme it is the client who sets up the anonymity path. It might be possible to reduce the load of the client by making the brokers select next hop of the path (e.g., introducing a counter of desirable length of the path, which would decrease hop-by-hop). However, in such scenario, if some brokers collude, they can create the path through themselves, thus revealing in the end the real values of the "Client – Broker" pair.

4.4 Conclusions

In this chapter a novel scheme for secure anonymous communication within networks based on MQTT protocol has been presented. The solution uses dynamic broker bridging mechanism described in Section 2.1. The proposed mechanism provides anonymity for MQTT clients by creating an anonymity path through a set of brokers and encapsulating and encrypting requests in such a way that none of the participants of communication knows which client is really subscribed/publishes to which topic and on which broker. The presented solution is simple and light in terms of interactions and state information. The proposed anonymization mechanism has been implemented as an extension of the dynamic broker bridging scheme presented in Section 2.3 and tested in order to prove its feasibility, analyze possible issues and find out the impact of anonymization features on the performance of the intercommunicating entities.

Conclusions

Nowadays, Internet of Things is applied to almost every sphere of life. The IoT devices are interconnected in order to collect, process and transfer various data. Due to the sensitivity of those data, providing IoT communications with security services is an essential task. However, it is usually difficult to design and implement reliable and strong security solutions for the IoT due to its heterogeneous nature and the constraints of IoT devices.

This thesis has focused on the study of security mechanisms for the Internet of Things environment. The IoT and its most popular protocols have been described. An overview of the most required security services, such as end-to-end authentication and authorization, data confidentiality, and anonymity, together with the solutions that have been proposed with the aim of providing those services, has been done.

Design of a novel dynamic broker bridging mechanism has been presented. It is based on the standard broker bridging concept, but without requiring a manual set-up of each broker. This feature makes it able to be dynamically adapted to various scalability requirements. Three authorization and authentication schemes have been designed for the proposed mechanisms. In all of them, the requests directed to brokers contain authorization tokens released by a trusted third-party authorization server, so only authorized entities can subscribe or publish to a particular topic. The proposed dynamic broker bridging mechanism has been implemented and tested in order to prove the feasibility of the proposed architecture. The proposed scheme has been applied to the Industrial IoT scenario with the aim of relaying M2M communications, based on publish/subscribe paradigm.

The work has also focused on providing IoT communications with anonymity. A novel anonymization protocol specifically designed for the IoT M2M communications has been described. The proposed protocol is based on the onion routing concept, however, unlike the other known OR-based schemes, it is completely datagram-oriented. Two modes for anonymity path (a sequence of IORs used to anonymously forward a datagram) setup have been designed: (i) two-way anonymity path mode in which communicating nodes use the same sequence of IORs in order to send datagrams; and (ii) one-way mode in which a new anonymity path is built when the responding node needs to send data. An important feature of the provided mechanism is the possibility to choose a different anonymity path for each packet, thus significantly increasing the overall level of anonymity. The proposed anonymization protocol has been implemented and tested in order to demonstrate its feasibility. The solution can be considered lightweight, due to relatively small cryptographic and protocol overhead.

A novel scheme for secure anonymous communication within networks based on publish/subscribe paradigm (e.g., MQTT protocol) has been described. The solution exploits the dynamic broker bridging mechanism. The anonymization mechanism provides anonymity for MQTT clients by creating an anonymity path through a set of brokers and encapsulating and encrypting requests in such a way that none of the participants of communication knows which client is really subscribed/publishes to which topic and on which broker. The proposed mechanism has been implemented as an extension of the dynamic broker bridging scheme implementation and tested in order to prove its feasibility. The anonymization mechanism has been analyzed in terms of possible issues and the impact of anonymization features on the performance of the intercommunicating entities. The proposed solution is simple and light in terms of interactions and state information.

The contribution of this work in the research is introducing novel approaches for providing security services for the Internet of Things systems. A novel dynamic broker bridging architecture has been proposed, which can be applied to various scenarios in the future. At the time of writing, there was no technique for providing publish/subscribe systems with the ability to communicate anonymously, and this

thesis proposes a reliable anonymization mechanism. An anonymization solution for datagram-oriented communications has been proposed in order to provide a decentralized system and to reduce the overhead of well-known anonymity solutions.

Bibliography

[1] Ovidiu Vermesan and Peter Friess, editors. *Internet of Things: Converging Technologies for Smart Environments and Integrated Ecosystems*. River Publishers Series in Communication. River, 2013. URL: http://www.internet-of-things-research.eu/pdf/Converging_Technologies_for_Smart_Environments_and_Integrated_Ecosystems_IERC_Book_Open_Access_2013.pdf.

[2] Roy Fielding and Julian Reschke. Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content. RFC 7231, June 2014. URL: https://rfc-editor.org/rfc/rfc7231.txt, doi:10.17487/RFC7231.

[3] LoRa Alliance Technical Committee. July, 2018. Protocol Specification. Version 1.0.3. *LoRaWAN 1.0.3 Specification*.

[4] Zach Shelby, Klaus Hartke, and Carsten Bormann. The Constrained Application Protocol (CoAP). RFC 7252, June 2014. URL: https://rfc-editor.org/rfc/rfc7252.txt, doi:10.17487/RFC7252.

[5] A. Banks et al. MQTT Version 5.0. Standard, OASIS, 2019.

[6] Ian Fette and Alexey Melnikov. The WebSocket Protocol. RFC 6455, December 2011. URL: https://rfc-editor.org/rfc/rfc6455.txt, doi:10.17487/RFC6455.

[7] Roy Thomas Fielding. *REST: Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California,

Irvine, 2000. URL: http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm.

[8] Eric Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446, August 2018. URL: https://rfc-editor.org/rfc/rfc8446.txt, doi:10.17487/RFC8446.

[9] Andy Stanford-Clark and Hong Linh Truong. November 14, 2013. Protocol Specification. Version 1.2. *MQTT For Sensor Networks (MQTT-SN)*.

[10] Eric Rescorla and Nagendra Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, January 2012. URL: https://rfc-editor.org/rfc/rfc6347.txt, doi:10.17487/RFC6347.

[11] Giederson Santos, Vinicius Guimaraes, Guilherme Rodrigues, Lisandro Granville, and Liane Tarouco. A DTLS-based security architecture for the Internet of Things. pages 809–815, 07 2015. doi:10.1109/ISCC.2015.7405613.

[12] Simone Cirani, Marco Picone, Pietro Gonizzi, Luca Veltri, and Gianluigi Ferrari. IoT-OAS: An OAuth-Based Authorization Service Architecture for Secure Services in IoT Scenarios. *Sensors Journal, IEEE*, 15:1224–1234, 02 2015. doi:10.1109/JSEN.2014.2361406.

[13] Ola Salman, Sarah Abdallah, Imad Elhajj, Ali Chehab, and Ayman Kayssi. Identity-based authentication scheme for the Internet of Things. pages 1109–1111, 06 2016. doi:10.1109/ISCC.2016.7543884.

[14] T. Claeys, F. Rousseau, and B. Tourancheau. Securing complex iot platforms with token based access control and authenticated key establishment. In *2017 International Workshop on Secure Internet of Things (SIoT)*, pages 1–9, Sep. 2017. doi:10.1109/SIoT.2017.00006.

[15] P. Musale, D. Baek, and B. J. Choi. Lightweight gait based authentication technique for iot using subconscious level activities. In *2018 IEEE 4th World*

Forum on Internet of Things (WF-IoT), pages 564–567, Feb 2018. doi:10.1109/WF-IoT.2018.8355210.

[16] P. Gope and B. Sikdar. Lightweight and privacy-preserving two-factor authentication scheme for iot devices. *IEEE Internet of Things Journal*, 6(1):580–589, Feb 2019. doi:10.1109/JIOT.2018.2846299.

[17] M. A. Gurabi, O. Alfandi, A. Bochem, and D. Hogrefe. Hardware based two-factor user authentication for the internet of things. In *2018 14th International Wireless Communications Mobile Computing Conference (IWCMC)*, pages 1081–1086, June 2018. doi:10.1109/IWCMC.2018.8450397.

[18] Ludwig Seitz, Goran Selander, Erik Wahlstroem, Samuel Erdtman, and Hannes Tschofenig. Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth). Internet-Draft draft-ietf-ace-oauth-authz-24, Internet Engineering Task Force, March 2019. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-ietf-ace-oauth-authz-24.

[19] Stefanie Gerdes, Olaf Bergmann, Carsten Bormann, Goran Selander, and Ludwig Seitz. Datagram Transport Layer Security (DTLS) Profile for Authentication and Authorization for Constrained Environments (ACE). Internet-Draft draft-ietf-ace-dtls-authorize-08, Internet Engineering Task Force, April 2019. Work in Progress. URL: https://datatracker.ietf.org/doc/html/draft-ietf-ace-dtls-authorize-08.

[20] Cigdem Sengul, Anthony Kirby, and Paul Fremantle. MQTT-TLS profile of ACE. Internet-Draft draft-ietf-ace-mqtt-tls-profile-00, Internet Engineering Task Force, May 2019. Work in Progress.

[21] Michael B. Jones, John Bradley, and Nat Sakimura. JSON Web Token (JWT). RFC 7519, May 2015. URL: https://rfc-editor.org/rfc/rfc7519.txt, doi:10.17487/RFC7519.

[22] Michael B. Jones, Erik Wahlstroem, Samuel Erdtman, and Hannes Tschofenig. CBOR Web Token (CWT). RFC 8392, May 2018. URL: https://rfc-editor.org/rfc/rfc8392.txt, doi:10.17487/RFC8392.

[23] Marco Calabretta, Riccardo Pecori, Massimo Vecchio, and Luca Veltri. MQTT-Auth: a Token-based Solution to Endow MQTT with Authentication and Authorization Capabilities. *Journal of Communications Software and Systems*, 14, 01 2018. doi:10.24138/jcomss.v14i4.604.

[24] M. Singh, M. A. Rajan, V. L. Shivraj, and P. Balamuralidhar. Secure MQTT for Internet of Things (IoT). In *2015 Fifth International Conference on Communication Systems and Network Technologies*, pages 746–751, April 2015. doi:10.1109/CSNT.2015.16.

[25] A. A. Wardana and R. S. Perdana. Access Control on Internet of Things based on Publish/Subscribe using Authentication Server and Secure Protocol. In *2018 10th International Conference on Information Technology and Electrical Engineering (ICITEE)*, pages 118–123, July 2018. doi:10.1109/ICITEED.2018.8534855.

[26] Kristian Beckers. *Pattern and Security Requirements: Engineering-Based Establishment of Security Standards*. Springer Publishing Company, Incorporated, 2015.

[27] W. Diffie and M. Hellman. New Directions in Cryptography. *IEEE Trans. Inf. Theor.*, 22(6):644–654, September 2006. URL: http://dx.doi.org/10.1109/TIT.1976.1055638, doi:10.1109/TIT.1976.1055638.

[28] Certicom Research. Standards for efficient cryptography, SEC 1: Elliptic Curve Cryptography, September 2000. Version 1.0.

[29] S. Jebri, M. Abid, and A. Bouallegue. STAC-protocol: Secure and Trust Anonymous Communication protocol for IoT. In *2017 13th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 365–370, June 2017. doi:10.1109/IWCMC.2017.7986314.

[30] Dijiang Huang. Pseudonym-based cryptography for anonymous communications in mobile ad hoc networks. *International Journal of Security and Networks*, 2(3-4):272–283, 2007.

[31] Taeho Lee, Christos Pappas, Pawel Szalachowski, and Adrian Perrig. Communication based on per-packet one-time addresses. In *Proceedings of the IEEE Conference on Network Protocols (ICNP)*, November 2016. URL: `/publications/papers/icnp16_ota.pdf`.

[32] A. I. Kouachi, S. Sahraoui, and A. Bachir. Per Packet Flow Anonymization in 6LoWPAN IoT Networks. In *2018 6th International Conference on Wireless Networks and Mobile Communications (WINCOM)*, pages 1–7, Oct 2018. `doi:10.1109/WINCOM.2018.8629719`.

[33] N. P. Hoang and D. Pishva. A TOR-based anonymous communication approach to secure smart home appliances. In *2015 17th International Conference on Advanced Communication Technology (ICACT)*, pages 517–525, July 2015. `doi:10.1109/ICACT.2015.7224918`.

[34] Ye Zhu, Xinwen Fu, Bryan Graham, Riccardo Bettati, and Wei Zhao. On flow correlation attacks and countermeasures in mix networks. In David Martin and Andrei Serjantov, editors, *Privacy Enhancing Technologies*, pages 207–225, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[35] Vitaly Shmatikov and Ming-Hsiu Wang. Timing analysis in low-latency mix networks: Attacks and defenses. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *Computer Security – ESORICS 2006*, pages 18–33, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[36] S. Janbabaei, H. Gharaee, and N. Mohammadzadeh. Lightweight, anonymous and mutual authentication in IoT infrastructure. In *2016 8th International Symposium on Telecommunications (IST)*, pages 162–166, Sep. 2016. `doi:10.1109/ISTEL.2016.7881802`.

[37] M. G. Reed, P. F. Syverson, and D. M. Goldschlag. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*, 16(4):482–494, May 1998. doi:10.1109/49.668972.

[38] Andrew Banks and Rahul Gupta. MQTT Version 3.1.1. Standard, OASIS, 2014.

[39] H. Boyes et al. The industrial internet of things (IIoT): An analysis framework. *Computers in Industry*, 101:1 – 12, 2018.

[40] K. Iwanicki. A Distributed Systems Perspective on Industrial IoT. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, pages 1164–1170, July 2018.

[41] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, pages 21–21, Berkeley, CA, USA, 2004. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251375.1251396.

[42] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Commun. ACM*, 42(2):39–41, February 1999. URL: http://doi.acm.org/10.1145/293411.293443, doi:10.1145/293411.293443.

[43] L. Belli, S. Cirani, L. Davoli, A. Gorrieri, M. Mancin, M. Picone, and G. Ferrari. Design and Deployment of an IoT Application-Oriented Testbed. *Computer*, 48(9):32–40, Sep. 2015. doi:10.1109/MC.2015.253.

[44] R. Housley. Cryptographic Message Syntax (CMS). RFC 5652 (Draft Standard), September 2009. URL: http://www.ietf.org/rfc/rfc5652.txt.

[45] Matthew Edman and Bülent Yener. On anonymity in an electronic society: A survey of anonymous communication systems. *ACM Computing Surveys*, 42(1):1–35, 2009. doi:http://doi.acm.org/10.1145/1592451.1592456.

[46] David L. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, February 1981. URL: http://dx.doi.org/10.1145/358549.358563, doi:10.1145/358549.358563.

[47] Andrei Serjantov, Roger Dingledine, and Paul F. Syverson. From a trickle to a flood: Active attacks on several mix types. In Fabien A. P. Petitcolas, editor, *Information Hiding*, volume 2578 of *Lecture Notes in Computer Science*, pages 36–52. Springer, 2002. URL: http://dblp.uni-trier.de/db/conf/ih/ih2002.html#SerjantovDS02.