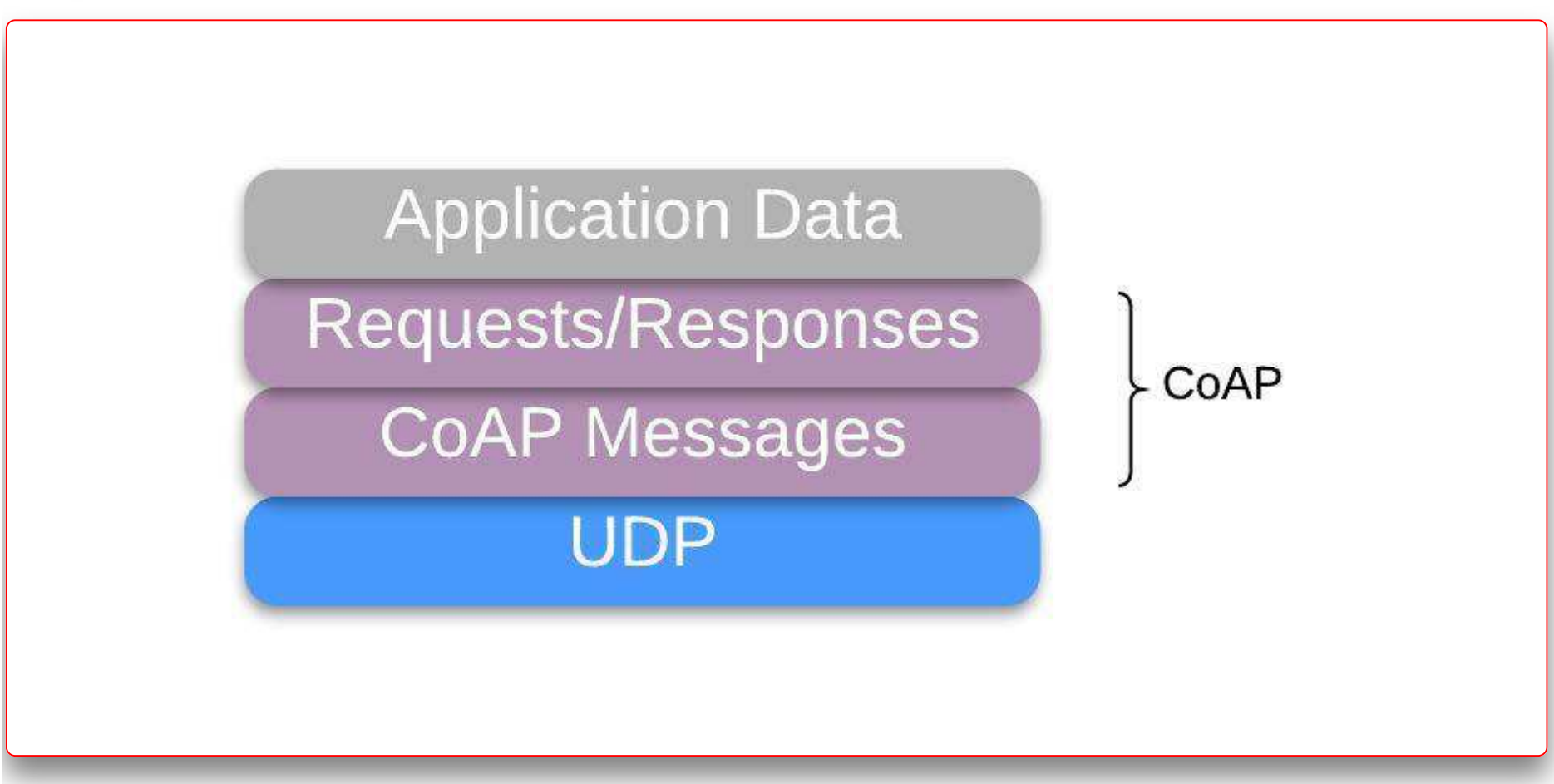
Payatu

IoT Security - Part 11 (Introduction To CoAP Protocol And Security)

Aseem

27-July-2020



This blog is part of the IoT Security series where we discuss the basic concepts pertaining to the IoT/IIoT eco-system and its security. If you have not gone through the previous blogs in the series, I would urge you to go through those first. In case you are only interested in CoAP, feel free to continue.

IoT Security - Part 1 (101 - IoT Introduction And Architecture)

IoT Security - Part 10 (Introduction To MQTT Protocol And Security)

In this blog, we are going to look at CoAP, a simple IoT protocol used for constrained environments, and security issues and attacks on CoAP protocol and its implementations. The power of CoAP is its simplicity and portability with HTTP protocol which will be evident when we discuss the protocol internals.

1 Introduction

CoAP is short for **Co**nstrained **A**pplication **P**rotocol. It is an IETF standard and the core protocol is defined in RFC 7252. There are further extensions that are defined in separate RFCs. It is well suited for nodes that run on simple microcontrollers, with limited ROM and RAM, and communicate over Low-Power Wireless Area Networks (LPWAN) e.g. 6LoWPAN. It works at the

application layer of the TCP/IP stack and utilizes UDP as the underlying transport protocol. In 2018 a new standard - RFC 8323 was released which describes CoAP over TCP, TLS and WebSockets.

UDP (or TCP) Port	Communication Type
5683	Plain text
5684	DTLS (or TLS over TCP)

1.1 CoAP Features

1. It provides a simple discovery mechanism
2. Integration with Web is easy
3. It provides asynchronous message exchange
4. Uses URIs to define resources/services
5. Uses REST-like request/response model

1.2 HTTP Lineage

CoAP was designed as a generic web protocol for constrained environments. It utilizes parts of HTTP more specifically the REST architecture in a compressed binary format. It can be easily integrated with the Web owing to simple translation/proxying between CoAP and HTTP. Think of it as a Poor man's HTTP. If you look at the HTTP protocol, it is quite extensive and Chatty owing to its text-based semantics. When working with constrained devices chatty protocols are not a good choice for long battery life and processing power. You will get a better idea when we discuss CoAP architecture and communication.

1.3 Use and Popularity

From our experience with IoT devices, CoAP functionality is provided on the RF side in various RF protocols. It is not quite as popular as MQTT for Ethernet/WiFi-based devices. However, you may still end up finding products that use CoAP for communication over WiFi/Ethernet.

2 CoAP Communication

2.1 Protocol Overview

CoAP uses the Client-Server communication model where nodes send requests and receive responses from other nodes. It uses a two-layer approach for communication where one layer deals with the UDP protocol and the other deals with the application data: 1. CoAP Messages - These define the type of CoAP packets and deal with UDP. 2. Requests/Response (Similar to HTTP) - These encapsulate the actual application payload/data using Request Methods and Response Codes.

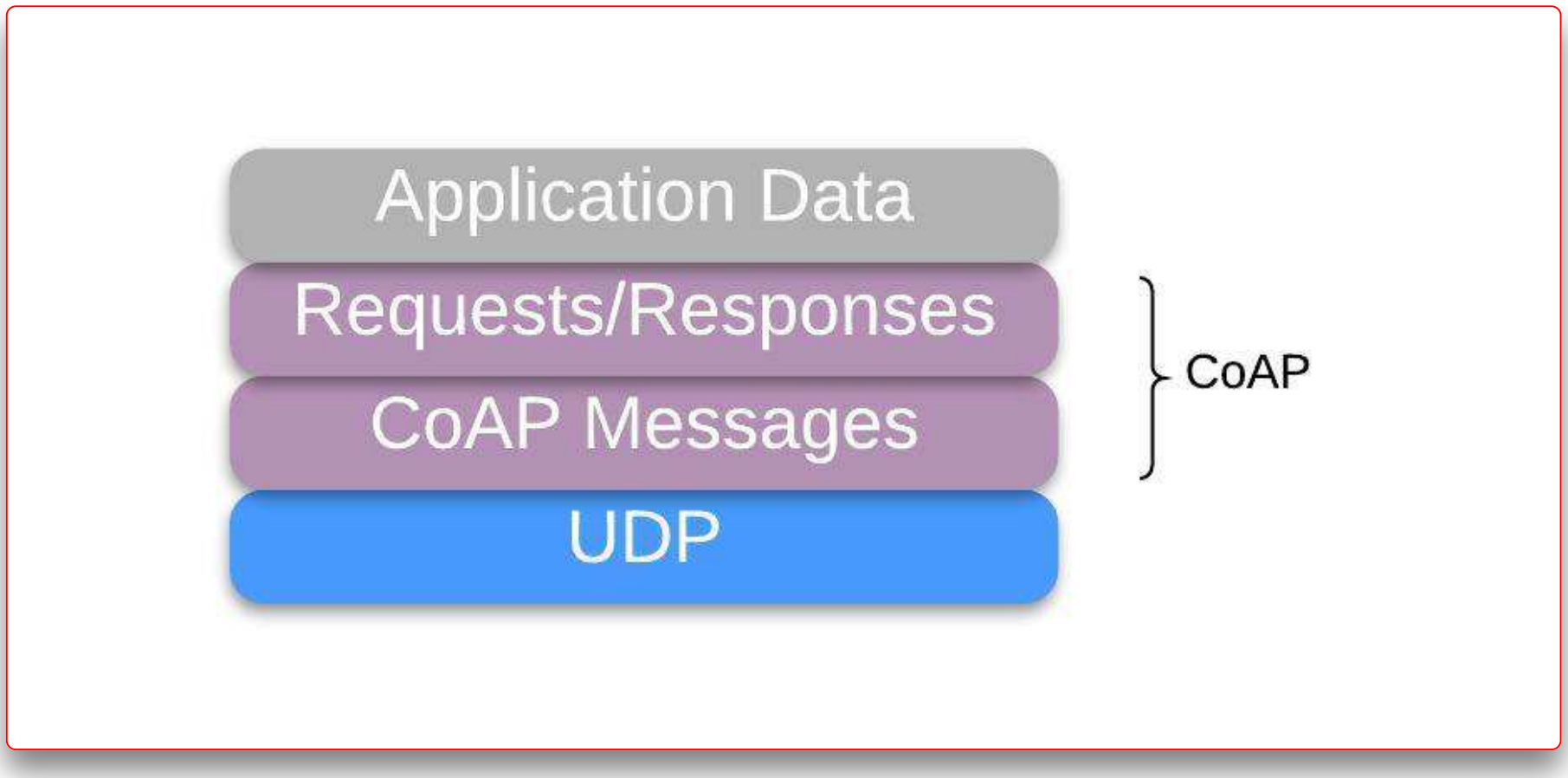


Fig 1. CoAP Protocol Layers

As we can see, CoAP uses UDP for message transfer, and encapsulates the request/response, application data in the messages. Since it uses UDP, it has to manage the reliability part. It does that using ACK messages.

2.1.1 CoAP Messages

The CoAP standard defines 4 different types of Messages: 1. Confirmable Message (CON) - This type of message requires that the receiver send an Acknowledgment message back to the sender for confirmation of the receipt of this message 2. Non-Confirmable Message (NON) - Does not require any ACK. This is used when no Acknowledgment is required i.e. reliability is not important.
3. Acknowledgment Message (ACK) - This message is sent, by a receiver, as an Acknowledgment for a Confirmable message that is received from the sender 4. Reset Message (RST) - This is message is typically sent when the receiver is not able to process a Confirmable or Non-Confirmable message due to some error.

Each message contains a 16-bit message ID to uniquely identify a message and map its corresponding ACK if any. This is typically used for identification and deduplication of messages as with other protocols as well. Below are some simple sequence diagrams that explain basic message exchange:

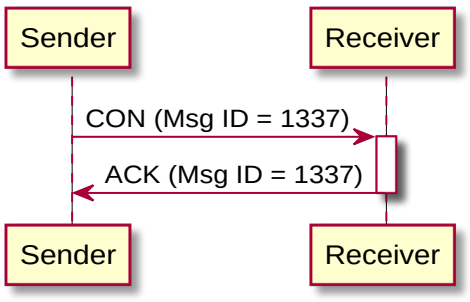


Fig 2. Confirmable Message Communication

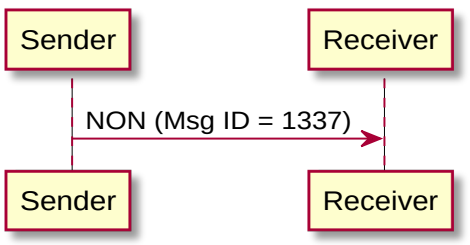


Fig 3. Non-Confirmable Message Communication

2.1.2 CoAP Request/Response

The Messages can contain a request, response, or maybe empty. There is no relation between the request/response and the message type other than the fact that some combinations are not possible or define some specific purpose. Let's first look at the Request and Response format and then delve deeper into the packet format.

1. Requests use Method Codes and Responses use response codes similar to HTTP but defined in binary format.

2. CoAP standard defines four Request methods - GET, PUT, POST, and DELETE.
3. CoAP standard defines Response code similar to HTTP - 2.xx for success, 4.xx for client error, and 5.xx for server error.
4. Unique 0-8 byte Tokens are used for identifying, mapping request and response much the same way Message-IDs are used to identify messages and map ACKs.
5. A response to a request sent in a CON message can be piggy-backed in the ACK message or may be sent as a separate CON or NON message.

Let's look at some examples of request/response communication to get a better idea about how applications exchange data. The first figure shows a request in a Confirmable message with the response piggybacked in the Acknowledgment Message.

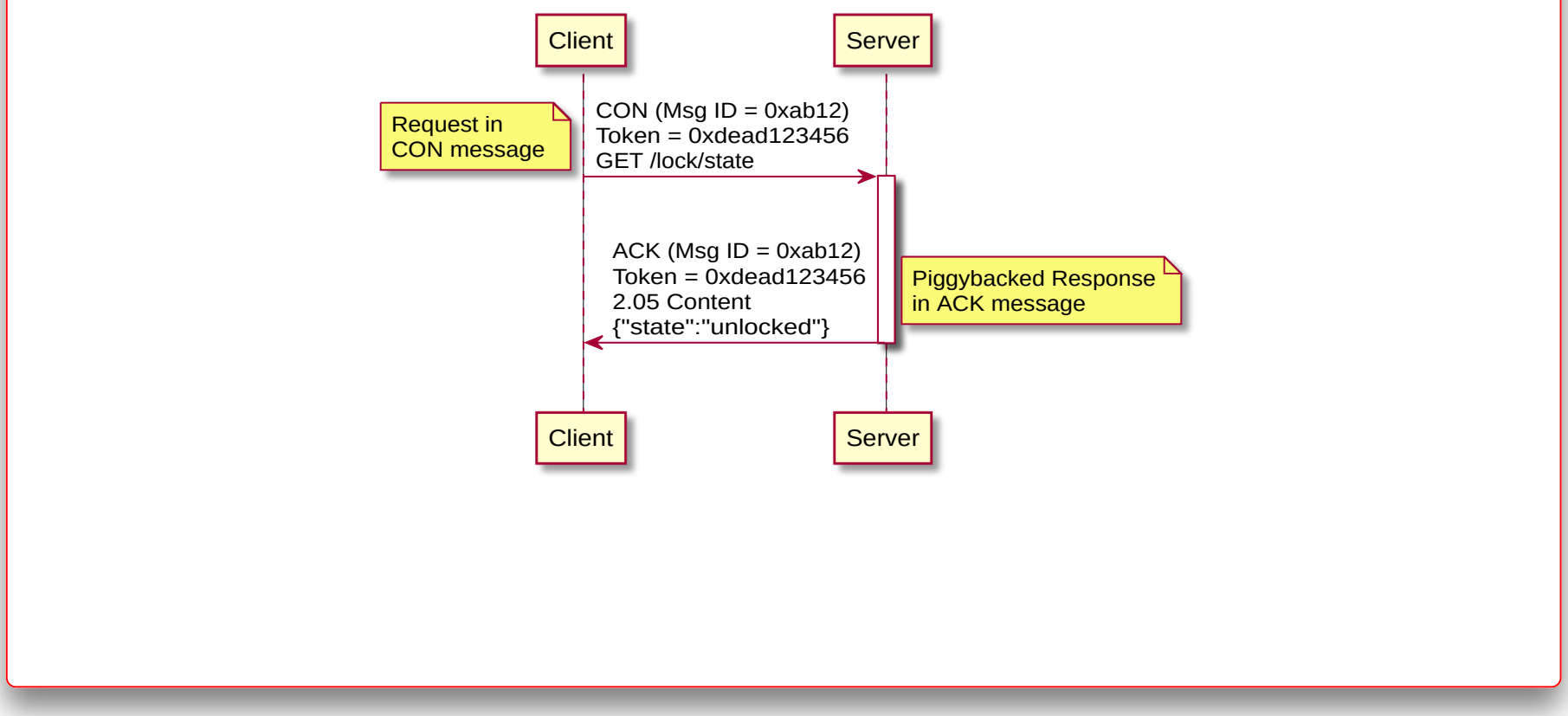


Fig 4. CON Message Request with Piggybacked Response

In the below figure, you can see the lifetime and mapping for Message-IDs and Tokens when request and response are sent in separate Confirmable messages and the Acknowledgment Messages are empty.

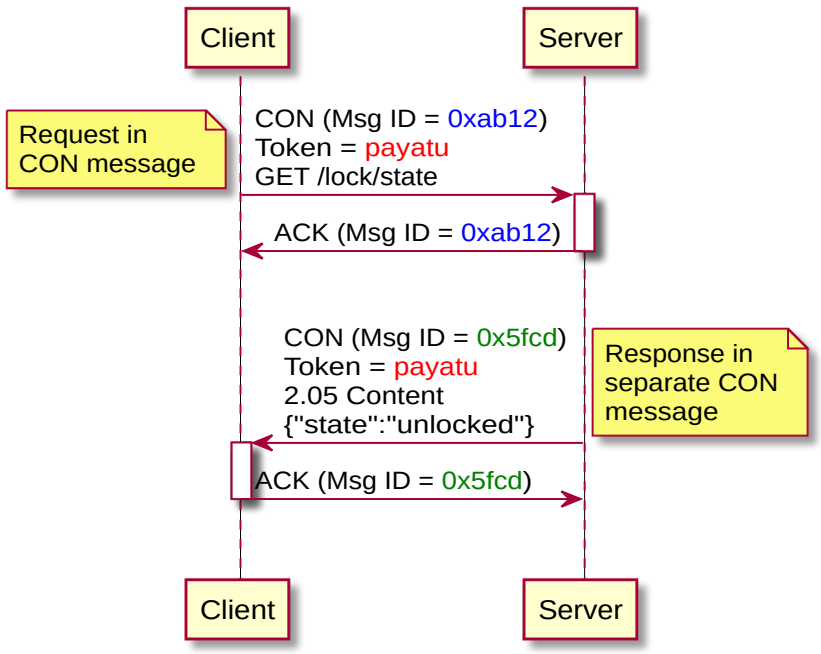


Fig 5. Separate CON Messages for Request and Response

The below figure shows an example when requests and responses are sent in separate Non-Confirmable messages.

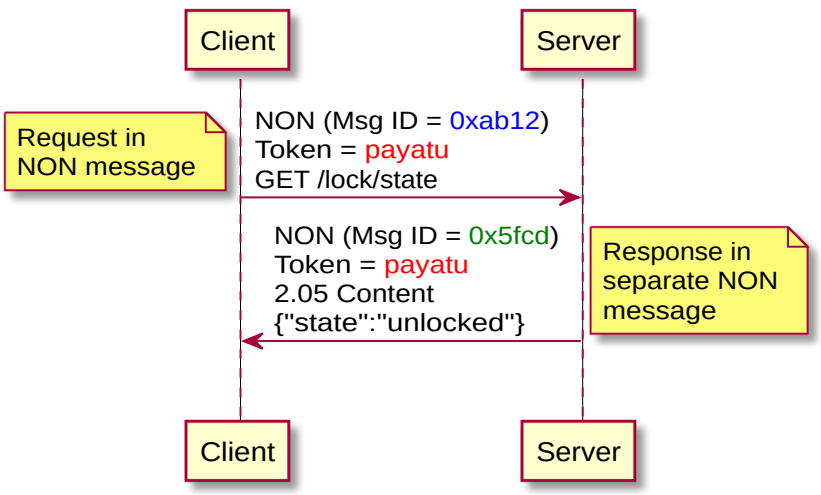


Fig 6. NON Message Request and Response

Of course, there are combinations of Message and Request/Response which are not allowed by the standard or have a specific meaning. The below table shows what is possible and what is not.

	CON	NON	ACK	RST
Request	Yes	Yes	No	No
Response	Yes	Yes	Yes	No
Empty	CoAP Ping	No	Yes	Yes

CoAP ping as suggested in the protocol can be used for heartbeat/health check of the server. A CoAP ping communication works as below:

1. The sender sends an empty Confirmable message.
2. The receiver responds with an empty Reset message.

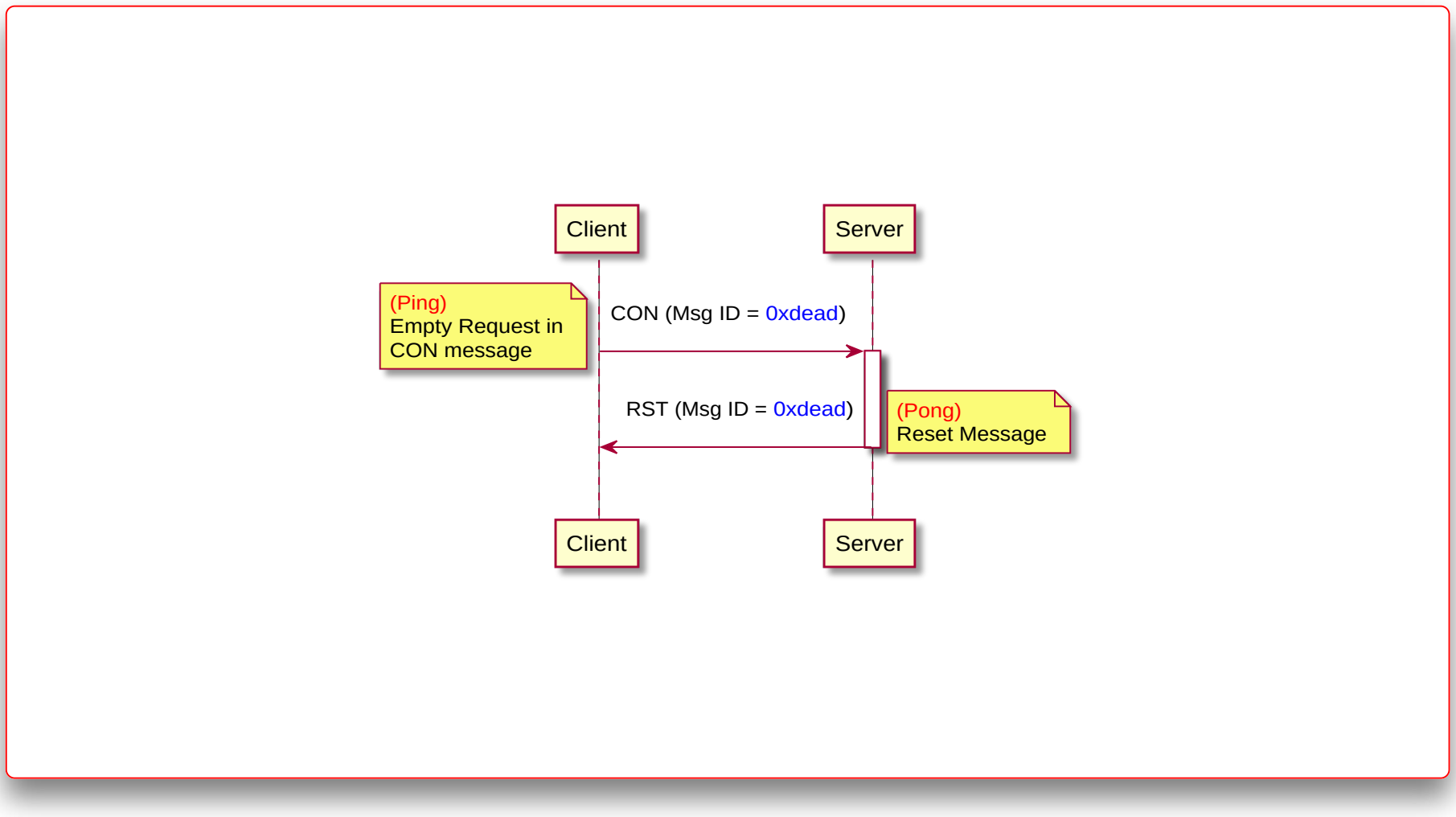


Fig 7. CoAP Ping Communication

2.1.3 CoAP Request and Response Codes

The requests and responses are identified by codes. The response codes are derived from HTTP. The code field is divided into two parts:

1. 3-bit class
2. 5-bit detail

The below tables describe the codes, denoted as c.dd format where c is the class and dd is the detail:

Request Codes

Code	Description
0.00	Empty message (Special Case)
0.01	GET

Code	Description
0.02	POST
0.03	PUT
0.04	DELETE

Response Codes

Code	Description
2.01	Created
2.02	Deleted
2.03	Valid
2.04	Changed
2.05	Content
4.00	Bad Request
4.01	Unauthorized
4.02	Bad Option
4.03	Forbidden
4.04	Not Found
4.05	Method Not Allowed
4.06	Not Acceptable
4.12	Precondition Failed
4.13	Request Entity Too Large
4.15	Unsupported Content-Format
5.00	Internal Server Error
5.01	Not Implemented
5.02	Bad Gateway
5.03	Service Unavailable
5.04	Gateway Timeout
5.05	Proxying Not Supported

2.1.4 CoAP Options

You might be thinking that we have discussed a lot about the Request/Response model and how CoAP is similar to HTTP. But there is still not much resemblance with HTTP other than request and response line, right? This section talks about the Options field of the CoAP protocol which is somewhat similar to HTTP headers albeit in binary format. Options define metadata that is sent to the other endpoint much like the data that is sent using headers such as host, content type, etc. Each Option field defines the option number as defined by the standard, its length, and its value. Let’s look at some of the options defined in the standard, which might sound familiar:

No.	Name	Format	Length(Bytes)	Description
3	Uri-Host	String	1-255	The hostname/IP address of the CoAP Server on which the resource is being requested
7	Uri-Port	uint	0-2	The UDP port no. of the CoAP server
11	Uri-Path	string	0-255	Each Uri-Path specifies a single segment of the absolute resource path of the URI
12	Content-Format	uint	0-2	ID of the content format of Application data (IDs are defined by the standard)
15	URI-Query	string	0-255	Each Uri-Query specifies single parameterized argument in the query part (query string) of the URI
17	Accept	uint	0-2	Indicates which Content-Format is acceptable to the client.

Following are the Content-Format types and their respective IDs defined by the standard:

Type	ID
text/plain;charset=utf-8	0
application/link-format	40
application/xml	41
application/octet-stream	42
application/exi	47
application/json	50

2.2 Discovery Mechanism

As with many other IoT protocols, CoAP also has a discovery mechanism. The discovery mechanism a very important aspect of m2m communication that aids automation without the need for human interaction. CoAP discovery is geared towards identifying the resources available on a CoAP server. It is supported using the **Co**nstrained **RE**STful Environments (CoRE) Link Format protocol RFC 6690. Cutting the long story short a CoAP Link Format of resources is supported on a CoAP server and queried by a client to identify the resources available on the server. This is performed by sending a GET request on /.well-known/core resource path as specified in the standard. When a server receives the request, it responds with a list of all the resource URIs available on the server.

Given below is an example taken from the standard itself. The response contains the comma-separated list of resource URI entries available on the server. Each entry has:

1. The resource URI in angular brackets ex. </sensors>
2. URI attributes separated by semicolons ; ex. title, rt, if.

Request GET coap://server:port/.well-known/core

Response

```
2.05 Content
</sensors>;title="Sensor Index",
</sensors/temp>;rt="temperature-c";if="sensor",
</sensors/light>;rt="light-lux";if="sensor"
```

3 CoAP Security

Now that we understand how CoAP works, let's focus on the security aspect of CoAP both from an offensive as well as defensive perspective. All the information below is based on our experience with CoAP security research and IoT product and infrastructure penetration testing projects. We will try to keep things simple in this blog. In future blog posts, we will dig deeper into the technical aspects of the different attacks on CoAP. Below we will try to answer some of the questions that generally come up when one encounters a new protocol in their IoT penetration tests.

3.1 Protocol Security/Authentication

We have discussed the protocol at length without a single mention of authentication. Surprised? The CoAP standard does not specify any authentication mechanism. It relies on DTLS to provide protocol-level security. The standard defines four security modes for devices post provisioning: 1. NoSec - This means there is no protocol-level security i.e. DTLS is disabled. 2. PreSharedKey - In this mode, DTLS is enabled and the device has a list of pre-shared keys with each key including a list of which nodes it can be used to communicate with. 3. RawPublicKey - In this mode, DTLS is enabled and the device has an asymmetric key pair without any certificate. The key is validated using an OOB (Out-of-bound) mechanism 4. Certificate - In this mode, DTLS is enabled and the device has an asymmetric key pair along with an X.509 certificate that is signed by a common and trusted Root CA.

The protocol standard clearly mentions in section 9 (page 69) - *"CoAP itself does not provide protocol primitives for authentication or authorization; where this is required, it can either be provided by communication security (i.e., IPsec or DTLS) or by object security (within the payload)"* There is another standard RFC 8613 that defines Object Security though.

3.2 Tools

Are there any tools available that assist in CoAP analysis and assessment? Thankfully there are some choices available:

1. Open-source libraries, clients, server implementations. The only issue here is that they are rigid and have no guidance from the security perspective and you may end up editing or writing custom scripts to do the job. The list of all libraries and packages can be found **here**.
2. **EXPLIoT Framework** We are planning to release basic plugins for CoAP assessment in the next couple of months.

3.3 Recon

Ok, so we have the tools, now what? Where do we start? The first thing is to perform recon on the CoAP Server to get as much information about the server as possible.

1. Is there any Authentication mechanism implemented?
2. Get a list of available resources on the server by sending a GET request to /.well-known/core URI path.
3. What do the URIs signify? Do their names indicate any action related to the physical environment (sensor/actuator)? Is there any URI for configuration/setting update?
4. Are you able to read (GET) any sensitive information?
5. Are you able to manipulate (PUT/POST/DELETE) any sensitive information?
6. What URIs are important and used in the IoT Eco-system? This can also be found by reverse-engineering the device firmware or Mobile App (if it communicates with the sensor).
7. Is DTLS used?
8. How is the telemetry data, sent from the device, utilized on the cloud?

3.4 Analysis

Once we gather the basic information about the CoAP implementation, we can utilize it to launch different attacks on the devices or the eco-system. By eco-system, I mean all the components of the IoT product infrastructure including the devices, gateways, Cloud and apps, mobile apps, etc. We will list down some of the ideas to get your grey cells working:

1. Brute-forcing/Fuzzing all URIs with different Request methods to identify which URIs respond to which methods. However, there is a risk of corrupting the data on the device using this method. Also, a bit more information about the query parameters and application data format would help in directed fuzzing to get meaningful results
2. If there is custom authentication implemented, you will need to reverse engineer it to identify the mechanism.
3. Reversing the firmware or the mobile app may give you sensitive information such as
    1. Resource URIs
    2. Parameters
    3. Methods
    4. Application Data Semantics.
    5. What data is exchanged and expected on specific URIs and how it affects the behavior of the components.

3.5 Attacks

3.5.1 Path Traversal (Old wine in new bottles)

During our research, a couple of years ago, one of the things we found was that few CoAP implementations are not (or incorrectly) parsing or ignoring double dots ".." in the URI. The issue here is the same as with Web, for example, if the URI is bound to a file, there are chances of breaking out of CoAP root and accessing or manipulating other files. There might be other issues in file-less URIs that researchers may find in the future, who knows. Below are direct quotes from the Standard about the dots:

1. Section 5.10.1 page 54: *"The Uri-Path and Uri-Query Option can contain any character sequence. No percent-encoding is performed. The value of a Uri-Path Option MUST NOT be "." or ".." (as the request URI must be resolved before parsing it into options)."*
2. Section 5.10.7 page 57 - *"Each Location-Path Option specifies one segment of the absolute path to the resource, and each Location-Query Option specifies one argument parameterizing the resource. The Location-Path and Location-Query Option can contain any character sequence. No percent-encoding is performed. The value of a Location-Path Option MUST NOT be "." or ".."."*

There are a few important things to note here: 1. No percent-encoding is performed. 2. The request URI must be resolved before parsing it into options. 3. The path options must not contain "." or ".." 4. Any CoAP compliant library/implementation should adhere to what is mentioned in the standard. 5. It does not mention how to handle the case when the value of the Option in the received packet is either "." or "..".

Based on the above information, lets quickly look at two implementations where we checked for double dots:

1. Californium - **Californium** is a well-known java-based CoAP implementation. The core implementation of Californium does not parse/ignore double dots. That is left to the users to implement. In one of their demo (sample) apps called **cf-simplefile-server**, they have implemented a method checkFileLocation() to detect this attack. It means that the core protocol implementation of californium does not provide any security against this. We sent an email to one of the core developers of californium about this on 26th Feb 2019. As usual, there was no response. We did not pursue it further, if someone files a bug and manages to get a CVE, please do add our name in the credits :).
2. libcoap - is also a famous C implementation of the CoAP protocol. They do have a provision for filtering dots. They have implemented a function dots() in the **code**. However, if the path is percent-encoded the check is bypassed. The issue is that the function responsible to do this coap_split_path_impl() checks for the dots first and then passes it to another function h() where the path is percent-encoding handling as can be seen in the below code snippet from the file **uri.c** if (!dots(p, q - p)) { h(p, q - p, data); } An example would be: GET coap://coap_server/foo/%2e%2e/%2e%2e/%2e%2e/%2e%2e/etc/passwd We sent the author an email on the same day as Californium i.e. 26 Feb 2019. In this case, the author replied, however, the response was that it will work only if the server implementation is broken. Again, we did not pursue it further as the author did not agree that it is an issue on their side.

3.5.2 Cross Protocol Attacks

The CoAP protocol has tight coupling with HTTP in many fronts including request/response format, caching, proxying, etc. This also gives rise to opportunities for HTTP-to-CoAP translation/proxying and vice versa. Due to the similarity, some of the traditional attacks that affect HTTP today could very well apply to CoAP now or in the future. As we have already seen in the above case we discovered Path Traversal issues in CoAP implementations, however, they were originally discovered in HTTP. It's only a matter of time when researchers start connecting the dots and finding interesting attacks that target headers/options, URIs, request/response,

etc. by way of either: 1. Finding it feasible to attack CoAP with traditional attacks 2. Finding ways to transfer an attack from one protocol to the other. Note: By Cross Protocol Attacks we mean attacking the protocol and not the high application which we discuss in the below scenario.

3.5.3 Attacking the Application via Malicious Input

In the CoAP application architecture, typically the server resides on the sensors and you can discover the resources/services available on the server and send the request. There are two places where we can attack the applications:
1. Server i.e. the application running on the sensor - This simply means sending it requests with malicious Application payload. You can do it blindly by fuzzing the payloads or a better approach is to reverse engineer the firmware and get a better understanding of what application data format is being used, how do the URI queries look like and then craft your malicious inputs.
2. Client i.e. the application running on the Cloud/Mobile - The idea behind this is quite similar to what we discussed in the previous blog post on MQTT (section 4.4.4). If you have not read it, we would advise you that you at least go through the section 4.4.4 if not the complete blog post. The premise of this attack is the fact that cloud/mobile application developers may not filter the input from a sensor for known application attacks and this can lead to exploiting those apps using traditional attacks like SQLi, XSS to name a few.

3.5.4 Accessing and Manipulating Devices Via Requests

Using various techniques mentioned above, if you can read sensitive data and/or can write data of your choice to sensitive resources by any of the request methods, you have complete control on the device. An example could be controlling the device by sending it specific commands that it expects to receive from the Gateway/User. Controlling or shutting down an IoT device performing a critical operation in itself is game over!

3.5.5 Cloning or MitM'ing the Server (Sensor)

Depending on whether the authentication/security is implemented or not and your access to the keys, Certificates, authentication credentials if any, you can have a rogue device pretend to be a Sensor and publish all the resources available on the legitimate sensor, you can send wrong or malicious data to the client.

3.5.6 Distributed Denial of Service using CoAP

The fact that CoAP runs over UDP makes it a perfect choice for attackers to cause DDoS on targetted nodes or machines within the network as well as on the Internet. In fact, utilizing CoAP devices available on the Internet for DDoS by malicious actors is already a known thing and attackers are doing it as we speak. The science behind this is pretty old and common - Attacker spoofs the IP of the victim and sends UDP datagram (CoAP request in this case) to the Server which then responds to the victim IP. If you send mass requests (Spoofed IP) to all available servers on the network/Internet, they will in turn respond to the victim and overload its kernel thereby causing Denial of Service on the victim.

3.6 Mitigations

The vulnerabilities and issues in CoAP are quite similar to other protocols, lack of authentication, access control. There are a few things that one should consider when trying to implement a secure CoAP service:
1. Use DTLS.
2. Prefer not to implement a custom authentication/encryption mechanism. History tells us that custom security implementations seldom work as desired :).
3. Prefer to utilize all 8-byte to generate random Tokens for Requests. It's good to utilize the max. length of the token to generate as much randomness as possible as smaller the size, easier it would be to brute-force/guess and spoof.
4. Filter double and single dots (".." and ".") in the Uri-Path to prevent against path traversal attacks. 5. Secure the keying material/Certificates etc properly on the devices.
6. Filter input (Request Application payload) on the Server (Device) side.
7. Filter input (Response Application payload) on the cloud side. Even though the device is sending simple telemetry data, it is good to filter it for known/traditional attacks.
8. Implement proper access control and Auth mechanism for accessing resources that perform a critical operation or provide critical information.
9. Log all Activity. 10. Have a way to notify cloud/user about unauthenticated/malicious looking requests. 11. Don't hardcode credentials/sensitive information in the device firmware.

4 Conclusion

We hope this blog post gave you a good high-level overview of CoAP protocol, how it works, how it is used and how would you go about performing security assessments of a CoAP based IoT/IIoT ecosystem. Some of the attacks mentioned in the blog may have not been exploited in the wild or reported to the vendor as the protocol and these techniques are fairly new. This is your chance to find as many vulnerabilities and new attack vectors as possible. The tools, attacks, and techniques mentioned above will improve assessment efficiency and give you a direction for deeper inspection and security research on any CoAP implementation.

Continue to the next part - IoT Security - Part 12 (MQTT Broker Security - 101)