Expertise
and insight
for the future

Hans-Christer Holmberg

NAT Traversal for Constrained IoT

Metropolia University of Applied Sciences

Master of Engineering

Information Technology

Thesis

25 October 2019

Author(s) Title Number of Pages Date	Hans-Christer Holmberg NAT Traversal for Constrained IoT 59 pages 28 October 2019
Degree	Master of Engineering
Degree Programme	Information Technology
Specialisation option	
Instructor(s)	Ari Keränen, Expert Ville Jääskeläinen, Head of Master's program in IT

The number of Internet of Things (IoT) devices is rapidly increasing, and it is predicted that the number of deployed IoT devices is going to be counted in tens or hundreds of billions in the years to come. A large amount of these devices is constrained in different ways, e.g., regarding battery power, CPU power, storage, accessibility or connectivity. The connectivity infrastructure might also be constrained, unreliable and limited in bandwidth. Similarly to many other devices connected to the Internet, many of these devices are deployed behind Network Address Translator (NAT) devices, which restrict access to the devices from other devices, e.g., management servers or other IoT devices.

Due to the constraints listed above, it has been realized that traditional NAT traversal mechanisms might not always be feasible for IoT devices and networks, as the mechanisms often require constant connectivity, or support of NAT traversal specific protocols in addition to the protocols used by the devices to perform their functionality. Due to that reason, problems related to NAT traversal have occurred in constrained IoT deployments.

The scope of the thesis was to provide NAT traversal mechanisms for IoT devices using the Constrained Application Protocol (CoAP). However, many of the solutions described in the thesis can be used with, or applied to, other IoT protocols. Part of the study was also to modify a generic NAT traversal mechanism to use only the CoAP protocol and infrastructure. The study was performed by studying the different protocols, and extensions to those protocols, that are used by CoAP, and see whether they can be used for NAT traversal purpose.

The study defines a novel mechanism for collecting and exchanging information required for effective NAT traversal in constrained environments, and how existing IoT standards can be extended in order to realize the mechanism.

Keywords	IoT, NAT, CoAP, ICE, constrained

Contents

Abstract

List of Abbreviations

1 Introduction 1

2 IoT Communication Challenges and Characteristics 4

2.1 Reachability 4

2.2 Identification 5

2.3 Machine-to-Machine Communication 8

2.4 IoT Characteristics and Traffic Patterns 8

2.4.1 Application Session Duration 8

2.4.2 Application Session Traffic Frequency 11

2.4.3 Number of IoT Endpoints 11

2.4.4 Gateways 11

3 IoT Communication Protocols 13

3.1 Transport Layer Security (TLS) 13

3.2 Datagram Transport Layer Security (DTLS) 15

3.3 Constrained Application Protocol (CoAP) 16

3.3.1 Resources and Representations 16

3.3.2 Transport 17

3.3.3 Security 18

3.3.4 Message Structure 19

3.3.5 Request/Response 23

3.3.6 Methods 23

3.3.7 Response Codes 24

3.3.8 Payload 24

3.3.9 Resource Observation 25

3.3.10 Resource Directory 27

3.4 Interactive Connectivity Establishment (ICE) 29

3.4.1 Collecting ICE Candidates 29

3.4.2 Exchange Candidates 30

3.4.3 Connectivity Check 31

3.4.4 Nominate Candidate 31

3.4.5 Keepalives 32

4 Solutions 33

4.1 Prevent NAT Binding Timeouts 33
4.1.1 Device Controlled NAT 33
4.1.2 KaaS (Keepalive as a Service) 36
4.2 Maintain Identity 39
4.2.1 TLS Session Resumption 41
4.2.2 Session Identifier 41
4.2.3 Session Ticket 42
4.2.4 TLS PSK Session Resumption 43
4.2.5 DTLS Connection ID 44
4.3 Thin-ICE 47
4.3.1 T-STUN 47
4.3.2 Candidate Distribution 49
4.3.3 Connectivity Checks 54
4.3.4 Keepalives 55
4.3.5 Example 56

5 Conclusions 58

List of Abbreviations

ACK	Acknowledgement
CoAP	Constrained Application Protocol
CID	Connection ID
CON	Confirmable
CoRE	Constrained RESTful Environments
DTLS	Datagram Transport Layer Security
HTTP	Hypertext Transfer Protocol
ICE	Interactive Connectivity Establishment
IoT	Internet of Things
IP	Internet Protocol
IRTF	Internet Research Task Force
JSON	JavaScript Object Notation
KaaS	Keepalive as a Service
M2M	Machine-to-Machine
NAT	Network Address Translator
NON	Non-confirmable
OMA	Open Mobile Alliance
OSCORE	Object Security for Constrained RESTful Environments
PCP	Port Control Protocol
PSK	Pre-Shared Key
RD	Resource Directory
RST	Reset
SDP	Session Description Protocol
SMS	Short Message Service
SRTP	Secure Real-time Transport Protocol
STUN	Session Traversal Utilities for NAT
TCP	Transmission Control Protocol
TLS	Transport Layer Security
TTL	Time-To-Live
TURN	Traversal Using Relays around NAT
UDP	User Datagram Protocol
URI	Unified Resource Identifier

1 Introduction

Network Address Translators (NATs) are devices used to map one IP address space into another. The most typical use-case is to map between a private address space and a public address spaces, but NATs are also used to map between different IP versions, e.g., between IPv4 and IPv6. A network that uses a private IP address space is referred to as a private network. A network that uses a public IP address space (i.e., the public Internet) is referred to as a public network. [1, 2, 3].

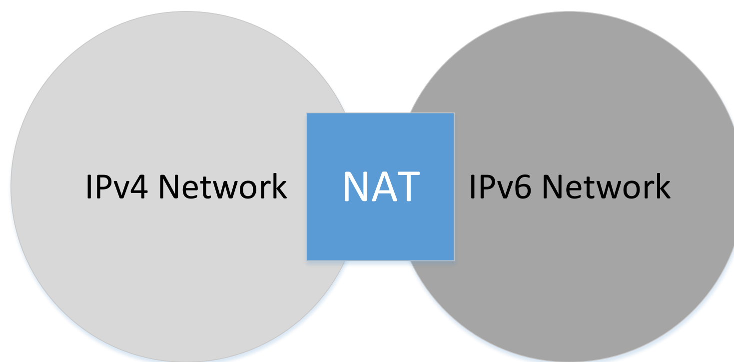


Figure 1: NAT between IPv4 and IPv6 networks.

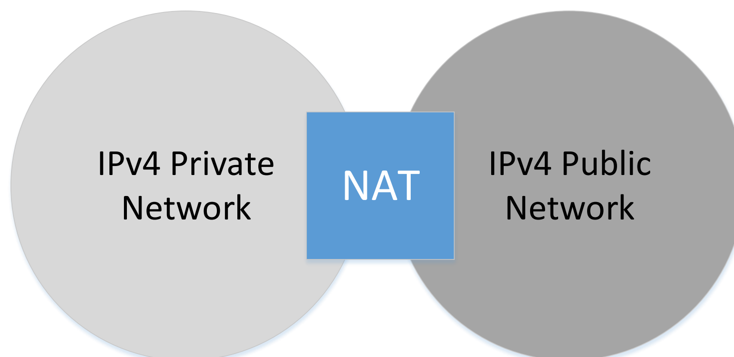


Figure 2: NAT between private network and public network.

When a device is located in a private network and has been assigned a private IP address and port, devices outside of that private network cannot use that IP address and port to route traffic to the device. Also, the same private IP address and port might be used in another private network, as private addresses and ports are only guaranteed to be unique within a given private network.

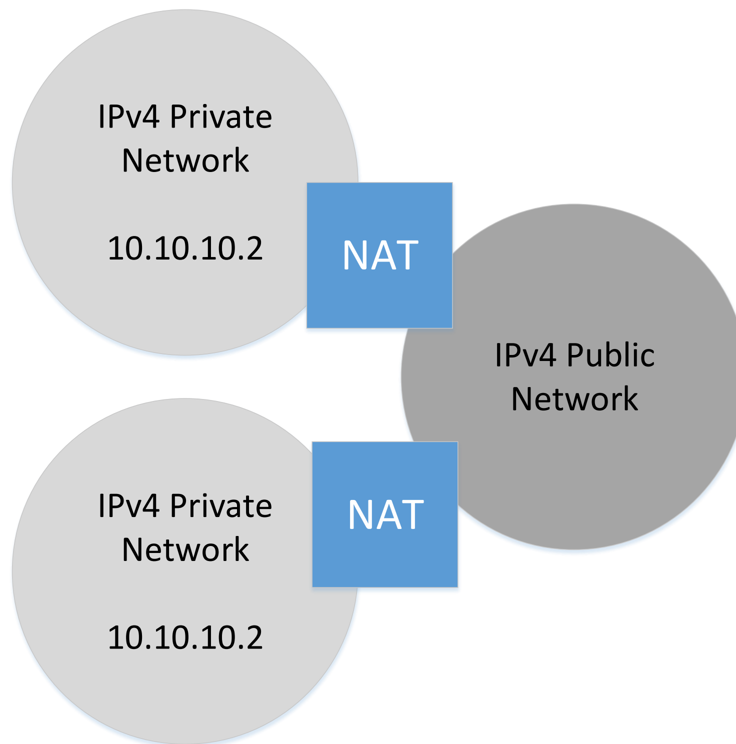


Figure 3: Multiple private networks sharing the same IP address space.

One reason why private networks are used is because of the shortage of public IPv4 addresses. The deployment of IPv6 will solve that problem, as it is expected that the number of IPv6 addresses will be enough for a very long time.

However, NATs are also used for privacy and security reasons, when a network operator does not want to expose the IP addresses and ports of the network devices to other networks. A NAT can be used to hide those IP addresses and ports. In addition, it also gives the operator of the network better control of incoming traffic to the network. IPv6 will not remove the need to use NATs for such purposes.

Internet of Things (IoT) is a term that is commonly used to describe where devices without direct human interaction are connected to the Internet. Such devices can be large industrial devices, or they can be small sensor type of devices. In many cases the IoT devices are also constrained in one way or another, e.g., when it comes to processing power, battery life or connectivity. [4].

It is expected that IoT is going to increase the number of devices connected to the Internet in very large numbers. In the industry people often talk about "billions of connected devices". Similar to legacy devices, and mostly for the same reasons, a large number of the IoT devices will be located in private networks, behind NATs. [5].

The purpose of this thesis is to investigate problems that NATs cause for deployments of IoT devices. The thesis will describe data traffic patterns and constraints associated with IoT devices, and whether legacy NAT traversal mechanisms are applicable for IoT devices.

The main focus of the thesis is on data exchange between IoT devices located in a private network and server devices located in a public network, referred to as client-server data exchange. However, the thesis also studies whether a legacy NAT traversal mechanism that is used for device-to-device data exchange can be optimized for IoT devices.

The thesis consists of 5 Sections. Section 2 describes the problems that the thesis focus on, and IoT specific characteristics and traffic patterns.

Section 3 describes existing protocols that are used for IoT communication.

Sections 4 describes mechanisms that can be used to solve the problems described in Section 2. While some of the mechanisms have been standardized, the thin-ICE mechanism described in Section 4.3 has been designed as part of this thesis.

2 IoT Communication Challenges and Characteristics

NAT bindings are typically created when the device in the private network is sending data through the NAT towards a device in the public network (as described later in the document, there are also mechanisms that allow devices to explicitly control NATs). The NAT binding will remain open as long as the device in the private network sends data through the NAT within a given time interval. Even if the device in the private network does not have any session data to send, it can send periodic packages, referred to as keepalives, which only purpose is to maintain the NAT binding. [1, 6].

The sending of periodic keepalives is normally not a problem for devices that do not have to consider battery resources, connectivity resources etc. However, as the sending of keepalives consumes resources (e.g., battery resources) it might not be feasible for a constrained device to send such keepalives. In such cases the NAT binding will timeout and be terminated.

This thesis focuses on three problems, described in Sections 2.1, 2.2 and 2.3, which can occur when NAT bindings are not maintained. While the problems as such are not specific to IoT, they are more likely to occur due to the constraint nature of many IoT devices.

Section 2.4 describes IoT specific traffic characteristics and patterns. Those need to be taken into consideration when designing NAT traversal mechanisms for IoT environments.

Section 4 describes mechanisms that can be used to overcome one or both of the problems described below.

2.1 Reachability

Typically, NATs only allow devices in private networks to create NAT bindings, which means that the device in the private network needs to initiate the communication with its peer in the public network. If the NAT receives packets from a public network, and there is no NAT binding associated with the destination IP address and port associated with the packets, the packets will simply be dropped by the NAT. Therefore, in order for a device in a public network to reach a device in a private network, the device in the private

Metropolia
University of Applied Sciences

networks needs to create and maintain a NAT binding that can be used. As described earlier, constrained IoT devices might not be able to maintain NAT bindings.

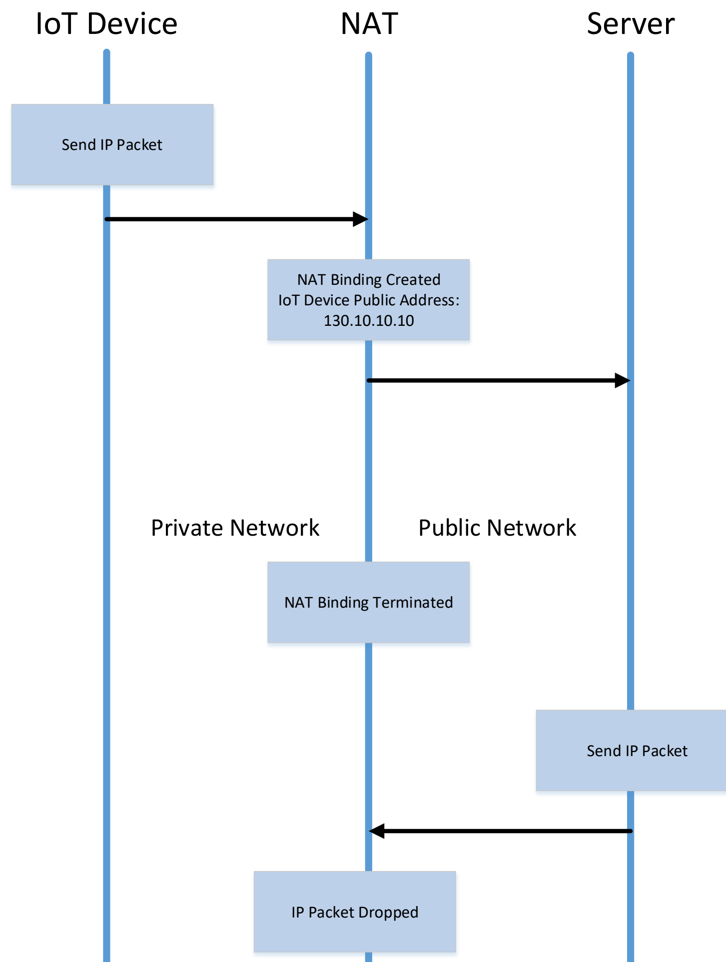


Figure 4: IP packet dropped by NAT.

Some services are designed in a way that devices in a private network always initiate the communication with a server located in the public network. However, that restricts the ability of the server to reach devices whenever it needs to.

2.2 Identification

There are cases where the IP address and port of the device in the network is used by a server device in a public network to identify the device (or a session or association established with the device) in the private network. As the device is located behind a NAT, the server device will only see the public IP address and port that the NAT has

assigned to the device (if there are multiple NATs between the device in the private network and the server device, the server device will see the IP address and port assigned by the NAT nearest to the server device).

If a NAT binding is not maintained, when the device in the private network later sends data in order to create a new NAT binding, the IP address and port that the NAT assigns to the device is most likely going to be different than the address and port associated with the previous binding. If the server device needs to be able to identify the same device both prior and after the new NAT binding was created, it will not be able to do so if the IP address and port used for that identification has changed.

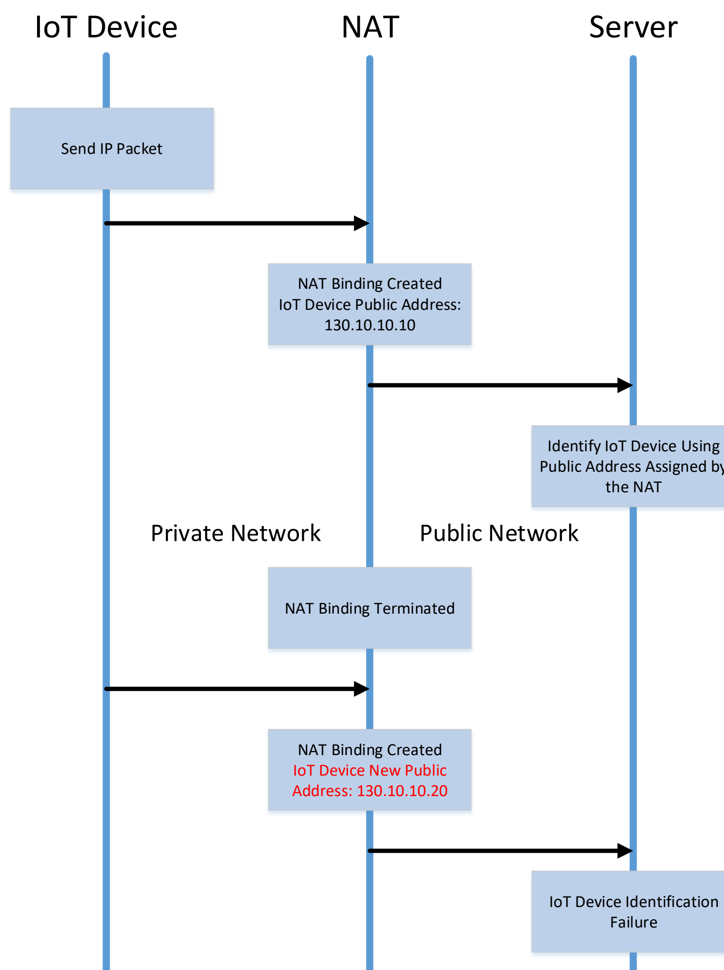


Figure 5: New public IP addresses assigned to device.

While the application level protocol often contains information that can be used to identify users, many servers still need to be able to identity the device.

2.3 Machine-to-Machine Communication

In some cases there is a need for IoT devices to communicate directly with each other, referred to as machine-to-machine (M2M) communication. If both devices are located behind NATs, in different private IP networks, none of the devices will be able to establish a connection with the other device. At least one of the devices will have to create a NAT binding, retrieve the public IP address and port associated with the NAT binding, and inform the IP address and port to the other device. The other device can then use that IP address and port and try to establish a connection between the devices.

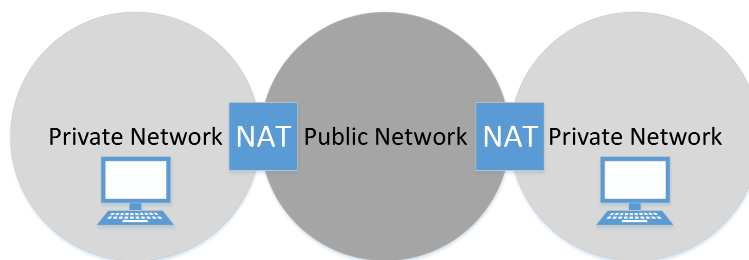


Figure 6: IoT devices in separate private networks.

In addition to being located in different private networks, the devices might also used different IP protocol versions (IPv4 or IPv6).

2.4 IoT Characteristics and Traffic Patterns

The data traffic patterns associated with constrained IoT devices might differ from traditional devices, e.g., regarding the duration of a session, the frequency of sent data etc. This Section will study different characteristics and traffic patterns of data exchange sessions, and how they apply to constrained IoT devices.

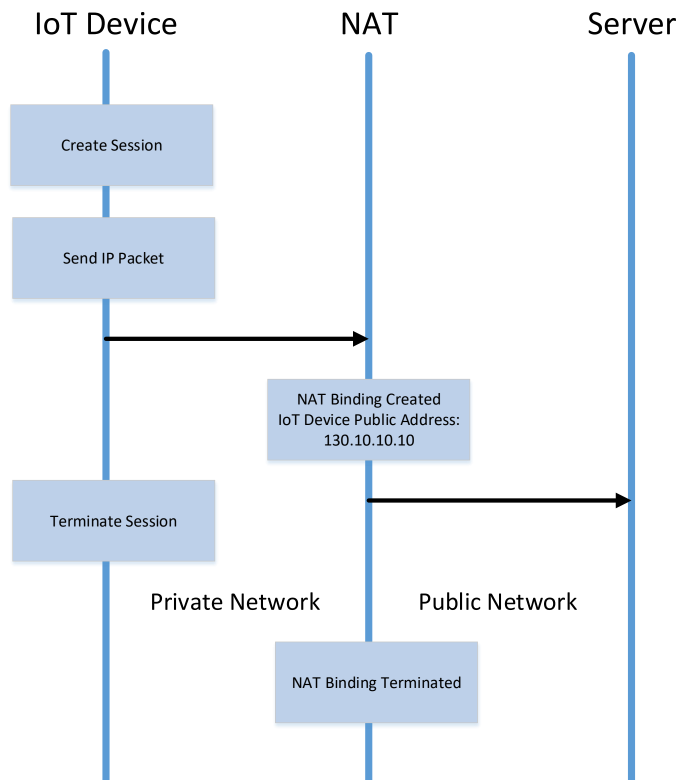
2.4.1 Application Session Duration

The session duration of an IoT device can vary widely.

For example, a device might establish a session, send some data, and then close the session. Depending on how many devices are served by a NAT, and how frequently a given device establishes a session, the NAT might end up creating a large number of

bindings that will only be used for a very short of time. The impact depends on how "heavy" the process of a binding is.

Alternatively, if a device is not battery constrained, and is able to maintain a NAT binding (by sending keepalives), a session might last for a very long time, compared to traditional data exchange sessions. Within such session, data might be sent frequently or unfrequently, as discussed in Section 2.4.2.



Metropolia
University of Applied Sciences

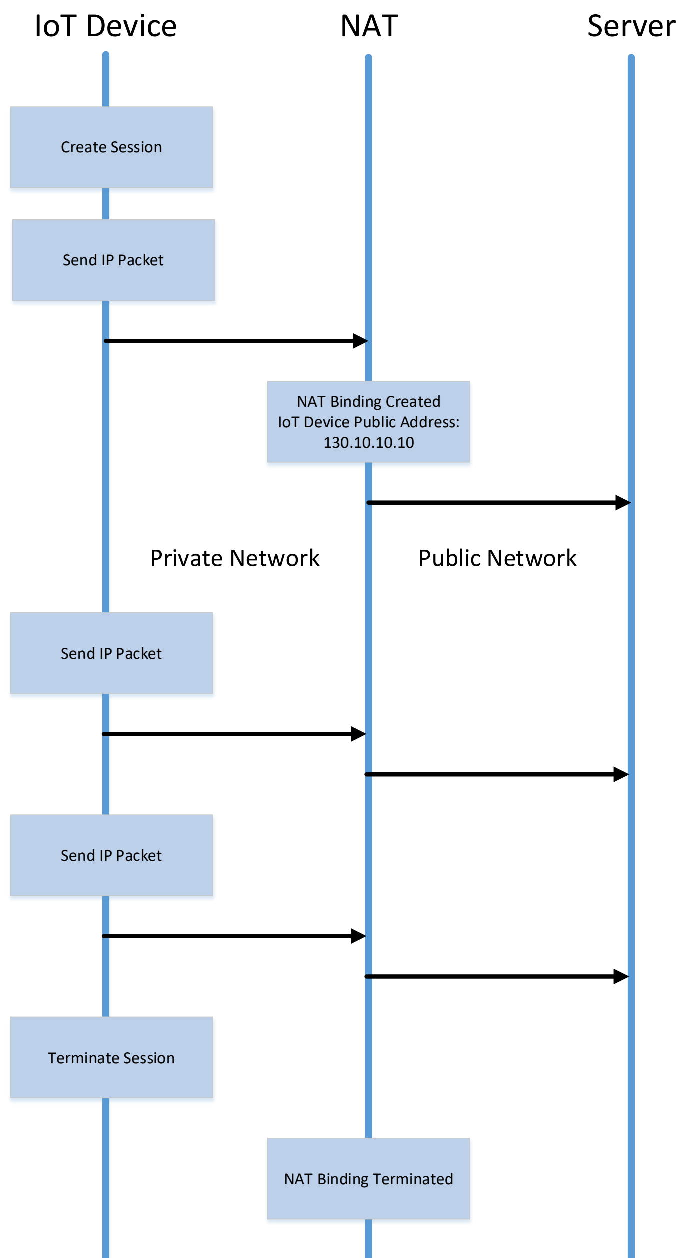


Figure 7: IP packets maintaining NAT binding.

Note that, unless the device is able to explicitly control the NAT (see Section 4.1.1), the NAT binding remains active until it eventually is terminated due to lack of data sent from the device.

2.4.2 Application Session Traffic Frequency

As described in Section 2.4.1, a data exchange session established by a constrained IoT device might last for a very long time. However, the amount of data sent by the device, and the frequency the data is sent, might vary widely. In some case the device might sent frequent, or even real-time, data. Examples are real-time video streaming or time-critical sensor measurement values. In other cases the device might send data unfrequently. In such cases, whenever the device does sent data, the size of the data might vary, from a single sensor value to a large data file (e.g., an image or video clip).

In case the duration of the data exchange sessions is very long, the NAT operator needs to take into account that there might be a large number of concurrent NAT bindings, even though there might not be any data passing through them.

2.4.3 Number of IoT Endpoints

When NATs are deployed in networks, the capacity of the NATs is based on assumptions and calculations based on the number of devices that the NAT will serve, the duration of the data exchange sessions passing through the NATs etc.

It has been predicted that, because of IoT, the number devices connected to the Internet (referred to as connected devices) is going to increase multifold. Unless the capacity of the NATs in the network is increased proportionally there will not be enough NAT capacity in order to serve all IoT devices. However, there is always an economical aspect associated with the capacity of a NAT, and it is unclear how much of that cost can be put on the owner or operator of the IoT devices served by the NAT. That might be one reason why operators choose to use short sessions that only last while data is exchanged (see Section 2.4.1).

2.4.4 Gateways

In some cases IoT devices might not be communicating directly with other devices on the Internet, but will instead be communicating with an intermediate node, referred to as a gateway, that will collect and forward data received from the device. In the case of constrained IoT devices, the reason might be that the device is not able to support all mechanisms (protocols etc.) that are needed in order to communicate with another device over the Internet.

A gateway might serve a single device, or multiple devices. The gateway can then either forward the data as it is (perhaps over a different transport mechanism) or collect data from multiple devices and aggregate the data towards a server device in a public network. The gateway might use a single data exchange session, or multiple sessions, to aggregate all data, depending on the amount of data etc. In case data will be sent frequently, the lifetime of the data exchange session might be very long (see Section 2.4.1).

From a NAT perspective, the gateway is typically seen as a single device. However, depending on how many IoT devices that are served by the gateway, how much data is sent by each individual device, and how the gateway aggregates that data, the traffic patterns might vary compared to a single IoT device.

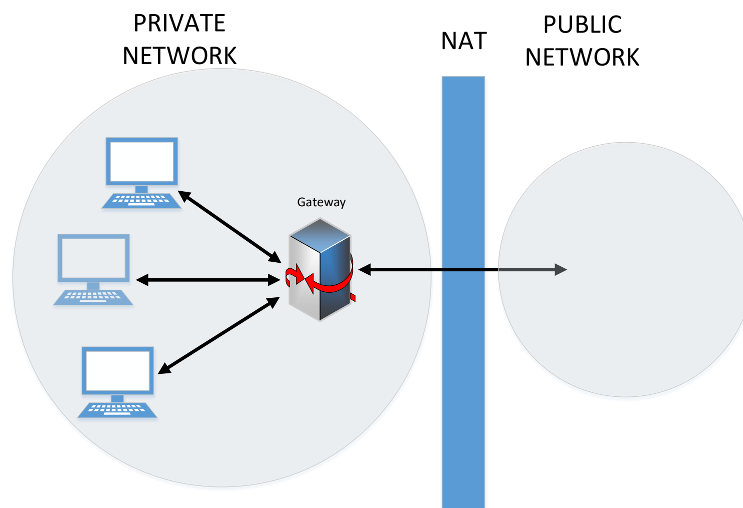


Figure 8: Gateway in private network.

3 IoT Communication Protocols

This section describes two secure transport protocols that are commonly used in IoT environments, one IoT protocol, and a NAT traversal mechanism for cases where both devices might be located behind different NATs.

3.1 Transport Layer Security (TLS)

Transport Layer Security (TLS) is a widely adopted security protocol that provides security for Transmission Control Protocol (TCP) connections. TLS provides authentication, confidentiality and integrity. Similar to TCP, data is delivered in a stream of bytes, without packet boundaries etc. Initially, TLS was mainly used to provide encryption for the Hypertext Transfer Protocol (HTTP), referred to as HTTPS, but nowadays TLS is also used to protect a number of different protocols. TLS defines two roles: TLS client and TLS server. The party that initiates a TLS session takes the TLS client role. [7, 8, 9, 10].

A TLS connection refers to a communication channel between a TLS client and a TLS server, e.g., a TCP socket. Each TLS connection is associated with a TLS session. A new TLS session can be created on a TLS connection that was previously used for another TLS session.

A TLS session refers to an association between a TLS client and a TLS server, and is defined by a set of cryptography parameters. A TLS session can be shared by multiple TLS connections. Some security parameters associated with a TLS session can be used for multiple TLS connections. However, different keys will be used for each TLS connection.

If a TLS connection is terminated, a TLS session associated with the TLS connection can be resumed on another TLS connection. Application protocols (e.g., HTTP) often use cookies for such session resumption. Section 4.2 describes the session resumption mechanisms defined for TLS, and how they can be used for NAT traversal.

Difference between connection and session is that connection is a live communication channel, and session is a set of negotiated cryptography parameters.

One can close connection, but keep session, even store it to disk, and subsequently resume it using another connection, may be in completely different process, or even after system reboot (of course, stored session should be kept both on the client and on the server).

On other hand, one can renegotiate TLS parameters and create entirely new session without interrupting connection.

Before application data can be encrypted, the TLS parties have to agree on the TLS version to use, the cryptographic algorithm to use, exchange certificates, generate a symmetric key and optionally authenticate each other. These steps take place during the handshake phase.

When the TLS client wants to initiate a TLS session, it initiates the handshake procedure by sending a ClientHello message to the other party. The ClientHello message contains e.g., the TLS versions and the security algorithms that the TLS client supports.

When the TLS server has received the ClientHello message, it sends a ServerHello message back to the TLS client. Based on the information that was provided in the ClientHello message, the ServerHello message contains the TLS version and the security algorithm that will be used for the TLS session. The ServerHello message also contains a TLS identifier that will be used for the TLS session. The ClientHello and ServerHello also contain other information associated with the TLS session, including data compression methods.

Once the TLS server has sent the ServerHello message, the parties will exchange certificates and generate symmetric keys that will be used to encrypt the data.

Once the handshake procedure is finished, the TLS parties can begin to exchange data that is encrypted with the shared key associated with the TLS session. The sending party splits data into blocks, encrypts the block and sends the encrypted data to the other party. When the other party receives encrypted data, it decrypts and reassembles the data.

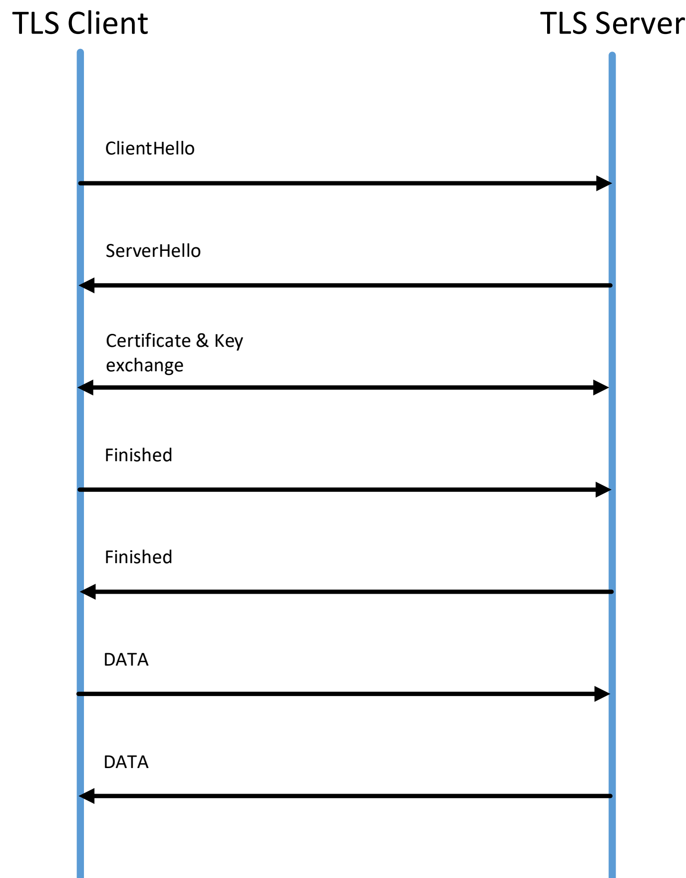


Figure 9: TLS message flow.

At the time of writing this document, the latest version of TLS is 1.3. The previous version, 1.2, is however still widely used.

3.2 Datagram Transport Layer Security (DTLS)

Datagram Transport Layer Security (DTLS) is a protocol that provides communications privacy for unreliable datagram protocols, e.g., the User Datagram Protocol (UDP). DTLS is based on Transport Layer Security (TLS) protocol. However, unlike TLS DTLS does not require reliable or in-order delivery transport protocol, e.g., Transmission Control Protocol (TCP). DTLS uses retransmission in order to deal with packet loss, and sequence numbers in order to deal with out-of-order delivery of packets. DTLS was designed to provide the same security features as TCP, and to minimize the need for new infrastructure in order to be deployed. In addition, DTLS is designed to run in the application space, and hence does not require kernel modifications. [11, 12].

DTLS does not provide DTLS association identifiers that can be used to associate a DTLS packet with a specific DTLS association. Therefore, if an endpoint establishes multiple DTLS associations, another mechanism in needed in order to associate a DTLS packet with a specific DTLS association. Traditionally the IP address and port is used for that purpose. However, that causes problems with Network Address Translators (NATs), as described later in this document.

Note that it is possible to use DTLS only to perform a key exchange and then use some other mechanism to protect the data. An example of this is the Secure Real-time Transport Protocol (SRTP) mechanism. When endpoints use SRTP, they can use DTLS to perform the key exchange, while the actual data encryption is done using other mechanisms. [13].

The DTLS handshake procedure is very similar to the TLS handshake procedure.

At the time of writing this document, the latest published version of DTLS is 1.2.

3.3 Constrained Application Protocol (CoAP)

The Constrained Application Protocol (CoAP) is a protocol designed for constrained devices, often also deployed in constrained networks. CoAP is very similar to HTTP, and uses the same client/server model. However, one of the main design goals of CoAP was to keep the message size small enough in order to be easy to process by memory and CPU constrained devices, and to minimize the need for message fragmentation in the network. CoAP also supports blockwise transfers that can be used to further reduce the need for message fragmentation. CoAP does not support all features supported by HTTP. Instead, only features that are considered needed for constrained IoT environments have been defined for CoAP. [14, 15].

The purpose of this Section is to give an overview of CoAP, in order to understand the thin-ICE mechanism defined in Section 4.3.

3.3.1 Resources and Representations

CoAP is designed based on RESTful principles. One key concept of REST is the use of resources. A resource is an object that contains different properties: type, data and links to other resources. A CoAP resource can represent physical things (e.g., a sensor), data

values (e.g., a sensor reading), properties (e.g., sensor setting) or an action (e.g., sensor reboot). A resource is identified by a unique Unified Resource Identifier (URI), and CoAP devices use such URIs to address and identify resources. [16, 17].

A resource representation contains the data associated with a resource, also referred to as the state of the resource. A representation is encoded in a machine readable format, e.g., JavaScript Object Notation (JSON). A resource can having multiple representations, each encoded in a different format. [18].

```
{
  ”temperature”: 34
}
```

Figure 10: Resource representation using JSON.

A CoAP client can read the current state of a resource from a CoAP server, or send the desired state of the resource to the server].

3.3.2 Transport

CoAP is a transport independent protocol. Initially only UDP transport was defined for CoAP. Later, TCP and WebSocket transports have been defined. In addition, the Open Mobile Alliance (OMA) Specworks standardization organization has defined the usage of non-IP CoAP transports, including using Short Message Service (SMS). SMS can be useful to wake up CoAP devices that do not maintain an IP connection while in sleep mode. Non-IP CoAP transports are outside the scope of this thesis. [19, 20, 21].

TCP typically require less frequent keepalives in order to maintain NAT bindings. In addition, TCP provides better congestion and flow control than UDP. However, UDP provides less overhead, and it is recommended to use UDP unless there is a reason why TCP has to be used.

Metropolia University of Applied Sciences

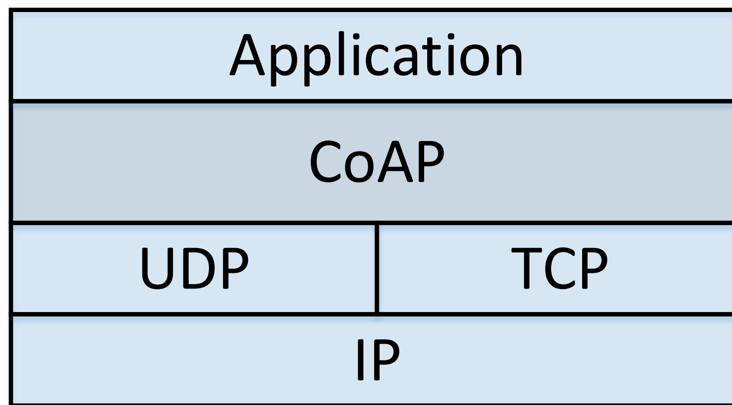


Figure 11: CoAP protocol stack (without security).

3.3.3 Security

CoAP provides both transport layer security, using of DTLS and TLS, and application layer security, using the Object Security for Constrained RESTful Environments (OSCORE) mechanism. OSCORE provides end-to-end security between CoAP devices, even if transport protocol translation occurs in the path between the devices. [22].

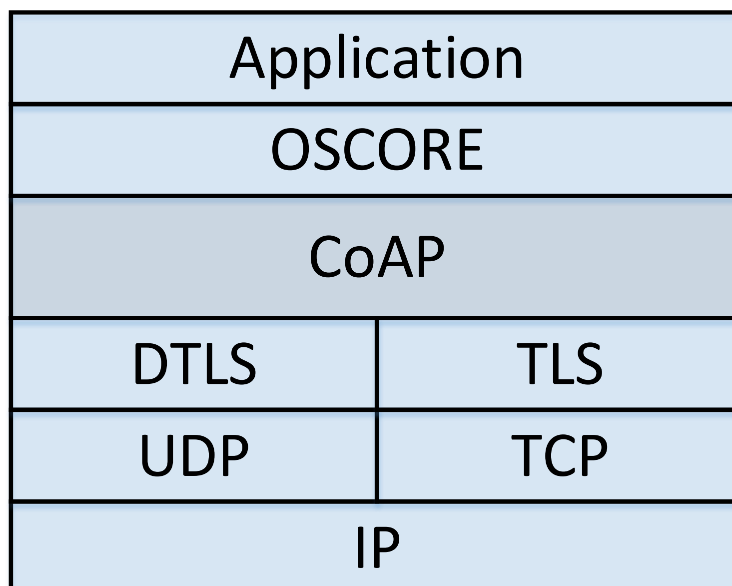


Figure 12: CoAP protocol stack with security.

3.3.4 Message Structure

CoAP messages are used to carry CoAP requests or CoAP responses. In addition, a message can also be “empty”, referred to as empty message, in which case it does not carry a request or a response. Each message shares the same message structure. An empty message only contains the 4 first bytes.

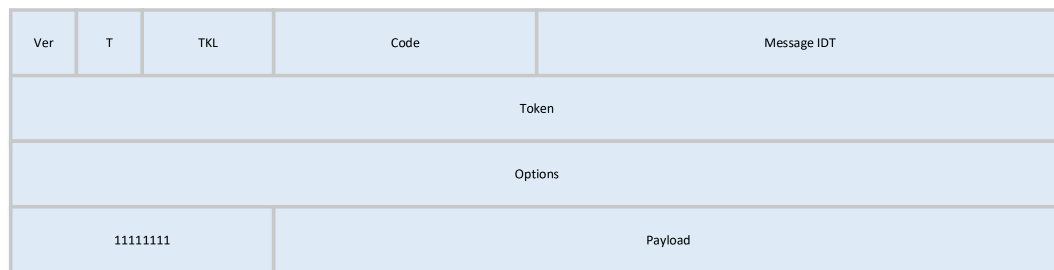


Figure 13: CoAP message structure.

The Version (Ver) indicates the version of the CoAP protocol. CoAP devices that implement IETF RFC 7252 must use a value of “1”.

The Type (T) indicates the message type. The message type can be Confirmable (CON), Non-confirmable (NON), Acknowledgement (ACK) or Reset (RST). A Confirmable message must be acknowledged by an ACK or a RST message, and a CoAP device will retransmit a Confirmable message until it receives the associated acknowledgement. A Non-confirmable message does not need to be acknowledged, and will not be retransmitted. Repeated messages (e.g., sensor readings) are typically non-confirmable, if it does not matter if some messages are not successfully delivered to the remote peer. An ACK message is sent by a device to indicate that it has received a Confirmable message. A Reset message is sent when a device has received a Confirmable or Non-Confirmable message that the device is not able to process. A CoAP server can piggypack a response in an ACK message, or it can first send an ACK message and later a standalone response message (e.g., if the server is not able to create a response immediately, but still needs to send an ACK message in order to acknowledge that it has received a Confirmable message).

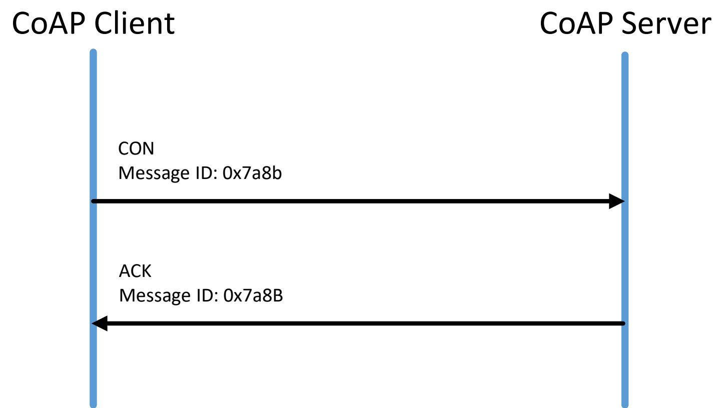


Figure 14: Reliable CoAP message transmission.

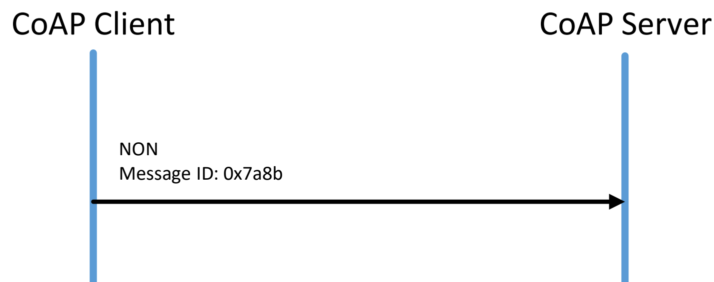


Figure 15: Non-reliable CoAP message transmission.

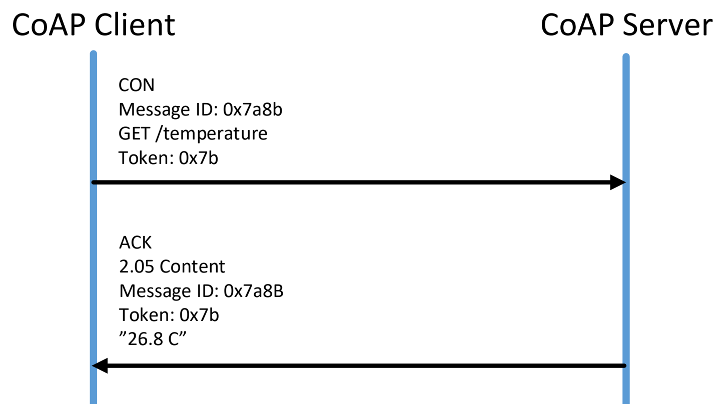


Figure 16: CoAP GET request message with piggybacked response.

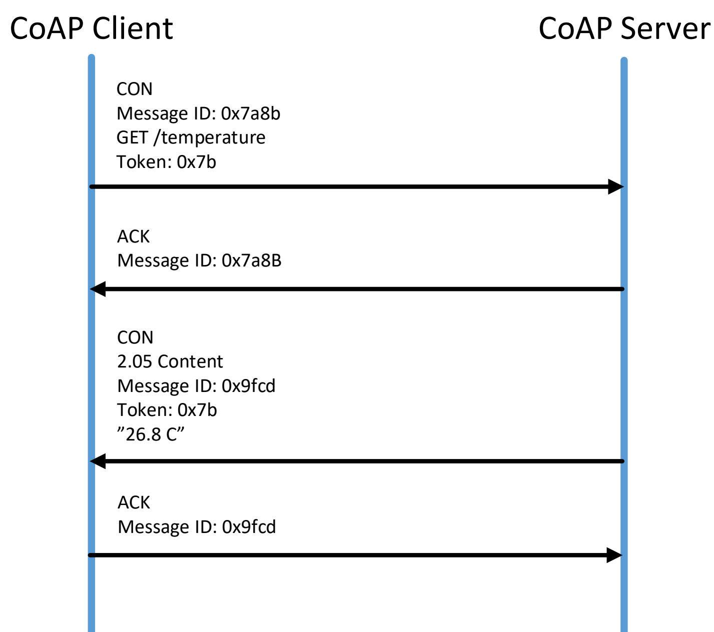


Figure 17: CoAP GET request message with separate response.

The Token Length (TKL) indicates the length of the Token field, in bytes. If the message does not contain Token information, the value is set to "0".

The Code is used to indicate the CoAP method (in the case of a CoAP request), the CoAP response code (in the case of a CoAP response) or an empty CoAP message (code 0.00).

The Message ID is unique value assigned to the message. It can be used to detect message duplication and to achieve reliability: a Confirmable message and the associated Acknowledgement message will contains the same Message ID value.

Metropolia
University of Applied Sciences

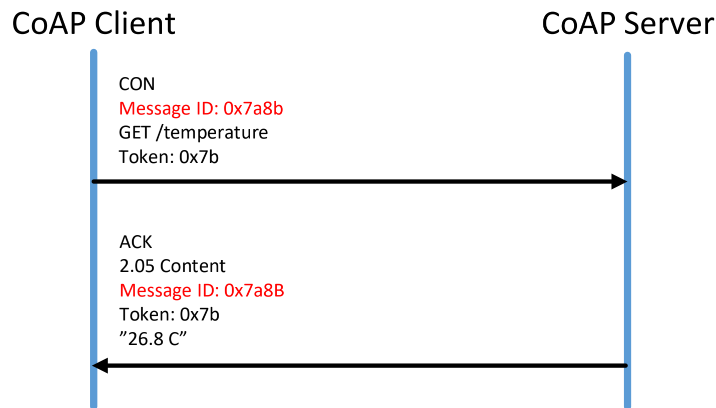


Figure 18: CoAP Message ID.

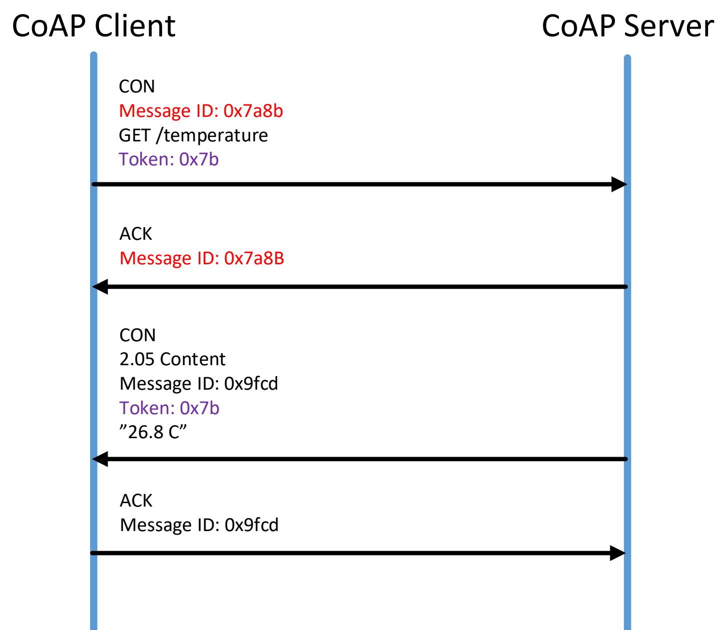


Figure 19: CoAP Message ID and Token.

The Token is used to associate CoAP requests and responses. Note that the Token and the Message ID are two separate values.

The Options is used to indicate CoAP Options associated with the CoAP message. A message can contain zero or more Options. Options can have different properties, e.g., whether it is required that the receiver of the message understands the Option.

The Payload Marker (PM) is used if the CoAP message contains a payload, in which case the payload is prefixed with the Payload Marker. The value of the Payload Marker is 0xFF (each bit is set to “1”).

The Payload contains the payload transported in the CoAP message.

3.3.5 Request/Response

CoAP messages can contain CoAP requests and CoAP responses. A CoAP client sends a request to a CoAP server, and the server sends back one or more (depending on the request type) responses to the client.

An empty messages does not contain a request or a response. For example, an ACK message or a RST message do not contain a request or a response. However, as described in Section 3.3.4, an ACK message can be used to piggyback a response.

3.3.6 Methods

The CoAP core specification defines 4 methods: GET, POST, PUT and DELETE. The semantics of the methods are similar to the HTTP methods with the same names. Each method has a property: safe and idempotent. A safe method will not modify a resource. If a method is idempotent, it means that even if the method is called multiple times, the outcome will be the same.

The GET method is used to read information associated with the resource to which the CoAP message is addressed. If the request succeeds (the response contains a response code that indicates success), the data associated with the resource will typically be included in the response message. The GET method is safe and idempotent.

The POST method is typically used to create a new resource. If the request is addressed to a previously created resource, the POST method can also be used to update that resource. If the request succeeds, the resource has been created or updated. In addition, if a new resource has been created, the response will contain a URI associated that identifies the new resource. The POST method is not safe, and it is not idempotent.

The PUT method is very similar to the POST method, and can also be used to create a new resource, or to update a previously created resource. The PUT method is not safe, but unlike the POST method it is idempotent.

The main difference between the POST and PUT is related to how the resource representation data provided in the CoAP request message is processed by the CoAP server. In the case of POST, the server will process the data using resource-specific logic, and the resulting resource representation data of the resource that is created or updated might vary from the data that was provided in the request. In the case of PUT, the resource representation data in the request will be assigned to the resource that is created or updated by the request. [23].

The DELETE method is used to delete the resource to which the CoAP request message is addressed. A DELETE request message is considered successful even if the addressed resource does not exist. Therefore the method is idempotent. The method is not safe.

3.3.7 Response Codes

Each response message will contain a response code. The response code contains two parts, separated by a dot. The first part indicates the type of response code, and the second part contains the rest of the code value.

A 2.xx response code indicates a success. Such response codes are: 2.01 (Created), 2.02 (Deleted), 2.03 (Valid), 2.04 (Changed) and 2.05 (Content).

A 4.xx response code is used to indicate that something was wrong with the request, or that the request was used to request something to be done for which the client was not authorized.

A 5.xx response indicates a server error.

3.3.8 Payload

A CoAP request and a CoAP response can contain a payload, depending on the method (in case of a request) and response code (in case of a response). The payload typically

carry the resource representation associated with the resource to which the CoAP message was addressed.

CoAP does not mandate a specific payload format. Instead, different formats can be used. The Content-Type option is used to indicate the format of a payload in a message. A CoAP client can use the Accept option in a request message to indicate to the CoAP server which payload formats the client supports.

3.3.9 Resource Observation

As described in Section 3.3.6, when a CoAP client wants to retrieve the representation of a resource it sends a GET request message, addressed to the resource. If the request is successful, the CoAP server will include the resource representation in the payload of the response message. If the client later wants to retrieve the resource representation of the same resource again, the client sends a new GET request message.

If a CoAP client wants to retrieve the resource representation of the same resource multiple times over a period of time, it is not very convenient and efficient to send a GET request message each time. In addition, if the client is only interested to retrieve the resource representation when some data has been modified, the client does not know when to send the GET request message. A resource observation CoAP option, referred to as Observation option, has been defined to solve this problem. The option allows a client to send a single GET request message, and receive the resource representation of the addressed resource multiple times, each time in a separate response message. The client is referred to as an Observer, and the resource that the client is observing is referred to as the Subject. The server will typically send a response message with the resource representation to the client, referred to as notifying the client, when some data has changed. The mechanism is very similar to the HTTP long polling mechanism. [10, 24].

The Observe option can be used in a GET request message, and the option can have two values. A value of “0” indicates that the CoAP client wants to observe the addressed resource. A value of “1” indicates that the client no longer wants to observe the resource. The GET request message, and each response message, referred to as notification, contains the same Token value, which allows the client to associate the notification with the request message. When the Observe option is included in a GET response message,

Metropolia
University of Applied Sciences

it indicates that the response message represents a notification. Within a response message the Observe option value is used as a sequence number, which allows the client to detect if notifications are received out of order. A notification can be Confirmable or Non-confirmable.

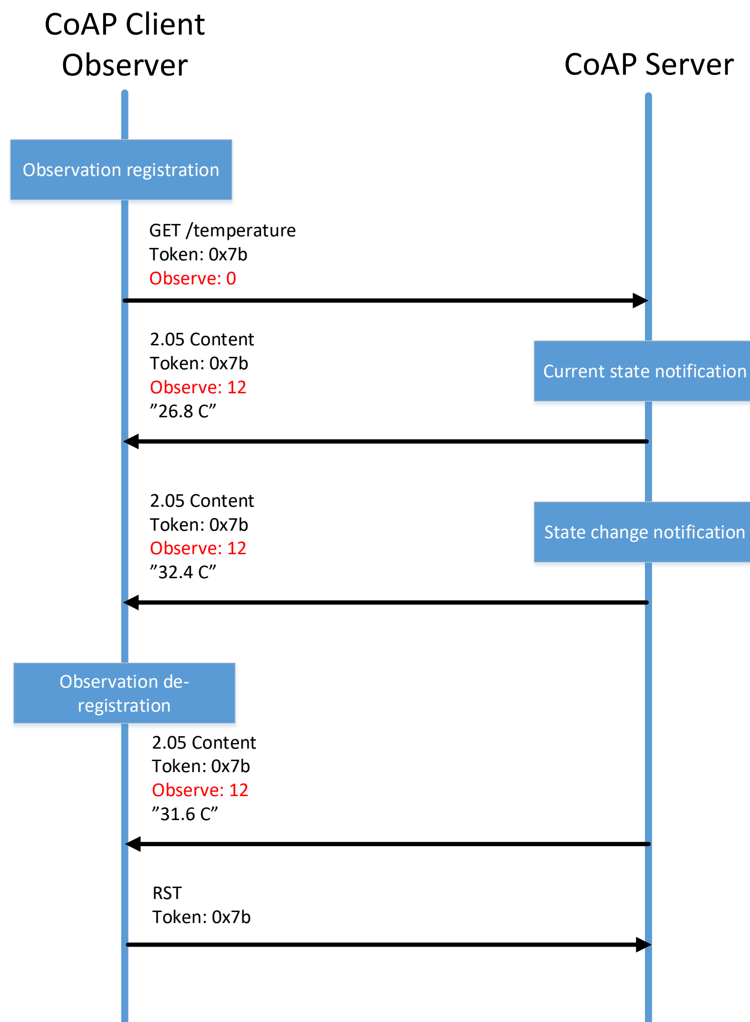


Figure 20: CoAP resource observation.

When a CoAP client no longer wants to observe a resource, there are two ways that the client can use to cancel the observation registration. The client can either return a RST message when it receives a notification, or the device can send a GET request message, addressed to the resource, with the Observe option value set to “1”.

3.3.10 Resource Directory

CoAP devices might not be aware of other devices, what resources they host, and how to reach them. A Resource Directory (RD) is an entity where devices can store information about the resources that they host. Other devices can then retrieve information about those resources from the RD. Note that the RD does not store the any resource values. The RD only stores information about the resources, using CoRE resource links. A resource link contains a URI that can be used by a device to access the resource associated with the resource link, and attributes to provide additional information about the resource link. Once a remote device has fetched a resource link, the device can use the resource link to access the associated resource. [25] [26].

Once a CoAP device has registered a resource link with an RD, the device can later update and remove the resource link data. Each resource link registration has a limited lifetime in the RD, so the device refreshes the resource link registration periodically.

When a CoAP device registers with an RD for the first time, the RD creates a resource associated with the registration, referred to as registration resource, and includes the path to the registration resource in a response sent back the device. The device will then use that registration resource in subsequent requests message that the device sends to the RD in order to manage the registered resource links.

In order to register a resource link, a CoAP device sends a POST request message to the RD. The POST request message contains e.g., a unique name associated with the device, referred to as the endpoint name (ep), and the lifetime of the registration. If no lifetime is provided, a default value is assumed. If the registration is successful, the RD will send 2.01 “Created” response message back to the device.

```
POST coap://rd.example.com/rd?ep=node1
<coap://10.10.10.10:5683/sensors/temp>
;ct=40;rt=”temperature-c”
```

Figure 21: CoAP POST request message for registering a link associated with a temperature sensor.

```
2.01 Created
Location-Path: /rd/4251
```

Figure 22: CoAP POST response from RD for successful link registration.

In order to fetch a resource link from an RD, a CoAP device performs a resource lookup by sending a GET request message to the RD. The device might provide different types of filter criteria to indicate what type of resources the device is interested in. The RD will then only return resource links associated with resources that match the filter criteria.

```
GET coap://rd.example.com/rd-
lookup?rt=”temperature-c”
```

Figure 23: CoAP GET request message to fetch links associated with temperature sensors from RD.

```
2.05 Content
<coap://10.10.10.10:5683/sensors/temp>
;ct=40;rt=”temperature-c”
```

Figure 24: CoAP GET response containing a link associated with a temperature sensor from RD.

Metropolia
University of Applied Sciences

The resource type (rt) link attribute contains a string value that is assign semantics to the resource associated with the resource link. In the example above, “temperature-c” indicates that the resource is a temperature sensor that provides measurements values in Celsius. Resource type values can be registered with the Internet Assigned Numbers Authority (IANA). [27].

3.4 Interactive Connectivity Establishment (ICE)

Interactive Connectivity Establishment (ICE) is a NAT traversal mechanism that allows two endpoints to establish a connection between themselves in cases where one or both endpoints are located in private IP networks behind one or more NATs. When ICE is used, the endpoints, referred to as ICE agents, try to establish connectivity using a set of tools of protocols. Once the agents have managed to establish connectivity, it can be used to exchange data. [28].

The following sub-sections describe the steps involved in the ICE procedure.

3.4.1 Collecting ICE Candidates

An ICE candidate represents an IP address, port and transport protocol associated with an ICE agent. There are different types of candidates.

A host candidate represents the local IP address and port of the ICE agent.

A server reflexive candidate represents an IP address and port associated by an ICE agent, as seen from the public network. In order to retrieve a server reflexive candidate, the ICE agent will send a request, using the Session Traversal Utilities for NAT (STUN) protocol, to a network entity, referred as STUN server. The STUN server will record the public IP address and port from where it receives the STUN request, and send that IP address and port back to the ICE agent in a STUN response. If the ICE agent is located in a public network, and there is no NAT between the agent and the STUN server, the server reflexive candidate will be identical to the host candidate. If the ICE agent is located in a private network, behind a NAT, the server reflexive candidate will reflect the public IP address and port associated with the NAT binding that is created when the ICE agent sends the message to the STUN server. [29].

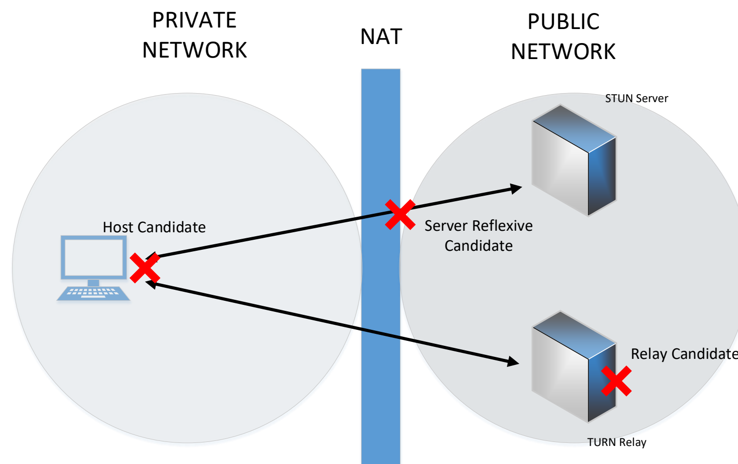


Figure 25: ICE candidate types.

It is possible to extend ICE and define new types of candidates. ICE does not mandate how many candidates, and what type of candidates, an ICE agent is required to collect. If an ICE agent is located in a public network, it will typically only have a host candidate.

No matter what the type of the candidate is, each candidate associated with an ICE agent is referred to as a local candidate. The candidates that the ICE agent will receive from its peer agent are referred to as remote candidates.

3.4.2 Exchange Candidates

Once an ICE agent has collected its set of candidates it will convey them to the peer agent, and receive the set of remote candidates from the peer agent.

The protocols and mechanisms that are used by the ICE agent to exchange the candidates, and the encoding of the candidates, are outside the scope of ICE. Draft-ietf-mmusic-ice-sip-sdp specifies how candidates are exchanged using the Session Description Protocol (SDP). [30].

When ICE agents only implement the core ICE specification, they must convey all local candidates at the same time. Draft-ietf-ice-trickle defines an extension to ICE, which allows ICE agents to convey additional candidates as they become available. [31].

Metropolia
University of Applied Sciences

3.4.3 Connectivity Check

When the ICE agents have exchanged their candidates, each agent will try to establish connectivity with the peer agent, using the IP addresses and ports associated with the remote candidates.

Each ICE agent will first create candidate pairs by pairing its own candidates with remote candidates of the peer agent. The ICE standard defines rules on how to give different priorities to candidate pairs, e.g., based on the candidate types, and in which orders candidate pairs will be tested. Some pairs will be discarded by the ICE agents, and will not be tested at all.

After an ICE agent have created the candidate pairs, it will start to test whether they can be used to establish connectivity with the peer agent. When an ICE agent tries to establish connectivity with the peer, it is performing a connectivity check. The ICE agents use the STUN protocol to perform the connectivity checks. For a given candidate pair, the STUN request will be sent from the IP address and port of the local candidate associated with the candidate pair, to the IP address and port of the remote candidate associated with the candidate pair.

If the STUN request used to perform a connectivity check reaches the peer agent, it will send a STUN response back. When the agent receives the response, it knows that the candidate pair for which the connectivity check was performed can used to establish connectivity. It will mark the candidate pair as a valid candidate.

The ICE agents will try to find at least one valid candidate pair for each data stream associated with the session. The ICE standard does not specify for how long the connectivity check procedure will last.

3.4.4 Nominate Candidate

One of the ICE agents is referred to as the controlling agent. At some point, if the controlling ICE agent has found at least one valid candidate pair associated with each data stream in the session, it will pick one candidate pair for each stream and inform the peer agent that those candidate pairs will be used for the session. This procedure is referred to as candidate nomination. After that the ICE procedure terminates and the ICE agents

can begin to exchange data using the IP addresses, ports and transport protocols associated with the nominated candidate pairs. The ICE agents can discard any resources associated with the non-nominated candidate pairs, and the agents will no longer send or receive data using those candidate pairs.

3.4.5 Keepalives

Throughout the lifetime of the session, the ICE standard specifies the sending of keepalives. Data exchange packets can be used as keepalives. If no data is exchanged STUN requests, referred to as STUN binding requests, can be used. Compared to STUN connectivity checks, STUN binding requests are not authenticated, as their solve purpose to maintain NAT bindings.

4 Solutions

This Section presents different solutions to the problems described in Section 2. The solutions are grouped into two main groups: solutions that are based on preventing NAT bindings to timeout, and solutions where a change of the public IP address, and port assigned to a device in a private network by a NAT does not impact how a server device can identity a device. Some of the solutions might solve both problems described in Section 2, while other solutions might solve a specific problem.

Section 4.3 defines a mechanism that can be used to establish connectivity between two devices when both devices are located behind different NATs. The mechanism is based on an existing mechanism, but has been modified for IoT devices and does not require support of additional protocols and infrastructure.

4.1 Prevent NAT Binding Timeouts

In theory, the most straight forward way to avoid the reachability- and identity problems caused described in Section 2 is to prevent NAT bindings from timing out. There are two ways of achieving this.

The most common way is that a device in the private network sends periodic keepalives that will maintain the NAT binding. Traditionally such keepalives are sent by the endpoint devices. However, as described in Section 2, it might not be feasible for a constrained IoT device to send keepalives. It requires that the IoT device is active, which consumes battery resources. In addition, if the IoT device is using constrained connectivity, the sending of keepalives will also consume connectivity resources.

When using the other method a device is able to explicitly control the NAT, using a dedicated NAT control protocol. Traditionally such control is also handled by the endpoint devices. Usage of a NAT control protocol typically does not require a device to send keepalives. However, it does require the device to support the NAT control protocol.

4.1.1 Device Controlled NAT

There are mechanism that allow endpoints to use a dedicated protocol to explicitly control NATs; request the NAT to create, modify and close NAT bindings. In order to use

such mechanisms, both the endpoint and the NAT(s) need to support the protocol. In addition, the endpoint need to be aware of the NAT(s).

Port Control Protocol (PCP)

The Port Control Protocol (PCP) allows a device to explicitly control a NAT, using a dedicated protocol. The device can control how incoming packets are translated and forwarded by the NAT, by creating mapping between the external and internal IP address and port. In addition to creating NAT mappings, PCP also allows a device to create different types of processing rules for incoming packets. [32].

PCP defines two roles: PCP client and PCP server. A PCP client is an entity that issues PCP requests to a PCP server in order to request and control resources. An IoT device that supports PCP will typically act as a PCP client. A PCP server is an entity that receives and processes PCP requests sent by a PCP client. A NAT that supports PCP will typically act as a PCP server.

Once the device has used PCP to create a NAT binding, it needs to inform its peer about the external IP address and port of the NAT. How the device informs the peer is outside the scope of PCP. It can be done e.g., using a signaling protocol that the device uses to communicate with its peer. If the device is using DNS based services it can use the DNS update method to update its DNS with the IP address.

When PCP is used, the device does not need to send keepalives in order to maintain the NAT binding. In addition, the creation of NAT mappings do not require the IoT device to send any data towards a server in the public network. It simply uses PCP to create a NAT mapping. However, the IoT device do need to inform anyone that wishes to establish a connection with the IoT device about the public IP address and port associated with the NAT mapping.

A PCP message contains a request- or response header, an opcode that identifies the operation associated with the message, information related to the operation and zero or more options. Response messages contain a response code, and can also contain additional information, e.g., for how long a created NAT mapping will last.

PCP can be extended by defining new OpCodes, or by defining new Options for existing OpCodes.

The following opcodes are defined:

MAP Creates or modifies a NAT mapping for inward forwarding.

PEER Creates or modifies a NAT mapping for outward mapping.

ANNOUNCE Announces different types of changes associated with the PCP server or NAT mappings.

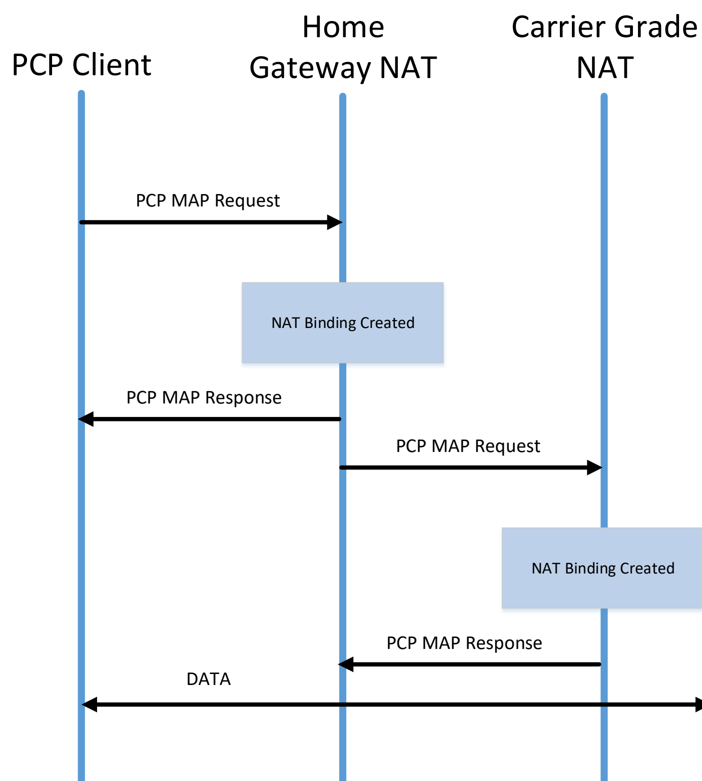


Figure 26: Port Control Protocol (PCP).

In many scenarios an IoT device will be located behind multiple NATs, e.g., a home gateway NAT and a carrier grade NAT. PCP can also be used by a NAT to control another NAT.

4.1.2 KaaS (Keepalive as a Service)

As mentioned earlier, if a device is responsible for sending keepalives, it will increase the power- and connectivity resources required by the device. The device needs to stay awake in order to send the keepalives, or at least be awaken every time a keepalive is to be sent. For that reason, it might not be feasible for a constrained IoT device to send keepalives. Many IoT devices are placed in “sleep” mode when not used, and are awaken e.g., only when they are going to send some data, e.g., a sensor reading, to a network server. A device might e.g., send data only once a day, while keepalives needs to be send more frequently in order to maintain NAT bindings.

KaaS (Keepalive as a Service) refers to mechanisms where another entity (typically a network entity) is sending keepalives on behalf of an IoT device. The network entity, referred to as KaaS device, can either be located in the private network between the IoT device and the NAT, or in the public network.

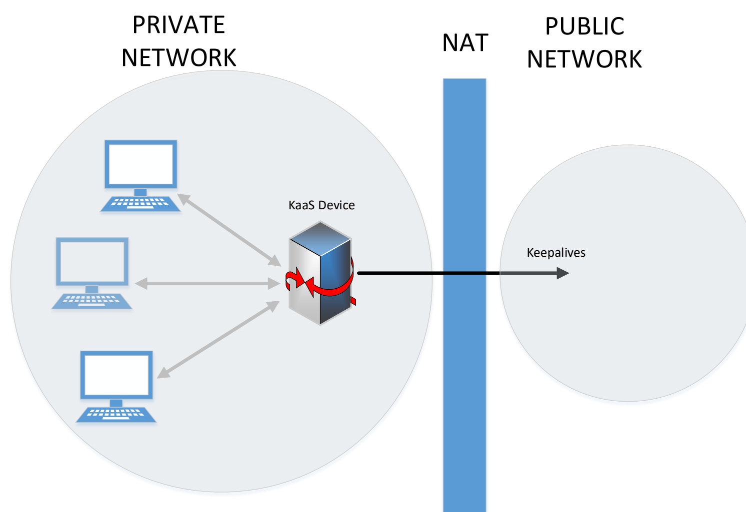


Figure 27: KaaS device located in private network.

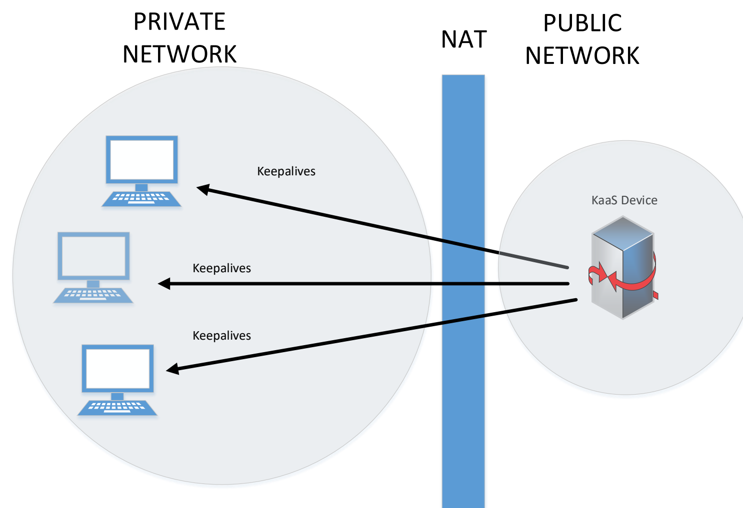


Figure 28: KaaS device located in public network.

Typically the KaaS device will also act as a relay for the data sent to and from the IoT devices that it serve. That way it can ensure that the NAT binding that it maintains is also used for the data.

If the KaaS device is located in the private network, it will send keepalives towards the public network in the same way an IoT device would. The only difference from a public network perspective is that all keepalives will come from the same IP address and port.

If the KaaS device is located in the public network, it will send keepalives towards the IoT device in the private network.

In some cases the KaaS device will send the keepalives all the way to the IoT device. However, if incoming traffic will wake up the IoT device, or if the connection link towards the IoT device is also constrained, the KaaS device might try to ensure that the keepalive does traverse the NAT, but is discarded before it reaches the IoT device.

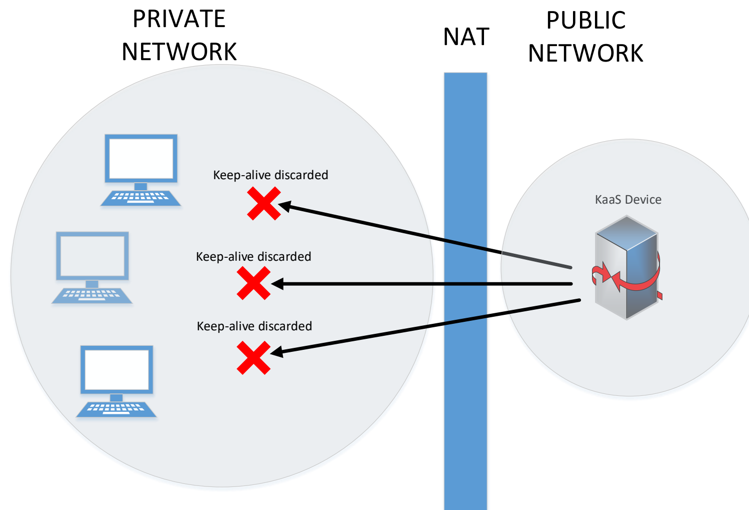


Figure 29: Keepalives dropped before they reach the IoT device.

One way to ensure that keepalives are discarded before they reach the IoT device, without having to deploy network devices especially for that purpose, is by limiting the number of hops the IP packets associated with the keepalives are allowed to travel before they are discarded. For IPv4 packets this is done by setting the Time-To-Live (TTL) field in the IP packet, and for IPv6 by setting the Hop Limit field. The KaaS device can calculate the correct hop value by starting by a small number, and then increase the number until the packet reaches the IoT device. After that, the KaaS device reduces the value and uses the new value for the keepalives.

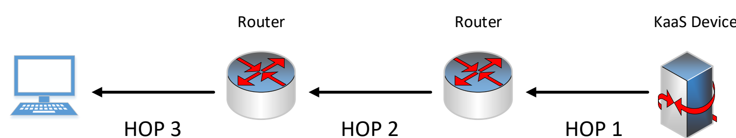


Figure 30: Keepalives dropped before they reach the IoT device.

As mentioned earlier, traffic sent from the public network may not trigger NATs to maintain NAT bindings; the IoT device (or a KaaS device located in the private network) needs to generate traffic in order to maintain the NAT binding. This needs to be taken into consideration if a KaaS device is to be used to maintain NAT bindings.

If a KaaS device sends keepalives on behalf of an IoT device, it might be useful for the KaaS device to know whether the IoT device is active. If the IoT device e.g., has been taken out of service there is obviously no idea to send keepalives. In some cases the KaaS device can be manually configured about the state of the IoT devices that it serves. In other cases the IoT devices might send some kind of information to inform the KaaS device (and other network devices) about its state. Such information will typically be sent less frequently than the keepalives; otherwise the IoT device could simply use the messages to send the information as keepalives.

4.2 Maintain Identity

In addition to not being able to reach an IoT device located in a private network, another problem caused by expired NAT bindings is related to identity.

A network server often recognizes an IoT device based on the IP address and port from where the server receive data from the device. If there is a NAT between the IoT device and the network server, the network server will use public IP address and port associated with the NAT binding created by the NAT. If the NAT binding is not maintained, it will eventually be closed. The IoT device can create a new NAT binding by sending data, but the public IP address and port associated with the new NAT binding will most likely be different than the previous one. For that reason, the network server will no longer be able to recognize the IoT device.

Metropolia
University of Applied Sciences

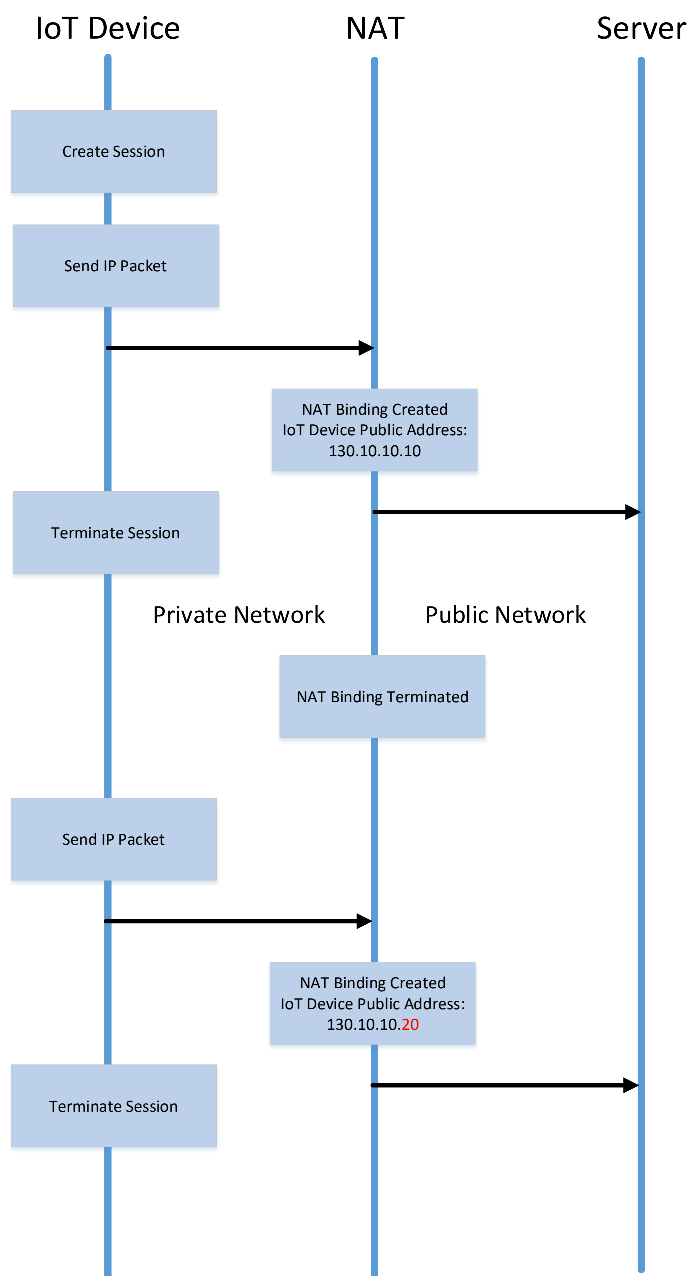


Figure 31: New public IP address assigned to IoT device by the NAT.

Sections 4.2.1, 4.2.2, 4.2.3, 4.2.4 and 4.2.5 defines TLS and DTLS mechanisms and extensions that can be used to maintain identity in cases where NAT bindings are terminated, without having to rely on the application layer protocol. Note that the mechanisms do not work with non-secure TCP or UDP.

4.2.1 TLS Session Resumption

TLS session resumption refers to the ability to resume a TLS session in cases where the underlying TLS connection layer has been interrupted or modified, e.g., if the public IP address and port that a NAT has assigned to a device in a private network has changed (because the device allowed the NAT binding to timeout).

Session resumption is a set of mechanisms that allows endpoint to resume a previously established TLS connection using less signaling than used during a normal handshake procedure. The resumption mechanisms do not need to use a 5-tuple to identify a TLS connection, which allows endpoint to resume a TLS connection even if the 5-tuple associated with the TLS connection has been modified due to a NAT binding timeout.

There are different ways to implement TLS Session Resumption, and the ways used depends on which version of TLS (1.2 or 1.3) is used.

For TLS 1.2, there exists two extension mechanisms for implementing session resumption: session identifier and session ticket. TLS 1.3 defines a mechanism referred to as TLS PSK session Resumption. Note that the TLS 1.2 mechanisms must not be used in TLS 1.3 deployments. Section 4.2.2 describes the session identifier mechanism, Section 4.2.3 describes the session ticket mechanism and Section 4.2.4 describes the TLS PSK session resumption mechanism.

4.2.2 Session Identifier

The session identifier mechanism is defined in the TLS 1.2 core specification. To use the mechanism, both TLS endpoints associate an identifier, referred to as session identifier, with the session state. The session identifier value is negotiated during the handshake procedure. [8].

The session identifier can be used to resume a TLS session on a new TLS connection, where the IP address and port of a DTLS endpoint has changed. Use of the session identifier mechanism allows the TLS session to be resumed with a single roundtrip of signaling, instead of having to perform a full handshake procedure.

The session identifier mechanism has been obsoleted in TLS 1.3.

4.2.3 Session Ticket

The session ticket mechanism is a TLS 1.2 extension. To use the mechanism, the TLS server provides a data structure, referred to as session ticket, to the TLS client. The session ticket contains information about the TLS session. [33].

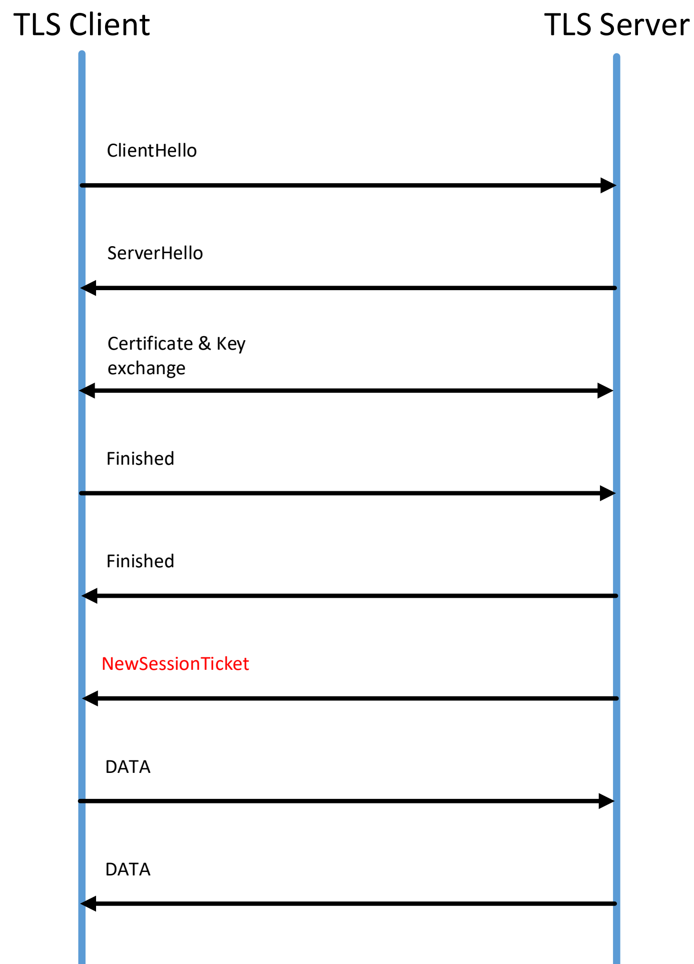


Figure 32: TLS server providing session ticket to TLS client.

In order to resume a TLS session, the TLS client can then include the session ticket in the ClientHello message of a subsequent handshake procedure.

Unlike the session identifier mechanism, a TLS server does not need to store a session ticket that it has provided to the TLS client. Instead the TLS server is able to retrieve the TLS session information from the session ticket.

Similar to the session identifier mechanism described in Section 4.2.2, use of the session ticket mechanism allows the TLS connection to be resumed with a single roundtrip of signaling, instead of having to perform a full handshake procedure.

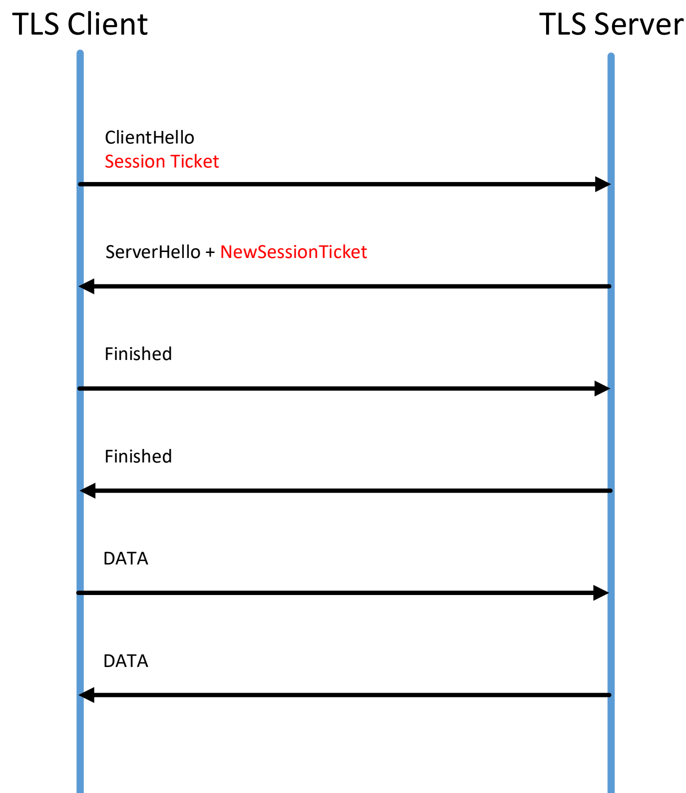


Figure 33: TLS session resumption with session ticket.

The session ticket mechanism has been obsoleted in TLS 1.3.

4.2.4 TLS PSK Session Resumption

A Pre-Shared Key (PSK), also referred to as a shared secret, is a key that is shared between two endpoints. [34].

Metropolia
University of Applied Sciences

In TLS 1.3 the session resumption mechanisms defined for TLS 1.2 have been obsoleted, and have been replaced by a similar mechanism, referred to as PSK session resumption. Instead of using a session identifier, or a session ticket, the TLS server generates a PSK identity, using a shared secret key derived from the initial handshake. The PSK identity can be used in future handshakes to resume a TLS session. From a TLS perspective, the PSK identity value is simply an opaque blob. TLS does not specify the content of the value. The value might e.g., point to an entry in the session cache of the TLS server, or it might be a session ticket that was encrypted by the TLS server. [7].

4.2.5 DTLS Connection ID

DTLS 1.2 does not specify an explicit session identifier parameter. Instead, the combination of the IP addresses and ports of the DTLS endpoints (DTLS client and DTLS server) has often been used to identify a DTLS "session", referred to as DTLS association.

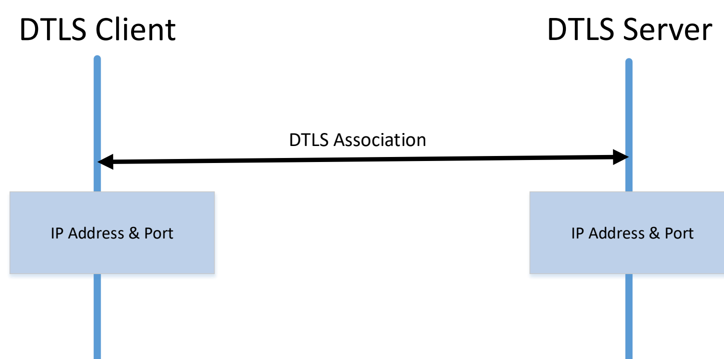


Figure 34: DTLS Association Identification.

As mentioned in Section 2.2, if there is a NAT between the DTLS endpoints, each endpoint will use the IP address and port associated with the NAT binding, together with its own IP address and port, to identify the DTLS association. However, if the NAT assigns a new IP address and port to a binding, the DTLS endpoints will no longer be able to use the IP address and port in order to identify a previously established DTLS association.

DSraft-ietf-tls-dtls-connection-id defines the Connection ID (CID). The CID is implemented as a DTLS extension type. The CID is a value that can be used to identify a DTLS association. The Connection ID value remains unchanged for the lifetime of the

Metropolia University of Applied Sciences

DTLS association, and it will not be affected by NAT binding timeouts and changes to the IP addresses and ports. [35].

If a DTLS client wants to use a CID, it includes the associated DTLS extension in the DTLS ClientHello message. The client indicates that value that it wants the DTLS server to use when it sends data towards the client. An empty value can be used to indicate that the client is willing to use the CID when sending data but does not want to receive the CID in data sent towards it. If the corresponding DTLS server wants to use a CID, it includes the associated DTLS extension in the DTLS ServerHello message. The server indicates the value that it wants the client to use when sending data towards the server. Similar to the client, the server can include an empty value if it does not want the client to use the CID when sending data towards the server.

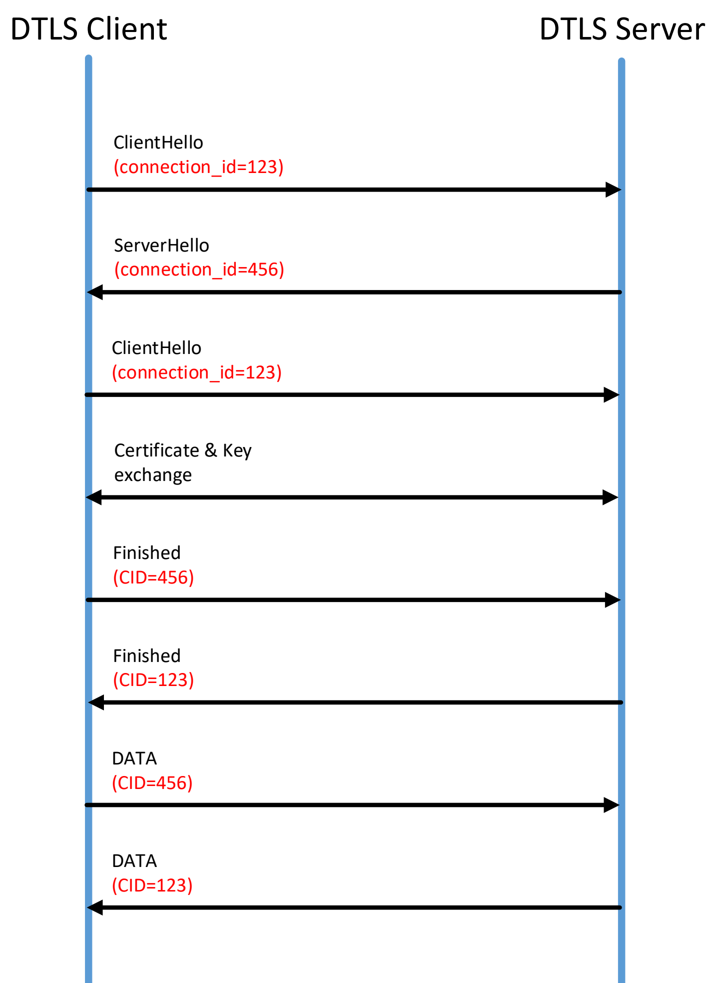


Figure 35: DTLS Connection ID negotiation and usage.

Once the Connection ID has been negotiated, endpoints can use it to identity DTLS associations, even if the IP address and port associated with NAT bindings change.

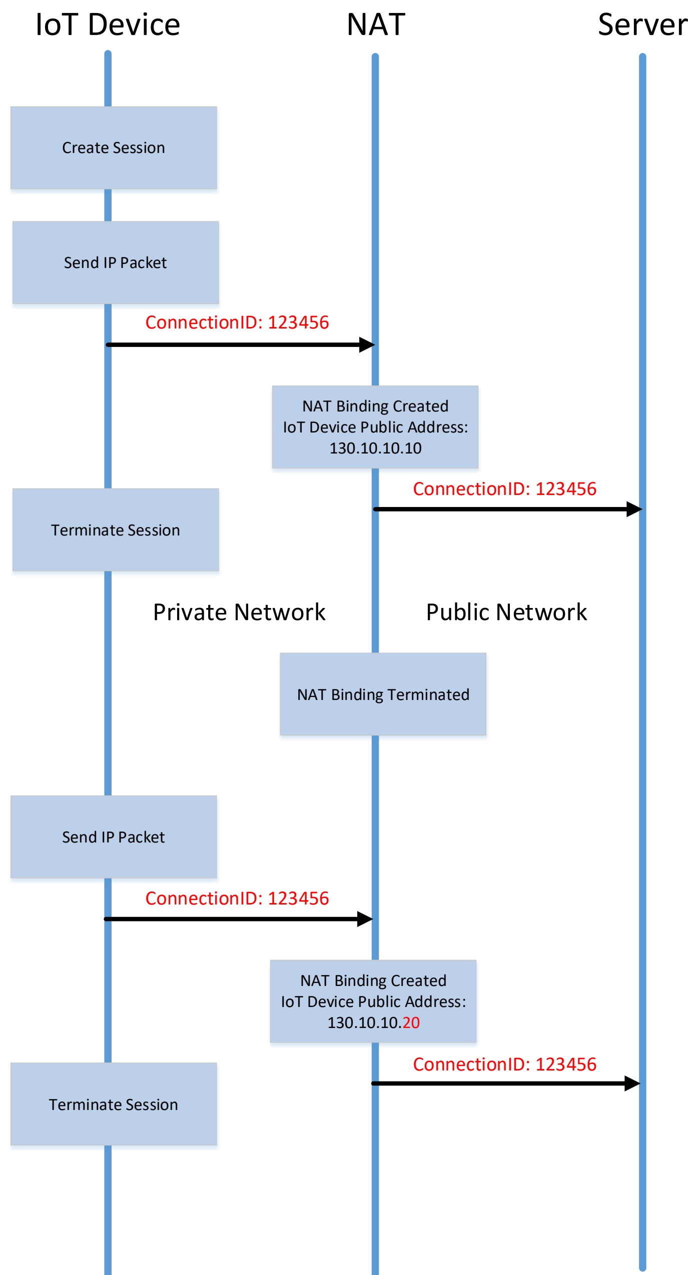


Figure 36: DTLS Connection ID with NAT rebinding.

In DTLS 1.2 it is not possible to negotiate a new CID value within a DTLS association. It requires a new handshake. However, in the next version of DTLS, DTLS 1.3, a DTLS endpoint can request the other endpoint to use a new value. [36].

In DTLS 1.3 the CID will be defined as part of the core specification. However, support and usage of the CID will still be optional.

4.3 Thin-ICE

Section 3.4 described the Interactive Connectivity Establishment (ICE) mechanism that can be used by endpoints located between NATs to establish a direct device-to-device connection.

As described in Section 3.4, the ICE standard does not mandate the usage of a specific protocol for ICE agents to exchange candidates. However, ICE does specify the usage of the STUN protocol for performing connectivity checks and keepalives. In addition, ICE specify the usage of dedicated infrastructure, including STUN servers and Traversal Using Relays around NAT (TURN) relays. These protocols and infrastructure were not designed with constrained IoT devices in mind, and it might not be feasible to implement support of them in such devices. [37].

The idea behind thin-ICE is to re-use the concept and procedures of ICE, but to use protocols and infrastructure that are more suitable for IoT devices to implement those concepts and procedures.

This Section describes how thin-ICE can be implemented using the CoAP protocol and infrastructure, including CoAP servers and CoAP resource directories. Thin-ICE might also be applicable to other IoT protocols.

The initial ideas behind the thin-ICE mechanism were presented and discussed in two of the Internet Research Task Force (IRTF) meetings, in 2016 and 2017. As part of this thesis, those initial ideas have been further expanded and enhanced, and the details of the thin-ICE mechanism documented. [38, 39].

4.3.1 T-STUN

With thin-ICE, the CoAP protocol will be used instead of STUN in order to create and distribute candidates, perform connectivity checks and keepalives.

Similar to a STUN server, a T-STUN server records the public IP address and port, assigned by a NAT, to an IoT device in a private network and returns the information to the

device. However, instead of using the STUN protocol, CoAP will be used between the IoT device and the T-STUN server.

From a CoAP protocol perspective, the T-STUN server is realized as a CoAP resource, referred to as a reflexive resource, hosted on a CoAP server. The T-STUN server must be deployed in a public network.

In order to create a candidate, a CoAP client will send a GET request message to the T-STUN server. The T-STUN server hosts a resource, referred to as the reflexive resource. When the T-STUN server receives the request, it checks the remote IP address and port from which the request message was received (the public IP address and port of the NAT), assigns the IP address and port values to the reflexive resource, and returns the IP address and port to the CoAP client in the response message payload. The client then creates a candidate associated with the returned IP address and port.

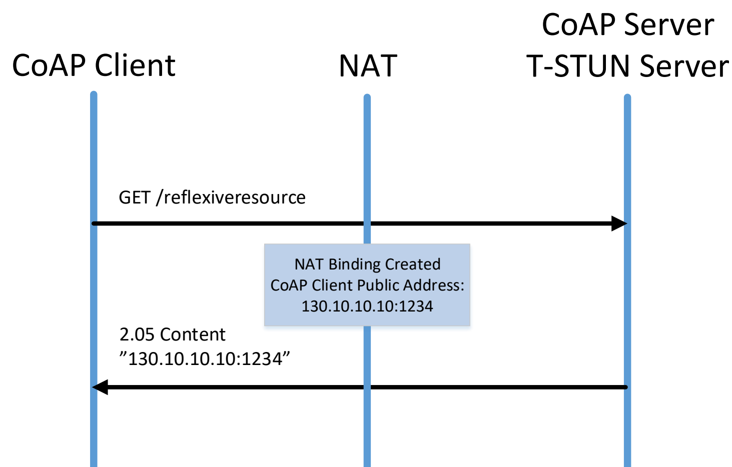


Figure 37: Fetch public IP address and port using T-STUN server.

Similar to STUN, the T-STUN server can be implemented in a standalone network entity, but it can also be implemented in existing CoAP infrastructure.

As described in Section 3.3.10, a Resource Directory (RD) stores resource links associated with resources hosted by CoAP devices. The resource links can then be used by other devices to access the resources associated with the resource links. A T-STUN

Metropolia
University of Applied Sciences

server can register a reflexive resource link with an RD, in order for other devices to find it.

4.3.2 Candidate Distribution

A CoAP device will use an RD to distribute its candidates, using resource links. Other devices can then fetch the resource links from the RD. There are two methods that a device can use to distribute its candidates using the RD. When a device uses the first method, referred to as the candidate resource link method, the device creates and hosts resources associated with each candidate, referred to as candidate resources. The device then registers candidate resource links with the RD. When a device uses the second method, referred to as the candidate embedding method, the device does not create candidate resources. Instead the device inserts the IP address and port associated with a candidate in the resource link URIs associated with other resource types that the device registers with the RD. Both methods are described below.

Candidate Resource Link

When a CoAP device uses the candidate resource link method, the device creates a resource for each candidate that it has generated. Such resource is referred to as candidate resource. The candidate resource representation data contains the IP address and port associated with the candidate. The device then registers candidate resource links with the RD, in the same way that the device registers other types of resource links (e.g., sensor resource links).

In addition to the IP address and port, additional data (e.g., how the candidate was generated, or the priority of the candidate) might be defined. However, such data is outside the scope of this thesis.

Unlike other resource type links, a candidate resource link contains the actual resource representation of the resource, as the candidate resource link URI contains the IP address and port associated with the candidate. Hence, a device that fetches a candidate resource link from the RD then retrieves the IP address and port associated with the candidate from the fetched candidate resource link URI.

IoT device:
IP address: 10.10.10.1
IP port: 5683

Candiate:
IP address: 130.10.20.30
IP port: 888

Temperature sensor resource link:
<coap://10.10.10.1:5683/sensors/temp>;ct=40;rt="temperature-c"

Candidate resource link:
<coap://130.10.10.10:1234candidates>;
rel="candidateof";anchor="coap://10.10.10.1:5683"

Figure 38: IoT device candidate resource link and temperature sensor resource links.

When a CoAP device has fetched a candidate resource link and another resource type link (e.g., a sensor resource link) associated with a remote device, and the device wants to access the other resource, the device replaces the IP address and port of the link URI associated with the resource that the device wants to access with the IP address and port of the candidate resource link URI.

IoT device:
IP address: 10.10.10.1
IP port: 5683

Candidate:
IP address: 130.10.10.10
IP port: 1234

Candidate resource link:
<coap://130.10.10.10:1234/candidates>;
rel="candidateof";anchor="coap://10.10.10.1:5683"

Temperature sensor resource link with candidate IP address and port:
<coap://130.10.10.10:1234/sensors/temp>;ct=40;rt="temperature-c"

Figure 39: Temperature sensor resource link with IP address and port of candidate resource link.

A CoAP device can also use the candidate resource link in order to access the candidate resource hosted by the remote device, e.g., in order to retrieve more information about the candidate resource.

If a candidate expires, e.g., because a NAT binding has terminated and the IP address and port associated with the candidate are no longer valid, the CoAP device that generated the candidate needs to generate a new candidate, update the associated candidate resource, and update the candidate resource link registration with the RD.

Similar to other resource type links registered with an RD, a candidate resource link registration might expire. Therefore, a CoAP device periodically refreshes the candidate resource link registration.

Link relationships are used to correlate a resource candidate link with other types of resource links associated with the same CoAP device. The candidate resource link contains a 'rel' link attribute and an 'anchor' link attribute. The value of the 'rel' attribute is "candidateof". The value of the 'anchor' attribute is the IP address and port used in the link URIs of the other resource type links. There is no need to use a 'rt' attribute, as the value of the 'rel' attribute indicates that the resource link is associated with a candidate resource.

Temperature sensor resource link:

```
<coap://10.10.10.1:5683/sensors/temp>;ct=40;rt="temperature-c"
```

Candidate resource link:

```
<coap://130.10.10.10:1234/candidates>;
rel="candidateof";anchor="coap://10.10.10.1:5683"
```

Figure 40: Candidate resource link correlated with temperature sensor link.

If a CoAP device generates multiple candidates, the devices creates a resource candidate and registers a candidate resource link associated with each candidate.

Candidate #1:
IP address: 130.10.10.10
IP port: 1234

Candidate #2:
IP address: 140.10.10.10
IP port: 5678

Candidate #1 resource link:
<coap://130.10.10.10:1234/candidates>;
rel=”candidateof”;anchor=”coap://10.10.10.1:5683"

Candidate #2 resource link:
<coap://140.10.10.10:5678/candidates>;
rel=”candidateof”;anchor=”coap://10.10.10.1:5683"

Figure 41: Multiple candidate resource links.

If the RD and the CoAP device that fetches links from the RD support the resource observation mechanism, the device can request the RD to automatically inform the device whenever a candidate resource link is modified.

Candidate Embedding

When a CoAP device uses the candidate embedding method, the device does not create candidate resources. Instead, the device inserts the IP address and port associated with a candidate in the link URIs of the other resource type links that the device registers with the RD. The candidate is said to be embedded into the other resource type links.

IoT device:
IP address: 10.10.10.1
IP port: 5683

Candidate:
IP address: 130.10.10.10
IP port: 1234

Temperature sensor resource link:
<coap://10.10.10.1:5683/sensors/temp>;ct=40;rt="temperature-c"

Temperature sensor resource link with embedded candidate IP address and port:
<coap://130.10.10.10:1234/sensors/temp>;ct=40;rt="temperature-c"

Figure 42: Temperature sensor resource link with embedded candidate IP address and port.

When a CoAP device has fetched a resource link that contains an embedded candidate, the device can use the link directly to access the associated resource hosted by the remote device. There is no need to replace the IP address and port of the link URI.

If a CoAP device generates multiple candidates, the device will register multiple links for each resource hosted by the device. Each link URI will contain the IP address and port associated with one of the candidates.

Candidate #1:
IP address: 130.10.10.10
IP port: 1234

Candidate #2:
IP address: 140.10.10.10
IP port: 5678

Temperature sensor resource link with embedded candidate #1 IP address and port:
<coap://130.10.10.10:1234/sensors/temp>;ct=40;rt="temperature-c"

Temperature sensor resource link with embedded candidate #2 IP address and port:
<coap://140.10.10.10:5678/sensors/temp>;ct=40;rt="temperature-c"

Figure 43: Temperature sensor resource links for multiple candidates.

If a candidate expires, a CoAP device needs to generate a new candidate, and update the registration for each resource link that contains an embedded candidate.

If the RD and the CoAP device that fetches links from the RD support the resource observation mechanism, the device can request the RD to automatically inform the device whenever a link that contains an embedded candidate is modified.

4.3.3 Connectivity Checks

When a CoAP device has fetched a candidate, either as a candidate resource link or embedded in the link associated with another resource type, the device performs a connectivity check simply by trying to access a resource hosted by the remote device, using the IP address and port associated with the candidate. If the device receives an ACK message, or a response message, the device knows that the candidate works, and can be used to access the remote device.

If a CoAP device has fetched multiple candidate associated with the same remote device, the device can perform a connectivity check for each candidate. Unless some priority information has been provided for each candidates, the device will choose which candidate to use based on local policy.

4.3.4 Keepalives

When a CoAP device has received its public IP address and port from a T-STUN serer, the device needs to ensure that the NAT binding that was created when the device contacted the T-STUN server is not terminated. Unless the transport protocol (e.g., TCP) will maintain the NAT binding the device will send periodic keepalives to the T-STUN server. The device can e.g., send periodic GET request messages, addressed to the reflexive resource hosted by the T-STUN server, as keepalive. That way, when the device receives the CoAP response, it can verify that the IP address and port associated with the candidate are still valid.

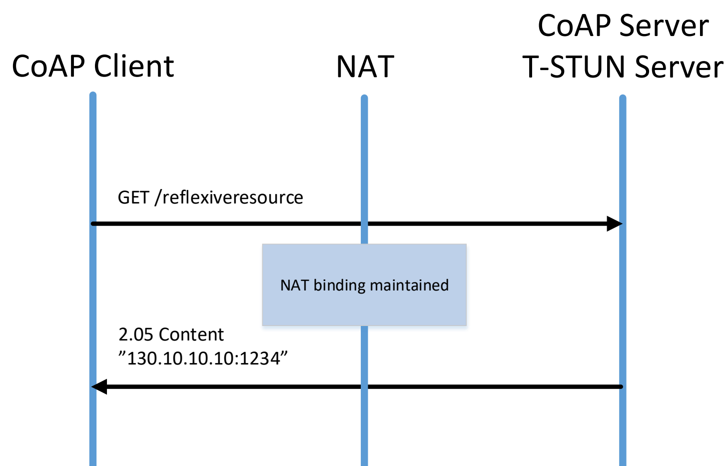


Figure 44: Keepalive using CoAP GET request.

As an alternative keepalive mechanism, the device can send periodic empty CoAP CON-messages to the T-STUN server. According to the CoAP specification, the T-STUN server should send a RST message back to the client when it receives the CON message.

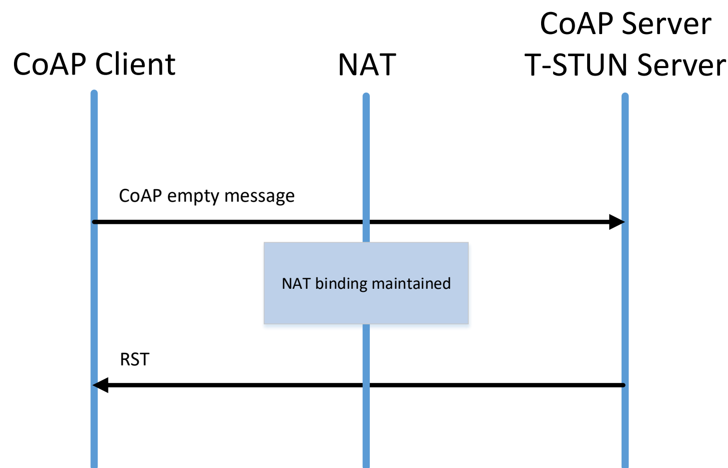


Figure 45: Keepalive using empty CoAP message.

If an empty CoAP message is used as keepalive, the device will not be able to verify that the IP address and port associated with the candidate are still valid, as the RST message does not contain that information.

4.3.5 Example

The figure below shows an example flow where a CoAP device (device #1) generates a candidate, and then a temperature sensor resource link that contains an embedded candidate with an RD. After that, another device (device #2) fetches the temperature sensor link from the RD, and uses the link URI to access the temperature sensor resource hosted by device #1.

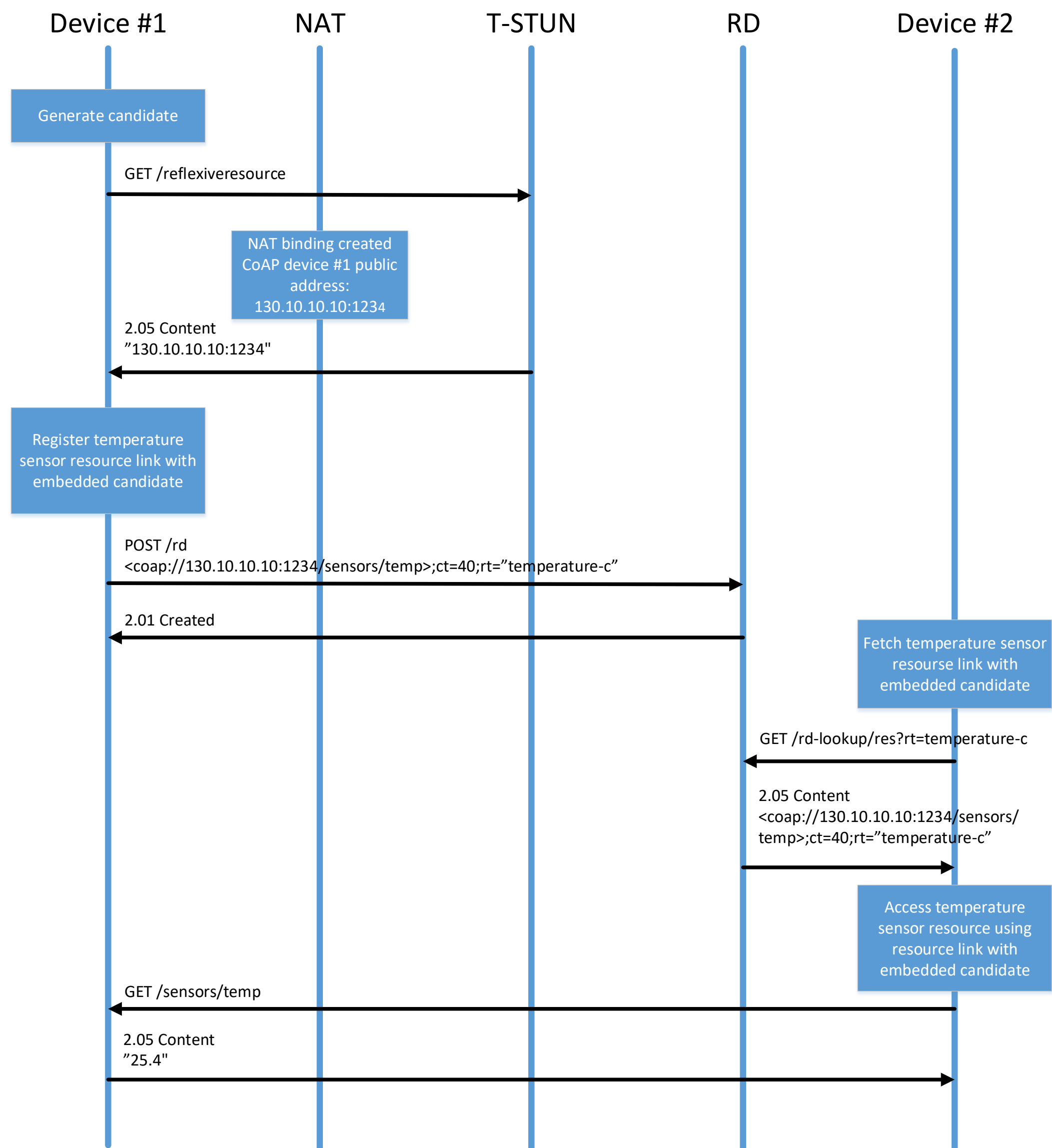


Figure 46: thin-ICE example flow with embedded candidate.

If a candidate resource link is used, CoAP device #1 registers both the candidate resource link and the temperature sensor link (without an embedded candidate) with the RD, and device #2 fetches both links from the RD.

5 Conclusions

While the deployment of IPv6 will remove the need for NATs for lack IP address availability reasons, NATs will still be deployed in IPv6 networks for other reasons, for example security and to keep network topologies similar to IPv4 network topologies. Also, as long as IPv6 and IPv4 continue to exist, NATs are needed in order to perform IP version translation. For that reasons, mechanisms for NAT traversal will be needed for the foreseeable future.

At the time of writing this thesis, there are already NATs deployed between constrained IoT devices and server devices that the IoT devices communicate with, and there is already a need for NAT traversal solutions suitable for such devices, e.g., because the server devices are located in another network, and in many cases hosted in cloud environments. In some cases the IoT devices maintain constant TCP connections with the server, which allows traffic in both directions. However, such solutions might not work for constrained IoT devices, or constrained networks, and as the number of such devices and networks is growing there is a need for NAT traversal solutions suitable for constrained environments.

There is yet not a big demand for NAT traversal mechanisms for machine-to-machine communication between constrained IoT devices. One reason is that direct communication between the constrained devices is not that common yet. Instead most traffic is sent via a server that is reachable from all IoT devices. Another reason is that most machine-to-machine communication takes place within a single subnet, where there a no NATs between the IoT devices. However, as machine-to-machine communication between IoT devices become more common, and the legacy communication technologies are replaced with IP-based protocols, it is expected that the communication will take place over a wider geographical area, and between different subnetworks. That means that there will be NATs deployed between the IoT devices, and NAT traversal mechanisms (e.g., thin-ICE) for machine-to-machine communication will be needed.

Usage of a NAT traversal mechanism such as ICE requires that devices implement support of new protocols (STUN), and that the needed infrastructure (STUN servers and TURN relays) are deployed in the network. While implementing support of a new protocol might not be a major issue for devices, it might have a big impact on constrained IoT devices. A big advantage of the thin-ICE mechanism described in this thesis is that it

allows usage of existing protocols that the device already supports, with minimal additional features to be implemented. In addition, as thin-ICE allows usage of existing IoT infrastructure (resource directories), the deployment of thin-ICE can be assumed to be faster and smoother than the deployment of ICE support in networks.

While this document describes how thin-ICE is implemented for CoAP, as thin-ICE mechanism can rather easily be adopted for other IoT protocols too, which is a big advantage in environments where operators have to run multiple IoT protocols simultaneously, or where operators have to upgrade the network from one IoT protocol to another.

Finally, while standardized NAT traversal mechanisms already exist, they are in general not well suited for constrained IoT environments. Thin-ICE is a potential candidate for a new standardized mechanism that has been designed for such environments.

References

1 NAT Traversal [online]. Wikipedia.
URL: https://en.wikipedia.org/wiki/NAT_traversal.
Accessed 29 September 2019.

2 Postel, J. Internet Protocol [online].
URL: https://www.ietf.org/rfc/rfc791.
Accessed 29 September 2019.

3 Deering, S and Hinden, R. Internet Protocol, Version 6 (IPv6) Specification [online].
URL: https://www.ietf.org/rfc/rfc2460.
Accessed 29 September 2019.

4 Internet of Things [online]. Wikipedia.
URL: https://en.wikipedia.org/wiki/internet_of_things.
Accessed 29 September 2019.

5 Internet of Things forecast [online]. Ericsson.
URL: https://www.ericsson.com/en/mobility-report/internet-of-things-forecast.
Accessed 29 September 2019.

6 Keepalive [online]. Wikipedia.
URL: https://en.wikipedia.org/wiki/Keepalive.
Accessed 29 September 2019.

7 Rescorla, E. The Transport Layer Security (TLS) Version 1.3 [online].
URL: https://tools.ietf.org/rfc/rfc8446.
Accessed 29 September 2019.

8 Rescorla, E. The Transport Layer Security (TLS) Version 1.2 [online].
URL: https://tools.ietf.org/rfc/rfc5246.
Accessed 29 September 2019.

9 Postel, J. Transmission Control Protocol [online].
URL: https://www.ietf.org/rfc/rfc793.
Accessed 29 September 2019.

10 Belshe, M, Peon, R and Thomson, M. Hypertext Transfer Protocol Version 2 (HTTP/2) [online].
URL: https://tools.ietf.org/rfc/rfc7540.
Accessed 29 September 2019.

11 Rescorla, E, Modadugu, N. Datagram Transport Layer Security Version 1.2 [online].
URL: https://tools.ietf.org/rfc/rfc6347.
Accessed 29 September 2019.

12 Postel, J. User Datagram Protocol [online].
URL: https://www.ietf.org/rfc/rfc768.
Accessed 29 September 2019.

13 Baugher, M, McGrew, D, Näslund, M, Carrara, E and Norrman, K. The Secure Real-time Transport Protocol (SRTP) [online].
URL: https://tools.ietf.org/rfc/rfc3711.
Accessed 29 September 2019.

14 Shelby, Z., Hartke, K. and Bormann, C. The Constrained Application Protocol (CoAP) [online].
URL: https://tools.ietf.org/rfc/rfc7252.
Accessed 29 September 2019.

15 Bormann, C and Shelby, Z. Block-Wise Transfers in the Constrained Application Protocol (CoAP) [online].
URL: https://tools.ietf.org/rfc/rfc7959.
Accessed 29 September 2019.

16 REST [online]. Wikipedia.
URL: https://en.wikipedia.org/wiki/REST.
Accessed 29 September 2019.

17 Berners-Lee, T, Fielding, R and Masinter, L. Uniform Resrouce Identifier (URI): Generic Syntax [online].
URL: https://tools.ietf.org/rfc/rfc3986.
Accessed 29 September 2019.

18 Bray T. The JavaScript Object Notation (JSON) Data Interchange Format [online].
URL: https://tools.ietf.org/rfc/rfc7159.
Accessed 29 September 2019.

19 Bormann, C, Lemay, S, Tschofenig, H, Hartke, K, Silverajan, B and Raymor, B. CoAP (Constrained Application Protocol) over TCP, TLS and WebSockets [online].
URL: https://tools.ietf.org/rfc/rfc8323.
Accessed 29 September 2019.

20 OMA SpecWorks [online].
URL: https://www.omaspecworks.org.
Accessed 29 September 2019.

21 SMS [online]. Wikipedia.
URL: https://en.wikipedia.org/wiki/SMS.
Accessed 29 September 2019.

22 Selander, G, Mattson, J, Palombini, F and Seitz, L. Object Security for Constrained RESTful Environments (OSCORE) [online].
URL: https://tools.ietf.org/rfc/rfc8613.
Accessed 29 September 2019.

23 Keränen, A, Kovatsch, M and Hartke, K. RESTful Design for Internet of Things Systems [online].
URL: https://tools.ietf.org/id/draft-irtf-t2trg-rest-iot.
Accessed 29 September 2019.

24 Hartke, K. Observing Resources in the Constrained Application Protocol (CoAP) [online].
URL: https://tools.ietf.org/rfc/rfc7641.
Accessed 29 September 2019.

25 Shelby, Z. Constrained RESTful Environments (CoRE) Link Format [online].
URL: https://www.ietf.org/rfc/rfc6690.
Accessed 29 September 2019.

26 Shelby, Z, Koster, M, Bormann, C, van der Stok, P and Amsuess, C. CoRE Resource Directory [online].
URL: https://tools.ietf.org/id/draft-ietf-core-resource-directory.
Accessed 29 September 2019.

27 Internet Assigned Numbers Authority (IANA) [online].
URL: https://www.iana.org.
Accessed 29 September 2019.

28 Keränen, A, Holmberg, C and Rosenberg J. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal [online].
URL: https://tools.ietf.org/rfc/rfc8445.
Accessed 29 September 2019.

29 Rosenberg, J, Mahy, R, Matthews, P and Wing, D. Session Traversal Utilities for NAT (STUN) [online].
URL: https://tools.ietf.org/rfc/rfc5389.
Accessed 29 September 2019.

30 Petit-Huguenin, M, Nandakumar, S, Holmberg, C, Keränen, A and Shpount, R. Session Description Protocol (SDP) Offer/Answer procedures for Interactive Connectivity Establishment (ICE) [online].
URL: https://tools.ietf.org/html/draft-ietf-mmusic-ice-sip-sdp.
Accessed 29 September 2019.

31 Ivov, E. Rescorla, E, Uberti, J and Saint-Andre, P. Trickle ICE: Incremental Provisioning of Candidates for the Interactive Connectivity Establishment (ICE) Protocol [online].
URL: https://tools.ietf.org/html/ draft-ietf-ice-trickle.
Accessed 29 September 2019.

32 Wing, D, Chesire, S, Boucadair, M, Penno, R and Selkirk, P. Port Control Protocol (PCP) [online].
URL: https://tools.ietf.org/rfc/rfc6887.
Accessed 29 September 2019.

33 Solowey, J, Zhou, H, Eronen, P and Tschofenig, H. Transport Layer Security (TLS) Session Resumption without Server-Side State [online].
URL: https://tools.ietf.org/rfc/rfc5077.
Accessed 29 September 2019.

34 Pre-shared key [online]. Wikipedia.
URL: https://en.wikipedia.org/ https://en.wikipedia.org/wiki/Pre-shared_key.
Accessed 29 September 2019.

35 Rescorla, E. Tschofenig, H and Fossati, T. Connection Identifiers for DTLS 1.2 [online].
URL: https://tools.ietf.org/html/draft-ietf-tls-dtls-connection-id.
Accessed 29 September 2019.

36 Rescorla, E. Tschofenig, H and Modadugu, N. The Datagram Transport Layer Security (DTLS) Protocol Version 1.3 [online].
URL: https://tools.ietf.org/html/ draft-ietf-tls-dtls13.
Accessed 29 September 2019.

37 Mahy R, Matthews P, Rosenberg J. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN) [online].
URL: https://tools.ietf.org/html/rfc5766.
Accessed 29 September 2019.

38 Keränen, A. Thin(g) ICE [online].
URL: https://datatracker.ietf.org/meeting/interim-2016-t2trg-03/materials/slides-interim-2016-t2trg-03-sessa-thin-ice.pdf.
Accessed 29 September 2019.

39 Holmberg, C. IoT NAT Traversal [online].
URL: https://github.com/t2trg/2017-09-berlin/blob/master/slides/33_T2TRG_Berlin_Thin-ICE.pdf.
Accessed 29 September 2019.