

Internet of Things Technologies

Table of Contents

Introduction.....	2
IoT Hardware devices.....	3
IoT Standards and Frameworks.....	4
The MQTT Standard.....	5
Introduction, Main Concepts.....	5
Terminology.....	5
How it works.....	6
Short Demo.....	8
Setting up the workspace.....	8
Building a publisher.....	9
Subscribing to a topic.....	11
Starting the Android sample.....	12
CoAP.....	13
Introduction.....	13
Features.....	13
Terminology.....	14
Architecture.....	14
Requests, Responses.....	15
Short demo.....	16
Setting up the workspace.....	16
Making the server.....	17
Creating a client.....	17
Adding request conditions.....	18
Conclusions and final thoughts.....	20
References.....	21

Introduction

The Internet of Things (IoT) is the interconnection of uniquely identifiable embedded computing devices within the existing Internet infrastructure. Typically, IoT is expected to offer advanced connectivity of devices, systems, and services that goes beyond machine-to-machine communications (M2M) and covers a variety of protocols, domains, and applications. The interconnection of these embedded devices (including smart objects), is expected to usher in automation in nearly all fields, while also enabling advanced applications like a Smart Grid.

Things, in the IoT, can refer to a wide variety of devices such as heart monitoring implants, biochip transponders on farm animals, automobiles with built-in sensors, or field operation devices that assist fire-fighters in search and rescue. Current market examples include smart thermostat systems and washer/dryers that utilize wifi for remote monitoring.

The embedded computing nature of many IoT devices means that low-cost computing platforms are likely to be used. In fact, to minimize the impact of such devices on the environment and energy consumption, low-power radios are likely to be used for connection to the Internet. Such low-power radios do not use WiFi, or well established Cellular Network technologies, and remain an actively developing research area. However, the IoT will not be composed only of embedded devices, since higher order computing devices will be needed to perform heavier duty tasks (routing, switching, data processing, etc.). Companies such as FreeWave Technologies have developed and manufactured low power wireless data radios (both embedded and standalone) for over 20 years to enable Machine-to-Machine applications for the industrial internet of things.

Besides the plethora of new application areas for Internet connected automation to expand into, IoT is also expected to generate large amounts of data from diverse locations that is aggregated and very high-velocity, thereby increasing the need to better index, store and process such data.

IoT Hardware devices

A lot of small, portable and relatively cheap hardware devices have recently emerged, that allow you to create your own small Internet of Things solutions.

Just to give you an idea of how easy it is to get started and build a simple IoT application, here is a tutorial of how to create your own [smart greenhouse](#).

This being said, the market offers a couple of nice hardware devices to get you started:

- **Arduino** is a tool for making computers that can sense and control more of the physical world than your desktop computer. It's an open-source physical computing platform based on a simple microcontroller board, and a development environment for writing software for the board. Arduino can be used to develop interactive objects, taking inputs from a variety of switches or sensors, and controlling a variety of lights, motors, and other physical outputs.
- **Mbed OS** is an operating system for IoT devices and is especially well-suited to run in energy constrained environments. TheOS includes the connectivity, security and device management functionalities required in every IoT device. This OS is designed for ARM® Cortex®-M-based MCUs
- **BeagleBone** is the low-cost, high-expansion focused BeagleBoard using a low cost Sitara AM335x Cortex A8 ARM processor from Texas Instruments. It is similar to the earlier BeagleBoards and can act as a USB or Ethernet connected expansion companion for your current BeagleBoard and BeagleBoard-xM or work stand-alone. The BeagleBone is small even by BeagleBoard standards and with the high-performance ARM capabilities you expect from a BeagleBoard, the BeagleBone brings full-featured Linux to places it has never gone before.
- **Raspberry Pi** is a low cost, **credit-card sized computer** that plugs into a computer monitor or TV, and uses a standard keyboard and mouse. It is a capable little device that enables people of all ages to explore computing, and to learn how to program in languages like Scratch and Python. It's capable of doing everything you'd expect a desktop computer to do, from browsing the internet and playing high-definition video, to making spreadsheets, word-processing, and playing games. What's more, the Raspberry Pi has the ability to interact with the outside world, and has been used in a wide array of digital maker projects, from music machines and parent detectors to weather stations and tweeting birdhouses with infra-red cameras. We want to see the Raspberry Pi being used by kids all over the world to learn to program and understand how computers work.

IoT Standards and Frameworks

As most of the targeted devices have low computational power, we need special standards and protocols to communicate with these devices.

Some of these protocols and standards are:

1. **MQTT** is an OASIS standard that implements a publish-subscribe communication model. It has several QoS levels making it easy to find the perfect tradeoff between reliability and resources/bandwidth usage.
The protocol runs over TCP/IP, or over other network protocols that provide ordered, lossless, bi-directional connections. Its features include:
 1. Use of the publish/subscribe message pattern which provides one-to-many message distribution and decoupling of applications.
 2. A messaging transport that is agnostic to the content of the payload.
 3. Three qualities of service for message delivery:
 1. "At most once", where messages are delivered according to the best efforts of the operating environment. Message loss can occur. This level could be used, for example, with ambient sensor data where it does not matter if an individual reading is lost as the next one will be published soon after.
 2. "At least once", where messages are assured to arrive but duplicates can occur.
 3. "Exactly once", where message are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied.
 4. A small transport overhead and protocol exchanges minimized to reduce network traffic.
 5. A mechanism to notify interested parties when an abnormal disconnection occurs.
2. **OMA Lightweight M2M** is another interesting standard from the Open Mobile Alliance that is getting lot of traction in the domain of Device Management. LwM2M is proposing a standard way to do things like: reboot a device, install a new software image (yes, similarly to what happens on your smartphone and that is based on an ancestor of LwM2M called OMA-DM), etc.
3. **CoAP** (Constrained Application Protocol), which is an IETF standard targetting very constrained environments in which it's still desirable to have the kind of features you would expect from HTTP. CoAP provides a request/response interaction model between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URIs and Internet media types. CoAP is designed to easily interface with HTTP for integration with the Web while meeting specialized requirements such as multicast support, very low overhead, and simplicity for constrained environments.

The standards presented above are also supported by the Eclipse community. They provide some sandbox servers and libraries for the protocols:

1. Eclipse [Paho project](#) libraries and [Mosquitto](#) server for the MQTT protocol
2. [Eclipse Californium](#) server and implementation for the CoAP protocol
3. Eclipse [Wakaama](#) and the proposed project [Leshan](#) are implementing LwM2M stacks in C and Java that you can use to manage your IoT devices

You can check out this "Getting Started" [guide](#) if you want to read more general things about these protocols, and their libraries.

Also, here is a list of [open source sandbox servers](#) provided by Eclipse. You can use them to test your IoT solutions, without creating and setting up the servers yourself.

Further on, I will present the MQTT and the CoAP protocols and their libraries, and I'll work through some examples to demonstrate their functionalities.

Some notes before going forward:

- I will not always get into a lot of details. I strongly suggest you read the specifications of the frameworks for a better understanding. You can find them in the References section.
- Most of what is presented here is taken from the specifications of the frameworks, or from the articles listed in the References section. Be sure to check them out as things may be explained better there.

The MQTT Standard

Introduction, Main Concepts

Terminology

As mentioned before, the MQTT protocol implements a publish/subscribe communication model, and a client-server system architecture. This means that an MQTT system will be split between the following two main components:

1. **Client:** A program or device that uses MQTT. A Client always establishes the Network Connection to the Server. It can:
 1. Publish Application Messages that other Clients might be interested in.
 2. Subscribe to request Application Messages that it is interested in receiving.
 3. Unsubscribe to remove a request for Application Messages.
 4. Disconnect from the Server.
2. **Server:** A program or device that acts as an intermediary between Clients which publish Application Messages and Clients which have made Subscriptions. A Server
 1. Accepts Network Connections from Clients.

2. Accepts Application Messages published by Clients.
3. Processes Subscribe and Unsubscribe requests from Clients.
4. Forwards Application Messages that match Client Subscriptions.

Also, the terminology corresponding to the publish/subscribe communication model and that specific to the MQTT protocol defines the following terms:

Subscription:

A Subscription comprises a Topic Filter and a maximum QoS. A Subscription is associated with a single Session. A Session can contain more than one Subscription. Each Subscription within a session has a different Topic Filter.

Topic Name:

The label attached to an Application Message which is matched against the Subscriptions known to the Server. The Server sends a copy of the Application Message to each Client that has a matching Subscription.

Topic Filter:

An expression contained in a Subscription, to indicate an interest in one or more topics. A Topic Filter can include wildcard characters.

Session:

A stateful interaction between a Client and a Server. Some Sessions last only as long as the Network Connection, others can span multiple consecutive Network Connections between a Client and a Server.

MQTT Control Packet:

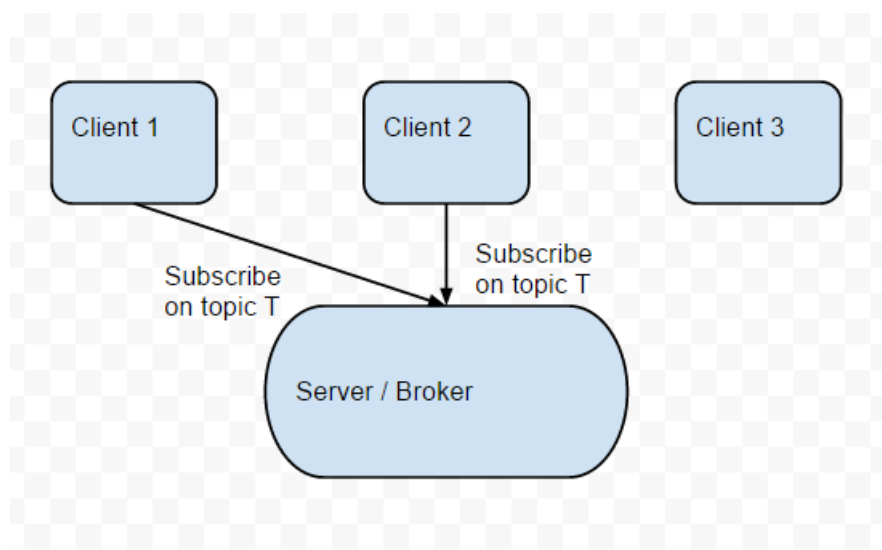
A packet of information that is sent across the Network Connection. The MQTT specification defines fourteen different types of Control Packet, one of which (the PUBLISH packet) is used to convey Application Messages.

All of the above are explained in more details in the specification of the MQTT protocol.

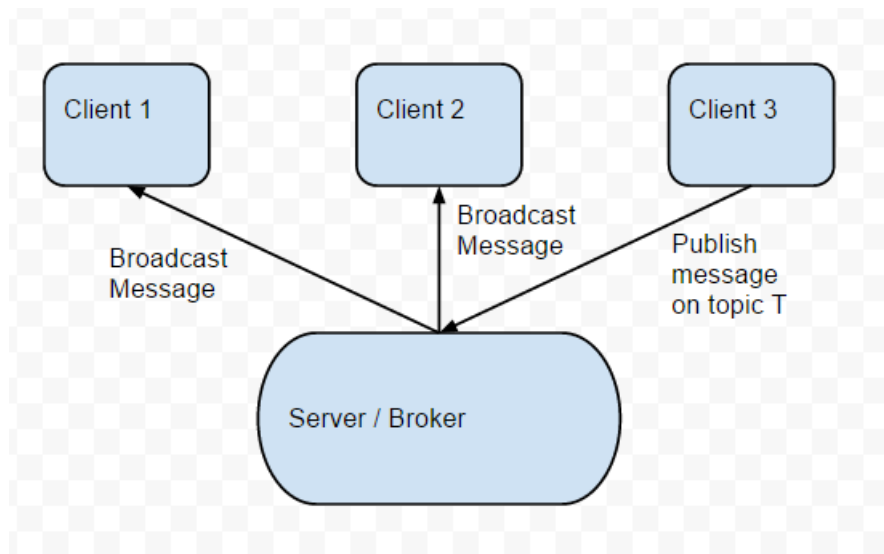
How it works

As mentioned above, MQTT implements a client/server architecture. More specifically it builds upon a broker architectural pattern. Here, the server will act as the broker, and broadcasts messages published to a certain topic to all the subscribers of that topic.

For example, let's say we have three clients that want to communicate on topic T:



First, Client 1 and Client 2 will subscribe to topic T on the server, then the third client will publish a message on that topic. At this point, the server will broadcast the message to all subscribers of topic T, that being Client 1 and Client 2:



The server will match topics in an hierarchical mode. For example, consider the topics *foo/bar/topic* and *foo/bar/foobar/topic*. You can subscribe to a whole group of topics by using wildcards:

- `+` will match a single entry name
e.g. *foo/+/topic* will match the first topic *foo/bar/topic*, but not the second *foo/bar/foobar/topic*
- `#` will match any number of entries
e.g. *foo/#* will match both of the topics. You can use *foo/#* to subscribe to the whole *foo* group.

Note that using wildcards will only work for subscriptions. When publishing, you'll have to use a specific topic.

Short Demo

Let's get down to business.

To show off the MQTT protocol, I will use the Eclipse Paho libraries. These contain some cool classes that make it simple to create a client, connect to a server, and communicate using the MQTT protocol.

You can find a short "Getting Started" tutorial for multiple languages on the project's website. Just go to <https://www.eclipse.org/paho/>, open the 'Downloads' section, and select the language you want.

I'll walk through the Java (not Android) tutorial from the website. And then I'll use the Android sample that comes with the project to publish and receive messages to and from the sandbox server.

Setting up the workspace

First, you'll need to clone the git repository of the Paho project. You can find it on [github](#).

you may skip this part for now, but make sure to use maven for dependency management

After you clone the repository, you should build the projects using maven (you'll need the .jar files for the Android app). To do so, go to the repository root from the command line, and run *mvn clean verify*. If you have any troubles with test not passing, add the *-Dmaven.test.skip=true* parameter to the verify command. After the build is done, you should have the jars in the *target* folders of each of the projects. We are mostly interested in the *org.eclipse.paho.client.mqttv3-1.0.1.jar* file. This is the jar containing the java classes required for the connection.

Now, open up the Eclipse IDE, create a new project (it can be a simple project) and add the required Paho libraries.

If you use maven, you can add the dependency to the Paho libraries in the .pom file:

```
<project ...>
<repositories>
  <repository>
    <id>Eclipse Paho Repo</id>
    <url>%REPOURL%</url>
  </repository>
</repositories>
...
<dependencies>
  <dependency>
    <groupId>org.eclipse.paho</groupId>
    <artifactId>org.eclipse.paho.client.mqttv3</artifactId>
    <version>%VERSION%</version>
  </dependency>
</dependencies>
</project>
```

Otherwise, you'll need to put the .jar file obtained from the maven build (*org.eclipse.paho.client.mqttv3-1.0.1.jar*) on the classpath.

Building a publisher.

The [java tutorial](#) from the Paho project website contains a code sample that publishes messages to a server.

This code is mostly self-explanatory. You have a bunch of settings like the topic, message contents, qos, client id and broker url. Besides these, there is a MemoryPersistence variable. This is used to store outbound and inbound messages while they are in flight, enabling delivery to the QoS specified.

Afterwards, you simply create a client, using the broker url, client id and the persistence variable:

```
MqttClient sampleClient = new MqttClient(broker, clientId, persistence);
```

Next, you set up some connection options (these are not mandatory), create a message, connect to the server and publish that message on the server.

```
MqttConnectOptions connOpts = new MqttConnectOptions();
connOpts.setCleanSession(true);
sampleClient.connect(connOpts);
MqttMessage message = new MqttMessage(content.getBytes());
message.setQos(qos);
sampleClient.publish(topic, message);
sampleClient.disconnect();
```

Also, don't forget to disconnect from the server.

Pretty easy to publish, right? Subscribing gets a little more tricky. :-)

Now, let's put this code in a fancy class, and make some methods!

First, we should declare all constants in a separate class, just to make them easier to work with:

```
package ro.stefanprisca.distsystems.iot.paho.constants;

public class Constants {
    public static final String TOPIC          = "ro.stefanprisca.distsystems.paho.test";
    public static final String CONTENT        = "Test message to be published to the
given topic";
    public static final int QUOS              = 2;
    public static final String BROKER         = "tcp://iot.eclipse.org:1883";
    public static final String CLIENTID      = "StefanPrisca";
}
```

Second, create a MQTTSample class, and put the following code into it:

```
import org.eclipse.paho.client.mqttv3.MqttClient;
import org.eclipse.paho.client.mqttv3.MqttConnectOptions;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;
import org.eclipse.paho.client.mqttv3.MqttSecurityException;
import org.eclipse.paho.client.mqttv3.persist.MemoryPersistence;

import ro.stefanprisca.distsystems.iot.paho.constants.Constants;

public class MQTTSample {
```

```

private MqttClient sampleClient;
public MQTTSample(){
    try {
        sampleClient = new MqttClient(Constants.BROKER, Constants.CLIENTID);
    } catch (MqttException e) {
        e.printStackTrace();
    }
}

public void publish(String topic, String contents) {
    try {
        System.out.println("Publishing message: " + contents);
        MqttMessage message = new MqttMessage(contents.getBytes());
        message.setQos(Constants.QUOS);
        sampleClient.publish(topic, message);
        System.out.println("Message published");
    } catch (MqttException me) {
        System.out.println("reason " + me.getReasonCode());
        System.out.println("msg " + me.getMessage());
        System.out.println("loc " + me.getLocalizedMessage());
        System.out.println("cause " + me.getCause());
        System.out.println("excep " + me);
        me.printStackTrace();
    }
}

public void initConnection() {
    MemoryPersistence persistence = new MemoryPersistence();
    try {
        sampleClient = new MqttClient(Constants.BROKER, Constants.CLIENTID,
            persistence);
        MqttConnectOptions connOpts = new MqttConnectOptions();
        connOpts.setCleanSession(true);
        System.out.println("Connecting to broker: " + Constants.BROKER);
        sampleClient.connect(connOpts);
    } catch (MqttException e) {
        e.printStackTrace();
    }
    System.out.println("Connected");
}

public void disconnect(){
    try {
        sampleClient.disconnect();
    } catch (MqttException e) {
        e.printStackTrace();
    }
    System.out.println("Disconnected");
}
}

```

Until now, we got methods for connecting, publishing a message on a topic, and disconnecting. Note that I used the constants from the above class to initialize the connection. You could also put in some parameters for a better control.

Subscribing to a topic.

Receiving messages from the server works in an asynchronous way. In order to receive a message from a topic you subscribed to, you will need to implement the *MqttCallback* interface. This has a method (*messageArrived*) that will be called when messages come in from the server. There are also methods for connection lost and delivery completed.

To do so, create a class *MyMqttCallback* that implements the *MqttCallback* interface, and add the following code:

```
import org.eclipse.paho.client.mqttv3.IMqttDeliveryToken;
import org.eclipse.paho.client.mqttv3.MqttCallback;
import org.eclipse.paho.client.mqttv3.MqttException;
import org.eclipse.paho.client.mqttv3.MqttMessage;

public class MyMqttCallback implements MqttCallback {
    public static String report;
    public void connectionLost(Throwable arg0) {
        report = "\n\n ----- Connection lost
-----\n"
            + "Connection was lost due to: " + arg0.getStackTrace();
        System.out.println(report);
    }
    public void deliveryComplete(IMqttDeliveryToken devToken) {
        report = "\n\n ----- Delivered Message
-----"
            + "\nMessage was sent by the client: "
            + devToken.getClient().getClientId()
            + "\nMessage was sent to the following topics: "
            + devToken.getTopics();
        try {
            report += "\nMessage content is "
                + devToken.getMessage().getPayload();
        } catch (MqttException e) {
            e.printStackTrace();
        }
        System.out.println(report);
    }
    public void messageArrived(String topic, MqttMessage message)
        throws Exception {
        report = "\n\n ----- Got Message
-----"
            + "\nA message arrived on the topic:" + topic +
            "\nThe message contents are: " + new
String(message.getPayload());
        System.out.println(report);
    }
}
```

Now, set the callback of your client to a new instance of this class:

```
sampleClient.setCallback(new MyMqttCallback());
```

You can do this in the *initConnection* method from the *MQTTSample* class.

You can now create a simple UI to play with these classes a little. If you want, you could start two clients, subscribe one of them to a topic, and publish a message on the same topic with the other. You'll have to modify the methods a bit to be able to set different client ids.

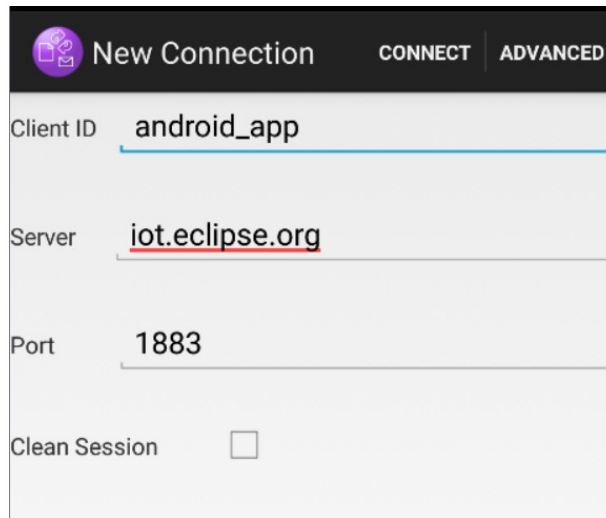
I'll use the android sample app to show how the clients communicate.

Starting the Android sample

To start this sample, open the project from an Eclipse IDE with Android tools installed, and copy the *org.eclipse.paho.android.service-1.0.1.jar* and the *org.eclipse.paho.client.mqttv3-1.0.1.jar* libraries in the *libs* folder of the android app project. Here's a more detailed [tutorial](#).

When you create your client, make sure to use the same broker url as the one defined in the *Constants* class! Afterward, subscribe using this client to the same topic as the one in the *Constants* class.

The settings would look something like this:

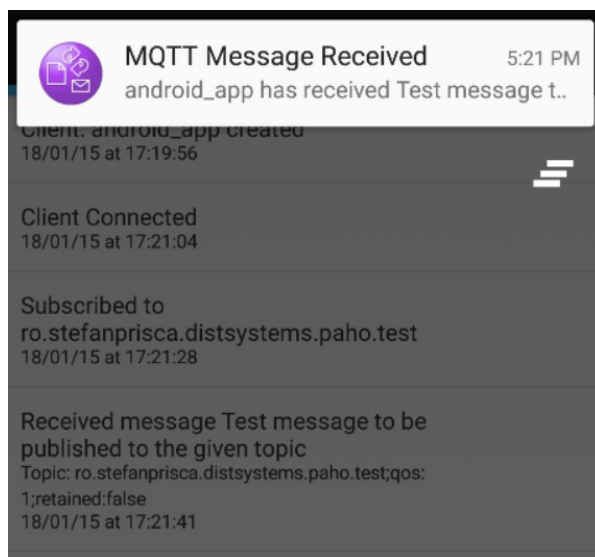


The screenshot shows the 'New Connection' dialog with the following fields:

- Client ID: `android_app`
- Server: `iot.eclipse.org`
- Port: `1883`
- Clean Session: ☐

After this, open up the connection and subscribe to the topic `ro.stefanprisca.distsystems.paho.test`

Now, from the other app, publish a message on this topic. A notification should appear in the android app saying that you received a message, and an entry with this will be added on the history page:



Now do the other way around. Subscribe the java application to the same topic, and publish a message to it from the android app. You should get the response that you set in the *messageArrived* method of the *MqttCallback* class. For example, the following:

```
----- Got Message -----  
A message arrived on the topic:ro.stefanprisca.distsystems.paho.test  
The message contents are: Hello from Android!
```

CoAP

Introduction

As REST-fult application are becomming more and more popular, it is desireble to extend the protocol to more constrained environments as well. The work on Constrained RESTful Environments (CoRE) aims at realizing the REST architecture in a suitable form for the most constrained nodes (e.g., 8-bit microcontrollers with limited RAM and ROM) and networks (e.g., 6LoWPAN). Constrained networks such as 6LoWPAN support the fragmentation of IPv6 packets into small link-layer frames; however, this causes significant reduction in packet delivery probability. One design goal of CoAP has been to keep message overhead small, thus limiting the need for fragmentation.

One of the main goals of CoAP is to design a generic web protocol for the special requirements of this constrained environment, especially considering energy, building automation, and other machine-to-machine (M2M) applications. The goal of CoAP is not to blindly compress HTTP, but rather to realize a subset of REST common with HTTP but optimized for M2M applications. Although CoAP could be used for refashioning simple HTTP interfaces into a more compact protocol, more importantly it also offers features for M2M such as built-in discovery, multicast support, and asynchronous message exchanges.

Features

Considering this, CoAP offers the following features

- Web protocol fulfilling M2M requirements for constrained environments
- UDP binding with optional reliability supporting unicast and multicast requests.
- Asynchronous message exchanges.
- Low header overhead and parsing complexity.
- URI and Content-type support.
- Simple proxy and caching capabilities.
- A stateless HTTP mapping, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP.
- Security binding to Datagram Transport Layer Security (DTLS).

Terminology

CoAP implements a basic client-server architecture, where the two are defined as:

- Client: The originating endpoint of a request; the destination endpoint of a response.
- Server: The destination endpoint of a request; the originating endpoint of a response.

In order to understand better, an endpoint is an entity participating in the CoAP protocol. Colloquially, an endpoint lives on a "Node", although "Host" would be more consistent with Internet standards usage, and is further identified by transport-layer multiplexing information that can include a UDP port number and a security association

There are also other terms like *Sender*, *Recipient*, etc. that I will not present here. You can read more about the Terminology used in the CoAP protocol in the [specification](#).

Architecture

CoAP packets are much smaller than HTTP TCP flows. Bitfields and mappings from strings to integers are used extensively to save space. Packets are simple to generate and can be parsed in place without consuming extra RAM in constrained devices.

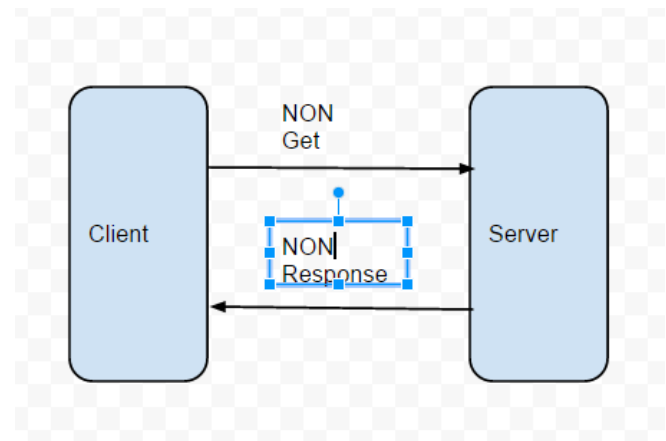
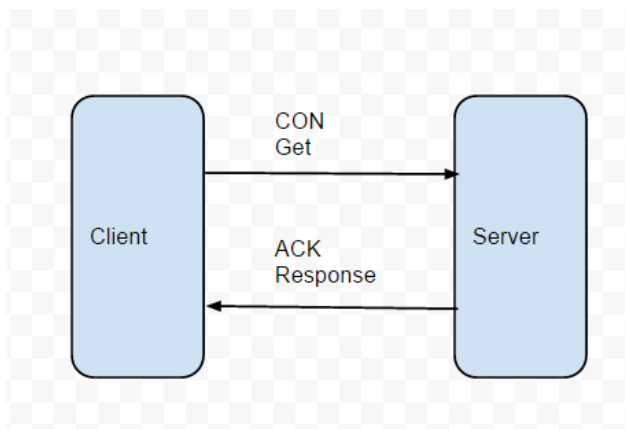
CoAP runs over UDP, not TCP. Clients and servers communicate through connectionless datagrams. Retries and reordering are implemented in the application stack. Removing the need for TCP may allow full IP networking in small microcontrollers. CoAP allows UDP broadcast and multicast to be used for addressing.

CoAP follows a client/server model. Clients make requests to servers, servers send back responses. Clients may GET, PUT, POST and DELETE resources.

The requests to the server are made in two modes:

- Confirmable (CON): The server response will be wrapped in an ACK (Acknowledge) message
 - if the server cannot respond immediately to the request, it will send an ACK message, and then send the response when it is ready.
- Non-confirmable (NON): The server response will be sent back as a Non-Confirmable or Confirmable message

Below are some illustrations of how the requests work:



Another interesting feature of the CoAP protocol is the Observe Capabilities.

This extends the HTTP request model with the ability to observe a resource. When the observe flag is set on a CoAP GET request, the server may continue to reply after the initial document has been transferred. This allows servers to stream state changes to clients as they occur. Either end may cancel the observation.

Requests, Responses

- A **CoAP request** consists of the method to be applied to the resource, the identifier of the resource, a payload and Internet media type (if any), and optional metadata about the request. CoAP supports the basic methods of GET, POST, PUT, and DELETE, which are easily mapped to HTTP. They have the same properties of safe (only retrieval) and idempotent (you can invoke it multiple times with the same effects) as HTTP. The GET method is safe; therefore, it **MUST NOT** take any other action on a resource other than retrieval.
- After receiving and interpreting a request, a server responds with a **CoAP response** that is matched to the request by means of a client-generated token; note that this is different from the Message ID that matches a Confirmable message to its Acknowledgement.

You can read more about requests and responses in the specification, at the [Requests/Responses](#) section.

Also, if you want more information about the GET, POST, PUT, DELETE methods, go to [their](#) section in the specification.

Short demo

I will use the Eclipse Californium libraries for this tutorial. These are really easy to use, and have a lot of helping classes.

You can read more about the Californium project (and about CoAP protocol) on this [google slides presentation](#).

Setting up the workspace

In order to get started, open the Eclipse IDE and create a new project (preferably a maven project). Afterward, add the required CoAP Californium dependency to the project:

```
<repositories>
  <repository>
    <id>repo.eclipse.org</id>
    <name>Californium Repository</name>
    <url>https://repo.eclipse.org/content/repositories/californium/</url>
  </repository>
</repositories>

<dependencies>
  ...
  <dependency>
    <groupId>org.eclipse.californium</groupId>
    <artifactId>californium-core</artifactId>
    <version>1.0.0-SNAPSHOT</version>
  </dependency>
  ...
</dependencies>
```

If you are not using maven for dependency management, you need to clone the [project repository](#) and build the sources to get the .jar files. You will then need to add the *californium-core.jar* file to your class path. You can find this in the target folder of the *californium-core* project. Anyway, see the README from the repository for details.

As you might guess, we need to set-up a server and create a client that will make requests to the server.

Making the server

There are a few key things to know about a Californium CoAP server:

- The classes that we will use are: CoapServer, CoapResource, CoapExchange
- You will need to implement a custom resource for the server by extending the class CoapResource.
 - This resource will implement the GET, POST, PUT, DELETE methods.
 - Note that by default, the implementations will return the METHOD_NOT_ALLOWED code (4.05). You will need to Override the methods, and provide the appropriate response.
- After you create your resource, you need to add it to the server.

So, first you need to create the resource. For now, create a class *MyCoapResource* that extends the *CoapResource* class. You'll need to have a constructor that takes the resource name. This name will be used to access the resource later.

Then, just override the GET/POST/PUT/DELETE methods. For example, here's how a basic server with a GET method would look:

```
public class MyCoapServer
{
    public static final String RESOURCE_NAME = "MyCoapResource";
    public static void main( String[] args )
    {
        CoapServer server = new CoapServer();
        server.add(new MyCoapResource(RESOURCE_NAME));
        server.start();
    }
}

class MyCoapResource extends CoapResource{
    public MyCoapResource(String name) {
        super(name);
    }
    @Override
    public void handleGET(CoapExchange exchange) {
        exchange.respond("The server salutes you!");
    }
}
```

Creating a client

To use a CoapClient, you just need to instantiate it with the server uri, and call the provided get/post/put/delete methods.

The following code will create a client that connects to our server, and makes a get request. It also has a fancy print method to show the response.

```
private static void printResponse(CoapResponse response){
    if(response.getStatusCode() != ResponseCode.NOT_FOUND){
        System.out.println("We got a good response code: " +
response.getStatusCode());
    }
}
```

```

        System.out.println("The contents of the response are: " +
response.getResponseText());
    }else{
        System.out.println("The resource was not found!");
    }
}

CoapClient client = new CoapClient("coap://localhost/"+MyCoapServer.RESOURCE_NAME);
CoapResponse response = client.get();
printResponse(response);

```

Note that your server will start on the localhost. So you don't need to connect to the sandbox server to access your resources.

The uri must respect the following format:

```
"coap:" "://" host [ ":" port ] path-abempty [ "?" query ]
```

The port part is optional.

The path is used to access resources on the server. In the above example, the resource I want to use is `RESOURCE_NAME` (= `"MyCoapResource"`) so I will access it with `"/RESOURCE_NAME"` (or `"MyCoapResource"`).

You can also try to access the eclipse sandbox server, and browse through their resources. The uri for this server is (`"coap://iot.eclipse.org:5683/<resource>"`) . There are some examples on slide 47 of the google slides [presentation](#). They also present the asynchronous access modes. Basically, for this you need to use a CoapHandler.

Adding request conditions

Of course, things can get a little more complicated. You can set-up conditional request options, that allow you to control what the method does. These are:

- **If-Match** The If-Match Option MAY be used to make a request conditional on the current existence or value of an ETag for one or more representations of the target resource. If-Match is generally useful for resource update requests, such as PUT requests, as a means for protecting against accidental overwrites.
- **If-None-Match** The If-None-Match Option MAY be used to make a request conditional on the nonexistence of the target resource. If-None-Match is useful for resource creation requests, such as PUT requests, as a means for protecting against accidental overwrites

For example, let's create the following two conditions, and put them in a list:

```
public static final String MY_FIRST_CONDITION = "Cond1";
public static final String MY_SECOND_CONDITION = "Cond2";
private List<String> ifMatchConditions = new ArrayList<String>();
public MyCoapResource(String name) {
    super(name);
    ifMatchConditions.add(MY_FIRST_CONDITION);
    ifMatchConditions.add(MY_SECOND_CONDITION);
}
```

Now, let's override the PUT method, and add the following code to it:

```
@Override
public void handlePUT(CoapExchange exchange) {
    exchange.accept();
    OptionSet options = exchange.getRequestOptions();
    List<byte[]> inConds = options.getIfMatch();
    boolean pass = false;
    for(byte[] bSet : inConds){
        if(ifMatchConditions.parallelStream().anyMatch((s) -> s.equals(new
            String(bSet))) ){
            pass = true;
            break;
        }
    }
    if(pass){
        exchange.respond(ResponseCode.CREATED, "Everything is ok, do not
            panic!");
    }else{
        exchange.respond(ResponseCode.BAD_OPTION, "Nothing is ok. Time to
            panic!");
    }
}
```

The above code does the following:

1. gets the request options from the exchange parameter
2. sets a *pass* flag
3. checks if any of the conditions are true.
 1. If any of the conditions are true, then respond with code CREATED, and some payload
 2. If none is true, then respond with the BAD_OPTION code, and some payload to induce panic to the client. Just because we are evil!

In order to make a request that uses requests options, the client can use the method *putIfMatch*:

```
response = client.putIfMatch("Hello from the client", APPLICATION_XML,
    MyCoapResource.MY_FIRST_CONDITION.getBytes());
```

Conclusions and final thoughts

MQTT and CoAP are both useful Internet of Things protocols, but they have some differences.

First of, MQTT uses a publish/subscribe communication model. This makes it better to use when clients need to broadcast messages to whole groups. For example, consider you want to cover a bunch of solar panels because a big ice storm is coming. You'll just have to publish the command message (e.g. "COVER") to the topic. Then all registered clients (the windmills) will receive that message and act accordingly.

On the other hand, CoAP is a Machine to Machine protocol. It is more useful if you want to send a message to a specific endpoint. To illustrate this, think that you are in charge of maintaining a field of windmills. One of them breaks, or acts weird, so you want to shut it down without affecting all of the other windmills. Or, even better, it can sense that it is malfunctioning, and it sends a shutdown request to the server, setting some request parameters.

So, in conclusion, it is up to you to decide what works best for your application. If you want to broadcast messages, maybe consider MQTT. If you want M2M communication, CoAP may be better.

Anyhow, you can find more about these protocols, and the Eclipse frameworks that I used, in the articles referenced below.

The sources for the demonstrations can be found on my repository:
<https://github.com/stefanprisca/FacultDistributedSystems/tree/master/App6>

The structure is the following:

- Doc: contains this document in .pdf format
- `iot.paho.desktop`: contains the desktop application for communicating over the MQTT protocol using Eclipse Paho
- `iot.californium`: contains the sources for the CoAP application.

Thank you for reading this. I hope you find it useful!

References

- [5 Things to get started with IoT](#) – Eclipse
- MQTT [Specification](#).
- COAP [Specification](#)
- Arduino [Introduction](#)
- [Mbed OS](#)
- [BeagleBone](#)
- [Raspberry Pi](#)
- [very nice presentation](#) for MQTT and CoAP
- Paho [android sample](#).
- Paho java start-up [guide](#)
- [Hands on with CoAP](#)
- Eclipse [Wakaama](#)
- Project [Leshan](#)
- [open source sandbox servers](#) provided by Eclipse