Understanding Constrained Application Protocol

Using CoAPSharp Library

2024

Understanding Constrained Application Protocol

Femtomax Inc.

Author : Vishnu Sharma

Understanding Constrained Application Protocol

Femtomax Inc. (“Femtomax”) is the company behind CoAPSharp[1], an implementation of the CoAP specification [5],a popular standard in machine to machine communications for resource constrained devices.
Femtomax Inc. provides Information Technology (IT) products and solutions with marked focus on measurable value. It offers services and products in Internet of Things and related technology solutions and products.
Femtomax Inc. was founded in 2024 and is headquartered in San Jose, CA, USA and has presence in India too. For more information, visit www.femtomax.com.

The source code for code examples outlined this book is available to readers at https://github.com/femtomaxinc/CoAPSharp.

DISCLAIMER:

This eBook is for informational purposes only and is provided “AS IS.” The information set forth in this document is intended as a guide and not as a step-by-step process, and does not represent an assessment of any specific compliance with laws or regulations or constitute advice. We strongly recommend that you engage additional expertise in order to further evaluate applicable requirements for your specific environment. FEMTOMAX INC. (“FEMTOMAX”) MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND, EXPRESS OR IMPLIED, AS TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS DOCUMENT AND RESERVES THE RIGHT TO MAKE CHANGES TO SPECIFICATIONS AND PRODUCT/SERVICES DESCRIPTION AT ANY TIME WITHOUT NOTICE.

FEMTOMAX RESERVES THE RIGHT TO DISCONTINUE OR MAKE CHANGES TO ITS SERVICES OFFERINGS AT ANY TIME WITHOUT NOTICE. USERS MUST TAKE FULL RESPONSIBILITY FOR APPLICATION OF ANY SERVICES AND/OR PROCESSES MENTIONED HEREIN. EXCEPT AS SET FORTH IN TERMS OF AGREEMENT YOU SIGN WITH FEMTOMAX, FEMTOMAX ASSUMES NO LIABILITY WHATSOEVER, AND DISCLAIMS ANY EXPRESS OR IMPLIED WARRANTY, RELATING TO ITS SERVICES INCLUDING, BUT NOTLIMITED TO, THE IMPLIED WARRANTY OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, AND NONINFRINGEMENT. Except as expressly provided in any written license agreement from FEMTOMAX, the furnishing of this document does not give you any license to patents, trademarks, copyrights, or other intellectual property.

All other product names and trademarks used in this document are for identification purposes only to refer to either the entities claiming the marks and names or their products, and are property of their respective owners. We do not intend our use or display of other companies’ trade names, trademarks, or service marks to imply a relationship with, or endorsement or sponsorship of us by, these other companies.

HOW TO SUBMIT ERRORS, OMISSIONS AND SUGGESTIONS

If you find errors, omissions or have improvement suggestions, you can send an e-mail to contactus@femtomax.com . Please clearly outline the following in your e-mail:

1. A detailed description of the error, omission or your suggestion
2. Your contact e-mail
3. Your Twitter or Facebook handle
4. Your picture
5. We firmly believe in giving credit where it's due. We would like to publish the names, pictures and Twitter or Facebook handles of individuals who help us improve this eBook. The details will be published in subsequent releases of this book for all incorporated feedback. Please do let us know whether you would like your details to be published or not. Also, please clearly indicate which details can be published in the version of the book that contains your feedback.

Table of Contents

Chapter 1

Brief Introduction of Machine-2-Machine Communication

Machine-2-Machine, or M2M as it's generally called, has a much older history. It simply means,one machine talking to another machine. This was not new, and it existed from early days of industrial automation, telemetry and SCADA systems. While these were point solutions, and enabled communication between similar type machines, the first thoughts on using the cellular network for M2M came up sometime in 1995[3] .

M2M is primarily driven by the need to connect more and more machines over large distances. So this is being viewed primarily as a connectivity problem and not as an application protocol level problem. If you look at companies joining the M2M bandwagon, you will find a lot of cellular network provider companies putting effort into this area.

Consider a simple example; you have sensors in your office conference rooms that monitor the temperature. These sensors will read the room temperature and make it available to other controllers deployed remotely. These controllers in turn will manage the cooling system centrally. This happens even today, but the difference is, that now you need to expand your thought and bring in Internet scale and reach. Can I have a reader sitting thousands of miles away from the temperature sensor? Weather channels will be very happy with this. They can have a listener installed in their office that will read temperature in real time from different sensors situated in different countries.

Brief Introduction of Internet of Things

While the term M2M is much older, a new term called "Internet of Things" (a.k.a IoT) has emerged. Given the current scenario, we may say that IoT is a subset of or is closely related to M2M, where machines communicate with each other but leverage the IP networks for such communication. Machines in M2M world can be considered "Things" in the IoT world. One day, we believe, both these will merge and become one.

The term "Internet of Things" was coined by Kevin Ashton in 1999 [2]. It refers to objects that can be uniquely identified and addressed using their "virtual representation" in an "Internet like" structure. Comparing this to today's Internet, visualize this as another Internet, where each person and a computing device (PC or tablets for example) together are replaced by "Things". Now, it is these "Things" that connect to the larger network of other "Things" and either consume

data or produce data for other's consumption. The "Thing" is assigned a unique IP address andit exposes a set of methods using which other things can discover, read or write to it.

Internet of Things requires "Things" that perform a useful function. In that process, they either produce valuable data for another "Thing's" consumption, or require data from another "Thing" to perform the function. Thus, the first basic requirement is that you need an object that represents a "Thing".

Assume another simplistic example of your house, where everything is connected to each other (these days, this is sometimes referred to Smart Homes). The microwave is aware of the freezer temperature and knows how much temperature to set, on its own to thaw a chicken. You do not need to think and decide. The television is connected to the microwave; perhaps, when the microwave switches on, it can give you information on what is the current program running on the food channel. Additionally, all these are connected to the electric meter at your home, whichin turn, sends data to the grid centrally in real time. The electric meter can inform the microwave what is the current load and is it safe to set it to a given temperature. All working in perfect harmony!

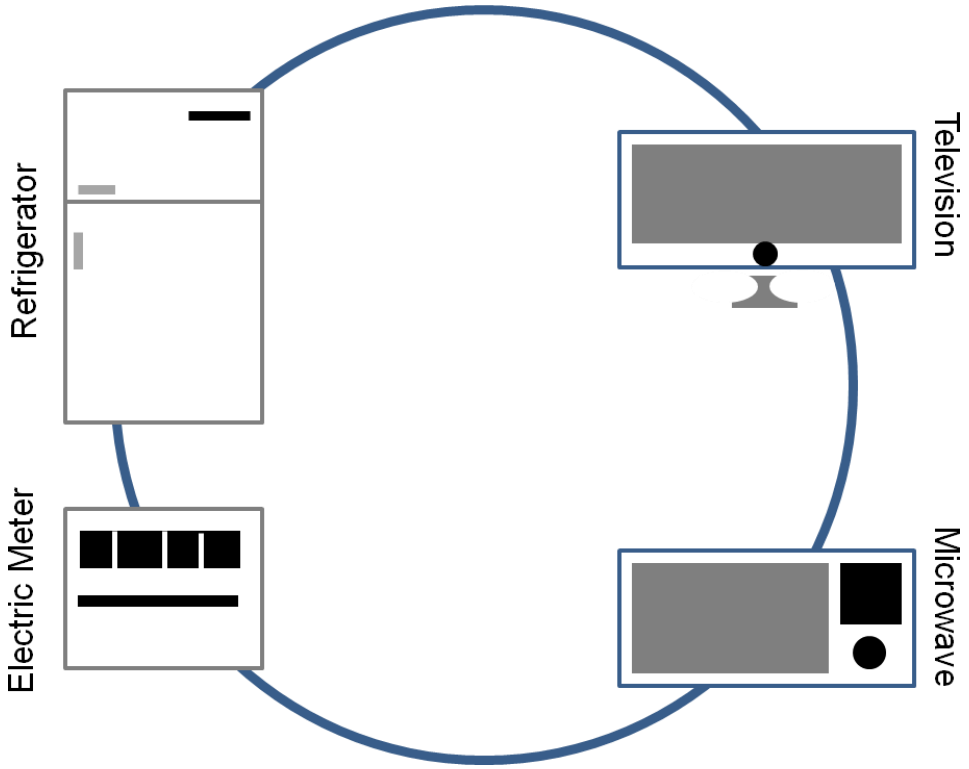

Figure 1-2. The connected things

So to summarize, we can define the following:

A. Things are objects that perform a useful function (like refrigerator, microware)
B. Internet is a means to exchange data and information between entities

C. When you club together “Internet” and “Objects” to create scenarios, where human life is made easier and comfortable, you get “Internet of Things”.

Birth of Constrained Application Protocol

As we move slowly to a paradigm where a large number of machines will interact with each other without the need for human intervention, we begin to realize that we need a standard wayto enable communication between these machines. Additionally, since a lot of good work has already been done in the past to setup the Internet and the IP networks, it makes sense to follow the lead and try to leverage the already existing infrastructure and build on top of it.

To standardize a communication methodology for machines to independently exchange information with each other, keeping in mind, that we want to build on and leverage existing knowledge and infrastructure, we need to address the following issues:

1. We must be able to leverage existing IP infrastructure
2. We must allow for a large number of machines on the “Internet” (possibly billions)
3. We should ideally enable small size machines to communicate. Putting a lot of requirements on the machine just for communication will make large scale adoption commercially unviable.
4. We need a way to easily leverage the existing HTTP based investments.

Many individuals and corporations have been working on finding solutions to the problems mentioned above. In due course, multiple enabling technologies have been created, with some notable achievements being:

1. IP V6 – This has allowed large scale deployment of devices with literally infinite addresses
2. 6LoWPAN – This has allowed smaller processing requirements to participate in an IP V6 based communication
3. Lightweight IP – This implementation has allowed have a very small software stack to participate in a TCP/IP based exchange.

With such advances, time was ripe to define a “lightweight HTTP”, something that does not require too much processing power and can be used by tiny devices. This is where IETF (Internet Engineering Task Force, www.ietf.org) came up with Constrained Application Protocol (a.k.a CoAP)

The Constrained Application Protocol defines the message format (arrangement of bytes) and message exchange rules between two or more participating “Things”. Given its simplicity, it is also easier to implement CoAP based systems on small embedded hardware. One of the best things about CoAP is that it closely resembles HTTP and it is therefore very easy to convert from one format to other.

Summary

In this chapter, I wanted you to understand the background and drivers of “Internet of Things”. While what we outlined covers only the tip of the iceberg, the remaining chapters will delve deeper into this subject. Being a programmer like you, I’m also not too interested in lot of much theory and want to jump straight to code. So, I will try to keep the theory to only what is an absolute must and focus more on programming and creating great samples to get you going.

The samples in this book have been built using a CoAP implementation called CoAPSharp[1]. The CoAPSharp library is written in C# and is designed to work on nanoFramework (https://github.com/nanoframework) runtime.

Chapter 2

Understanding the Constrained Application Protocol

The Internet Engineering Task Force created a working group called Constrained RESTful Environment Group (or CoRE) group. This group was assigned the task to define a mechanism using which a large number of small, resource constrained, low power devices, can communicate over lossy networks.[4]

This group defined a set of specifications that is known today collectively as – Constrained Application Protocol or CoAP in short.

This is perhaps the most important chapter in the entire book. My attempt is not to cover the entire specification, but to highlight some of the important aspects that will help you create better applications. Coming from the web world, we found it easier to map the concepts with HTTP to better understand CoAP. We will do the same in this chapter.

What is CoAP

Constrained Application Protocol is a protocol at the application level that is designed to allow message exchange between resource constrained devices over resource constrained networks. Resource constrained devices are small devices that lack the processing power, memory footprint and speed that we generally expect from our computing devices. These devices often are built using 8-bit microcontrollers or low-cost, general purpose 32-bit microcontrollers. Resource constrained networks are network stacks and configurations that do not have the full capabilities of TCP/IP stack and have lower transfer rates. CoAP runs over UDP and not TCP. 6LoWPAN is an example of such a constrained network configuration setup.

CoAP provides an HTTP-like request and response paradigm where devices can interact by sending a request and receiving a response. Like the web, devices are addressed using IP address and port number. Access to services exposed by the device is via RESTful URIs. It's very much similar to HTTP, where method type (e.g. GET, PUT), response codes (e.g. 404, 500) and content-type are used to convey information. Given the protocol's close similarity to HTTP, it's obvious that it was designed for easy web integration.

CoAP does not replace HTTP, instead, it implements a small subset of widely accepted and implemented HTTP practices and optimizes them for M2M message exchange. Think of CoAP as a method to access and invoke RESTful services exposed by “Things” over a network.

As an example, let’s consider our temperature sensor installed in the conference room of our office. The temperature sensor works like a “server” (Thing) which any CoAP based client (another Thing) can query to get the temperature.

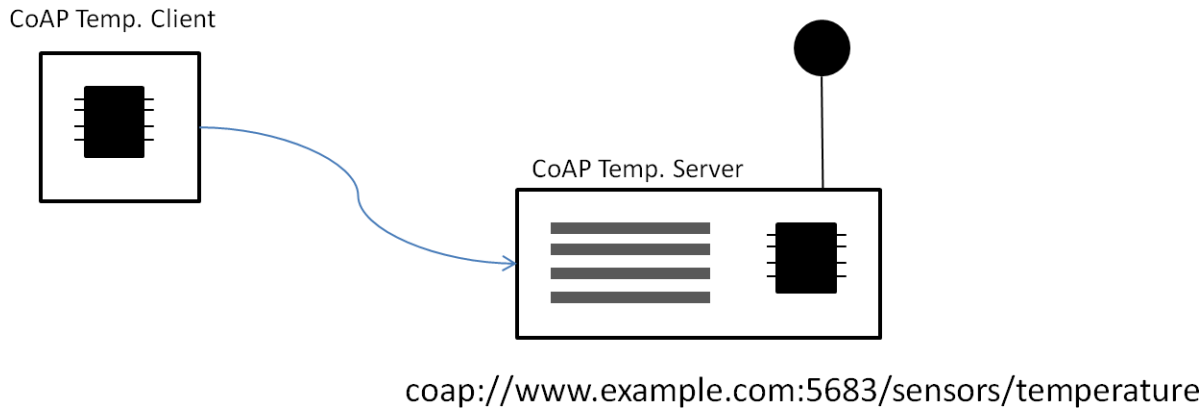

Figure 2-1. CoAP Client and Server

In the figure above, the server exposes the interface to query the temperature as a RESTful URL with the path as “sensors/temperature”. By this time, you would have noticed the new scheme – coap. Instead of using http as the scheme, a new scheme called “coap” is introduced.There is also a secure version, just like https, you can use coaps.

In the figure above, the full provides the scheme name, the DNS name, the port number and the path. Remember, the default port suggested for CoAP is 5683. Also remember, that the communication will use UDP and not TCP.

Therefore, the client needs to establish a UDP connection with the server, send a GET requestto the server over the given URL path and get a response. Just like HTTP response, you can get a response in various formats (remember HTTP content-type?). The specification allows forvarious content formats, notable amongst them is JSON, XML and plain text.

The next sections will outline key concepts of CoAP. The descriptions are based on draft-18, which is the most recent draft at the time of this writing this eBook.

The CoAP Request and Response Messaging Model

As stated before, CoAP is similar to HTTP. One party can send a request to the remote party, and the remote party may respond back. There are four kinds of message types defined by the specification:

1. CON – This represents a confirmable message. A confirmable message requires a response, either a positive acknowledgement or a negative acknowledgement. In case

2. acknowledgement is not received, retransmissions are made until all attempts are exhausted. The retransmissions use a non-linear, exponential strategy between attempts.
3. NON – This represents a non-confirmable message. A non-confirmable request is used for unreliable transmission (like a request for a sensor measurement made in periodic basis. Even if one value is missed, there is not too much impact). Such a message is notgenerally acknowledged.
4. ACK – This represents an acknowledgement. It is sent to acknowledge a confirmable (CON) message.
5. RST – This represents a negative acknowledgement and means “Reset”. It generally indicates, some kind of failure (like unable to parse received data)

A typical confirmable message exchange could look like as shown in Figure 2-2.

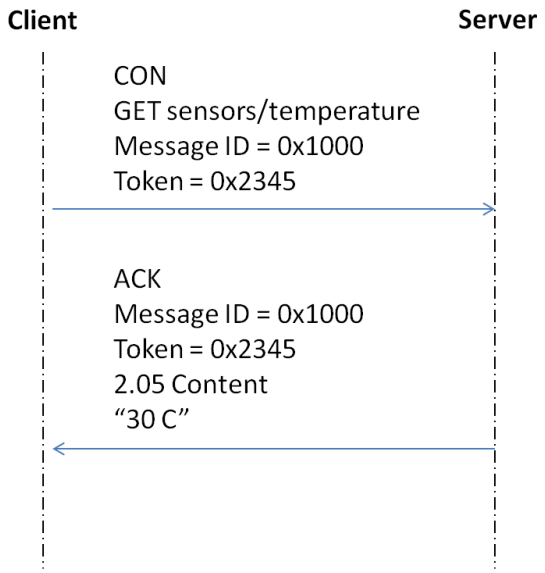

Figure 2-2. CON Message Exchange

The client sends a CON message to the server. The method type is GET, the path URL is “sensors/temperature”. The message ID is a 16-bit number used to uniquely identify a message and help the server in duplicate detection. Token is used to correlate messages. We will soon see an example of message correlation.

Once the server gets the message, it measures the temperature and returns an acknowledgement. The acknowledgement contains the same message ID and the token that was received in the request. Along with the acknowledgement, the message also contains the temperature data (in the above figure, its 30 C). Sending response data, along with the acknowledgement is also called “piggy-backed response”. Finally, the response also has a message code, in this case it’s “2.05 Content”. These are very similar to HTTP status codes (there is a 4.04 message code to indicate not found, like the HTTP 404 not found status code).

The previous example indicates a success, but sometimes CON message might result in failure. For example, if the path URL is incorrect, there is no way the server can serve the request. In HTTP world, if the URL is incorrect, we get a 404 as response code. In the same manner, in CoAP also, we get a “Not Found” response. Figure 2-3 indicates such a situation.

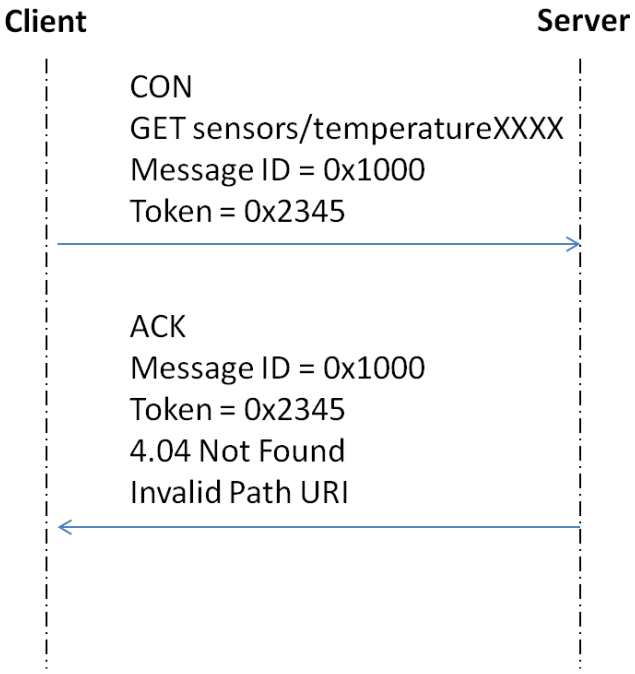

Figure 2-3. CON Message Exchange Failure

In the figure 2-3, the request is made to an unrecognized path URL“sensors/temperatureXXXX”. Since there is no way server can handle that path, it acknowledges the receipt of the message, however, it sets the message code as 4.04 (we will explain these codes later on) which means “not found”. Additionally, implementations may add diagnostic message in the response payload, and in this example, the string “Invalid Path URL” was added as the diagnostic payload.

Sometimes, the request is not mission critical and it’s acceptable if some values are never received. Classic example is temperature sensing request for room cooling that continues 24 x 7.Even if some queries to the temperature server are lost there would hardly be an issue. In those cases, NON (or Non-Confirmable) requests are used. Figure 2-4 depicts such a scenario.

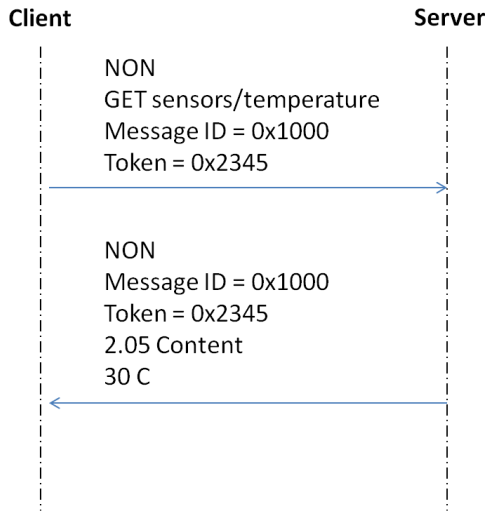

Figure 2-4. NON Message Exchange

The client sends a NON based GET request. The server also responds back with a NON message. In either direction, the message may get lost, but unlike a CON message, no attempt will be made to retransmit the lost message. A new NON message will simply be sent by the client when it's due.

Natural question to ask in this case would be what if NON message cannot be understood by the server (for example, the URL path is incorrect), will the server respond back? The specification states that a NON message, being non-confirmable, must not be acknowledged bythe recipient. The recipient, at best, may send a RST (reset) message and must silently ignore. Therefore, if a NON message carries incorrect path, there is no guarantee that the sender will be informed of the error. The sender may choose to re-transmit the message up to a limit, but the sender cannot expect a guaranteed response.

The CoAP Message Format

Knowing the internal details of the CoAP message format is best left for implementers of the CoAP protocol stack, however, you must have some understanding of the structure to truly appreciate the protocol. It will also help you make the right decisions.

The CoAP message consists of a series of bytes, a four-byte header that is mandatory, followed by a set of bytes that are optional. Therefore, the smallest size of a CoAP message is just 4 bytes. Figure 2-5 provides an overview of the CoAP message byte structure.

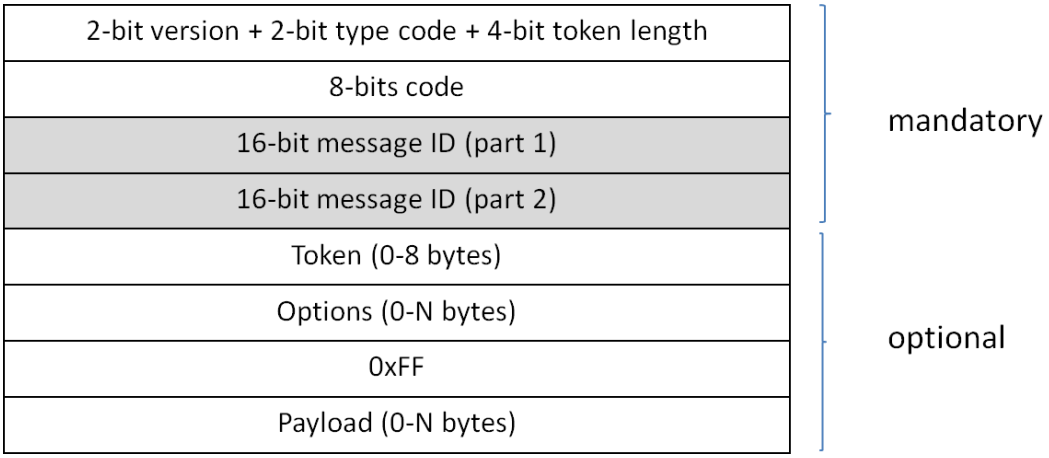

Figure 2-5. CoAP Message Byte Structure

The CoAP message is transmitted to the remote recipient over UDP. The byte ordering is in network byte order.

The first 4 bytes that are mandatory contain the following pieces of information:

1. Byte 0
   (i). 2-bit version: The first two bits indicate the CoAP version number. As of now,only version 1 is supported.
   (ii). 2-bit type code: The next two bits indicate the message type. This can take oneof 4 values – CON, NON, ACK, RST
   (iii).4-bit token length: The next 4 bits indicate the length of the token value in bytes.As explained before, the token is used to correlate messages. The length of token can be between 0-8 bytes. Other values are reserved.
2. Byte 1 – This contains the message code. The message code values can be GET, PUT, POST, NOT FOUND etc. I will talk about other possible message codes later on in this book.
3. Byte 2,3 – The next two bytes together make up a 16-bit number. This is where the message ID is carried. This is an unsigned number.

After the first 4 bytes, based on the context, the message may contain additional bytes. It is recommended that we send a token with every message (especially in cases where correlationis required. We will see such cases later on in this book). Therefore, if the 0^{th} byte indicates a token length value that is between 1-8, then 5^{th} byte onwards, we will have a token. For example, if last 4 bits of byte 0 in the 4-byte header have a value 5, then it indicates that this message contains a token whose value is 5-bytes long. Thus, byte number 4,5,6,7,8 that follow the 4-byte header (byte-0 to byte-3) will contain the token.

Following the token, there could be multiple options. Think of options as pieces of data that provide additional information about the message. If you compare with HTTP world, options are like request headers, however, there is one big difference. Unlike HTTP world, where you can add any arbitrary HTTP header, in CoAP, you are limited to what is defined. For new headers,

you need to go through the process of submitting your recommendation for a review to IETF groups.

Examples of options defined by CoAP are “Max-Age”, “Content-Format” and “ETag”. I will cover some of the key the options along with the examples in later chapters.

Once the options are complete, there is a separator marked by the presence of the value 0xFF. If this value is present, there must be at-least 1-byte of payload data. The value 0xFF is followedby one or more bytes of payload data. This is where you can put any data that you want to transfer (like in previous examples, we put measured temperature value).

The last point to note is that CoAP does not define an “End of Message” identifier. So, it’s pretty much left to the lower level implementations to take care of when the message ends.

Request Response Guidelines

The CoAP specification at the time of this writing was draft-18. In this draft, there is still some work to be done around clear set of rules that govern what is the response message type for a given request. If you read the specification, such request response rules are outlined in various places. However, I believe, much work has to be done in this area to bring in more clarity. While the specification will continued to get refined over a period of time, I have come up with simple guidelines as tabulated below to help you decide what response to send back for a given request type.

Client Sends	Message Successfully Parsed and Understood By Server ?	Server Has all Information to Process the Request and can Successfully Process ?	Server Sends Message Type	Server Sends Message Code	Remarks
CON	YES	YES	ACK	One of success response codes (e.g. CONTENT)	Happy day scenario
CON	YES	NO	ACK	One of failed response codes (e.g.	URL path in request is wrong and

Client Sends	Message Successfully Parsed and Understood By Server ?	Server Has all Information to Process the Request and can Successfully Process ?	Server Sends Message Type	Server Sends Message Code	Remarks
				NOT FOUND)	4.04 not found is sentin ACK
CON	NO	NO	RST	One of failed response codes (e.g. BAD OPTION)	e.g. Unknown option number
NON	YES	YES	-	-	No response sent back
NON	YES	YES	NON	One of success response codes (e.g. CONTENT)	Response sent back as NON message
NON	YES	NO	RST	One of failed response codes (e.g. NOT FOUND)	URL path in request is wrong and 4.04 not found is sentin RST
NON	NO	NO	RST	One of failed response codes (e.g. BAD	e.g. Unknown option number

Client Sends	Message Successfully Parsed and Understood By Server ?	Server Has all Information to Process the Request and can Successfully Process ?	Server Sends Message Type	Server Sends Message Code	Remarks
				OPTION)	
NON	YES	YES	CON	One of request codes (e.g. PUT)	e.g. Previous NON request was for a data that requires confirmation from the sender on whether it reached the client or not

Table 2-1. Guidelines for Request and Response Messages

Maintaining Message Uniqueness

Every CoAP message is at-least 4-byte long. This is called the CoAP header. The last 2-bytes of the CoAP header contain the 16-bit message identifier. This is a simple 16-bit unsigned integer. Every new CoAP NON and CON message must have a unique message identifier. Theobvious question to ask here is when we reach the value 0xFFFF, the message identifier will rollover and thus message identifiers will now have duplicate values. This is not a problem.
Once a message is sent, after a certain period has elapsed, the message is assumed to be consumed. Once the message is consumed, the identifier can be reused.

For a CON or confirmable message this elapsed time is called “EXCHANGE_LIFETIME” in CoAP specification. All you need to take care is that the same message identifier is not re-used during the exchange lifetime.

EXCHANGE_LIFETIME is the time that starts when you begin to send a CoAP message and ends when you no-longer expect a CoAP response. One cased could be when a CON message

is sent and a response is received. The entire time period is EXCHANGE_LIFETIME. Another case could be that a CON message was sent but response was not received. Therefore, the message was re-transmitted until all re-transmission attempts exhausted. In this case, the EXCHANGE_LIFETIME is from the point the message was sent to the point a decision was made that we have exhausted the re-transmission attempts.

With default values of all time delays that make up the EXCHANGE_LIFETIME that value comes out to be 247 seconds (draft-18). Therefore, to be on the safe side, if you send a CON message and reuse the same message identifier after 300 seconds, you should be fine.

CoAPSharp library takes care of internally calculating the lifetimes for a confirmable message. It also provides you option to tweak these values. You can specify what is the acknowledgement timeout (the time to wait for an ACK or RST message to come back) and what is the maximum retry count (how many times to re-transmit if we do not get back an ACK or RST message). You can use the APIs provided by CoAPSharp library to get the next available message identifier (more on this in later chapters)

For a NON message where no response is generally expected, the EXCHANGE_LIFETIME would be shorter and it's called NON_LIFETIME. With the default values (draft-18) it is 145 seconds. Therefore, if you send a NON message, you can safely reuse the message identifier after about 150 seconds.

NOTE
Every response (ACK or RST) should contain the same message identifier value as received in the request (but in RST cases this may not be possible)

Message Co-relation in CoAP and Separate Response

For a given CON request, a response carries the message identifier, the purpose of which is to inform the client, against which request the response was sent. While this works with cases where a CON message can be immediately responded back to, there are cases where the server might require time to process the message (for example, the client sends a CON message to shutdown a valve and server required about 10 seconds to shutdown the valve). When server needs time, it needs to inform the client that it has registered the request and will respond back shortly. This approach in CoAP is called "Separate Response". The server will respond back to the client with an empty ACK and sometime later, it will send back a CON message to the client indicating task was done.

Now such an example poses a challenge – how does the server inform the client against which request the response was sent because the response was sent so late, that it crossed the EXCHANGE_LIFETIME boundary and by this time, client would have started to reuse the message identifier.

This is where the “Token” comes in. Token carries an opaque value (meaning no attempt is made to understand the value. It’s reused as is). Client sends a CON request with the message identifier and a token value. Server responds back with an empty ACK containing the message identifier, indicating to the client that it has registered the request and will respond back shortly. Once the server has completed the task, it responds back with another CON message with a new message identifier (because now the CON response has to be tracked at the server end and should be re-transmitted if previous attempts to get an ACK from client fail) but with the same token value.

Same happens in a NON request. The server can receive a NON request with a message identifier. Since it’s a NON request, the server registers the requests and goes on for processing. No response is sent back. After a while, when the server completes the task, it sends back another NON request with a new message identifier, however, the token value is same as what was received from the client in the original request (The server might decide to respond back with a CON request. This decision is based on the implementation.)

Thus, token is used to co-relate messages that participate in a single set of logically related transactions, even if the message identifiers are different).

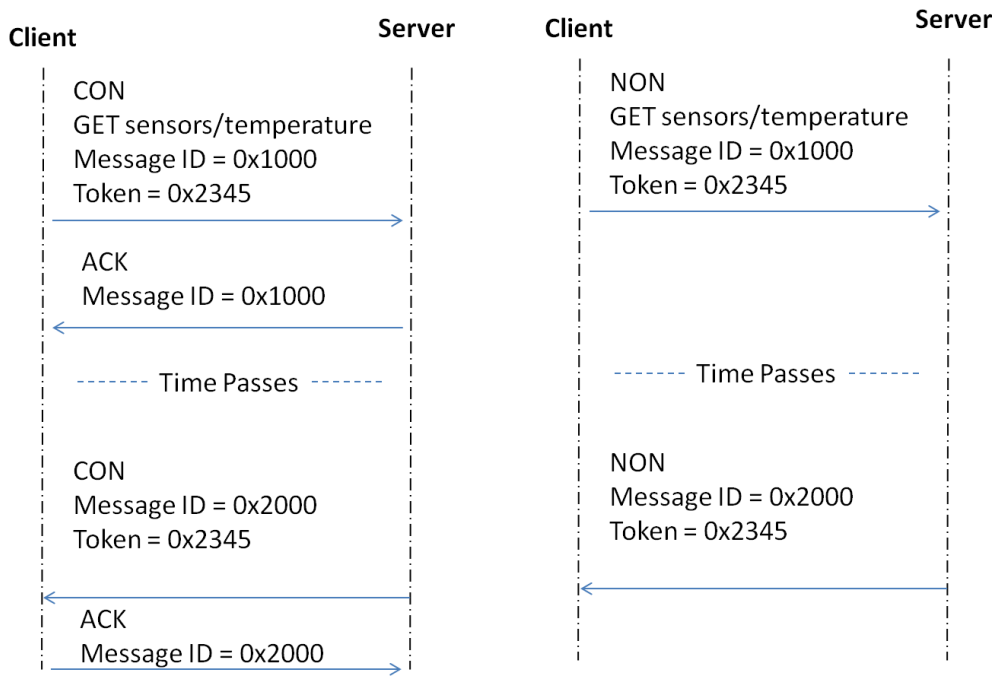

Figure 2-6. CON and NON Separate Response with Message Co-relation

Passing Metadata with Message using Options

CoAP message format makes provision for passing options from the sender to the recipient. By the very name “options”, it indicates that these values are optional. Options in CoAP resemble to “Request Headers” in HTTP, however, they a bit different. CoAP has a pre-defined set of

options with each option defined by an option number and a value type. One is not free to use any free number as a custom option (something you can do with request headers). The options carry a lot of useful information about the request. These options are maintained in “Option Number Registry”. In draft-18, there are about 16 different options that have been defined. There is an option to indicate the “Content Format” of the payload being carried in the message (This is similar to Content-Type header in HTTP). There is another option to indicate the “Max-Age” of the payload in the message after which, the payload should be considered stale. You can use these options to pass additional information about the request between communicating parties.

Exchanging Large Payloads using Block Transfer

While CoAP is designed for small exchanges, sometimes it might become necessary to exchange bigger loads. This is made possible by a concept called “Block Transfer”. Client can send large data sets in smaller chunks over multiple calls to the server (e.g. PUT case) or client can get a large dataset from the server in small chunks (e.g. GET case). I will cover this topic in detail in later chapters with examples.

Summary

In this chapter, we learnt basics of CoAP protocol and an outline of the message structure. The specification defines a lot more things, however, our purpose is to enable you build meaningful applications as fast as possible. With that goal in mind, I have chosen an implementation of the CoAP protocol that takes care of lot of underlying details without you having to worry about them. I will focus more on building application using the CoAP concepts and will cover only those aspects that are relevant for an application developer.

Chapter 3

Setting Up the Development Environment

The last two chapters introduced you to the concept of Internet of Things and the Constrained Application Protocol. By this time, you would probably be eager to get your hands on some code. You will start by building the basic “Hello CoAP” client and server example.

There are a few basic components and tools that you need to get you started with CoAP examples. I have selected C# as the programming language to build the examples. This leads us to the determine how can we build microcontroller based applications using C#. nanoFramework runtime is a great choice for this. It is supported on many microcontrollers and development boards. ESP32 is one such popular board that runs nanoFramework. Finally, we would also need a library that implements CoAP specifications and is available for C#. For this, we will use CoAPSharp.

The IDE

There is no better IDE than Visual Studio if you want to work on .NET based projects. You will need the IDE to work with nanoFramework. At the time of this writing, Visual Studio 2017, Visual Studio 2019 and Visual Studio 2022 can be used to work with nanoFramework. Visit this link https://marketplace.visualstudio.com/items?itemName=nanoframework.nanoFramework-VS2022-Extension to understand how can you install this extension.

Once the extension is installed, you can create nanoFramework based projects.

Setting up the hardware

After the IDE is installed, you may work with a microcontroller board with connectivity features. ESP32 (DevKit C, ESP32 WROOM 32) is one such great board that comes with WiFi connectivity. So get yourself a couple of these boards and head over to https://nanoframework.net to get started, if you want to jump right to writing CoAP based code on microcontrollers.

For the first time, you would need to flash the nanoFramework runtime on these boards. To do so, use the dotnet tool named “nanoff”. Visit this link https://docs.nanoframework.net/content/getting-started-guides/getting-started-managed.html

to learn how to flash the nanoFramework runtime on the ESP32 board using nanoff tool. Once you connect the ESP32 board on your computer, you should see a COM port entry in device manager (on Windows). For ESP32 DevKit C, with the ESP32 WROOM 32 board, I found that the build that targets ESP32_BLE_REV0 or ESP32_BLE_REV3 works best. Once you have flashed the nanoFramework runtime, you are ready for the next step.

Setting up the CoAPSharp library

CoAPSharp library is an open source project and the entire source code is available at https://github.com/femtomaxinc/CoAPSharp . You can download and build locally. Alternatively, if you are only interested in using the library, then this is also available as a Nuget package. There are two versions of this library:

1. Femtomax.nanoFramework.CoAPSharp : This is designed to be targeted for nanoFramework. You want to include this if you are writing CoAP applications on a microcontroller.
2. Femtomax.NetStandard.CoAPSharp : This is designed to be targeted for .NET Standard 2.0. If you are building CoAP applications for the .NET environment, then you should select this.

The samples associated with this book are built on a Windows computer so that you do not have to wait for an ESP32 board. These samples use the netCore.CoAPSharp nuget package.

There are two more libraries named nanoFramework.CoAPSharp.Security and netCore.CoAPSharp.Security. There are not open sourced and provide patented secure communication capability between CoAP nodes.

Examples in this book

The examples in this book are built on .NET Standard version of CoAPSharp library. This allows you to get started quickly without the need to wait for microcontroller boards. Additional examples for the ESP32 board are posted on our Github repository.
To run the example, simply compile the examples project and run

dotnet Femtomax.NetStandard.CoAPSharp.Samples.dll

This will prompt you with a menu item of what example to run. Since all cases need a client and server, you can open two different terminals and run the server in one terminal and client in another.

Summary

This chapter outlines the basic tools you need to get started. It provides the list of things you must have ; the IDE, nanoFramework runtime ,CoAPSharp library and ESP32 board. If you have setup everything, it’s time to get started. From next chapter onwards, I will pick up one concept at a time and write the code to implement the concept.

Chapter 4

Your First CoAP Server

Now that we understand some basic aspects of the CoAP protocol and have the development environment ready, it's time to write some code.

We will build the server using NON request. This is the simplest case. If you remember, a NON request represents a non-confirmable message, meaning, even if the message is lost in transmission, hell doesn't break loose.

Example Scenario

Let's assume that you have a temperature sensor installed in your office conference room. This temperature sensor sends the room temperature to a remote machine installed in the cooling controller station. Thus, the machine installed in the cooling controller station acts like a client and the sensor installed in the conference room acts like a server. We will build a server based on CoAP that will return the latest temperature to clients. Clients will send a NON request to the server and server will respond back with a NON response.

Hardware requirements

We will build this on an ESP32. So after you successfully compile, connect the ESP32 and flash the code.

Server Design and Logic

The server side will be built using the CoAPSharp library. We will open a UDP socket that listens on the default CoAP port 5683. CoAPSharp provides an event based mechanism as well as a read-write mechanism. There are three events that it exposes; one event when a request is received from remote client, another when a response is received from a remote client and third, when an error occurs within the CoAPSharp library during various internal operations (like parsing an input stream and sending a message to the remote client)

In this example, we will focus only on the event that is raised when a request is received fromthe remote client. Given below is the basic algorithm:

1. Server receives the requests and extracts the URI path submitted in the request (Remember, CoAP is like HTTP and thus resources are identified using a URI)
2. Check that the request message type must be NON
3. If the message type is not NON, server responds back with a RST message with the message code as “NOT IMPLEMENTED”
4. If the message type is NON, check that the message code must be GET. Just like HTTPGET request, we are also expecting a GET request and for sending back the temperature, GET is the only request type that makes sense in this case (other types are PUT, POST, DELETE)
5. If the message type is NON and request type is GET, we check if the URL path in the request is correct (as we need a corresponding handler, else it’s a 404 not found case). This is same as an HTTP server checking if the path is valid or not. In this example, the GET request must be made to the URL coap://[sensor IP address or DNS name]:5683/sensors/temp. The last part “sensors/temp” is the path. If the path in the request does not match “sensors/temp”, the server respond back with a RST message with the message code as “NOT FOUND” (Same as HTTP NOT FOUND 404 response)
6. Finally, if the message type is NON, message code is GET and the URL path is correct,then the server constructs another response message of type NON. This time around, the message code is “CONTENT” (like HTTP 200 OK code)
7. The temperature value is measured (in real world scenario, you will have a temperature sensor connected to your hardware, which will issue the command to measure the temperature) and interestingly, a JSON response message is created. CoAPSharp supports basic, single level JSON object creation. This should be sufficient for most constrained applications. The JSON response string is added to the payload of the NON response. Additionally, just like HTTP Content-Type, a CoAP option named “CONTENT TYPE” is added to the response message with the value indicating that the payload content type is in JSON format.

Starting the CoAP UDP Server

To implement the server, we will use the class named CoAPServerChannel. It has a method named “Initialize” which takes two parameters. The first parameter is only for diagnostics and accepts the DNS name or the IP address of the server this code is running on. You can simply pass a null value. The second parameter accepts which port you want this UDP server to listen on. We are using the default port for CoAP UDP server which is 5683.

```
public class Program
{
        /// <summary>
        /// Holds the instance of the server channel class
        /// </summary>
        private CoAPServerChannel _coapServer = null;
        /// <summary>
        /// The main entry point for the program
        /// </summary>
        public static void Main()
        {
            Program p = new Program();
            p.StartServer();
            Thread.Sleep(Timeout.Infinite);//block here
        }
        /// <summary>
        /// Start the server
        /// </summary>
        public void StartServer()
        {
            this._coapServer = new CoAPServerChannel();
            this._coapServer.Initialize(null, 5683);//Initialize and listen on
default CoAP port
            //Setup event listeners
            this._coapServer.CoAPRequestReceived += new
CoAPRequestReceivedHandler(OnCoAPRequestReceived);
            this._coapServer.CoAPResponseReceived += new
CoAPResponseReceivedHandler(OnCoAPResponseReceived);
            this._coapServer.CoAPError += new CoAPErrorHandler(OnCoAPError);
        }
```

Listing 4-1 Setup CoAP UDP Server

After the server is initialized, we register the event listeners. The sequence is not important. You can first register and then initialize or vice versa. In this example, we will focus only on the “OnCoAPRequestReceived” event handler.

Getting URL Path in Request

Once a request arrives, the CoAP server library will parse the request and convert that into an object of type CoAPRequest. This object is then passed on to the request handler method that has been registered. We will extract the URL path from the request object by calling the "GetPath" method.

```
/// <summary>
/// Called when a request is received
/// </summary>
/// <param name="coapReq">The CoAPRequest object</param>
void OnCoAPRequestReceived(CoAPRequest coapReq)
{
       string reqURLPath = (coapReq.GetPath() != null) ?
coapReq.GetPath().ToLower() : "";
```

Listing 4-2 Getting URL Path in Request

Checking the Message Type

As stated before, our server will only respond to NON requests. Therefore, the first thing we check in the request event handler (after getting the path) is the message type. If it's not a NON, we send back a RST message to the client indicating we do not understand the request.Being a NON message, you can simply ignore this message, however, that way, the client will never know what went wrong. Since we can only send either a NON or a RST (and not an ACK), we chose to send the RST message.

```
if (coapReq.MessageType.Value != CoAPMessageType.NON)
{
    //only NON  combination supported..we do not understand this send a RST back
    CoAPResponse msgTypeNotSupported = new CoAPResponse(CoAPMessageType.RST,
/*Message type*/

CoAPMessageCode.NOT_IMPLEMENTED, /*Not implemented*/
                                                        coapReq.ID.Value /*copy
message Id*/);
    msgTypeNotSupported.Token = coapReq.Token; //Always match the request/response
token
    msgTypeNotSupported.RemoteSender = coapReq.RemoteSender;
    //send response to client
    this._coapServer.Send(msgTypeNotSupported);
}
```

Listing 4-3 Handling Incorrect Message Type

It’s important to note that we can copying over the message identifier and the token in the response. Additionally, we also need to copy over the details of the remote sender. This is not a CoAP requirement, but a requirement imposed by the CoAPSharp library.

Checking the Message Code

If the message type check passes through, we will next check the message code. Again, in this example, we are only supporting GET requests, therefore, if the message code is anything other than GET, we will once again respond back with an RST message indicating that the method is not allowed.

```
else if (coapReq.Code.Value != CoAPMessageCode.GET)
{
    //only GET method supported..we do not understand this send a RST back
    CoAPResponse unsupportedCType = new CoAPResponse(CoAPMessageType.RST,
/*Message type*/
                                        CoAPMessageCode.METHOD_NOT_ALLOWED,
/*Method not allowed*/
                                        coapReq.ID.Value /*copy message Id*/);
    unsupportedCType.Token = coapReq.Token; //Always match the request/response
token
    unsupportedCType.RemoteSender = coapReq.RemoteSender;
    //send response to client
    this._coapServer.Send(unsupportedCType);
}
```

Listing 4-4 Handling Incorrect Message Code

Checking the URI Request Path

CoAP is designed to expose a machine as a set of RESTful URIs to the external world. Therefore, all interaction with the machine happens over a set of URIs. Just like an HTTP server, it becomes important to check if the request URI is valid or not.

In this sample, we have already extracted the URI path present in the request. Now we justhave to compare that with the path we are expecting. In this example scenario, we have exposed the temperature sensor with the path “sensors/temp”. So as a final validation, we check if the request path is valid or not.

```
else if (reqURIPath != "sensors/temp")
{
    //classic 404 not found..we do not understand this send a RST back
    CoAPResponse unsupportedPath = new CoAPResponse(CoAPMessageType.RST, /*Message
type*/
                                        CoAPMessageCode.NOT_FOUND, /*Not found*/
                                        coapReq.ID.Value /*copy message Id*/);
    unsupportedPath.Token = coapReq.Token; //Always match the request/response
token
    unsupportedPath.RemoteSender = coapReq.RemoteSender;
    //send response to client
    this._coapServer.Send(unsupportedPath);
}
```

Listing 4-5 Handling Incorrect Request URI Path

If the path is not valid, we send back a RST with the message code “NOT_FOUND”.

Sending a Response Back

When all validations are through, it’s time to send a valid response back to the client. For now we will first write a small method to simulate temperature reading.

```
/// <summary>
/// A dummy method to get the room temperature...in real life,
/// this would be the real work the machine/sensor is required to do
/// </summary>
/// <returns>Temp in degree C</returns>
private int GetRoomTemperature()
{
    int temp = DateTime.Now.Second;
    if (temp < 15) temp = 25;// Dummy temperature value generation
    return temp;
}
```

Listing 4-6 Temperature Simulation

With temperature reading taken care of, now it's time to write the code that will send this temperature back to the client.

```
else
{
//All is well...send the measured temperature back
//Again, this is a NON message...we will send this message as a JSON
//string
Hashtable valuesForJSON = new Hashtable();
   valuesForJSON.Add("temp", this.GetRoomTemperature());string
   tempAsJSON = JSONResult.ToJSON(valuesForJSON);
//Now prepare the object
CoAPResponse measuredTemp = new CoAPResponse(CoAPMessageType.NON, /*Messagetype*/
                                        CoAPMessageCode.CONTENT, /*Carries
content*/
                                        coapReq.ID.Value/*copy message Id*/);
measuredTemp.Token = coapReq.Token; //Always match the request/response token
//Add the payload
measuredTemp.Payload = new CoAPPayload(tempAsJSON);
//Indicate the content-type of the payload
   measuredTemp.AddOption(CoAPHeaderOption.CONTENT_FORMAT,

AbstractByteUtils.GetBytes(CoAPContentFormatOption.APPLICATION_JSON));
//Add remote sender address details measuredTemp.RemoteSender =
   coapReq.RemoteSender;
//send response to client
   this._coapServer.Send(measuredTemp);
}
```

Listing 4-7 Sending Response Back to Client

The temperature value is first read and a Hashtable is created. In this Hashtable, the temperature value is added using the key "temp".

CoAPSharp provides a helper class named JSONResult, that can create a JSON string from a Hashtable or can convert a simple JSON string into a Hashtable. The measured temperature is converted into a JSON string and is added as a payload. The JSON string will look like

{ "temp" : 25 }

Once the string is added to the payload, we also add an option to indicate what the content-type of the payload is. After that, the message is sent.

Point to note here is how we have used the message identifier for this NON message. Ideally, we should generate a NON message to be used. The NON message identifier can only be reused after NON_LIFETIME time period has expired. Using the default values as suggested byCoAP draft-18, this value comes to about 150 seconds.

You can choose to write your own ID generator and use them in NON messages.

In this example, we are simply using the ID that was given to us to keep the sample very simple and focus on aspects to handle the request.

Summary

In this chapter, we learnt about basic classes offered by CoAPSharp and elementary request handling using a NON message. This should set the foundation for next chapters. Concepts learned in this chapter will be reused in subsequent discussions.

Chapter 5

Your First CoAP Client

Now that we understand some basic aspects of about how to write a CoAP UDP server andhandle NON requests, it's time to write the client code.

We will build the client that sends NON requests. This is the simplest case. We will send a NON request and the client will hope to get a NON response back from the server.

Example Scenario

We will use the same example scenario that we used while building the NON server. That way, we will cover both the client and the server for a given case. Let's assume that you have a temperature sensor installed in your office conference room. This temperature sensor sends the room temperature to a remote machine installed in the cooling controller station. Thus, the machine installed in the cooling controller station acts like a client and the sensor installed in the conference room acts like a server. We will build a client based on CoAP that will query the server to get the latest temperature. Client will send a NON request to the server and server will respond back with a NON response.

Hardware and Software Requirements

In this book, we will build all samples using the emulator. Therefore, there are no hardware requirements for this example.

You will need Visual Studio IDE with .NET Micro Framework installed. You will also need the CoAPSharp binaries.

Client Design and Logic

The slient side will be built using the CoAPSharp library. We will open a UDP client socket that send message to the server socket that is listening on the default CoAP port 5683. To build the client, CoAPSharp provides a synchronous and an asynchronous class. We will use the asynchronous class as that takes care of many other aspects. Using the synchronous class is for advance users. You should be able to use the synchronous version after you finish the examples in this book. CoAPSharp client provides an event based mechanism. There are threeevents that it exposes; one event when a request is received from remote server, another whena response is received from a remote server and third, when an error occurs within the CoAPSharp library during various internal operations (like parsing an input stream and sendinga message to the remote server)

In this example, we will focus only on the event that is raised when a response is received from the remote server. Given below is the basic algorithm:

1. Client receives the response and extracts the token
2. Client checks if the token matches with the token sent in the request.
3. If there is a match, client checks if the response code is “CONTENT” (meaning success).
4. Upon success, client extracts the payload from the message. The payload is a set of bytes that is converted to a string using a utility provided by CoAPSharp.
5. Once the string is extracted, we know it is in JSON format. We will use the JSONResultclass to parse the JSON string and convert it into a Hashtable of key value pairs.

Starting the CoAP UDP Client

To implement the client, we will use the class named CoAPClientChannel. It has a method named “Initialize” which takes two parameters. The first parameter accepts the DNS name or the IP address of the server and the second parameter accepts which port on which the server is listening. We are using the default port for CoAP UDP server which is 5683.

```
public class Program
    {
        /// <summary>
        /// Holds the instance of the CoAP client channel object
        /// </summary>
        private static CoAPClientChannel _coapClient = null;
        /// <summary>
        /// Used for matching request / response / associated request
        /// </summary>
        private static string _mToken = "";
        /// <summary>
        /// The entry point method
        /// </summary>
        public static void Main()
        {
            string serverIP = "127.0.0.1";
            int serverPort = 5683;

            _coapClient = new CoAPClientChannel();
            _coapClient.Initialize(serverIP, serverPort);
            _coapClient.CoAPResponseReceived += new
CoAPResponseReceivedHandler(OnCoAPResponseReceived);
            _coapClient.CoAPRequestReceived += new
CoAPRequestReceivedHandler(OnCoAPRequestReceived);
            _coapClient.CoAPError += new CoAPErrorHandler(OnCoAPError);
        }
```

Listing 5-1 Setup CoAP UDP Client

After the client is initialized, we have registered the event listeners. The sequence is not important. You can first register and then initialize or vice versa. In this example, we will focus only on the “OnCoAPResponseReceived” event handler.

Sending the NON Request to the Server

Once the client is initialized, we will construct a CoAPRequest object that will carry the NON request to query the temperature at the server side. As explained before, each machine in CoAP is exposed to other machines via a set of RESTful URIs. Therefore, we need to first construct the URI (like an HTTP web service call). The URI will look like

coap://localhost:5683/sensors/temp

Please note that the scheme is now “coap”. For a secure call, the scheme is “coaps”.

Since this is a simple sample, and we are sending only one message, we will hardcode the message identifier for now. Also, for message correlation, we need to add the token. For the token, we are using the DateTime object. Remember, the maximum length of the token can be8 bytes.

Once the CoAPRequest object is setup fully, it is sent to the server. The modified “Main” method now looks like as shown below.

```
/// <summary>
/// The entry point method
/// </summary>
public static void Main()
{
string serverIP = "127.0.0.1";int
   serverPort = 5683;

_coapClient = new CoAPClientChannel();
_coapClient.Initialize(serverIP, serverPort);
_coapClient.CoAPResponseReceived += new
   CoAPResponseReceivedHandler(OnCoAPResponseReceived);
_coapClient.CoAPRequestReceived += new
   CoAPRequestReceivedHandler(OnCoAPRequestReceived);
_coapClient.CoAPError += new CoAPErrorHandler(OnCoAPError);
//Send a NON request to get the temperature...in return we will get a NONrequest
   from the server
CoAPRequest coapReq = new CoAPRequest(CoAPMessageType.NON,
CoAPMessageCode.GET, 100);//hardcoded message ID as we are
using only once
string uriToCall = "coap://" + serverIP + ":" + serverPort + "/sensors/temp";
   coapReq.SetURL(uriToCall);
_mToken = DateTime.Now.ToString("HHmmss");//Token value must be less than 8bytes
coapReq.Token = new CoAPToken(_mToken);//A random token
_coapClient.Send(coapReq);
   Thread.Sleep(Timeout.Infinite);//blocks
}
```

Listing 5-2 Send NON Message to Server

Once this message is sent to the server, we now have to wait for the response.

Handling Response from the Server

If everything goes well, we should get a NON response from the server containing the temperature. The code for the response handler is shown below and is exactly in line with the steps outlined in the previous section on client design and logic explanation.

```
/// <summary>
/// Called when a response is received against a sent request
/// </summary>
/// <param name="coapResp">The CoAPResponse object</param>
static void OnCoAPResponseReceived(CoAPResponse coapResp)
{
    string tokenRx = (coapResp.Token != null && coapResp.Token.Value != null) ?
AbstractByteUtils.ByteToStringUTF8(coapResp.Token.Value) : "";
    if (tokenRx == _mToken)
    {
        //This response is against the NON request for getting temperature we
issued earlier
        if (coapResp.Code.Value == CoAPMessageCode.CONTENT)
        {
            //Get the temperature
            string tempAsJSON =
AbstractByteUtils.ByteToStringUTF8(coapResp.Payload.Value);
            Hashtable tempValues = JSONResult.FromJSON(tempAsJSON);
            int temp = Convert.ToInt32(tempValues["temp"].ToString());
            //Now do something with this temperature received from the server
        }
        else
        {
            //Will come here if an error occurred..
        }
    }
}
```

Listing 5-3 Handling Response from Server

We extract the token from the response and match it with the token sent earlier. If there is a match and if the response code is “CONTENT” (meaning success), we extract the temperature value.

Conversion from JSON string (the payload was sent in JSON string, see chapter 3) to a Hashtable is done via the JSONResult helper class provided by CoAPSharp.

Congratulations, you have written your first CoAP Client !

Summary

In this chapter, we learnt about basic classes offered by CoAPSharp and elementary request sending and response handling using a NON message. I hope the theory presented earlier is now slowly matching up with the implementations. As we go along, I will cover more examples and attempt to explain other aspects of the CoAP specification.

Chapter 6

Timing Considerations

CoAP is a request and response based system. For a given request, generally, the requester is expecting a response. However, due to practical considerations, there are many reasons where the response is not received. In that case, the request is sent again. Without a set of rules, this could become a problem and the network might get flooded with such re-transmitted requests.

Another aspect is to protect the resource constrained device from request flooding. CoAP specification briefly mentions that the device should perform some sort of throttling to prevent from request flood. Beyond that, the specification is largely silent.

However, from the perspective of setting rules on the sender side, the specification has gone into great degree of depth to explain the timing requirements. The timing requirements are largely imposed on the sender side. There are two kinds of messages a sender could send; CON and NON. We will look into each message type and understand what the timing requirements are. I have already mentioned some timing requirements briefly while describing the message identification number, but I will cover them in details here. Please note, the currenttext is based on draft-18 of the proposed specification.

We will start by understanding the different timing parameters that are involved in message exchange. The values given below are the base parameter values. From these base parameters we will derive a few more parameters.

1. ACK_TIMEOUT: This is the number of seconds the sender must wait after sending a CON message for an ACK or RST message before trying again. The default value as per the specification is 2 seconds.

2. ACK_RANDOM_FACTOR: This value is a number that is multiplied to the ACK_TIMEOUT to ensure there is no clashing in timeout values used for subsequent transmission and avoid clash. This value must not be decreased below 1.0 and shouldbe sufficiently different from 1.0. The default value as per the specification is 1.5 seconds.

3. MAX_RETRANSMIT: This is a number that indicates how many times the client shouldretry after the first transmission fails (meaning no ACK or RST was received within the ACK_TIMEOUT period). The default value as per the specification is 4. Therefore, if weuse the default values, a total of 5 transmissions will be made if all transmissions fail (The first one and then subsequently four additional attempts)

4. NSTART: This parameter specifies how many simultaneous pending transmissions the sender should keep with a given server. The default value as per the specification is 1. Thus, it means that you should not have more than one CON message pending for an ACK or RST with a given server.

5. DEFAULT_LEISURE: This is an interesting parameter. If the server receives a request,that is a multicast request, then the server can either not respond at all (depends on the application) or if the server decides to respond, then it must wait for a while before responding. The default value as per the specification is 5 seconds.

6. PROBING_RATE: This is the rate at which a client might send data to an endpoint thatis not responding. The default value as per the specification is 1 byte/sec.

Of all the above values, the most important ones are ACK_TIMEOUT, and MAX_RETRANSMIT. We will focus on these two. We will understand the timing requirements for a CON message and then extend the concepts for a NON message.

The first calculated parameter is called MAX_TRANSMIT_SPAN.

Assume that the first transmission failed, and then, all subsequent four retransmission attempts also failed (with a MAX_RETRANSMIT value of 4). In such a case, from the time the first message was sent, till the time the last message was sent is called MAX_TRANSMIT_SPAN. In the figure below, the situation has been highlighted.

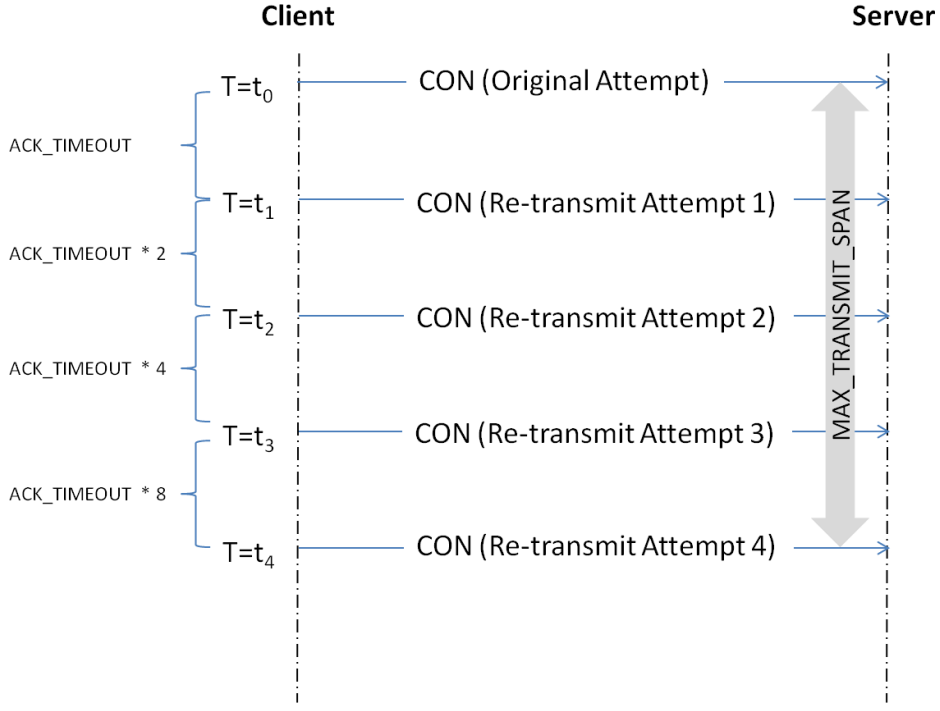

Figure 6-1. The MAX_TRANSMIT_SPAN Period

At time $T=t_0$, the first CON message was sent. The client failed to get a response, and thus, sent the same CON message again after waiting for ACK_TIMEOUT period. The next transmission also did not result in an ACK or a RST message. Therefore, after waiting for (ACK_TIMEOUT * 2) another attempt was made. With each attempt, the timeout value was increased non-linearly. Finally, the 4^{th} re-transmission was sent resulting in total 5 transmissions.

The total time taken to transmit and retransmit the message is therefore

MAX_TRANSMIT_SPAN = (ACK_TIMEOUT * 1.5) + (ACK_TIMEOUT * 2 * 1.5) +(ACK_TIMEOUT * 4 * 1.5) + (ACK_TIMEOUT * 8* 1.5)

For the case outlined above with the default values, the above equation becomes

MAX_TRANSMIT_SPAN = (2 + 4 + 8 + 16) * 1.5 = 45 seconds.

The generalized equation thus becomes

MAX_TRANSMIT_SPAN = ACK_TIMEOUT * ((2 ** MAX_RETRANSMIT) - 1) * ACK_RANDOM_FACTOR

(Please note, the ** symbol bas been used to denote ‘raised to the power of’)

Another derived parameter that we are interested in is called MAX_TRANSMIT_WAIT. This indicates the maximum amount of time the sender will wait before giving up. This is same as MAX_TRANSMIT_SPAN with the addition of wait time for the last transmission. The figure below outlines the situation.

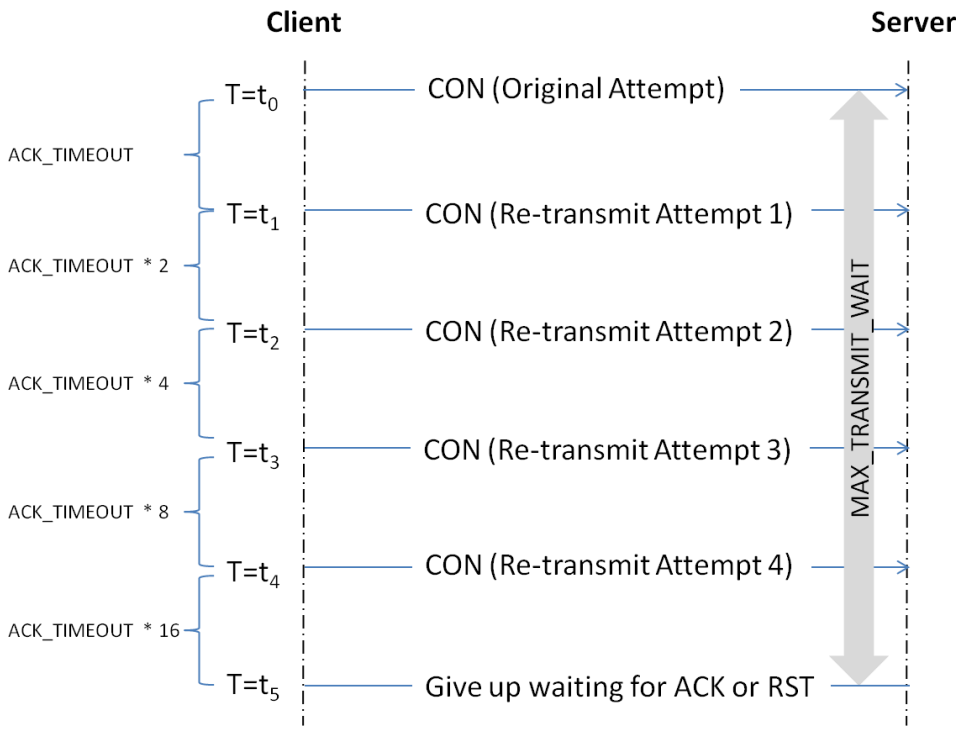

Figure 6-2. MAX_TRANSMIT_WAIT Period

Thus, on the same lines as MAX_TRANSMIT_SPAN calculation, if we use the default values, the MAX_TRANSMIT_WAIT comes out to be

MAX_TRANSMIT_WAIT = (2+4+8+16+32) * 1.5 = 93 seconds

More generally, it can be expressed as

MAX_TRANSMIT_WAIT = ACK_TIMEOUT * ((2 ** (MAX_RETRANSMIT + 1)) - 1) *

ACK_RANDOM_FACTOR

(Please note, the ** symbol bas been used to denote ‘raised to the power of’)

Another parameter that we need to take into account is the amount of time it takes a datagram to reach from sender to recipient. This is called MAX_LATENCY. The value of MAX_LATENCY has been arbitrarily chosen as 100 seconds in the specification, which is reasonably realistic.

The next parameter that we need to account for is called PROCESSING_DELAY. This is the amount of time that a machine will take to process the request. Given that the recipient machine will attempt to process and send the response within the ACK_TIMEOUT, this is set equal to ACK_TIMEOUT value.

Finally, the last parameter that we define is MAX_RTT which indicates the maximum round-trip time. MAX_RTT can be expressed as

MAX_RTT = MAX_LATENCY (request) + PROCESSING_DELAY + MAX_LATENCY (response)

Or more generally,

MAX_RTT = (2 * MAX_LATENCY) + PROCESSING_DELAY

With all the above parameters, we are now ready to define what is the maximum time required for a CON message to stay in the system before it is either acknowledged (ACK or RST) or it is trashed (meaning recipient is no longer responding). This time is called EXCHANGE_LIFETIME.

It is defined as

EXCHANGE_LIFETIME = MAX_TRANSMIT_SPAN + MAX_RTT

With the default values, it comes to 247 seconds. Therefore, for a CON message, it is safe toreuse the message identifier after 247 seconds if we are using the default values.

While EXCHANGE_LIFETIME is important for a CON message, a NON message is treated slightly differently. If you do not wish to retransmit the same message, you can safely reuse the message identifier after MAX_LATENCY period (the time it takes for your message to reach thesender). However, if you wish to resend the NON message, you need to define a lifetime for a NON message. Not surprisingly it is called NON_LIFETIME. It is defined as

NON_LIFETIME = MAX_TRANSMIT_SPAN + MAX_LATENCY

With the default values, this is equal to 145 seconds.

Summary

While this chapter seemed too theoretical, it gives you the full details of how you should manage the message identifier generation. CoAPSharp library provide a default implementation of re-transmissions. You can change the ACK_TIMEOUT and MAX_RETRANSMIT values for each message that you send to server. If a transmission fails, CoAPSharp library will automatically resend the message until all retransmission attempts fail. If that happens, the CoAPSharp library calls the error event providing the details of which message transmission failed.
Additionally, it also provides a safe way to generate message identifiers. More on this in later chapters.

Chapter 7

Sending Confirmable (CON) Requests

The Constrained Application Protocol is designed to run over UDP. By definition, a datagram is not reliable. To bring in a degree of reliability in CoAP exchanges, the Confirmable (CON) message concept was defined. When a CON message is sent to the remote machine, the sender expects a response back. This way, there is some confirmation that the sender received the message. If a response is not received, the sender continues to send the same CON message until all retry attempts are exhausted. Sending CON messages requires adherence to various set of rules. In this chapter, we will understand the various rules and also write a client implementation that sends a CON message as per the rules defined.

Rules for Sending a CON Message

A CON message is sent to a remote machine to solicit a response. Therefore, you need to clearly understand if you really need a CON message for a given scenario. Sending and processing a CON message requires considerable resources (in the Constrained Application world) and you must evaluate all options before deciding on using a CON message.

A CON message must wait for a response before the message is assumed to be consumed by the remote recipient. If a CON message is received by the recipient, it will either send an ACK, indicating it understood the message and ACK has the necessary response details, or it will send a RST indicating the recipient lacks context (or is not in the right state currently) to processthe message. In either case, if an ACK or a RST is received, the CON message is assumed to be consumed and is therefore considered “confirmed”.

As long as an ACK or an RST does not arrive, the sender needs to wait before proceeding further with the message. In previous chapters, we understood the timing considerations and those rules must apply here. The CON message sender must wait for at-least ACK_TIMEOUT seconds, before it decides to resend the message. Also, when the message is resent, the timeout needs to be increased in a non-liner fashion. With each retry attempt, the ACK_TIMEOUT value is re-calculated and increased non-linearly. The retry mechanism continues until an ACK or a RST is received, or, all retry attempts are exhausted.

Given that this is a processing heavy procedure, it is recommended that the sender keeps only one pending CON message that is waiting for a response.

The CoAPSharp library takes care of almost all the requirements for sending a CON message. Once you send the CON message, the CoAPSharp library keeps that message pending in its internal queue. The internal queue does not have any hard limit. It is up to the application programmer to ensure that no too many messages are sent at once to overload the queue. Once an ACK or a RST is received, the CON message is cleared from the queue. Additionally, there is a timer that runs internally within the CoAPSharp asynchronous client implementation. Once the CON message that is waiting for response times out, CoAPSharp implementation automatically re-sends the CON message. This loop continues until all attempts are exhausted. When this happens, CoAPSharp library invokes the error handler and provides the CON message that was not delivered. From there on, it's your responsibility to handle the error. The error thrown by the CoAPSharp implementation is an exception of type UndeliveredException.

It might happen that you receive multiple responses for the same CON message. This may happen if multiple tries were made to send the CON message and both arrived at the server. The server in turn, instead of discarding one, sent response for both. Therefore, when you builda client, you should prepare for such a situation on what to duo when you receive a duplicate response.

Building the Client to Send a CON Message

We will again take an example to understand the sample code. Once again, let's take the conference room temperature sensor example. The temperature sensor in the conference roomis the server that responds back to a client request to get the temperature. In this example, the client will send a CON request to the server to get the temperature. Once the CON request is received from the client, the server responds back with an ACK response, and in this ACK response, the server piggy-backs the measured temperature value also.

```
/// <summary>
/// Entry point
/// </summary>
public static void Main()
{
    Program p = new Program();
    p.SetupClient();
    p.SendRequest();
    Thread.Sleep(Timeout.Infinite);
}
```

Listing 7-1 The main method

The main method is very simple; it sets up the CoAP client object, sends a CON request and then waits for a response. In this example, we are using the asynchronous implementation of CoAP client provided by CoAPSharp library.

Setup the CoAP Client Object

As a first step, we need to setup a socket client that connects with the server over UDP. CoAPSharp provides a built-in asynchronous socket client class called CoAPClientChannel.

We will use this class to setup our client side. We need to specify the IP address (or Domain name) and the port number on which the UDP connection will be made. Given that this is an asynchronous class, the communication with server is managed via events. There are three events that this class exposes:

1. CoAPRequestReceived – This event is raised when a request is received from the server (CON or a NON with message code as GET, POST, PUT or DELETE)
2. CoAPResponseReceived – This event is raised when a response is received from the server (NON, ACK, RST and message code is not GET, POST, PUT or DELETE)
3. CoAPError – This event is raised when an error occurs during processing

There is another client class provided by CoAPSharp which is synchronous and is called CoAPSyncClientChannel. Since it works synchronously, there is no event based mechanism and thus, you are responsible for taking care of all rules. This is for advance users who clearly understand CoAP specification and want a custom implementation. For now, we will use the asynchronous implementation. The code to setup the client class is shown below.

```
/// <summary>
/// Setup the client
/// </summary>
public void SetupClient()
{
    this._client = new CoAPClientChannel();
    this._client.Initialize("127.0.0.1", 5683);
    this._client.CoAPError += new CoAPErrorHandler(OnCoAPError);
    this._client.CoAPRequestReceived += new
CoAPRequestReceivedHandler(OnCoAPRequestReceived);
    this._client.CoAPResponseReceived += new
CoAPResponseReceivedHandler(OnCoAPResponseReceived);
}
```

Listing 7-2. Setup the client class for communication

This code simply provides the address and port number and initializes the client UDP socket. It also registers event handlers for all the three events supported by the client class. For sending a CON request, this should be sufficient for now.

Send the CON Request to Server

Once we have setup the client, it's time to send the CON message to the server. Once a CON message is sent, CoAPSharp library stores the message internally. Only when a matching ACK or RST message is received, the message is removed from the storage. This is done to ensure automatic re-transmissions. CoAPSharp library contains a timer object that continuously checks for timed-out CON messages. Once a timed-out CON message is found, it sends the message

again after making appropriate calculations on the new value of timeout. When the message is sent, it is again stored internally and the loop continues. Upon timeout, the library checks if any further re-transmission attempts are pending. If all re-transmission attempts have exhausted, the CoAPError event is raised. In this event, the exception of type UndeliveredException is passed as one argument, while the other argument provides the original CON message that theclient intended to send.

Given that there could be pending CON messages in the send queue, it becomes important to choose a message identifier that does not conflict with pending messages. For this, we need to ensure that we choose a number that is not already being used. The most simplistic approach is to read through all pending messages that are stored internally, and then use a number that does not match with any pending message identifier. Since the CoAPSharp library maintains all pending messages internally, it has access to all “in-use” message identifier. It exposes a simple method to get the next message identifier that does not conflict with the blocked message identifiers.

Thus, to generate the message identifier, we will use the method provided by the CoAPSharp library.

The code listing to prepare a CON request and to send the message is given below.

```
/// <summary>
/// Send the request to get the temperature
/// </summary>
public void SendRequest()
{
    string urlToCall = "coap://127.0.0.1:5683/sensors/temp";
    UInt16 mId = this._client.GetNextMessageID();//Using this method to get the
next message id takes care of pending CON requests
    CoAPRequest tempReq = new CoAPRequest(CoAPMessageType.CON,
CoAPMessageCode.GET, mId);
    tempReq.SetURL(urlToCall);

    /*Uncomment the two lines below to use non-default values for timeout and
retransmission count*/
    /*Dafault value for timeout is 2 secs and retransmission count is 4*/
    //tempReq.Timeout = 10;
    //tempReq.RetransmissionCount = 5;

    this._client.Send(tempReq);
}
```

Listing 7-3. Sending a CON request

The important aspects to note in the code is that we are using a method “GetNextMessageID()”to get the next message identifier. This ensures we get an identifier that is not currently in use.

Additionally, the commented code outlines how you can provide customized values for the timeout and the maximum retransmission attempts. The default values used by CoAPSharp for the timeout is 2 seconds and maximum retransmissions is 4. This is in line with the suggested

values in the CoAP specification. If you change these values, you must be absolutely sure of what you are doing. One on hand, you do not want to wait too long for an acknowledgement andon the other hand, you do not want to continue to retransmit and close the already constrained network.

Outmost care must be taken while choosing custom values of acknowledgement timeout and retransmission attempts.

Once we have sent a CON request, we expect a response back, either an ACK message or a RST message. Once a response is received from the server, the CoAPResponseReceived event is raised.

Handling Server Response

A CON message expects a response back from the server. Either an ACK or an RST is expected. This is handled in the CoAPResponseReceived event. A response object can have message type as RST, ACK or NON and the message code must not be GET, PUT, POST or delete.

What you do with the response is entirely based on the use case you are implementing. In this sample, we simply want to show you how you can receive the response. The code listing for the response handler is shown below.

```
/// <summary>
/// We should receive the temperature from sever in the response
/// </summary>
/// <param name="coapResp">CoAPResponse</param>
void OnCoAPResponseReceived(CoAPResponse coapResp)
{
    if (coapResp.MessageType.Value == CoAPMessageType.ACK &&
        coapResp.Code.Value == CoAPMessageCode.CONTENT)
    {
        //We got the temperature..it will be in payload in JSON
        string payload =
AbstractByteUtils.ByteToStringUTF8(coapResp.Payload.Value);
        Hashtable keyVal = JSONResult.FromJSON(payload);
        int temp = Convert.ToInt32(keyVal["temp"].ToString());
        //do something with the temperature now
    }
}
```

Listing 7-4. Handling response from server

The response from server is provided as a CoAPResponse class instance. If we expect a piggy-backed response in an ACK message, then the payload will contain the response data. In this specific example, the payload comes back as a JSON response.

CoAPSharp contains a very basic implementation of JSON parser. This should be sufficient given that we are working with constrained devices. The bytes in the payload are first converted to string. The string is then parsed to get a Hashtable with key value pairs translated nicely.

Note on Using CON Messages

CON messages require a response from the server. This puts additional processing requirements on the hardware, software and the network. Given that we are talking about constrained devices, it is best to use CON messages sparingly. You should use CON messageswhen you really require a confirmation, like, was the actuator closed or did we turn on the oxygen cylinder. Requesting temperature data over CON messages is generally not a good idea.

Summary

In this chapter, we took the first steps towards working with confirmable messages. Confirmable messages are the most complicated and consume more resources that other message types. A CON message must be used only as a last resort. We also learnt various timing requirements, re-transmissions and how to ensure unique message identifiers.

In the next chapter, we will see how to write a server that can handle a CON message.

Chapter 8

Receiving Confirmable (CON) Requests

In the previous chapter we learnt how to write a client application that sends a confirmable request. In this chapter, we will build a server application that receives the confirmable request. Receiving a confirmable request is no different from receiving any other request, however, what the receiver must know is that it is required send a response back to a CON message type. The response message should be of type ACK or RST.

Sending an ACK back indicates to the client that the message was understood and processed as per rules (Please remember, sending back an ACK message with an error message code also means that the message was understood and appropriate action was taken by the server). Only in rare cases, where the message is not understood at all, or there is lack of context at the server end, a RST message can be sent back informing the client that we do not understand this message. The phrase “lack of context” has been used multiple times in the CoAP specifications without pointing out what could be a real use case for that scenario. At this point of time, the specification is not clear. It could be that a CON message was sent by the client to get the temperature value, but the server is still initializing after a reboot (or waking up from sleep mode).

Another point to note is that the server cannot take too long to process the CON message. When we covered timing considerations in previous chapters, we introduced a term called PROCESSING_DELAY. This is the amount of time a recipient will take to process a message. As per the specification, the default value is set equal to the ACK_TIMEOUT value. The problem is that ACK_TIMEOUT is set at the client end and server is not aware of this. Therefore, both client and server, must agree on the timeouts used prior to starting the communication. This will ensure, that the client, which sent the CON message does not resend the CON message thinking it was lost, while the server is still processing the message.

As of now, the specification is silent on how to exchange the timeout values, therefore, till that happens, I suggest that each server designer must clearly outline in the documentation, what isa realistic processing delay that can occur. This will help the client caller to adjust it’s timeouts before making the call.

Well, that’s as much theory you need before jumping on to code.

Setting up the Server

Create a new .NET Micro Framework type console project (or a Netduino Plus 2 project if you want to run this on a Netduino board)

Setting up a UDP server in CoAPSharp is very simple. Given that the server is based on events, you simply start the server and suspend the main thread (well, for this sample. What you do with the main thread will vary based on the context).

```
/// <summary>
/// Entry point
/// </summary>
public static void Main()
{
    Program p = new Program();
    p.StartServer();
    Thread.Sleep(Timeout.Infinite);
}
/// <summary>
/// Start the server
/// </summary>
public void StartServer()
{
    this._server = new CoAPServerChannel();
    this._server.Initialize(null, 5683);
    this._server.CoAPResponseReceived += new
CoAPResponseReceivedHandler(OnCoAPResponseReceived);
    this._server.CoAPRequestReceived += new
CoAPRequestReceivedHandler(OnCoAPRequestReceived);
    this._server.CoAPError += new CoAPErrorHandler(OnCoAPError);
}
```

Listing 8-1. Setup and start CoAP server

For this chapter, we will focus on the “OnCoAPRequestReceived” event. This is the event that will get called when we receive a CON message from the client.

Processing a message received, required ensuring that the message we received is a message that we support and has all the information to process the message.

Therefore, the very first things that we need to do in this handler are to perform validation checks.

Again, we are taking the example of the temperature server. In this example, temperature is returned by the server against a CON request of type GET issued at the path sensors/temp.

Thus, we first need to validate if the message type is CON, whether the message code is GET and if the path is indeed sensors/temp. The code listing given below performs the validation functions.

```
/// <summary>
/// Called when a CoAP request is received...we will only support CON requests
/// of type GET... the path is sensors/temp
/// </summary>
/// <param name="coapReq">CoAPRequest object</param>
void OnCoAPRequestReceived(CoAPRequest coapReq)
{
    string reqPath = (coapReq.GetPath() != null) ? coapReq.GetPath().ToLower() :
"";

    if (coapReq.MessageType.Value == CoAPMessageType.CON)
    {
        if (coapReq.Code.Value != CoAPMessageCode.GET)
        {
            CoAPResponse resp = new CoAPResponse(CoAPMessageType.ACK,

CoAPMessageCode.METHOD_NOT_ALLOWED,
                                                 coapReq /*Copy all necessary
values from request in the response*/);
            //When you use the constructor that accepts a request, then
automatically
            //the message id , token and remote sender values are copied over to
the response
            this._server.Send(resp);
        }
        else if (reqPath != "sensors/temp")
        {
            //We do not understand this..
            CoAPResponse resp = new CoAPResponse(CoAPMessageType.ACK,
                                                 CoAPMessageCode.NOT_FOUND,
                                                 coapReq /*Copy all necessary
values from request in the response*/);
            this._server.Send(resp);
        }
```

Listing 8-2. Validating received message

We first check if the message type is CON. In this example, we are not even responding back if the message is NON (Please note, the event will be raised only when the incoming message is of type NON or CON and the message code is GET, PUT, POST, DELETE, EMPTY)

You can decide what you want to do if the message type is not what you expect (meaning you are expecting a CON and a NON arrives or vice versa). Either you can respond back with a message code as BAD_REQUEST or take any other appropriate action appropriate to your context. Additionally, you might end up getting a duplicate message (when client sends the same message twice thinking earlier message was lost). At the server side, you need to decidebased on your context as to how to address such duplicate messages.

Let's come back to the code listing above. We first check if the incoming request is of type CON. After that, the very first check we make is to see if the request code is GET. If it's anything other

than GET, we respond back with an ACK (and not a RST as explained before) indicating that the method code is not allowed. Only GET method is allowed.

Next, we check if the URI path is the right one, and if not, we again respond back with an ACK but the message code is class NOT FOUND (Like 404 not found in HTTP).

When both checks are successful, it's time to respond back with the measured temperature.

```
CoAPResponse resp = new CoAPResponse(CoAPMessageType.ACK,
                                     CoAPMessageCode.CONTENT,
                                     coapReq /*Copy all necessary values from
request in the response*/);
//The payload will be JSON
Hashtable ht = new Hashtable();
ht.Add("temp", this.GetRoomTemperature());
string jsonStr = JSONResult.ToJSON(ht);
resp.AddPayload(jsonStr);
//Tell recipient about the content-type of the response
resp.AddOption(CoAPHeaderOption.CONTENT_FORMAT,
AbstractByteUtils.GetBytes(CoAPContentFormatOption.APPLICATION_JSON));
//send it
this._server.Send(resp);
```

Listing 8-3. Sending back the response

Once all checks are successful, we create a response message of type ACK. The method code this time is CONTENT, indicating this response has data that was originally requested.

The third argument is original request that was received. This is a shortcut to completely create the response message. As outlined in the previous chapters, message correlation is done through the token and the message identifier. When you create the response message, you need to copy over the message identifier and the token to the response. This way, both values are sent back to the client to help it match the response against a specific request.

The additional task, this constructor performs, is to copy over the remote sender IP address and port number to the response object. This is used internally by CoAPSharp. When the Send method of the server is called, it extracts the remote sender details and sends the message to the client identified by the remote endpoint. This value is never passed back to the client and only for internal usage.

If you notice, this method gets the temperature and creates a JSON string using the JSONResult helper class (how temperature is measured is outside the scope of the topic. Just assume there is a method that knows how to get the temperature from the attached temperaturesensor)

Finally, the content type identifier is added to the response options (as APPLICATION_JSON) and the response is sent back to the client.

As an implementer, you need to define what is a realistic timeframe for this method to completeand that is what you should publish in your documentation as the processing delay at your end.

This will help client callers adjust the timeout values so that they do not keep sending duplicate CON messages thinking that the message was lost.

Summary

In this chapter, we understood how to receive a CON (or confirmable) request, what is the importance of processing delay, how does we process an incoming CON request and how to respond back. This gives you enough information to jump into more advance topics. In subsequent chapters, I will cover the advance topics like resource discovery, separate response, observe notifications and block transfers.

Chapter 9

Supporting Resource Discovery

With time, we will have more and more CoAP based devices deployed in the field. Soon, these deployed devices will face the problems humans faced during the early stages of Internet; how do you find what devices are deployed and what are their capabilities. The problem becomes more complicated given the fact that no humans are involved in the process to fire a web searchand interpret the results.

This is where the term “Resource Discovery” comes into picture. A client device needs to learn about the services offered by a CoAP server. As per CoAP specification [6], a client device learns about the server by learning a URI that references a resource in the namespace of the server. Alternatively, a client device can use Multicast CoAP and “All CoAP Nodes” multicast address to find CoAP servers.

The rules state, that any server that offers resource discovery must support the default CoAP port 5683 and the data format for the results returned should be in CoRE Link Format [7]. CoRE link format is a way to serialize links that describe relationship between resources, and is also called “Web Linking”.. The Internet media type is “application/link-format” for the CoRE Link Format.

The response of a resource discovery query in CoAP world returns the result in the payload rather than the usual known way of returning HTTP link headers. The specifications indicates that it’s perfectly possible to have a resource discovery server (e.g. www.coapfinder.com) that only allows resource discovery, whereas, actual services are served by different servers.

Basic Discovery Process

A client issues a GET to a server, whose address is known to the client and uses the path “/.well-known/core”. A server supporting resource discovery must support this URI. The server returns the resources in CoRE Link format.

In cases where multicast resource discovery is required within a limited scope, a GET request to the multicast address is made. Given that querying a multicast address might result in large responses, it is advisable to provide some kind of filtering information in the query. I will describe more on the parameters later on in this chapter.

Before we delve deeper into an understanding of details of the format, we need to first understand what are the different use cases where such discovery process will be used. There are primarily 3 different cases; the first one is the obvious case of resource discovery, the second one is about resource collections and the third one is about maintaining a resource directory.

Resource Collections

In many cases, the device may host multiple instances of the same service type. For example, a device may contain many temperature sensors or it may contain multiple alarms. How to get access to the collection needs to be clearly outlined. The /.well-known/core discovery service must indicate the entry point for such a collection. One can define the collection in two ways – either by simply providing the interface description with a size parameter, or by describing each resource as one entry in the response payload for a resource discovery query.

Resource Directory

It is perfectly possible to maintain a resource directory, something like a phone directory, which other devices can query and get information. This resource directory can also support a POST and PUT to /.well-known/core URL to allow creation or update of the resources held by the actual device.

Sample Message Exchange

A client sends the following to the server that supports resource discovery:

1. Message Type – CON
2. Message Code – GET
3. URL - /.well-known/core

The server responds back with the following:

1. Message Type – ACK
2. Message Code – CONTENT
3. Payload – A payload in CoRE Link Format

The CoRE Link Format is a complex format. For simplicity, I have outlined the sample below with relevant attributes only. The response payload is of type

```
<sensors/temp>;title=temperature sensor; sz=3200; obs,
<sensors/humidity>;title=humidity sensor; sz=8000,
<actuators/temp>;title=temperature valve actuator; sz=8000,
<actuators/humidity>;title=humidity control valve actuator; sz=8000
```

Listing 9-1. Sample response in CoRE Link Format

The listing above outlines how multiple resources hosted by the server are exposed to the client. Each resource description is separate by a comma and attributes within the resource description are separated by a semi-colon. The first entry is the URL using which the client can access the resource. The second entry with the key “title” is just a human readable description. While this will not be used by machines, this just provides a hint to humans developing or searching the device. The third attribute indicates an approximate size of the payload in bytes for a GET request.

In the first line, there is another parameter called “obs”, but this is not associated with any value. This is simply an indicator that the URL supports observation.

Important Link Attributes

As described before, each entry in the comma separated list contains the URI-reference in angle brackets (“<>”) followed by a set of attributes separated by semi-colon. Some of the key attributes are:

1. Resource Type Attribute ”rt” – This attribute indicates the application specific semantic tothe resource. You can think of this as additional information regarding the resource. For example, if the URI-reference is <sensor/temp>, the “rt” attribute could have a value thatstates “kitchen-temperature”. Alternatively, this can also be a URI-reference. Multiple resource type values can be assigned to this attribute, with the separator being space. This attribute must not appear more than once in the link description. Also, this is not for humans, instead, the “title” attribute is for humans.
2. Interface Description Attribute “if” – This attribute describes the specified interface. Thisis an opaque string that describes the details of how to interact with the target resource.Think of this as the place where you can get the details of the “API” (like in API docs). Consider for example that a device hosts multiple temperature sensors (e.g. kitchen temperature, microwave temperature, living room temperature), then to know how to access these resources, you need to refer to the interface description that is made available via the key “if”. As of now the specification is not very clear on how this will used by the devices. One example the specification provides is that this could describe the input/output details (like WADL), but then, for a constrained device, having so muchof processing power to infer from a description on actions could be too much to ask for.
3. Maximum Size Attribute “sz” – This indicates the maximum size of a representation that will be returned during a GET request. The recommendation is that this attribute should not be included for resources that can return the representation that are smaller than the

single MTU (Maximum Transmission Unit). For resources, that will return representations larger than a single MTU, this attribute must be present (as it will help inblock transfers). This attribute must not appear more than once. There is no upper limit to the value of this parameter.

4. The Observe Attribute “obs” – This attribute does not have a value. When present, it indicates that the resource described by the link can be observed.

Summary

In this chapter, we understood basics of resource discovery and how clients can query and understand the resources hosted by the server. I believe that the resource discovery and the CoRE Link Format has to do more before this concept can really be useful and can be automated fully. As of now, our suggestion is to have good documentation of the server capabilities in addition to a basic resource discovery support.

Chapter 10

Separate Response and Requests

So far, we learned a great deal about sending and receiving messages between two or more participating machines using the CoAP protocol. In all the examples, we assumed that if a response is expected, then within a reasonable period of time, we will get a response back. In case we do not get a response back, we can always retransmit the request (assuming that the last request probably never made it). Sometimes, such an assumption is not valid. Remember, when we learnt about various timing considerations, we talked about the term PROCESSING_DELAY. This was assumed to be around 2 seconds. What if, the machine, to whom we have sent the request, takes longer, much longer that 2 seconds. In that case, it will not be able to respond within a reasonable time but that should not be interpreted by the calling machine as dead request. This is where, the concept of a “separate response” originates. This mechanism allows more time to the responder to provide a response, without the client thinking that it’s request was probably lost of there was an error.

Separate Response and Non-Confirmable Messages

In cases where the response is sent back as a NON message, it’s already a separate response. The client does not expect the response to come back piggy-backed in the ACK message. The client simply sends the request and hopes for a NON message at some point of time. This is the most simplistic example of a separate response.

Separate Response and Confirmable Messages

When the client sends a CON message, it expects a response back. In such a case, the server can immediately send an empty ACK message. This will indicate to the caller that the message was received by the server and will be acted upon. Now, once the server has the information requested by the client, it can send it back. However, it must be noted that the separate response must be either a NON or a CON message. We should not send the response in a ACK message as it breaks the protocol. Whether to send the response back as a CON or a NON message is based on the context. If we need guaranteed delivery, we should use CON message else we should use a NON message.

Separate Response Exchange Mechanism Identification

While the separate response mechanism is just about how various messages are exchanged, the specification does not provide any guidance on how to inform both client and server that they need to use the separate response mechanism. There is no option that the client can send,or rather, server can send back in the ACK message indicating that it will respond using the separate response mechanism. As of today, the best we can do is to provide this information in the documentation. During the implementation, you can create a special URL (e.g. /sensors/temp/separate) to indicate that the response from this URL will be separate response. Additionally, during documentation, you can outline this special need.

Implementing a Server that uses Separate Response Mechanism

Implementation a server that sends a separate response against a request is a bit different from normal request response mechanism. Assume that we have a temperature sensor that requires some sort of time consuming processing before it can respond back to the temperature request. Also, there could be more than one client requesting for the same data.

We can devise a simple mechanism to address both the issues. First, every time a request is received on the specific URL, the server will immediately send and ACK message. Next, we will start a new thread, passing the request received as a parameter to the thread (so that the original request is available to the thread procedure). Please note, this activity is being done in aresource constrained device. You may want to limit how many such requests your server can handle. If you receive more requests than that, you may want to send a “Server Busy” response.

By the time you reach this chapter, you are already familiar with basic aspects of setting up a server, wiring the various events and performing validations. Thus, we are only listing the relevant code snippet.

```
CoAPResponse resp = new CoAPResponse(CoAPMessageType.ACK,
                                     CoAPMessageCode.EMPTY,
                                     coapReq.ID.Value /*Copy over request message
Id*/);
                    //Add the remote sender details..this is used internally by
CoAPSharp
resp.RemoteSender = coapReq.RemoteSender;
//send it
this._server.Send(resp);
/* do some heavy work to get the response data*/

Thread heavyWork = new Thread(() => DoTimeTakingTask(coapReq));
heavyWork.Start();
```

Listing 10-1 Responding to request and initiating a thread for separate response

The above code snippet shows that once you receive the request, respond back with an empty ACK. How the time-consuming work is performed is purely based on your context. In this example, we are simply creating a thread and we pass the original request to the thread.

In this thread, we will respond back with a CON message. Again, whether to use CON or NONis purely based on the needs. The sample below provides information on how to send the response back.

```
private void DoTimeTakingTask(CoAPRequest sepReq)
{
       Thread.Sleep(10000);//simulate a time taking task
        //Send the response back as another CON request
       CoAPRequest req = new CoAPRequest(CoAPMessageType.CON,
       CoAPMessageCode.POST, 100);
        //Copy over needed values
        req.RemoteSender = sepReq.RemoteSender;
        req.Token = sepReq.Token;
        //Add some value that is the result of this heavy work
        req.AddOption(CoAPHeaderOption.CONTENT_FORMAT,
AbstractByteUtils.GetBytes(CoAPContentFormatOption.TEXT_PLAIN));
       req.AddPayload("Result of heavy work");
       this._server.Send(req);
}
```

Listing 10-2 Sending the separate response

The important aspect to note here is that we are sending another request and not a response object. Given that CoAP specification does not do a good job of defining what is a request and what is a response, we have followed our own convention that any message code of type GET, PUT, POST and DELETE, is a request. Additionally, we want to send a confirmable message and want a response back, therefore, we should indeed a send a request and not a response. With the same logic, a NON message can be treated as a special request where we are not interested in the acknowledgement response. Also note that the ID is hardcoded in this sample.

Ideally, you should use your own ID generation mechanism. You must make sure that the token of the original request must be copied over in this message. This will help in matching at the client end.

Implementing a Client to Handle Separate Response

A client that handles separate response is no different from any other client. The difference liesin how you handle a response and how do you map the separate response with the original request. The server example we gave earlier returns a CON message. Therefore, on the client side, we should expect to handle a response of type CON. To map this response, we can look at the token. The token of the response should match the token of the request.

Sending a request to a URL that will have a separate response is same as making any other call.

```
/// <summary>
/// Send the request to get the temperature over a separate response
/// </summary>
public void SendRequest()
{
    string urlToCall = "coap://[ip address]:[port]/sensors/temp/separate";
    UInt16 mId = this._client.GetNextMessageID();//Using this method to get the
next message id takes care of pending CON requests
    CoAPRequest tempReq = new CoAPRequest(CoAPMessageType.CON,
CoAPMessageCode.GET, mId);
    tempReq.SetURL(urlToCall);
    tempReq.AddTokenValue(“123”);

    /*Uncomment the two lines below to use non-default values for timeout and
retransmission count*/
    /*Dafault value for timeout is 2 secs and retransmission count is 4*/
    //tempReq.Timeout = 10;
    //tempReq.RetransmissionCount = 5;
    this._client.Send(tempReq);
}
```

Listing 10-3 Sending a request to a URL that will have separate response

We will get an empty ACK message as the first response. After a while, we should get a CON message whose token is same as the token of the request that was sent originally.

On the client side, we will receive two messages (for this example). The first message will be an empty ACK message and the second message will be the CON message.

```
/// <summary>
/// We should receive the ACK from server
/// </summary>
/// <param name="coapResp">CoAPResponse</param>
void OnCoAPResponseReceived(CoAPResponse coapResp)
{
    if (coapResp.MessageType.Value == CoAPMessageType.ACK && coapResp.Code.Value
== CoAPMessageCode.EMPTY)
    {
        //Server wants some time to process our request...wait
    }
}
/// <summary>
/// Server sends the separate response as a CON request
/// </summary>
/// <param name="coapReq"></param>
void OnCoAPRequestReceived(CoAPRequest coapReq)
{
    //For a separate response, we get a CON request back...
    if (coapReq.MessageType.Value == CoAPMessageType.CON &&
        coapReq.Code.Value == CoAPMessageCode.POST)
    {
        //Do token matching here...TODO::
        Debug.Print(coapReq.ToString());
        //Send the acknowledgement that we received the separate response
        //We are sending the "CREATED" status code to indicate to server
        //we did something with the response
        CoAPResponse resp = new CoAPResponse(CoAPMessageType.ACK,
                                             CoAPMessageCode.CREATED,
                                             coapReq);
        this._client.Send(resp);
    }
```

Listing 10-4 Handling messages received from server

Summary

In this chapter we learnt about separate responses. Separate response mechanism allows client and server to elongate the time span of the conversation. This is a way for the server to ask for additional time to process the message. The specification does not indicate how long a client should wait, even for a separate response, neither does it provide any mechanism to identify that a request or response is actually a part of separate response. For now, all of these are decisions that you have to take during your design.

Chapter 11

Observable Resources

When we start to install various machines, we believe there would be a large number of machines that will simply be sending data that others will consume and use for actionable decisions. Consider this, if we will have a large number of devices that are simply responsible for sending data, does it make sense to repeatedly ask that device for the data? If there are a lotof “listeners” for the data, the problem becomes even more complicated. A large number of listeners would simply be asking for data and would be clogging the network.

In essence, there could be a large number of use cases, where a parameter is continuously being monitored or acted upon. With each read or action, there could be a set of entities that are interested to know “what happened”. Given that this is a repetitive action, it makes more sense to simply send the results of the measure or the action to “interested listeners”. Therefore, we can say that the “interested listeners” are observing a resource.

What is Observing a Resource

When a resource is either being monitored (like a temperature sensor measuring temperature variations of a specific area) continuously, or is being acted upon continuously (like regular opening and closing of valves), what we really want is to simply be “informed” when the change happens (e.g. when the temperature changes more than 1 centigrade or the valve opens or closes). The concept of “Observing a Resource” essentially means to get notified when a change occurs at the resource.

To get notified, the observer can simply register itself as an interested party. When the resource changes its state, the observer can be sent a notification. If the observer is no longer interested,it can send a de-registration request. This approach greatly reduces the burden of unwanted traffic, unnecessary processing and unnecessary polling. The resource being observed is simplycalled “Observed Resources” and interested parties are called “Observers”.

CoAP Specifications for Observable Resources

CoAP specification has made provision for observing a resource. There are very simple rules. The observer sends a registration message and then simply waits for notifications. The observed resource then sends notifications on every change until the observer sends an un-register message.

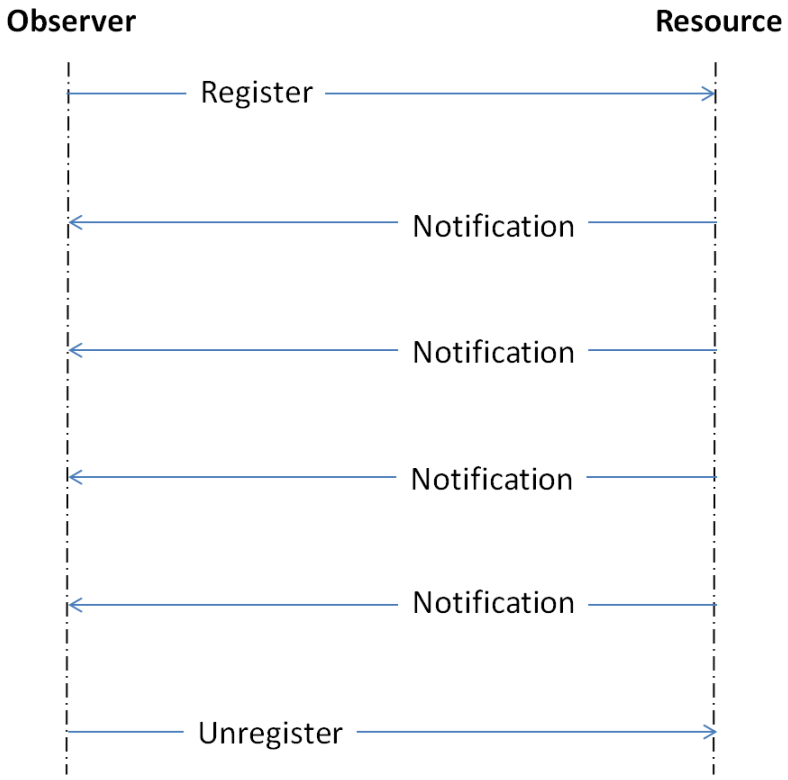

Figure 11-1 Basic message flow for observable resources

The CoAP specification introduces an option called OBSERVE option. The observer will send a request for registering itself with the observable resource. In this request, it should send the OBSERVE option and must set the value to zero.

When the observable resource receives the request, it should check for an OBSERVE option. The value of the OBSERVE option must be ignored by the observable resource. The observable resource would then mark the caller as one more observer in its internal list. From this point onwards, when the observable resource has data to send, it will iterate through its list of registered observers and send the observers, the resource representation as either a CON or a NON message.

The resource representation sent by the observed resource will also contain the OBSERVE option, but this time around, it will have an integer value (24-bit). This value is used for sequencing the received values by the observers. The observable resource must set this 24-bit value in a strictly increasing fashion.

If an observer is no longer willing to observe a request, it should send a RST message as a response. If the observed resource is sending NON message and is ignoring the RST responses, the observer may have to wait for a CON message from the observed resource to hope that its RST response will result in de-registration. It should be noted that the observer should take care of token values during registration request. A server should treat a combinationof endpoint and the token as a unique observer. Therefore, one observer, using different values for the token will get registered multiple times.

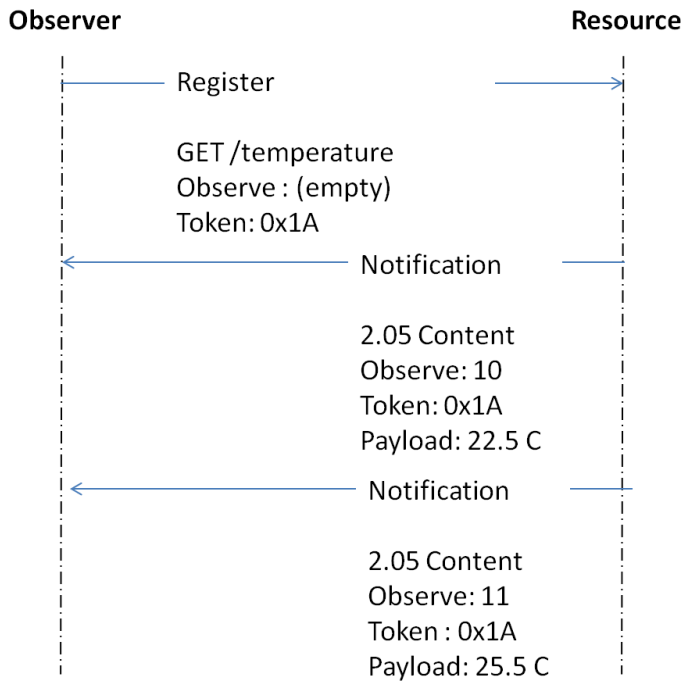

Figure 11-2 Observing a Resource

It should be noted that the server (observable resource) is the authority to decide when the resource state has changed and whether the state change needs to be communicated back to the observers or not. The entire logic is left to implementers. Additionally, the resource representation sent back to the observers may contain freshness information via the addition of the option MAX_AGE.

Sequencing the Observed Values

As stated before, each observed notification contains a 24-bit integer that will help the recipients sequence the observed values. The goal would be to keep the observed values as synchronized as possible with the observed resource. Since the requirement on the observed resource side is to set the 24-bit value of the OBSERVE option in strictly increasing fashion, theobservers can employ a simple logic to find out if the currently received notification contains the latest representation or not. The logic outlined below is take from the observe specification [8] .

Assume V1 is the value that you last received and V2 is the value you received just now or the newly received value. Also assume that T1 is the time you received the value V1 and T2 is the time you received the next value V2. Now you need to decide if V2 is newer than V1. The following condition is being described in the specification:

(V1 < V2 And V2 – V1 < 2 ^ 23) Or(V1 > V2 And V1 – V2 > 2 ^ 23)

Or

(T2 > T1 + 128 seconds)

What the above condition specifies is that either V2 is a larger sequence number than V1 or, V2is smaller than V2 (due to rollover). Alternatively, if a long time has elapsed between when V1 was received and V2 is received, we simply ignore the values V1 and V2 and accept the new value. The value of 128 seconds was arbitrarily chosen to be a number sufficiently larger than the MAX_LATENCY value.

Implementing the Observable Server

After having gone through the theory, it's time to create a CoAP server that supports observingof resources. In order to support observation, a server needs to address the following:

1. Register all listeners
2. Send the observed value to all the registered listeners
3. Un-register an observer

To implement the first requirement, we can simply choose to hold the list of listeners in an array. We need to remember that a good implementation should have three independent threads (if possible), one to accept incoming registration requests, another to perform the actual processing and third to send the notification messages back to the listeners.

What is the exact implementation can vary by context and the capabilities of the device. The sample server provides a very simplistic implementation. The objective is to show how to implement various requirements without being too elaborate.

```
public static void Main()
{
    Program p = new Program();
    p.StartServer();
    int lastMeasuredTemp = 0;
    /*
        * Read the temperature every 10 seconds and notify all listeners
        * if there is a change
        */
    while (true)
    {
        int temp = p.GetRoomTemperature();
        if (temp != lastMeasuredTemp)
        {
            p.NotifyListeners(temp);
            lastMeasuredTemp = temp;
        }
        Thread.Sleep(30000);
    }
}
```

Listing 11-1 The various threads in an Observable Server

If you carefully look at the code above, there are two threads; the main thread and the thread that runs the server. The server starts to listen when the method “StartServer” is called. The infinite loop continues to measure the temperature. Whenever there is a change in the temperature, the listeners are notified.

As stated before, we first need a mechanism to register the listeners. A listener registers itselffor notifications by sending a message with the OBSERVE option in the request message. Ideally, the observer option should not have any value in the request. Also, the server should only look at the OBSERVE option in the incoming request and ignore the value.

The server should expose a URL (this should be in the documentation) that clients should call with the OBSERVE option in the request to register themselves.

CoAPSharp provides a default list to manage the listeners. The list contains a collection of all listeners for a given observable URL. When the server starts, you must add all the observable URLs to the list (if you plan to use the default implementation from CoAPSharp). The “StartServer” method is same as illustrated in previous chapters with just one more extra lined added. This line adds the URL being observed to the map.

```
public void StartServer()
{
this._server = new CoAPServerChannel();
…..Other initialization code…..
//Add all the resources that this server allows observing
this._server.ObserversList.AddObservableResource(OBSERVED_RESOURCE_URI);
}
```

Listing 11-2 Adding information about observable resource

When you receive a request, after performing all the validations, you need to add the caller to the list of listeners. If you are using CoAPSharp library, this is as simple as calling just one method passing the request. You do not need to specify against which URL the listener has to be registered (because the request already contains the request URL). Once you add the listeners successfully, we recommend that you send a response back in the form of ACK. The current CoAP specification is not very strong in setting up form rules about registration and un-registration. Therefore, to keep things uniform and simple, we recommend that you accept a CON request for registration and you should send back an ACK after successful registration. Please note, during validation, you must check that the incoming request contains the OBSERVE option. The listing below provides a basic implementation of such a registration service.

```
void OnCoAPRequestReceived(CoAPRequest coapReq)
{
    string reqPath = (coapReq.GetPath() != null) ? coapReq.GetPath().ToLower() :
"";
    /*We have skipped error handling in code below to just focus on observe
option*/
    if (coapReq.MessageType.Value == CoAPMessageType.CON && coapReq.Code.Value ==
CoAPMessageCode.GET)
    {
        if (!coapReq.IsObservable() /*Does the request have "Observe" option*/)
        {
            /*Request is not to observe...we do not support no-observable*/
            CoAPResponse resp = new CoAPResponse(CoAPMessageType.ACK,
CoAPMessageCode.NOT_IMPLEMENTED,
coapReq /*Copy all necessary values from request in the response*/);
            this._server.Send(resp);
        }
        else if
(!this._server.ObserversList.IsResourceBeingObserved(coapReq.GetURL()))
        {/*do we support observation on this path*/
            //Observation is not supported on this path..
            CoAPResponse resp = new CoAPResponse(CoAPMessageType.ACK,
                                                 CoAPMessageCode.NOT_FOUND,
                                                 coapReq);
            this._server.Send(resp);
        }
        else
        {
            //This is a request to observe this resource...register this client
            this._server.ObserversList.AddResourceObserver(coapReq);
            CoAPResponse resp = new CoAPResponse(CoAPMessageType.ACK,
                                                 CoAPMessageCode.EMPTY,
                                                 coapReq);
            this._server.Send(resp); //send it..tell client we registered it's
        }
    }
}
```

Listing 11-3 Method to register observable resource listeners

Please note, the code above runs in a separate thread. The main thread continues to measure the temperature and on change, notifies the listeners. To notify the listener, you need to decide if you want to use a NON or a CON message. Once again, the CoAP specification is not very clear on this. If a listener wants to un-register, it needs to send a RST against the observe notification. If the observe notification is not of type NON, then the sender is not waiting for a response. Therefore, we are kind of tied to the fact that can only use a CON message to send anotification. If the response to the notification is RST, we will remove the listeners from our list ofresource observers.

```
public void NotifyListeners(int temp)
{
ArrayList resObservers =
       this._server.ObserversList.GetResourceObservers(OBSERVED_RESOURCE_URI);
if(resObservers == null || resObservers.Count == 0) return;

//The next observe sequence number
UInt16 obsSeq = (UInt16)Convert.ToInt16(DateTime.Now.ToString("mmss"));//Willget
       accomodated in 24-bits limit and will give good sequence
foreach (CoAPRequest obsReq in resObservers)
{
UInt16 mId = this._server.GetNextMessageID();
CoAPRequest notifyReq = new CoAPRequest(CoAPMessageType.CON ,
       CoAPMessageCode.PUT , mId);
notifyReq.RemoteSender = obsReq.RemoteSender;notifyReq.Token =
       obsReq.Token;
//Add observe option with sequence number notifyReq.AddOption(CoAPHeaderOption.OBSERVE,
AbstractByteUtils.GetBytes(obsSeq));

//The payload will be JSON Hashtable ht =
       new Hashtable();ht.Add("temp",
       temp);
string jsonStr = JSONResult.ToJSON(ht);
       notifyReq.AddPayload(jsonStr);
//Tell recipient about the content-type of the response
       notifyReq.AddOption(CoAPHeaderOption.CONTENT_FORMAT,
AbstractByteUtils.GetBytes(CoAPContentFormatOption.APPLICATION_JSON));
//send it this._server.Send(notifyReq);
}
```

Listing 11-4 How to notify listeners

The code above extracts all the listeners from the internal list for the given observed URL. It then creates a CON message and sends the message to each client. The only important difference is the addition of the OBSERVE option with a value. If you remember, the OBSERVE option is a 24-bit value that carries an incrementing value for each notification. For this example,we are using the minute and second value to provide the incrementing sequence number. You need to devise your own way for your context.

Lastly, we need a mechanism to remove clients from our internal list. Clients can be removed if the response to a observe notification is RST or if the client is dead and no longer responding. The code sample below provides insight into how to remove the client when a RST is received.

```
void OnCoAPResponseReceived(CoAPResponse coapResp)
{
//If we receive a RST, then remve that client from notificationsif
(coapResp.MessageType.Value == CoAPMessageType.RST)
{
this._server.ObserversList.RemoveResourceObserver(coapResp);
}
}
```

Listing 11-5 Un-registering a listener

The simple implementation above can be expanded to suit your requirement. The key message is that when you receive a RST as a response from the listener, it indicates that the listener is no longer interested in observable resource.

Another place to remove the listener is when the server fails to send messages to the listener. Given that only CON messages are re-tried if the sender fails to receive a response, you are left only with a CON message to properly clean the list of dead listeners.

If you are using CoAPSharp, then an exception of type UndeliveredException is raised with the details of the message that failed to be delivered. You can use this exception to remove the listeners.

Implementing the Listener

Implementing the observable resource listener is very similar to sending any other CON message and receiving a CON message. The listener should perform the following:

1. Send one message for registering as a listener with the observable resource
2. Receive the notification and re-arrange the sequence.
3. Send message to indicate its interested in more messages or it wants to un-register.

The first task for a listener is to request for registration. For this, the listener must call an appropriate URL with the right kind of message type and the message code (you should read the documentation from the provider of the observable resource).

```
public void SendRequest()
 {
     string urlToCall = "coap://127.0.0.1:5683/sensors/temp/observe";
     UInt16 mId = this._client.GetNextMessageID();//Using this method to get the
 next message id takes care of pending CON requests
     CoAPRequest tempReq = new CoAPRequest(CoAPMessageType.CON,
 CoAPMessageCode.GET, mId);
     tempReq.SetURL(urlToCall);
     //Important::Add the Observe option
     tempReq.AddOption(CoAPHeaderOption.OBSERVE , null);//Value of observe option
 has no meaning in request
     tempReq.AddTokenValue( DateTime.Now.ToString("HHmmss") );//must be <= 8 bytes
     /*Uncomment the two lines below to use non-default values for timeout and
 retransmission count*/
     /*Dafault value for timeout is 2 secs and retransmission count is 4*/
     //tempReq.Timeout = 10;
     //tempReq.RetransmissionCount = 5;

     this._client.Send(tempReq);
 }
```

Listing 11-6 Making a registration request to receive observe notifications

The only extra addition in this case is inclusion of the OBSERVE option with a null value. Rest all remains more or less the same as other examples.

As a next task, we need to be able to receive the response. If we receive an ACK, we know we have successfully registered.

Once the registration succeeds, from that point onwards, we will receive CON messages (as per the server example in this chapter) which will contain the observe sequence number and the observed value. In this listener sample, we will accept 5 values and then we will send a request for un-registration. The code listing is given below:

```
void OnCoAPRequestReceived(CoAPRequest coapReq)
{
if (coapReq.MessageType.Value == CoAPMessageType.CON)
{
//Extract the temperature..but first, check if the notification is fresh
//The server sends a 4-digit sequence numberint
newObsSeq =
AbstractByteUtils.ToUInt16(coapReq.Options.GetOption(CoAPHeaderOption.OBSERVE).Val
ue);
        if ((lastObsSeq < newObsSeq && ((newObsSeq - lastObsSeq) <
(System.Math.Pow(2,23)))) ||
            (lastObsSeq > newObsSeq && ((lastObsSeq - newObsSeq) >
(System.Math.Pow(2,23)))) ||
DateTime.Now > lastObsRx.AddSeconds(128))
{
```

```
            //The value received from server is new....read the new temperature
            //We got the temperature..it will be in payload in JSON
            string payload =
AbstractByteUtils.ByteToStringUTF8(coapReq.Payload.Value);
            Hashtable keyVal = JSONResult.FromJSON(payload);
            int temp = Convert.ToInt32(keyVal["temp"].ToString());
            //do something with the temperature now
            Debug.Print(coapReq.ToString());
        }
        //update how many notifications received
        this.countOfNotifications++;
        if (this.countOfNotifications > 5)
        {
            //We are no longer interested...send RST to de-register
            CoAPResponse resp = new CoAPResponse(CoAPMessageType.RST,
CoAPMessageCode.EMPTY, coapReq.ID.Value);
            resp.RemoteSender = coapReq.RemoteSender;
            resp.Token = coapReq.Token;//Do not forget this...this is how messages
are correlated
            this._client.Send(resp);
        }
        else
        {
            //we are still interested...send ACK
            CoAPResponse resp = new CoAPResponse(CoAPMessageType.ACK,
CoAPMessageCode.EMPTY, coapReq.ID.Value);
            resp.RemoteSender = coapReq.RemoteSender;
            resp.Token = coapReq.Token;//Do not forget this...this is how messages
are correlated
            this._client.Send(resp);
        }
        lastObsSeq = newObsSeq;
        lastObsRx = DateTime.Now;
    }
}
```

Listing 11-7 Receiving observed notifications

Once you receive a CON message, you need to find out if this message is newer than the previous message you received. The code follows the same logic described before in this chapter. From that point onwards, we are simply checking if we have received 5 notifications or not. If we have not yet received 5 notifications we send an ACK back to the server to indicate we continue to be interested in the messages, else, we send back a RST message telling the server to un-register us.

Summary

This chapter outlined the important concept of observable resources. While the specification does not provide answers to a few questions on this topic, we still have enough information to create a robust observer/listener combination and create great systems.

Chapter 12

Block Transfer Basics

So far I have been talking about exchanging small amounts of data between two or more participating machines. A large number of cases should be addressed by the schemes explained in previous chapters. However, there might be cases where we need to send larger chunks of data from one machine to another. The data is large enough that it mandates some kind of chunking of the data stream before it is transferred. One question that is difficult to answer is what is “large”. CoAP is based on datagram transports such as UDP or DTLS, which limits the maximum size of resource representations that can be transferred without too much fragmentation. Although UDP supports larger payloads through IP fragmentation, it is limited to 64 KB. This value seems too large for a constrained environment. The specification does not define a threshold beyond which, one can clearly state what is large. It largely depends on the context. However, our suggestion is that we should really keep our messages small (to be precise, tiny) and anything larger than 1 KB on IP networks should be treated large (an arbitrary value to that takes into consideration the size of the MTU over IP networks) or over 50 bytes over 6LoWPAN networks.

Therefore, the question arises – how do we safely transmit large data chunks over the constrained network in a reliable manner. This is where the concept of block transfer [9] comes in. The specification states that we can create small chunks of data and then, transfer the chunks in multiple transmissions. The specification also takes care of transferring a sequence number along with the data for appropriate merging of the data at the receiver end. Additionally,the specification also does a good job in alleviating the needs for maintaining the state during the transfer. Each call can be stateless, like a true RESTful call. While it is impossible to completely alleviate the need to maintain state, we can safely say that there is minimal or very simple state management required across multiple calls to transfer large data in smaller chunks.

Key Aspects of Block Transfer

Block transfer is designed to ensure reliable transfer of chunks, without putting too much requirements on the constrained devices. A few key aspects are:

1. A large data set can be transferred in small block over multiple calls
2. Managing conversation state is easy
3. Transfer of each block is acknowledged (CON message)
4. Both the sender and the recipient and mutually agree on the maximum size of the blockfor such a transfer.

Whether a message is participating in a block transfer or not is identified by the presence or absence of one of two options – Block1 and Block2. There are two options that are used duringa block transfer. Please note, both the options can be a part of either the request or the response. Under what circumstances which option to use, depends on the context. I will shortlyvisit the scenarios later in this chapter.

When a block of data is transferred from the sender to the recipient, three key pieces ofinformation are required:

1. The block number (for sequencing) – This is a simple numeric starting with the valuezero.
2. The size of the block in the incoming message – This is 3-bit integer that defines the exponent (explained later)
3. Whether or not this is the last block – This is a single bit value

The three pieces of information are present as a part of the block option value. The size of the option value could be from 0-3 bytes. The value is represented as a 3-byte unsigned integer. The size of the block is represented by a 3-bit exponent. The formula to calculate the size is

SIZE = 2 ^ (SZX + 4); SZX = The 3-bit size exponent.

The allowed values for the size exponent ranges from 0- 6. Therefore, when the size exponentis zero, it indicates a maximum payload size of 16 bytes (2 ^ 4) and when the size exponent is6, it denotes the maximum payload size of 1024 bytes (2 ^ 10). The size exponent simply indicates the maximum bytes in the payload. During each transfer, the size of payload must match the size as denoted by the size exponent. The only exception being, when the last blockis transferred. The last block can be smaller than the size denoted by the size exponent.

The 4th bit when zero, indicates that the block being transferred is the last one, else it indicates that there are more blocks to transfer.

The rest of the bits in the block option value indicate the sequence number. As stated before, the block option value could be 0-3 bytes. When the block option value is zero bytes, it is assumed that the sequence number is zero, the more flag is zero (meaning last block) and the size exponent as zero (indicating 16 bytes of block size).

The figure below indicates the case when the option value is one byte.

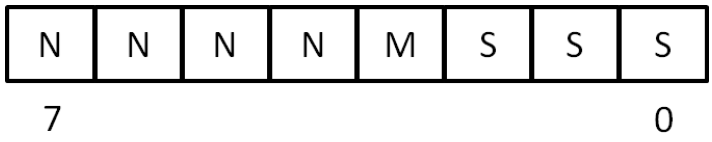

Figure 12-1 One Byte Block Option

The first 3 bits denoted by ‘S’, starting from left, (0-2) hold the size exponent. The 4th bit, denoted by ‘M’ holds the more flag and the rest of the bits hold the sequence number.

The figure below indicates the case when the option value is of two bytes.

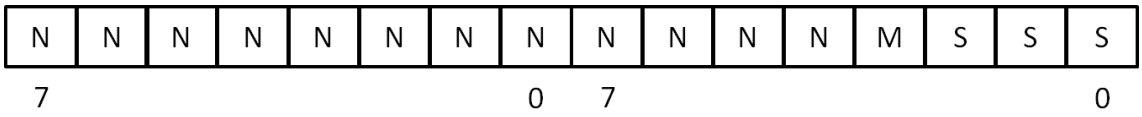

Figure 12-2 Two Byte Block Option

In this case, once again, the first 3 bits (starting from left) contain the size exponent, the 4th bit contains the more flag and the rest of the bits are used to denote the sequence number.

When to Use Block1 and Block2 Options

Both the Block1 and Block2 option value formats are same; however, which option to use under which circumstances requires careful consideration. I believe, this is one aspect in the specification that has confused users more than what it has managed to clarify. To understand this better, we need to first understand the basic use cases. There could be only two cases – either a client machine is sending data to a server machine for storage using block transfer, or a client machine is requesting data from a server machine through a query operation. In each of these cases, the block option is used differently.

Client Requests Server to Store Data Block

Consider the case where a client is sending data in blocks to a server that is required to store the data. This case is called “descriptive usage” in the specification. In this case, the client will use Block1 in the request (either a PUT or a POST). The server will respond back with a Block1 option in the response. In the request, the client provides the information about the block being transferred to the server, whereas, in the response, the server provides information (in the acknowledgement) about the block recently being accepted.

Client Requests Server for Data in Blocks

Another case could be when a client requests the server to provide it the data in block wise fashion. This case is called “control usage”. In this case, the client uses the Block2 option. The value of the Block2 option in such a request contains the sequence number of the block being requested. The M bit must be set to zero and has no meaning. The server will respond back using the Block2 option and the option value will have the information about the block being transferred.

Please note, the specification says that block options are used in one of three roles, however, with such an approach, it has managed to confuse readers more. Therefore, I have chosen a simpler explanation as described above, that considers only two cases.

Block Wise GET Request

Let us examine in detail the flow of messages between the client and the server when the client requests for a block wise data transfer.

At first, the client can simply send a GET request on a resource. The server can respond back with the block and set the Block2 option in the response. This will indicate to the client that the server is server wants a block wise transfer. The client will continue to issue GET requests untilit sees that the M bit (more flag) is not set anymore.

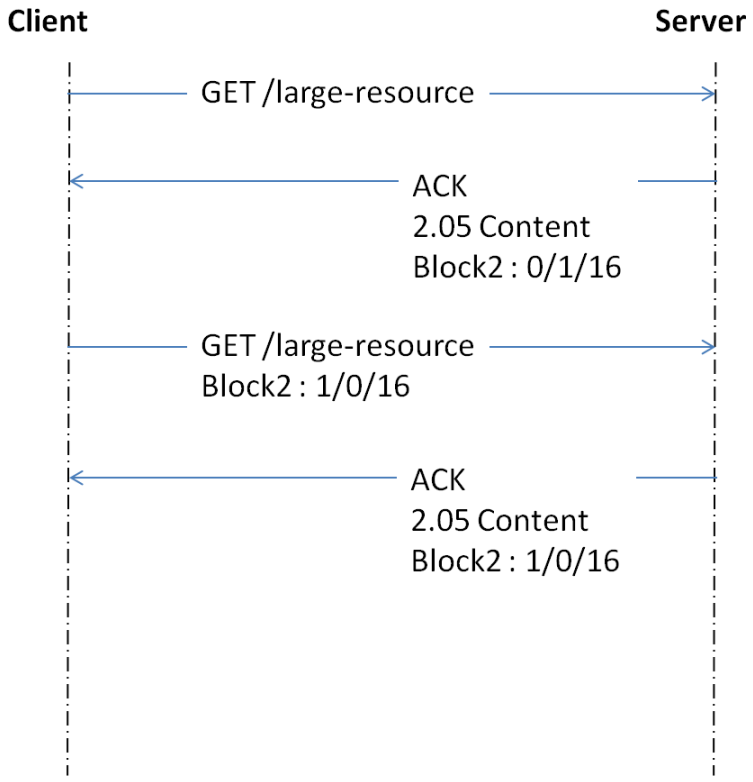

Figure 12-3 Block Wise GET Request

The client first sends a simple GET request on a resource. The server responds back with only 16 bytes of the resource. It additionally sets the Block2 option with value of sequence number as zero, the more bit set and the size exponent as zero (The third value of 16 is the value of the payload size, instead of the size exponent for better understanding. In a real response, this would contain the size exponent value). The client realizes that a block was transferred (since it sees the Block2 option) and requests for more blocks (since the more flag is set). Additionally, it now requests the next sequence number. The server again responds back with the block containing the next set of bytes. However, since it does not have any more bytes to send, the

more flag is now zero. Client sees the more flag and realizes that it has received the last block and the transfer terminates.

Summary

In this chapter, we learnt the basics of transferring large data in smaller blocks. While this chapter was purely theoretical, it is important to understand that block transfer is probably the most complicated exchange defined in CoAP. Thus, I have dedicated a whole chapter to this topic. In the next chapter, you will learn to write client and server implementations that will exchange data using the concept of block transfers.

Chapter 13

Block Transfer PUT Cases

In the previous chapter, I explained the basics of data exchange using block transfer. In this chapter, we will take an example, where the client wants to send data to server using the block transfer. We will build both the client as well as the server to provide you the full example. Please note, we will focus only on the important parts of the code and we will skip other aspects like initializing and starting the client and server. The assumption is, that if you have reached this far, you already know such basics.

Writing the Client

Let us take the example of a client machine that wants to send 35 bytes of data to the server using block transfer. I have purposefully taken the number 35. This number does not divide by 2 and therefore will represent a generic case where the last block will be smaller than what the size exponent denotes.

We will transfer the data in block size of 16 bytes per block. This is the smallest size we can transfer. With this, you can now clearly see that we need three calls. The first two calls will transfer 32 bytes whereas the last call will transfer the remaining 3 bytes.

We will write a simple client that will transfer the bytes in block size of 16 bytes each. Therefore, the size exponent will be zero (Remember, size exponent = 2 ^ (SZX + 4)). The client (or the sender) must maintain the sequence number of the blocks being transferred. As a first step, we will define a few class level variables as shown in the listing below.

```
/// <summary>
 /// What is our block size in terms of bytes
 /// </summary>
 private const int BLOCK_SIZE_BYTES = 16;
 /// <summary>
 /// We want to transfer 35 bytes, in 16 byte blocks..3 transfers
 /// </summary>
 private byte[] _dataToTransfer = new byte[35];
 /// <summary>
 /// Holds the sequence number of the block
 /// </summary>
 private UInt32 _blockSeqNo = 0;
 /// <summary>
 /// Holds the client channel instance
 /// </summary>
 private CoAPClientChannel _client = null;
 /// <summary>
 /// Holds how many bytes got transferred to server
 /// </summary>
 private int _totalBytesTransferred = 0;
```

Listing 13-1 Setting up the required variables

In this chapter, we will focus only on the key code required to exchange data using block transfer. Basic aspects of setting up a client and wiring up the event handlers is already explained in previous chapters.

We are also going to define a method to calculate if the client has more data to send. The listing below outlines the code.

```
/// <summary>
/// Check if we have data to send
/// </summary>
/// <returns>bool</returns>
      public bool
      HasDataToSend()
{
//In this example, we are only transferring on large data set... return
      (this._totalBytesTransferred < this._dataToTransfer.Length);
}
```

Listing 13-2 Determine if we have more data to send

Once we have the basic setup, it's time to write code to send the data in blocks using block transfer. The generic logic is that we will calculate the start and end index of the byte array to determine what block to send.

The key aspect from the perspective of the protocol is to specify BLOCK1 option in the request. The BLOCK1 option in the request will contain the block sequence number, an indicator to specify whether there are more blocks after the current block and the size exponent.

The listing below performs the logic outlined before.

```
/// <summary>
/// Transfer data to server in blocks
/// </summary>
public void TransferToServerInBlocks()
{
CoAPRequest blockPUTReq = new CoAPRequest(CoAPMessageType.CON,
       CoAPMessageCode.PUT, this._client.GetNextMessageID());
blockPUTReq.SetURL("coap://127.0.0.1:5683/largedata/blockput");
       blockPUTReq.AddTokenValue(DateTime.Now.ToString("HHmm"));
//Get needed bytes from source
int copyBeginIdx = (int)(this._blockSeqNo * BLOCK_SIZE_BYTES);int
       bytesToCopy = ((copyBeginIdx + BLOCK_SIZE_BYTES) <
this._dataToTransfer.Length) ? BLOCK_SIZE_BYTES : (this._dataToTransfer.Length -
       copyBeginIdx);
byte[] blockToSend = new byte[bytesToCopy];
Array.Copy(this._dataToTransfer, copyBeginIdx, blockToSend, 0, bytesToCopy);
//Calculate how many more bytes left to transfer
bool hasMore = (this._totalBytesTransferred + bytesToCopy <
       this._dataToTransfer.Length);
//Add the bytes to the payload
blockPUTReq.Payload = new CoAPPayload(blockToSend);
//Now add block option to the request
       blockPUTReq.SetBlockOption(CoAPHeaderOption.BLOCK1,new
CoAPBlockOption(this._blockSeqNo, hasMore, CoAPBlockOption.BLOCK_SIZE_16));
//send this._client.Send(blockPUTReq);
//Updated bytes transferred this._totalBytesTransferred +=
       bytesToCopy;
}
```

Listing 13-3 Transferring data in blocks

We first create the request object. After the request object is created, we copy a subset of the entire byte block from the source into another array. This subset of bytes is added to the payload. Lastly, the block option is added to the request and sent to the server. As a housekeeping measure, we are also updating the number of bytes transferred.

Once this request is successfully received by the server, it would copy the bytes into it's own internal buffer and respond back with an ACK. The response from server will also contain a BLOCK1 option. Important thing to note here is that the block option in the response from server must not have its "more" flag set. If the more flag is set, it indicates server is still processing the data it received and we should wait for that processing to complete. Once we receive a response from the server with the "more" flag not set, we can send the next block. The code listing below outlines the process.

```
void OnCoAPResponseReceived(CoAPResponse coapResp)
{
    if (coapResp.MessageType.Value == CoAPMessageType.ACK)
    {
        CoAPBlockOption returnedBlockOption =
coapResp.GetBlockOption(CoAPHeaderOption.BLOCK1);
        if (returnedBlockOption != null && !returnedBlockOption.HasMoreBlocks)
        {
            //send the next block
            this._blockSeqNo++;
            if(this.HasDataToSend()) this.TransferToServerInBlocks();
        }
    }
}
```

Listing 13-4 Handling block response from server

Once the server indicates that it has successfully received the block sent and is ready for the next block, we send the next block.

This simple example should provide a glimpse of how you make a block-wise transfer using CoAP. Next, let us focus on the server side that is receiving the blocks being sent by the client.

Writing the Server

Writing the server for block transfer case is much easier that writing the client. Once again, I will only focus on the code that accepts the block from the sender. Other aspects like setting the server and initializing it is already covered in previous chapters. On the server side, receive a block required adhering to a few rules as outlined below:

1. If the block sequence number that is received currently, was previously received, we should update the internal buffer, else we should add to the buffer at appropriate place.
2. We might receive the block sequences out of order. For this, the server must ensure it is correctly ordering the received blocks.
3. Once the server receives the block, and it requires time to process it, it can send the response back with block option set and the more flag set it in. This will indicate to the client that the server is still processing the received block.
4. Once the processing of the received block is complete, the server should send the block option in the response with the more flag not set.

With these simple rules, we can construct a request receiver method. The listing below outlinesa simple block processing request receive code.

```
void OnCoAPRequestReceived(CoAPRequest coapReq)
{
string path = coapReq.GetPath();
/*Error checking not done to simplify example*/
if (coapReq.MessageType.Value == CoAPMessageType.CON &&
       coapReq.Code.Value == CoAPMessageCode.PUT &&
path == "largedata/blockput")
{
if (this._rxBytes == null) this._rxBytes = new Hashtable();
       CoAPBlockOption rxBlockOption =
coapReq.GetBlockOption(CoAPHeaderOption.BLOCK1);if
       (rxBlockOption != null)
{
byte[] rxBytes = coapReq.Payload.Value;
if (this._rxBytes.Contains(rxBlockOption.SequenceNumber))
       this._rxBytes[rxBlockOption.SequenceNumber] = rxBytes;//Update
else
this._rxBytes.Add(rxBlockOption.SequenceNumber, rxBytes);//Add
//Now send an ACK
CoAPBlockOption ackBlockOption = new
       CoAPBlockOption(rxBlockOption.SequenceNumber
       ,
                                                    false /*incidate to client that we
have guzzled all the bytes*/,
                                                    rxBlockOption.SizeExponent);
CoAPResponse resp = new CoAPResponse(CoAPMessageType.ACK,CoAPMessageCode.CONTENT,
       coapReq.ID.Value);
resp.Token = coapReq.Token; resp.RemoteSender =
       coapReq.RemoteSender;
resp.SetBlockOption(CoAPHeaderOption.BLOCK1, ackBlockOption);this._server.Send(resp);
}
}
```

Listing 13-5 Receiving Block Transfer Requests

The above example provides a simplistic case to outline how to receive and respond back. The example does not include ensuring order of the received sequence as that is not important to the concept we are trying to learn.

The server receives the bytes, updates its internal buffer and responds back with an ACK message. The ACK message contains a BLOCK1 option with more flag not set, to indicate that the block was received successfully by the server.

Summary

In this chapter we learnt how to exchange large data streams, using the concept of block transfer between a sender and a recipient. While I encourage keeping payload size small, theremight be cases where such a block transfer may be useful. Therefore, I suggest that block transfer be chosen as a last resort.

Chapter 14

Security Support

In all the previous chapters we learnt about various concepts of the Constrained Application Protocol and we also learnt how to develop CoAP based systems using the CoAPSharp library. So far, we have purposefully not delved into the complex topic of security.

Given the draft (draft 18), I believe there is much work to be done in CoAP specificationto clearly define security aspects for practical implementation. As of now, CoAP specification isadvocating using DTLS to implement security. The DTLS specification is also an in-progress work and thus, I believe it would take some more time for all aspects of the system to be in place.

We all know that the default port for HTTP is 80 and for HTTPS is 443. Similarly, the default port for CoAP is 5683 but the specification is yet to define a default port for CoAPS. Given the current state of the specification and the fact that must is required to be done in this area, this chapter is purely theoretical in nature. I will update this chapter as and when the specifications and associated implementations start to take shape for practical usage.

As per the current specification, there are four modes:

1. NoSec – This essentially means there is no protocol level security and DTLS is disabled (or not available). Alternative means to provide security should be used.
2. PreSharedKey – In this mode, it is assumed that DTLS is enabled and a set of keys are available. Each key can be used for a set of nodes. In the extreme case, it may happen that one key is used on one node only.
3. RawPublickKey - DTLS is enabled and the device has an asymmetric key pair without a certificate (a raw public key) that is validated using an out-of-band mechanism (described later). The device also has an identity calculated from the public key and a listof identities of the nodes it can communicate with.
4. Certificate – In this case, it is assumed that DTLS is enabled and the device has an asymmetric key pair with X.509 certificate which has been verified by a trusted authority.

CoAPSharp security library

To authenticate CoAP nodes and to secure data during transmission, we have created a companion library to CoAPSharp. It implements our patented security mechanisms to help in data security and authentication. The library is available for both .NET and nanoFramework platforms. Please write to us for a commercial license.

Summary

Security in CoAP is still an open subject and much work is required to be done. The specification is a good starting point and work is underway to address implementation issues around security aspects. Once I get better answers to security questions, I will update this chapter with details that is relevant for programmers.

Chapter 15

Applications of CoAP

I must admit that it is a little too early to write a chapter on applications of CoAP, given that the technology is still evolving and is in early stages. However, I believe that unless engineering knowledge is used in practical field, it has no worth. Therefore, this chapter is an attempt to imagine where and how we can use CoAP in real-life scenarios.

Basic CoAP Based System Setup

Before we start to look at CoAP based applications, let's understand how a basic CoAP based system is setup. At the lowest level, you will have a set of sensors and actuators that are either measuring some value or are performing some work. These "things" will be interconnected to each other in a small network (some people refer this as mesh). The collection of these machines (or "things") will eventually need to interact with external world. Therefore, one or more of these machines will also connect with a distance machine over CoAP. Alternatively, youmight setup a proxy which finally sends data to an HTTP server. The figure below, illustrates thesetup.

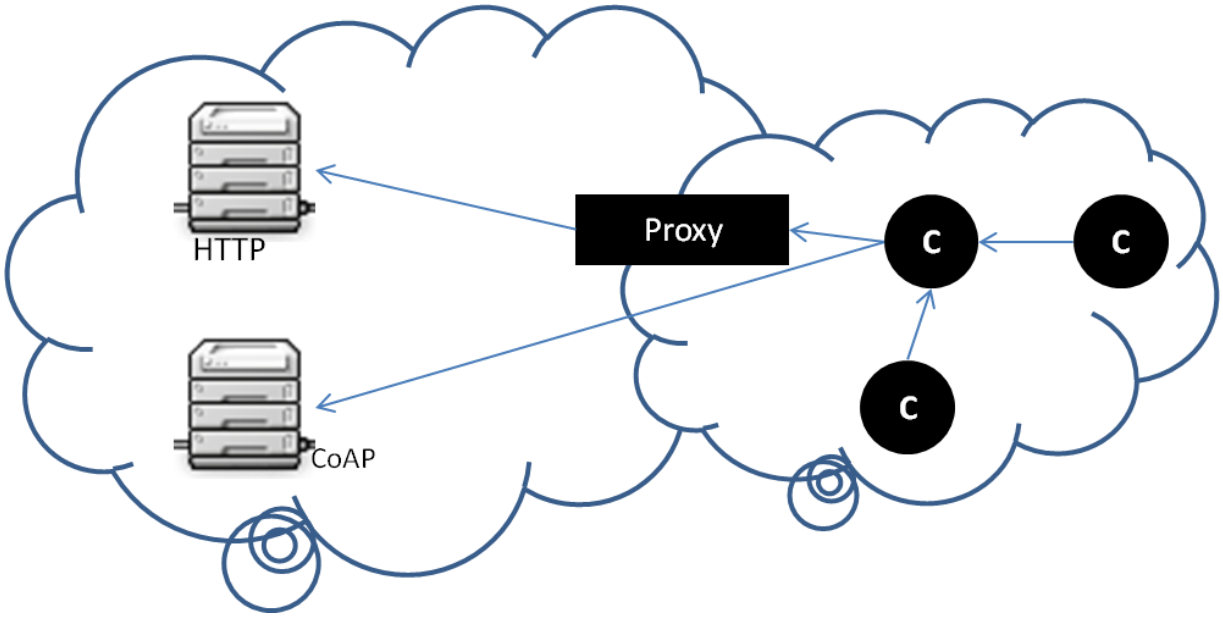

Figure 15-1 Basic CoAP System Setup

The circles marked with "C" indicate CoAP based machines.

Having setup a basic system, let's look at some applications (please note, these are purely fictional in nature for now)

Real Time Condition Based Monitoring in SmartGrid

A SmartGrid is an intelligent power generation, distribution and monitoring system. It uses the modern information and communication technologies to gather and act on data. The world is slowly putting up systems and processes in place to build a global SmartGrid. Imagine, one day, everyone in this would be able to pre-pay for their electricity usage. There will never be any power cuts because authorities would be made aware of breaks before they occur.

Utility companies are moving towards the paradigm of preventive maintenance. They would like to know about health of distribution transformers before a break happens. This is sometimes referred to as Rt-CBM (Real Time Condition Based Monitoring).

As of today, multiple sensors are being placed on the transformer and are collected in a unit. The unit, then in turn, sends the data to the data centers via different means (PLC, GPRS, Ethernet etc.). The simple message exchange itself has so many different protocols that you would get lost in the jargon (MODBUS, DNP3 etc.). To simplify, here is my imagination on how CoAP can be leveraged.

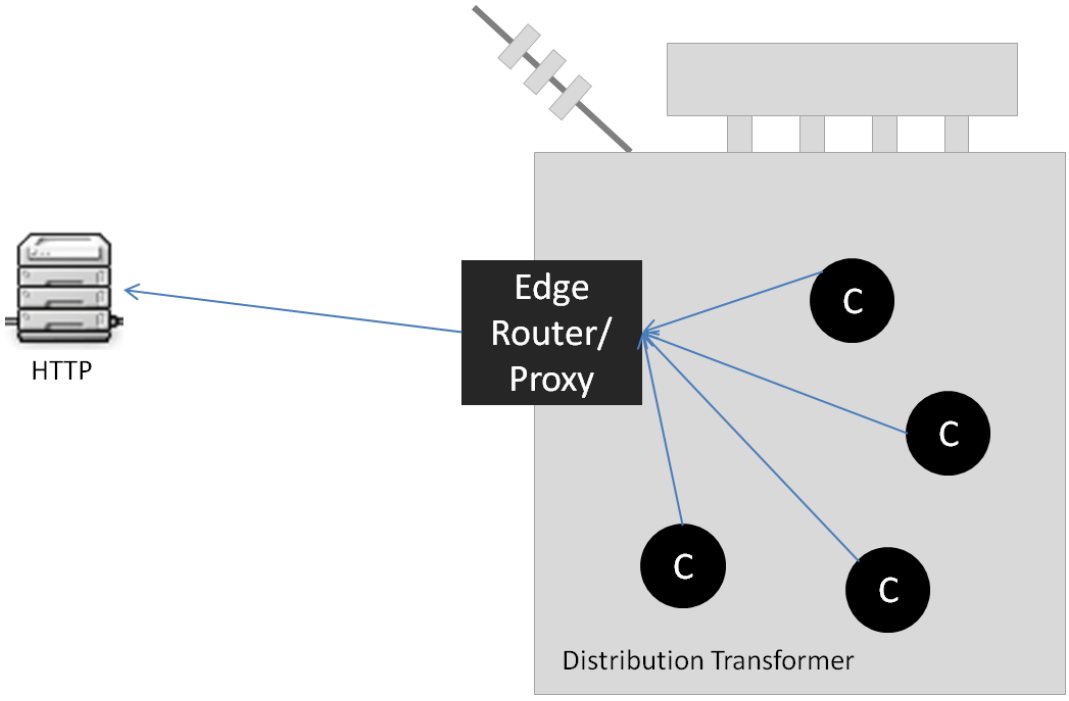

Figure 15-2 Transformer Monitoring using CoAP

All the circles marked with a "C" represent the various CoAP based sensors (temperature, dissolved gas, voltage etc.). They send the data over 6LowPAN via CoAP to the edge router

and proxy combination. The router then sends the data to the data center over HTTP. This will standardize the way manufacturers create the sensors.

Building Automation

This is an old subject, but with advances in technology, solutions to the same problems become less and less complex. Inside large building complexes, I can imagine that there are far too many sensors and actuators that are potential candidate for replacement that use the single standard CoAP.

Defense Equipment

Imagine a battle tank today. It has thousands of sensors each connected via wires (as of today).If we simply replace these tiny sensors with another tiny set of sensors but they all work on 6LowPAN, we would have achieved a great deal.

Aircraft Equipment

This is same as the battle tank analogy. As of today, a commercial aircraft contains miles and miles of wiring just connecting a large number of sensors and actuators with each other. If we can build these smart CoAP based sensors and actuators that work over 6LoWPAN, I'm sure we will save a lot of weight in the aircraft, making it more efficient.

Factory Instrumentation

A large number of manufacturing factories deploy various kinds of instruments for measuring different parameters that are then centrally managed and controlled. Using CoAP, we can miniaturize and simplify the local panels as well as standardize the communication and data exchange patterns.

Summary

The application examples in this chapter are largely theoretical and a fig of imagination. The idea is to encourage you to think what is possible and where can CoAP be beneficial. In the coming year, I'm sure we will see far greater adoption and usage of CoAP.

References

[1] CoAPSharp – https://github.com/CoAPSharp

[2] That 'Internet of Things' Thing - http://www.rfidjournal.com/articles/view?4986

[3] The cellular M2M - http://en.wikipedia.org/wiki/Machine_to_machine

[4] The IETF CoRE WG Charter - http://datatracker.ietf.org/wg/core/charter/

[5] CoAP Implementations http://en.wikipedia.org/wiki/Constrained_Application_Protocol

[6] CoAP Specification Draft (18) – http://tools.ietf.org/html/draft-ietf-core-coap

[7] CoRE Link Format - http://tools.ietf.org/html/rfc6690

[8] CoAP Observe Specification (11) - http://tools.ietf.org/html/draft-ietf-core-observe-11

[9] CoAP Block Transfer (12) - http://tools.ietf.org/html/draft-ietf-core-block-12