An overview of the CoAP protocol architecture, features and its extensions

Stefano Ivancich
Department of Information Engineering (DEI)
University of Padova, Italy
Email: stefano.ivancich@studenti.unipd.it
Student ID: 1227846

Fig. 1: Stefano Ivancich in 2021

***Abstract*—Today (2021), several billions of devices are connected daily to the internet, so a very huge amount of a data is sent and managed every second. In the next years, it is expected that even more devices like sensors, actuators for domestic or industrial usage will connect to the internet. In this context, we review an HTTP like protocol called Constrained Application Protocol (CoAP) that was developed by the Internet of Engineering Task (IETF) to work on constrained IoT devices operating in lossy environments. Although this protocol is lightweight and efficient compared to other IoT protocols such as HTTP or MQTT, it has some limitation in some scenarios, for example when strong reliability is needed or in streaming applications. For this reason we also overview, from the most recent literature some modifications, one is CoAP over TCP that guarantee reliability, another is CoAP-SC that has a good mechanism to handle with error and flow control that are crucial in streaming applications.**

***Index Terms*—CoAP, IoT, TCP, UDP, Streaming, enhancements in CoAP**

I. Introduction

The Hypertext Transfer Protocol (HTTP) [1] is one of the most widely used application layer protocol for distributed, collaborative, hypermedia information systems, which rules the communication between Web clients and Web servers. It's used mostly for transfer HTML documents between nodes. At its basis it is a textual request-response protocol where clients and servers exchange messages constituted by an header and an optional body. Is a stateless protocol, that basically means that each request-response is independent and neither the client nor the server has to keep trace of the exchanged messages. This simplifies the implementation of the protocol and makes it more scalable. And is designed to favors the use of intermediaries or proxies, typically for caching or security purposes. It is appropriate for devices with relatively high computational power, but often it can be too demanding in terms of bandwidth and processing requirements for the constrained devices that we are considering in the IoT scenario. So, a protocol called Constrained Application Protocol (CoAP) defined in the RFC 7252 [2] was developed as a lightweight counterpart of HTTP in order to simplify integration with the web but at the same time address the needs of constrained devices and constrained networks, indeed those kind of nodes usually carry 8-bit microcontrollers with very small amounts of ROM/RAM, while constrained networks generally suffer of high packet error rates and have an indicative throughput of 10s kbit/s. This protocol is used for machine-to-machine (M2M) communication and, as said before, is quite similar to HTTP, but with some relevant differences. It uses smaller and simplified headers, supports asynchronous REST, publish/subscribe message exchanges, provides support for multicast, extremely low overhead, and is easy to implement for constrained environments. The aim of CoAP is not to indiscriminately compress the HTTP protocol, but instead to provide a subgroup of REST in common with HTTP that are optimized for M2M communications.
In this work, we focus on the IoT communication pro-

tocol CoAP; in particular, we overview its architecture, features and extensions.
The report is structured as follows: in Section II we describe the communication problems in constrained networks, in Section III we give an overview of the CoAP protocol that tackle those problems, and in Section IV we describe it's message format, it's transmission and semantic. In Section V we describe the CoAP methods definitions, caching and proxying. In Section VI we review some extension and modifications of the standard CoAP protocol. And finally in Section VII we make some extra considerations on future developments and possible improvements.

II. Constrained Devices and Networks

Edge networks may operate in a huge variety of environments and conditions that requires a careful attention on their use. Usually, data flowing throw edge networks are ultimately destined for the Internet, so they should be designed with that transition in mind and therefore it makes some sense to emulate some ideas in IoT edge network design. The so-called Constrained networks are characterized as Low-Power and Lossy Networks (LLNs). Those kinds of networks are used often in combination with constrained devices that, have limited processing and powers supply, in Table 1 you can see a classification of constrained devices. LLNs are constrained because the cost, the limited nodes capabilities, limited power, limited spectrum, high density and interference. So they need smaller (compressed) headers and smaller payloads/packets to keep bit error rates low and permit media sharing. That's where lightweight communication protocols like CoAP comes into play.

III. Constrained Application Protocol (CoAP)

The way CoAP interact is very similar to the HTTP client/server model, but is common to use a node performing both client and server. A CoAP request is sent by a client to a server in order to request an action (specified with a Method Code parameter) on a particular resource identified by an URI. Then the server returns a response containing a Response Code, with the resource or an error message. CoAP differs from HTTP when dealing with those messages exchanges because it handles them asynchronously over the transport layer (in this case the UDP, in section VI we will describe how CoAP can also work over TCP). 4 types of message are supported by CoAP: Confirmable, Non-confirmable, Acknowledgement and Reset. Those messages can transport request or responses based on their Method Codes or Response Codes. Both request and responses can be confirmable or non-confirmable. We can think of CoAP as a two-layer protocol, one layer for messaging that deal with the transport layer and with asynchronous messages, and a Request/Response layer that manages request/response interactions through Method and Response Codes. You can see this in Fig. 2.

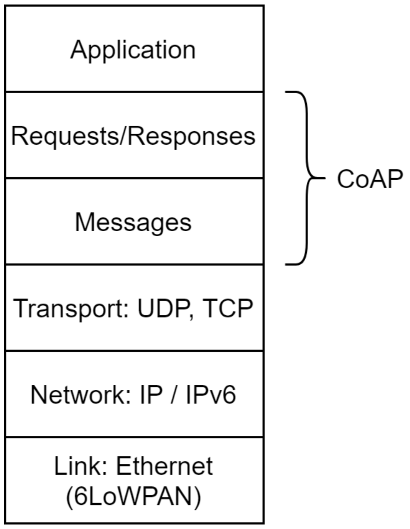

Fig. 2: Abstract Layering of CoAP

Now we are going to explain these 2 layers in depth.

A. Messaging Model

The message layer is the bottom layer of CoAP and it deals with the exchange of messages over the transport layer, in the base case the UDP datagrams, but in the following section we will see how it adapts to TCP. CoAP uses a binary header of fixed length (4 bytes), after which a compact binary options and the payload might follow. Every message in CoAP has a 16-bit unique ID used for the detection of duplicates and for reliability, allowing an exchange of up to 250 messages per second. A message can be sent reliably if its type is Confirmable (CON), a response will be provided with the Acknowledgement message (ACK) type carrying the same Message ID or with the Reset message (RST) if the recipient is not capable to handle/process the message that was sent. This process can be seen in Fig. 3.

A non-reliable message can be sent as Non-confirmable message (NON) that won't have any acknowledge response, but if the recipient has some troubles processing such message, it can reply with the Reset message (RST). Even though these kind of messages are

Name	Data size (eg. RAM)	Code Size (eg. ROM)	Functionality
Class 0, C0	$\ll 10KB$	$\ll 100KB$	Very constrained devices. Cannot communicate with the Internet directly.
Class 1, C1	~10KB	~100KB	IP and security capable, cannot easily communicate using full IP stacks, such as HTTP. May be able to use CoAP over UDP.
Class 2, C2	~50KB	~250KB	Support most protocol stacks.

TABLE 1: Classification of constrained devices

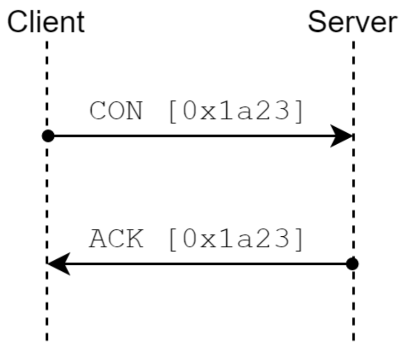

Fig. 3: Reliable message transmission

not reliable, they carry a unique ID. This process can be seen in Fig. 4. CoAP message types are summarized in Table 2.

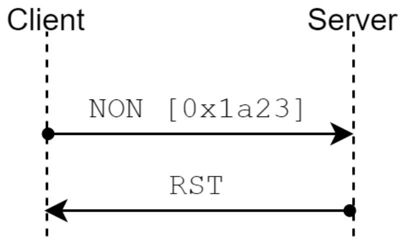

Fig. 4: Unreliable message transmission

B. Request/Response Model

The top abstract layer of CoAP is the Request/Response layer. Request and responses are carried by the CoAP messages, that includes a Method Code or a Response code. A Token is used to pair the response to the corresponding request. The request is sent by means of a Confirmable (CON) or a Non-confirmable (NON) message. If the request is sent through a Confirmable message, the server replies with an Acknowledgement (ACK) message with the resource or with an Error Code, this process is called piggybacked Response and can be seen in Fig. 5.

If the server is not capable to reply rapidly to the Confirmable message, it sends back and empty ACK

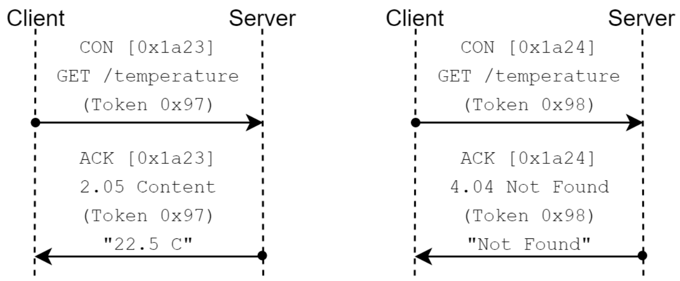

Fig. 5: Example of GET Requests that receive Piggy-backed Responses

message, otherwise the client will continue to retransmit the request. And when it's ready to respond, it sends a Confirmable message including the content that must be confirmed with an ACK message from the client. This mechanism is called ”separate response” and can be seen in Fig. 6.

Instead, if the request is sent by using a Non-confirmable message (NON), the response is sent also as Non-confirmable that you can see in Fig. 7.

C. Intermediaries, Caching and Resource Discovery

To fulfill request efficiently CoAP offer the caching of responses. Since we are working on constrained networks, CoAP also offer the possibility of creating a proxy, this is due to restrain network traffic, boost performances, to access to the data of devices in sleep mode, and to provide security. Like in HTTP proxying, the destination IP is the proxy address while the resource's URI is inside the request. There is also the possibility to map CoAP to HTTP and the other way around. This conversion can be realized by a cross-proxy (a cross-protocol proxy) that converts Method/Response codes, options o the corresponding in HTTP. As final note of this section, since in the context of M2M interactions Resource Discovery is pretty important, CoAP support it using the CoRE Link Format [3].

Message type	Description
Confirmable	Reliable message delivery, the recipient is required to confirm with an acknowledgment
Non-Confirmable	Not acknowledged, delivery is not guaranteed
Reset	Indicates that a message was received, but the receiver is not capable to process/handle it
Acknowledgment	Indicates that the message was received and processed correctly

TABLE 2: CoAP message types

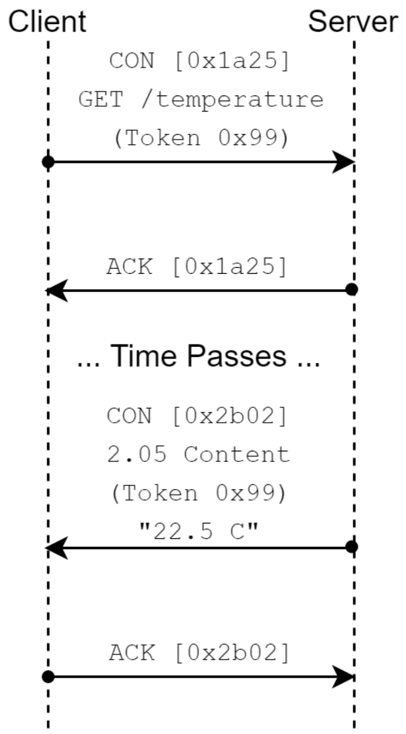

Fig. 6: A GET Request with a Separate Response

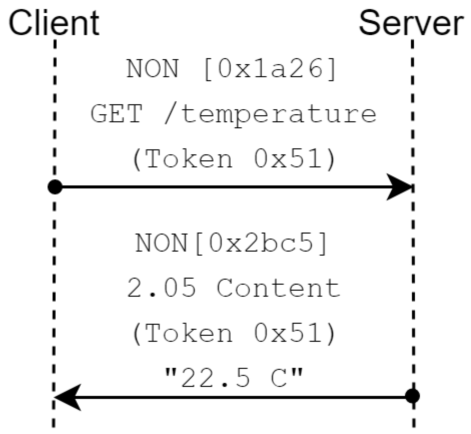

Fig. 7: Request and Response sent by Non-confirmable Messages

IV. Message Format, Transmission and Semantic

In this section we are going to cover the CoAP message format. Since this protocol was created to work on constrained networks, it implements compact messages and, to avoid the fragmentation, it uses the data section of just one UDP datagram, but it can also be transported over TCP.

A. Message Format

The message format is composed by a 4-byte header, followed by a Token value which length is stated in the TKL header field (0 to 8 bytes), as said before the role of this token is to match requests with responses. Then a sequence of 0 or more options in the Type-Length-Value (TLV) format. An Option might be followed by other Options, by the end of message, or by the $0xFF$ (the Payload Marker) which express the end of the options section and the start of the payload which may take the remaining datagram size. This format can be seen in Fig. 9. In particular the header fields are defined as follow:

- Version (Ver): 2-bit unsigned integer.
- Type (T): 2-bit unsigned integer. 0=Confirmable, 1=Non-confirmable, 2=Acknowledgement, or 3=Reset
- Token Length (TKL): 4-bit unsigned integer. Represents the length of the variable-length Token field which size can be between 0 and 8 bytes. Lengths 9-15 must be reserved, not sent, and are interpreted as a message format error.
- Code: 8-bit unsigned integer, divided into a 3-bit class (most significant bits) and a 5-bit detail (least significant bits). Class are 0=request, 2=success response, 4=client error response, or 5=server error response. For example 0.00=Empty message. In the case of a request, this field express the Request Method; while in case of a response, it express a Response Code.
- Message ID: 16-bit unsigned integer in network byte order.

The option format requires to each option instance to specify its Option Number, its Value field length and the Option Value itself. More than one instance of the same option can be set by specifying an option delta of zero. We don't go further on this concept because its quite complicated and long, we limit ourselves to show the structure in Fig. 8, for further information, read the RFC7252 [2].

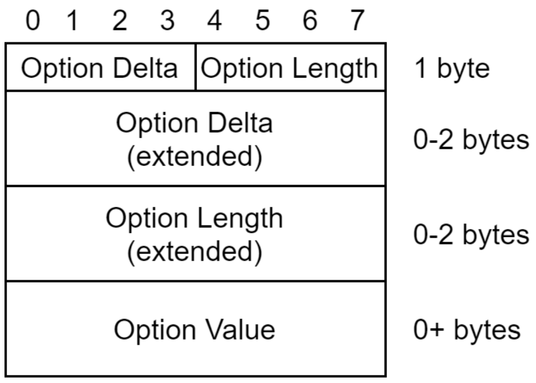

Fig. 8: CoAP Message Option Format

B. Message Transmission

CoAP messages are used to carry request and responses between CoAP endpoints, and they are exchanged asynchronously. Since UDP is not a reliable transport protocol, those messages may not arrive in order, they can be missing or can be duplicated. So, CoAP uses a small mechanism to assure reliability that can detect duplicates for both Confirmable (CON) and Non-Confirmable (NON) messages, in particular, implements for Confirmable messages a basic reliability system based on stop-and-wait retransmission with exponential back-off.

Messages and Endpoints: an endpoint, identified by an IP address plus an UDP port number, is the destination or the source of a CoAP message. As we said, there are various types of messages, that can transport a request, a response or are empty. If a message is empty, it's Code field is set to 0.00, the Token Length field (TKL) to 0 and if there are bytes present in the payload, they are interpreted as error message.

Messages Transmitted Reliably: to transimt with reliability the message has to be marked as Confirmable (CON) in the proper header field. This type of message always carries a request, a response or a Reset, so a receiver must return an ACK or reject it. The rejection of a confirmable message contains the Message ID and the payload is empty. The sender keeps retransmitting the message at increasing intervals until it receives an ACK, a RST or reach the limit of attempts. At start, the timeout is initialized to a random duration between `ACK_TIMEOUT` (default is 2 seconds) and (`ACK_TIMEOUT` * `ACK_RANDOM_FACTOR` (default is 1.5)), and the retransmission counter starts from 0. When the timeout is reached and the retransmission counter is less than `MAX_RETRANSMIT` (default is 4), the message is sent again, the counter for retransmission is incremented, and the timeout is doubled. This continues until the counter reach `MAX_RETRANSMIT` or the sender receive a RST message.

Messages transmitted without reliability: usually this are messages that are sent regularly, like data coming from a sensor where the eventual transmission success is enough. Those kinds of messages can be sent marking the proper header field as Non-Confirmable (NON) message. This kind of messages must not be empty.

C. Message semantic

CoAP uses an analogous request/response model of HTTP, basically a client might sends several CoAP requests to a server, which sends back CoAP responses. But, unlike HTTP, those request and responses are not sent after a pre-established connection, there are sent asynchronously.

Requests: consist of a method to be applied to a specific resource, the resource identifier, the payload and the internet media type (if any), and the request's metadata (optional). The methods supported by CoAP are GET, POST, PUT, and DELETE. They are safe (only on retrieval) and idempotent (when invoked multiple times have the same effect) like for the equivalent HTTP.

Responses: they are matched with the request by a Token generated by the client. They can be identified using the Code field in the header. Like for the HTTP Status Code, in the same way the CoAP Response Code express the result of trying to satisfy the client request. The (8-bit) Response Code (Fig. 10) define the class (3-bit) and the detail (5-bit) of the response. There are 3 main classes of Response Code:

- 2 (Success): indicates that the request has been successfully received, understood and accepted.
- 4 (Client Error): the request cannot be fulfilled or contains syntax that the server is not capable to understand.
- 5 (Server Error): the request is valid but the server is not able to fulfill it.

V. CoAP Methods, Caching and Proxying

A. Method definitions

Resources are organized with a hierarchy and directed by a CoAP origin server that listen for CoAP requests ("coap" or "coaps"). CoAP URI scheme is defined as

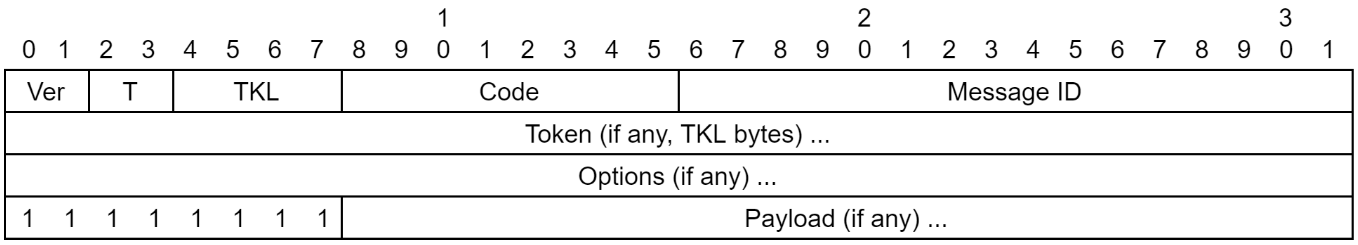

Fig. 9: CoAP Message Format

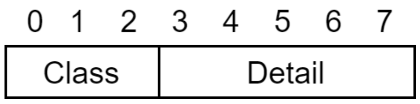

Fig. 10: Response Code structure

follows: coap-URI = ”coap:” ”//” host [”:” port] path-abempty [”?” query]
Where HOST must not be empty and PORT indicates the UDP port, if it's empty the default 5683 is used.
CoAP implements methods, very similar to HTTP, with which clients can call an action on a specific resource identified the previously mentioned URIs. A request with a method that is not supported, returns the 4.05 (Method Not Allowed) response. The methods supported by CoAP are:

- GET: receives a representation of the resource that is identified by the URI. If this is possible, the server returns a 2.03 (Valid) or 2.05 (Content) Response Code. This method is safe (it MUST NOT take other action on a resource other than retrieval) and idempotent.
- POST: results in a creation, update or deletion of a resource. If a resource is created, the server returns s 2.01 (Created) Response Code with the URI. If there is an update of the resource, the server returns a 2.04 (Changed) Response Code. Instead, if a resource is deleted, the server returns a 2.02 (Deleted) Response Code. POST is not safe and not idempotent. Is not idempotent because its effect is established by the origin server and dependent on the target resource.
- PUT: result in the creation or update of the resource specified by the URI given. Its response codes are the same of POST. PUT is idempotent but not safe.
- DELETE: results in the deletion of the resource identified by the URI given. If succeed, a 2.02 (Deleted) Response code is returned. DELETE is idempotent but not safe.

The difference between POST and PUT is that PUT is generally used to replace the existing content of a resource, while POST is used to send new data. Some other IoT frameworks uses the use of CRUD methods (CREATE, RETRIEVE, UPDATE, DELETE) to overcome this ambiguity.
A brief summary of CoAP methods can also be seen in Table 3.

B. Caching

We just briefly note that CoAP endpoints have the possibility to cache the responses in order to reduce the time of response and bandwidth usage. The main purpose of CoAP caching is to reuse a response message to fulfill a current request that is the same of a past one. The cacheability of a response is determined by the Response Code.

C. Proxying

Proxies can serve many different purposes; we can distinguish them in 3 main types:

- Forward-proxies: CoAP request on such proxies can be made as Confirmable (CON) or Non-Confirmable (NON), but the request URI is specified in the Proxy-Uri Option instead of being split in URI-Host, URI-Port, URI-Path and URI-Query Options. If the endpoint is not able to act as a proxy, it returns a 5.05 (Proxying Not Supported) response.
- Reverse-Proxies: offer various resources as if they were its own resources.
- Cross-Proxies: can translate a CoAP request/response to a different protocol.

More specifically, a particular kind of cross-proxying is of interest in IoT when using CoAP, that is the Cross-Protocol Proxying between CoAP and HTTP, that's because CoAP support a limited subset of HTTP functionalities. Only the request/response model of CoAP is mapped to HTTP. The Confirmable/Non-confirmable messages model is invisible and has no effect on a proxy function.

Methods	**Description**
GET	Retrieves a representation of information corresponding to the specified URI
PUT	Requests to update the identified resource be updated with the enclosed representation, or create new
POST	Request to process the enclosed representation
DELETE	Delete the resource identified by the URI

TABLE 3: CoAP methods

CoAP-HTTP Proxying: enables CoAP clients to access resources that are hosted on a HTTP server. When sending a request to the CoAP-HTTP proxy, a CoAP client has to set the Proxy-URI or Proxy-Scheme Option to ”http” or ”https”. Since the basic methods of CoAP are very similar to HTTP, performing a request is not much different. If the proxy is not able to service the request with the HTTP URI, it returns a 5.05 (Proxying Not Supported) response. If the proxy is unable to get the requested resource by the HTTP server in a given timeframe, it returns a 5.04 (Gateway Timeout) response. If the resource is not understood, it returns a 5.02 (Bad Gateway) response. The response payload is a representation of the resource, and the Content-Format Option should be set accordingly.

HTTP-CoAP Proxying: enables HTTP clients to access resources hosted on CoAP servers. When sending a request to the HTTP-CoAP proxy, an HTTP client has to set the Request-Line to ”coap” or ”coaps”. If the proxy is not able to service the request with the CoAP URI, it returns a 501 (Not Implemented) response. If the proxy is not able to get the requested resource by the CoAP server in a given timeframe, it returns a 504 (Gateway Timeout) response. If the resource is not understood, it returns a 502 (Bad Gateway) response. Since the methods OPTION, TRACE and CONNECT are not implemented in CoAP, if those one are called, the proxy returns a 501 (Not Implemented) error.

VI. CoAP ENHANCEMENTS

In this section we are going to review some enhancements and extensions of the standard CoAP protocol. In particular we explore the CoAP over TCP and CoAP for streaming.

A. CoAP over TCP

The standard CoAP operates above the User Datagram Protocol (UDP) accomplishing lightweight messaging. But the main downsides are that UDP cannot provide reliability and some networks, especially enterprise networks, do not froward UDP packets. For those reasons and that the demand for the use of TCP in IoT is increasing, CoAP over TCP was proposed in [4]. TCP has congestion control and flow control mechanism, more sophisticated that the one of CoAP over UDP, but uses a larger packet size, more round trips, and increased RAM requirements. The main difference between CoAP over TCP and over UDP is on the message layer, while the request/response model remains the same.

Messaging Model: since TCP provides reliable transmission, CoAP Confirmable and Acknowledge message are no longer needed. So, the Message ID field and the type field in the header are not present anymore, instead, 2 fields that indicate the length of the message (Length and Extended Length) are present because TCP does not provide this information.

Message Format: is very similar of the one over UDP, except for Type, Message ID and Version fields that are removed as previously said. This format can be seen in Fig. 11.

Message Transmission: after the TCP connection is established, the CoAP endpoints send a CSM (Capabilities and Settings Message) as first message of the connection. This special message initializes the settings and capabilities of the endpoints, if there are no options set in this message, the defaults are used. Request and response are sent asynchronously over the Transport Connection. So, a client can send multiple requests without waiting for responses, and those responses can be returned in any order but in the same connection. The TCP protocol is bidirectional, this implies that requests and responses have the capability to be sent by both endpoints. TCP also support retransmission and duplication of messages.

B. CoAP for Streaming

In the last years, a variety of streaming application in IoT are becoming more common. In this paper [5] the authors explore the usage of the CoAP protocol for this particular type of flow of data. The conventional CoAP over UDP or CoAP over TCP protocols can be used for reliable services as we seen. However, they do not implement error handling and flow controls suitable to help the transmission of streaming data at the sender, this

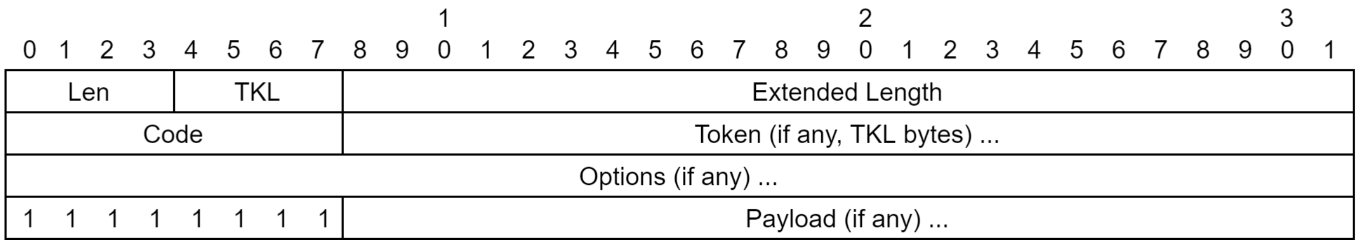

Fig. 11: CoAP over TCP Message Format

leads to a degradation of the throughput performances, especially in sensor wireless lossy networks. For example, if in CoAP over UDP a message gets lost, its retransmission will happen after a timeout, making the error recovery to increase transmission delays. In CoAP over TCP the packet can be recovered quickly using the TCP fast recovery, but this mechanism adds overhead in the IoT environment. Moreover, the complexity of TCP is not very suitable for real time streaming in IoT environment. To overcome these issues, the authors in [5] proposed a streaming control based on the CoAP, called CoAP-SC, which extends the CoAP over UDP protocol that enhance the throughput by adding error handling and flow controls mechanism. The scheme is designed by assigning a sequence number (SN) to each data message, and an ACK number (AN) is returned by the receiver.

Initialization for CoAP-SC: with a POST message the sender requests the creation of a new resource, this request include authentication information, buffer size and other parameters correlated with the streaming service. After the creation of the resource, the receiver returns a 2.01 response message with the URL of this new resource. Then, the sender sends a GET request to that URL, and the receiver responds back with a 2.05 response message. All GET and their response messages contains a sequence number (SN) and ACK number (AN), where SN is sequentially assigned on each message sent by the sender, and the AN is set by the receiver to tell that that message was received correctly. The AN is cumulative. In the initialization process SN and AN are set to 0. This initialization process can be seen in Fig 12.

Error Handling for CoAP-SC: the first message sent by the sender has SN=1 and AN=0. Every time the receiver gets a message, it updates its own AN number to be equal to the highest SN value that has been received and cumulatively. So, if the receiver is not getting any data message for a certain amount of time, it sends an ACK message to tell the sender the

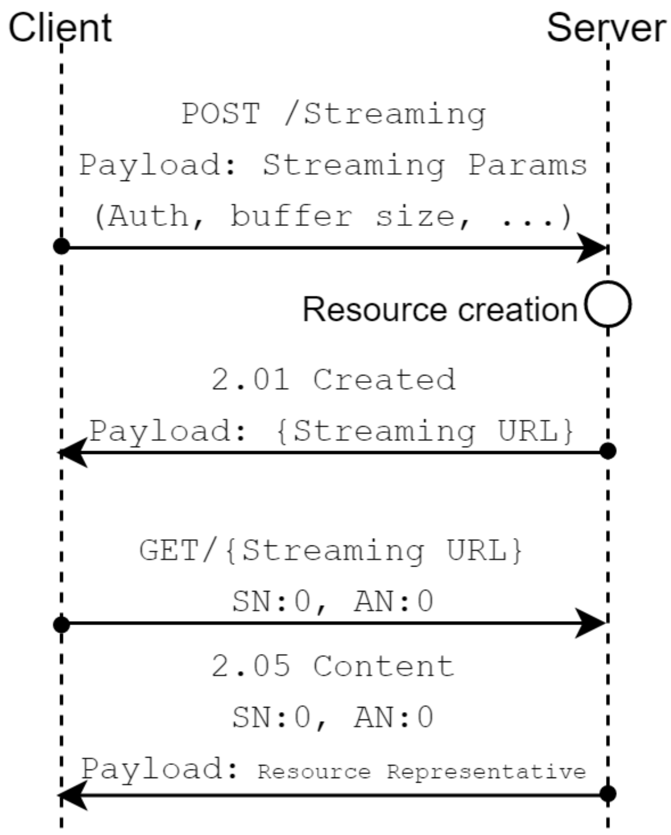

Fig. 12: CoAP-SC Initialization

AN status. In the normal streaming flow, the recipient will receive a message that has $SN = AN_{sent} + 1$, and the next message it will send, will have $AN = SN_{received} = AN_{sent} + 1$. So, if a message gets lost, the receiver will determine that by checking if this condition is true: $SN_{received} - AN_{sent} > 1$. When this message loss is found, the receiver sends an ACK message for a retransmission request that includes the SN of the message to be transmitted, and this ACK is retransmitted until the data lost is correctly received. An example of this mechanism can be seen in Fig. 13.

Flow Control for CoAP-SC: ACK messages are used also for flow control, by providing the newest AN information to the sender, in this way facilitating it to

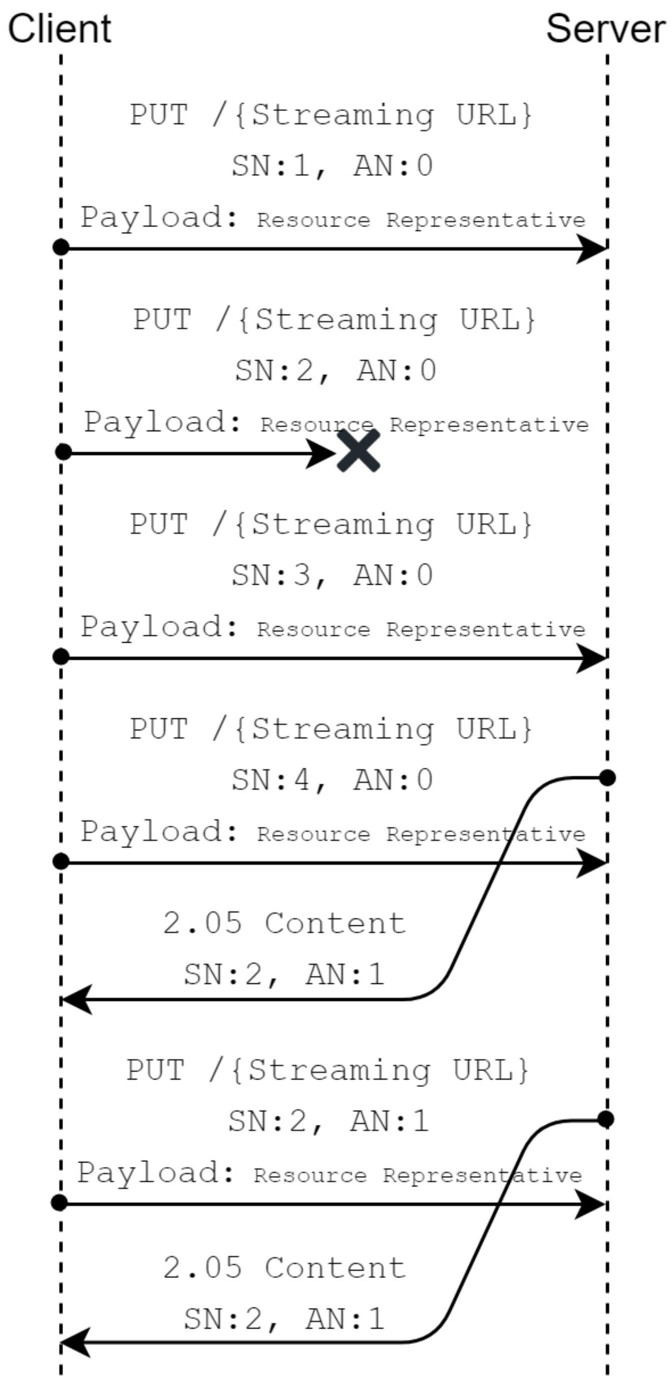

Fig. 13: CoAP-SC Error Handling example

transfer as much data as possible, resulting a throughput enhancement.

CoAP Option for CoAP-SC: the Option header is composed by 4bit Option Delta, 4-bit Option Length, 4-byte Sequence number and 4-byte ACK number. This option format can be seen in Fig. 14.

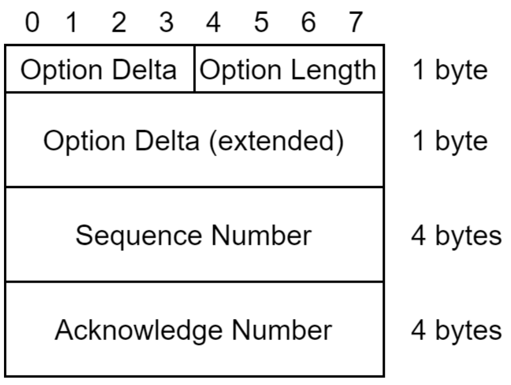

Fig. 14: CoAP-SC Message Option Format

VII. Concluding Remarks

In this paper an overview of the CoAP protocol has given. Since the basic architecture of CoAP may perform inefficiently in some scenarios like when strong reliability is needed and in the case of streaming applications, we proposed possible solutions for those problems taken from the most recent literature. In particular for reliability, a RFC called ”CoAP over TCP” that as its title says, modifies CoAP to work on the reliable transport protocol TCP. While for the streaming scenario, a modification called CoAP-SC was reviewed, that propose good a mechanism to deal with errors and flow control that are crucial in streaming. However, it is quite clear that this Streaming scheme needs some other modifications to reduce the packet size to be applied in the IoT environment.

VIII. Acknowledgment

All images are made by Stefano Ivancich.
This work is presented for the final grade of the ”Internet of Things and Smart Cities” course, held by prof. Lorenzo Vangelista at the University of Padova in 2021.

References

[1] M. Belshe, R. Peon, and M. Thomson, “Hypertext Transfer Protocol Version 2 (HTTP/2).” RFC 7540, May 2015.
[2] Z. Shelby, K. Hartke, and C. Bormann, “The Constrained Application Protocol (CoAP).” RFC 7252, June 2014.
[3] Z. Shelby, “Constrained RESTful Environments (CoRE) Link Format.” RFC 6690, Aug. 2012.
[4] C. Bormann, S. Lemay, H. Tschofenig, K. Hartke, B. Silverajan, and B. Raymor, “CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets.” RFC 8323, Feb. 2018.
[5] J.-H. Jung, M. Gohar, and S.-J. Koh, “Coap-based streaming control for iot applications,” *Electronics*, vol. 9, no. 8, 2020.