adafruit learning system

All the Internet of Things - Episode Two

Created by lady ada

Last updated on 2019-02-28 06:47:23 PM UTC

Introduction

Hello!

I'd like to welcome you to the second episode of our new series of videos and guides, designed to help you learn about and make your very own connected objects

This is Adafruit and Digi-Key's **ALL THE INTERNET OF THINGS** - a six-part series, covering everything you need to know about the Internet of Things (which we will shorten to **IoT**).

For our second guide, we'll go over the most popular *protocols* used in the IoT industry, as well as the upsides and downsides of each type of protocol to help you decide what you'll use to connect your devices to the internet and exchange data

PROTOCOLS!

In TRANSPORTS we talked about - "How to Get From Your Device to the Internet" - POWER, DISTANCE, AND BITS. (https://adafru.it/ChW)

Transports are the *physical* and *wireless* means to get the data around - how to get from *here* to *there* and back again. But transports often are not structured - your data shows up in a pile of bits. You may not get your data in order, correctly or even at all!

On top of the transport layer, we can add another level of quality assurance - **PROTOCOL**! A protocol can be thought of as the language each machine, or machines, use to talk to each other. Protocols don't deal with the messy business of moving bits to and from, they are a higher level set of rules the machines use to talk to each other.

Protocols

The Language of Machines

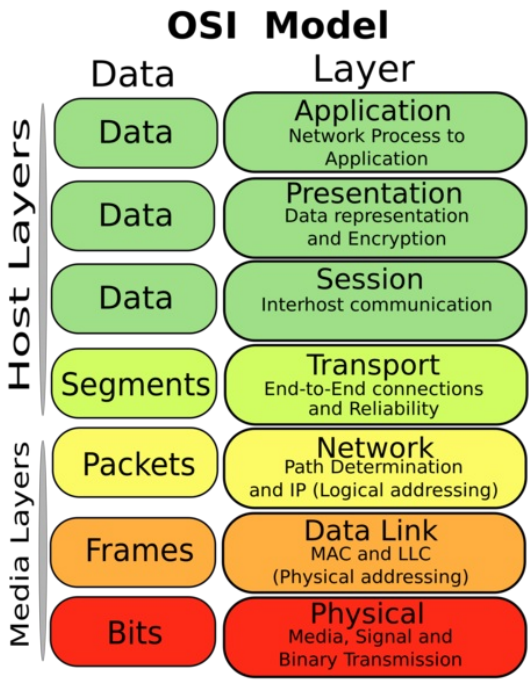
Let's bring this back to something we can easily relate to, something we all know and use. When humans greet each other we say hello, we wave, we shake hands. It would be odd if we greeted each-other by throwing an apple, or whistling a show-tune, or doing a hand-stand. Every language and culture has a way to greet another person and a way to communicate with human language. And we learn at a very young age how to do this. It makes it easier for people to work together and share information.

You can think of protocols as the language machines use when they talk to each other.

With the machines we have built and the Internet of Things, we also have a need to communicate. So, we have a language, and guidelines. There are millions of computers, sensors, actuators, systems on the internet, and for all things to communicate we need a *shared language and rule-set* A shared language allows quick and efficient cooperation between things, using less programming, power, memory and data transmission.

Protocol Layer

You may remember 7-layer OSI model (still not to be confused with the delicious 7-layer burrito (https://adafru.it/Bss)) from the last episode. We mentioned that this design is very idealized, and that you can't really completely separate the layers, so transports is pretty much the bottom four layers:



Again, it's really specific to the transport - almost all have built in error correction and reliability management (Ethernet, WiFi, Cellular...) but some do not (mostly, radios) and so that part of the transport may be something you have to manage.

The next two layers, Session & Presentation plus maybe a bit of the Segments layer, is what we deal with in the Protocols section. That's where you'll actually package data up, encode or encrypt it so that you can present it to the Application layer which is the final product you're making.

Transports vs Protocols

Going back to our comparison of languages for humans as a way to understand the languages for computers, we can also use the OSI layer division when talking about human language:

The **transport** of language can be air - in which case the language will be vocal.

The **transport** of language can be light - in which case the language will be visual - this is what sign language uses.

The **transport** of language can also be paper - in which case the language will be written!

Either way, *the transport varies but the language itself is the same:* English (or Indonesian! or Spanish! etc.) You can always get the same *data* across no matter the transport. In this sense, the protocol (language) is transport-agnostic, just like that OSI layer model above.

Except - its not really....

The way we talk and the way we write does vary. You get intonation when using air, exactness with writing. ASL even uses the space around the speaker to indicate relationships (https://adafru.it/ChX) and the movements to convey emotion, irony and puns. So sometimes the transport and protocol bleed into each other a bit.

Likewise, there are some transports that are really incompatible with some protocols - for example you really don't want to try to run HTTP over Bluetooth Classic, as it would be terribly slow. And SigFox won't work with CoAP due to the packet size restrictions.

We'll try to call out some of these differences, but be aware that that just because you may be able to interchange the layers in theory, it can be very difficult to maintain a good customer experience in practice.

HTTP

Chances are you're familiar with HTTP, but even if you're not, you've seen the letters at the beginning of every website. Like this one - **http:**//www.adafruit.com

HTTP stands for **H**yper **T**ext **T**ransfer **P**rotocol (hey will you look at that, it's even called a protocol!) It's used by devices connected internet to send website data back and forth. Note that you may connect to the internet using Ethernet at work, WiFi at home, or Cellular on the go - but the HTTP part is the same for every website.

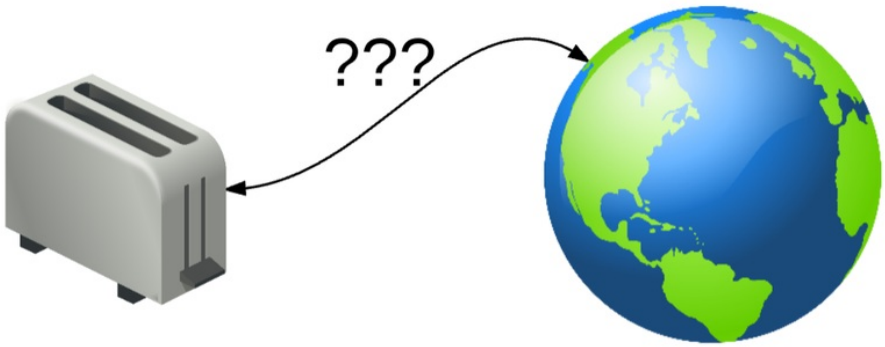
Note that while many people assume that the internet is all HTTP, it isn't! It's just the most popular procotol. There are other protocols you may bump into in your travels, such as the once-popular **FTP** (File Transfer Protocol) (https://adafru.it/ChY) which is used to transfer files around, and **SMTP** (Simple Mail Transfer Protocol) (https://adafru.it/ChZ)which is how your e-mail gets delivered.

Stateless-ness

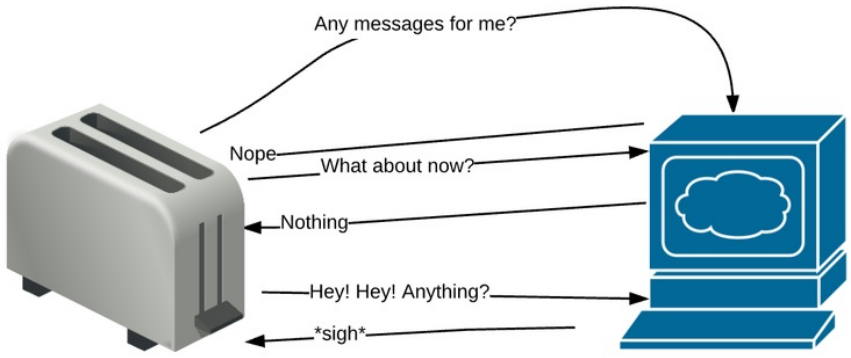
HTTP is stateless, so you have to have a connection per data transfer - one connection every time you want to write data, one connection for reading. (You can use persistent to keep the connection open (https://adafru.it/Ch-) but many simplified microcontroller stacks assume one connection per request, and its considered bad taste to keep it open indefinitely - eventually the server will kick you out!) HTTP is optimized for huge amounts of data such as used for websites, and it can be used for IoT connections. But it's not lightweight: there's a lot of headers and encoding per request. And it's not that fast when it comes to .

Internet of TOAST

Another problem with HTTP is that it's pull only. Let's say you have an IoT device- *Internet of Toaster* - this toaster is connected to the Internet and can send you toast updates, as well as have its settings managed securely online.



With HTTP, your toaster can easy send data to the server whenever it wants (e.g. "Toast is done!") by connecting and submitting a REST PUT request. But if it wants to pull data from the server via REST GET, it has to constantly connect and ask ("Any updates to the Toast darkness level?" "What about now?" "Anything now?") which is really data and time consuming. Pull updates are either slow (check only every few minutes) or data/power intensive (check constantly).



To Summarize!

HTTP is:

- asynchronous protocol, this means the client waits for the server to send the data.
- a one way street, only the client makes the request.
- one to one protocol, one client request at a time.
- data heavy, filled with headers and rules.

Take a REST

REST (**Re**presentational **S**tate **T**ransfer) isn't exactly a protocol, technically that's **HTTP**'s domain only. It lives a layer *above* HTTP. REST is more like a data-interfacing style that works for well IoT. But since it's based so heavily on top of HTTP, the two words are often used interchangeably.

REST is a well established standard for transmitting data, it is very internet friendly. If you're online, just about every web service uses REST both internally and externally. Every modern computer & programming language has REST API (application program interface) support, so it's very fast to get started, and if your device is going to talk to public 'utilities' like IFTTT, Facebook, Twitter, etc it will need to use REST.

Four Magic Words

There are four ways to move data around:

1. GET (request data)
2. PUT (send data)
3. POST (create data)
4. DELETE (natch)

Every web-browser uses these commands to get and put the data use use on the internet. And the data you get/put/post/delete is stored in 'buckets' called resources. The names of these buckets are also known as **URL**s - **U**niversal **R**esource **L**ocation!

Restful URLs

So lets say you want to query the adafruit twitter account. You can go to the twitter.com server using http (**http:**//www.twitter.com (https://adafru.it/Ci0)) and then request the **adafruit** resource (http://www.twitter.com/**adafruit** (https://adafru.it/Ci2)). This will give you a huge amount of data, but you can 'drill down' by adding more /'s and subcategories. So to read the post with our video visiting the NYC Formula E Series race go to https://twitter.com/adafruit/status/900750127425630208 (https://adafru.it/Ci6) - that big long number is a unique identifier for the tweet.

What makes it RESTful is that each / section gets more specific - the adafruit account, then the statuses, and finally the unique status #.

What your web browser gets when it goes to those URLs is a huge amount of HTML (Hyper Text Markup Language). You can see it by using **View Source...**

which looks like this (we pulled out just a section that shows a tweet)

```
<div class="js-tweet-text-container">
<p class="TweetTextSize TweetTextSize--normal js-tweet-text tweet-text" lang="en" data-aria-label-part="0">2017 is almost over, and we're celebrating another great year with a Circuit Playground Express sale! See <a href="https://t.co/lMXdvG9KLo" rel="nofollow noopener" dir="ltr" data-expanded-url="http://adafruit.com/sale" class="twitter-timeline-link" target="_blank" title="http://adafruit.com/sale" ><span class="tco-ellipsis"></span><span class="invisible">http://</span><span class="js-display-url">adafruit.com/sale</span><span class="invisible"></span><span class="tco-ellipsis"><span class="invisible"> </span></span></a> for deets

<a href="/hashtag/adafruit?src=hash" data-query-source="hashtag_click" class="twitter-hashtag pretty-link js-nav" dir="ltr" ><s>#</s><b>adafruit</b></a> <a href="/hashtag/adafruitsale?src=hash" data-query-source="hashtag_click" class="twitter-hashtag pretty-link js-nav" dir="ltr" ><s>#</s><b>adafruitsale</b></a> <a href="/hashtag/circuitplayground?src=hash" data-query-source="hashtag_click" class="twitter-hashtag pretty-link js-nav" dir="ltr" ><s>#</s><b>circuitplayground</b></a></p>
</div>
```

You can see the text of the tweet (https://adafru.it/Ci7) is embedded in there

```
2017 is almost over, and we're celebrating another great year with a Circuit Playground Express sale! See <a href="https://t.co/lMXdvG9KLo" rel="nofollow noopener" dir="ltr" data-expanded-url="http://adafruit.com/sale" class="twitter-timeline-link" target="_blank" title="http://adafruit.com/sale" ><span class="tco-ellipsis"></span><span class="invisible">http://</span><span class="js-display-url">adafruit.com/sale</span>
```

While you could, in theory, parse a webpage's HTML via your device, its a hairy and unpleasant business that breaks very easily (https://adafru.it/Ci8). So, instead, most services online provide a **REST API** which is simplified and machine-friendly. In those cases, the data is usually encoded with JSON or XML, rather than HTML, but still using HTTP.

Here are some examples of REST APIs:

- IFTTT (https://adafru.it/Ci9)
- Twitter (https://adafru.it/Cii)
- GitHub (https://adafru.it/Cij)

Since we are using HTTP, the connection does not have 'state': the client connects to the REST API when needed,

transferring data to/from the resource. When it's done, it closes the connection.

On one hand, this is nice because you have a burst of data transmission, and then a disconnection so the server doesn't need to hold the connection open. But the downside is you have to setup and tear-down the connection each time, and that can take up a ton of data.

XML

XML

XML stands for e**X**tensible **M**arkup **L**anguage. XML is a markup language much like HTML that you saw in the previous page, and looks the same too! XML was designed to be self-descriptive, in other words, when you look at it, it should be clear what is is based on the tags you define, sorta like HTML uses `<image>` when it wants to describe an image, and `<b>` to mark up bold text. But, XML has fallen out of favor due to it's parsing complexity and verboseness.

JSON

JSON

These days, you'll see JSON most often. (**J**ava**S**cript **O**bject **N**otation) is a lightweight data-interchange format. It is easy for humans to read and write, and despite being based on JavaScipt language standard you do not need to use JavaScript - you can use any language. It is easy for machines to parse and generate.

What's the Difference?

XML is a markup language, JSON is a way of representing objects. Generally speaking, JSON is preferred for IoT applications since it can self-describe and is more programmatic, where XML was made for document mark up like HTML.

API Example

JSON API Example

For example, lets look at a public JSON API example. There's not a ton of public APIs, most require authentication, but github has some public data available: https://api.github.com/users/adafruit (https://adafru.it/Cik) here's what you'll get (of course some dates and numbers may vary)


```
{
  "login": "adafruit",
  "id": 181069,
  "avatar_url": "https://avatars3.githubusercontent.com/u/181069?v=4",
  "gravatar_id": "",
  "url": "https://api.github.com/users/adafruit",
  "html_url": "https://github.com/adafruit",
  "followers_url": "https://api.github.com/users/adafruit/followers",
  "following_url": "https://api.github.com/users/adafruit/following{/other_user}",
  "gists_url": "https://api.github.com/users/adafruit/gists{/gist_id}",
  "starred_url": "https://api.github.com/users/adafruit/starred{/owner}{/repo}",
  "subscriptions_url": "https://api.github.com/users/adafruit/subscriptions",
  "organizations_url": "https://api.github.com/users/adafruit/orgs",
  "repos_url": "https://api.github.com/users/adafruit/repos",
  "events_url": "https://api.github.com/users/adafruit/events{/privacy}",
  "received_events_url": "https://api.github.com/users/adafruit/received_events",
  "type": "Organization",
  "site_admin": false,
  "name": "Adafruit Industries",
  "company": null,
  "blog": "www.adafruit.com",
  "location": "New york city",
  "email": null,
  "hireable": null,
  "bio": null,
  "public_repos": 838,
  "public_gists": 1,
  "followers": 0,
  "following": 0,
  "created_at": "2010-01-12T23:57:58Z",
  "updated_at": "2017-08-23T10:11:22Z"
}
```


As you can see, the JSON data is very structured but pretty simple. You can see the location is a string:

```
"location": "New york city",
```

"New York City" but the number of public repositories (public_repos)

```
"public_repos": 838,
```

is a number 838 and other entries are URLs of their own,

```
"avatar_url": "https://avatars3.githubusercontent.com/u/181069?v=4",
```

boolean values,

```
"site_admin": false,
```

etc.

Since JSON is text-based you can really toss whatever data you like in there. If you need to store binary data, you'll need to textify it, using something like Base64 encoding. (Compare this to MQTT where the data is not text based but raw binary and can be parsed however you like)

XML API Example

The URL itself can end up becoming a data transport of its own. For example Yahoo's weather API has URLs like this:

```
https://query.yahooapis.com/v1/public/yql?q=select%20item.condition.text%20from%20weather.forecast%20wher
```

Note the format=json (https://adafru.it/Cim) part of the URL. When you go here you'll get JSON data (we've added line breaks for legibility)

```
{
  "query": {
    "count": 1,
    "created": "2017-09-05T04:27:15Z",
    "lang": "en-US",
    "results": {
      "channel": {
        "item": {
          "condition": {
            "text": "Clear"
          }
        }
      }
    }
  }
}
```

If you rewrite the URL to have format=xml () you can see the XML output:

```
<query yahoo:count="1" yahoo:created="2017-09-05T04:28:53Z" yahoo:lang="en-US">
  <results>
    <channel>
      <item>
      <yweather:condition text="Clear"></yweather:condition>
      </item>
    </channel>
  </results>
</query>
<!-- total: 11 -->
```

You can see the data is the same, but the format is a little different.

RESTfull

If a web service/API supports REST it's called RESTful, this means it sticks to some common architectures so many programming languages can work with it.

The *vast majority* of online APIs are RESTfull

Why Use REST?

The number one reason you'll be using REST is that REST runs over HTTP, so any device that is on the web can use a REST API.

For example, a single board computer with Linux on it, or a cellular module, will already have a TCP/IP or HTTP stack built in. You've got an easy way to hook into a wide variety of services, both for storing data but also to grab data!

Also, many applications for IoT are about the "state" of device, and that's why REST is used so much. Say you have a light-bulb that you are making IoT. The light-bulb has a *state* - whether it's on or off. The light-bulb can transmit it's state state "I'm a light, I am on" with a GET request to http://www.server.com/lightbulb/3292 (for lightbulb # 3292). You can also PUT the state back to that URL to turn the bulb on, or off. In these examples the programming and model fit the model of the device you are internetifying!

Yay! We're Done!

OL, so we're done, this solves everything for IoT, just use REST and that's it? Not so fast! In fact, that's the problem, with REST it's ALL request / response, this means every time you want to do something you need to request it and get the response. That can lead to tons of requests, lots of data usages, battery drain, and slow responses.

Surely there's another way?

As you've seen from the TRANSPORTs episode (https://adafru.it/Cio), some of our transports like cellular or LoRa, you end up paying per byte, per message, or working with a very restrictive bandwidth. In those IoT transports, you'll want a simple & lightweight protocol with push *and* pull.

Let's take a look at a few more options!

MQTT

MQTT (**M**essage **Q**ueue **T**elemetry **T**ransport) has become the most popular and essentially the second standard IoT protocol (besides HTTP+REST).

Note that even though it is called M.Q.T.*Transport*, we'll be referring to it as IoT *protocol* since it does session management.

One confusing thing about the name, it does not really queue messages (it's more of a store-retrieve model). This is why you may see it called MQ Telemetry Transport.

MQTT is an open, royalty-free and an OASIS open standard as of 2014. (https://adafru.it/f29)

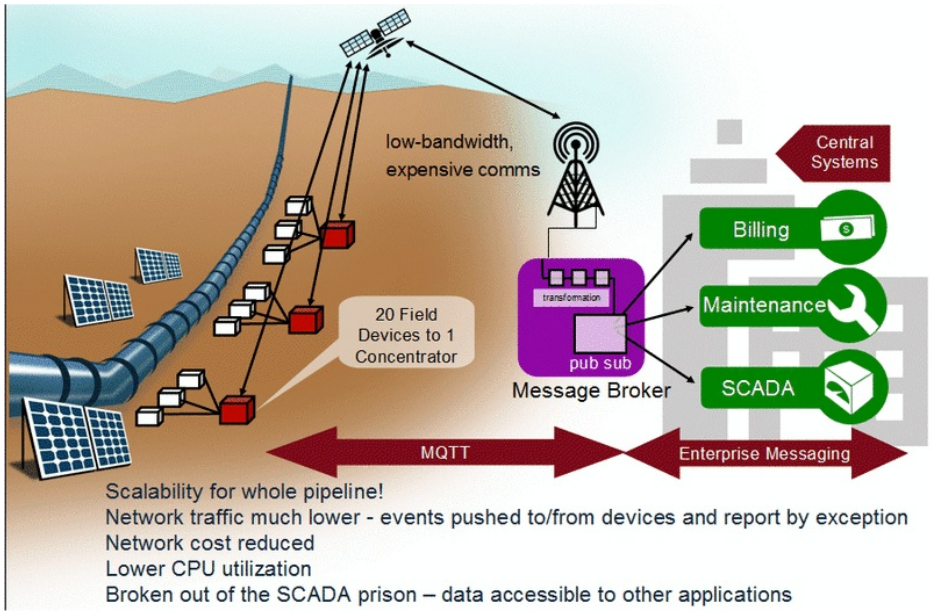
The Origins of MQTT

> *MQTT stands for MQ Telemetry Transport. It is a publish/subscribe, extremely simple and lightweight messaging protocol, designed for constrained devices and low-bandwidth, high-latency or unreliable networks. The design principles are to minimise network bandwidth and device resource requirements whilst also attempting to ensure reliability and some degree of assurance of delivery. These principles also turn out to make the protocol ideal of the emerging "machine-to-machine" (M2M) or "Internet of Things" world of connected devices, and for mobile applications where bandwidth and battery power are at a premium.*
>
> *MQTT was invented by Dr Andy Stanford-Clark of IBM, and Arlen Nipper of Arcom (now Eurotech), in 1999.*

MQTT was developed at IBM in the late 1990s, it was used to link oil pipeline sensors with satellites. The oil pipeline sensors and controls were not high speed, did not require a lot of data and were not near infrastructure.

The up/downlink used satellites that would move in and out of range, so HTTP would have been a very bad protocol to use because it requires a durable underlying transport.

Instead, the network needed something more flexible and understanding of failures and dropouts. One interesting piece of MQTT is that the protocol manages messages *asynchronously*. That is: the server/broker in an MQTT network can hold and forward messages from client to client so if one gets disconnected, it will be able to fetch the message when it reconnects later.



To keep things simple and allow bi-directional message passing, MQTT uses a **publish** and **subscribe** model.

Simple & Light

MQTT is a simple and well designed protocol, and it turns out that the same protocol used for oil pipeline sensors and satellites is handy for IoT. It's extremely simple and lightweight. The packet structure uses binary as much as possible for compactness. Compare this to HTTP and REST where data is encoded in unicode or ASCII.

Here's the structure of the CONNECT packet, which you use when initializing the connection.

	Description	7	6	5	4	3	2	1	0
Protocol Name									
byte 1	Length MSB (0)	0	0	0	0	0	0	0	0
byte 2	Length LSB (6)	0	0	0	0	0	1	1	0
byte 3	'M'	0	1	0	0	1	1	0	1
byte 4	'Q'	0	1	0	1	0	0	0	1
byte 5	'I'	0	1	0	0	1	0	0	1
byte 6	's'	0	1	1	1	0	0	1	1
byte 7	'd'	0	1	1	0	0	1	0	0
byte 8	'p'	0	1	1	1	0	0	0	0
Protocol Version Number									
byte 9	Version (3)	0	0	0	0	0	0	1	1
Connect Flags									
byte 10	User name flag (1) Password flag (1) Will RETAIN (0) Will QoS (01) Will flag (1) Clean Session (1)	1	1	0	0	1	1	1	x
Keep Alive timer									
byte 11	Keep Alive MSB (0)	0	0	0	0	0	0	0	0
byte 12	Keep Alive LSB (10)	0	0	0	0	1	0	1	0

You'll also need to transmit some subscription packets, but overall, connecting to a server only takes about 80 bytes (certainly less than 1KB).

You stay can connected all the time and send 2-byte ping packets every few seconds to verify connectivity, or connect once in awhile to manage messages.

bit	7	6	5	4	3	2	1	0
byte 1	Message Type (12)				DUP flag	QoS level		RETAIN
	1	1	0	0	x	x	x	x
byte 2	Remaining Length (0)							
	0	0	0	0	0	0	0	0

Every data 'publication' (push data from client to broker) and data 'subscription' (push data from broker to clients) is as little as 20 bytes.

Publication to subscription delay - the time it takes from when one client publishes data, to when that data is made available to subscriber clients, is instant.

The connection is single ‘socket’ so overhead is at a minimum. This makes connections and data super fast, Facebook uses it for the billions of people who use messenger on mobile (https://adafru.it/Cip)

The single-socket state-full connection does have a tradeoff in that, while not a lot of data is transmitted, it does have to hold that socket connection open. On the other hand, connections are so light and fast, maybe your publication application is OK with connecting every time needs to send data. There's flexibility for you to experiment with!

MQTT Topics

Just like REST has text-based URLs
(like https://twitter.com/adafruit/status/900750127425630208 (https://adafru.it/Ci6)) MQTT can has text-based **topics**.

Like REST, MQTT can manage arbitrary-data payloads to and from those topics.

You can send text or binary (a photo for example) - you don't have to encode the data if you don't want. As long as the clients can parse the data, you can use it. You can transmit any packet of data up to a theoretical max of 256 MB (although, in reality most brokers and clients don’t expect more than a few KB since for huge payloads you would want a more durable protocol that manages re-transmits, better error correction, etc.)

We'll cover topics in more detail in the next sections

Small, Code-wise

Unlike HTTP, there isn’t a lot of "build up a connection" and then "tear it down" overhead, as you saw above, the packets are small, and once you CONNECT you can PUBLISH and SUBSCRIBE immediately without any other headers. That means we can stream data in and out of multiple 'topics' quickly and easily.

The compactness makes it easy to adapt to any transport since there are no underlying layers that you have to depend on - while the vast majority of MQTT clients run on top of TCP/IP, *it isn't a requirement* - you can run on any low level transport with bi-directional data.

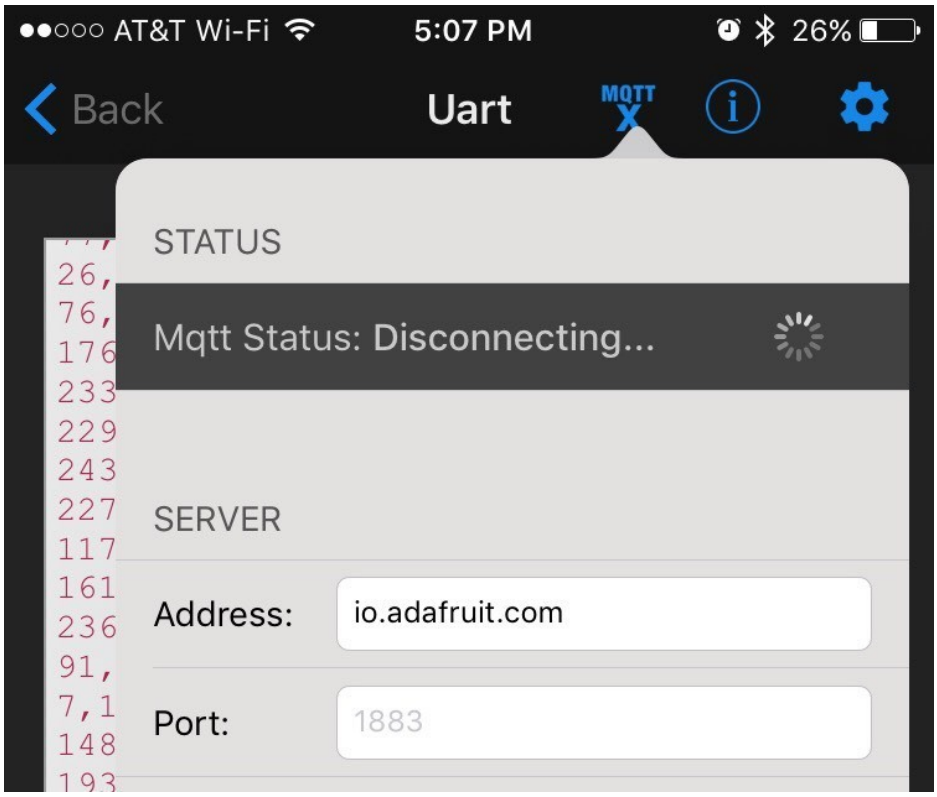
Since the protocol's state-machine is simple, full MQTT clients can fit in microcontrollers with as little as 32KB of Flash and 2KB of RAM, such as our Adafruit_MQTT (https://adafru.it/fp6) client.

The 'official' client is Paho, which has a wide variety of ports to your favorite language (https://adafru.it/Ciq)

Don't Forget Your Gateway!

While MQTT can run on top of any kind of transport, whether it be a mesh network, TCP/IP, Bluetooth, etc. If you are using Bluetooth, XBee, Bluetooth LE, or another non-Internet-connected protocol & device, you will need a gateway to get data to and from the Internet.

For example, the Adafruit Bluefruit Connect app has a BLE to Internet gateway that you can use for MQTT 'tunneling' (https://adafru.it/yb7)


●●○○○ AT&T Wi-Fi
5:07 PM
26%
Back
Uart
MQTT
STATUS
Mqtt Status: Disconnecting...
SERVER
Address:
io.adafruit.com
Port:
1883

Brokers & Clients

MQTT has some terms that make it easy to remember, what does what.

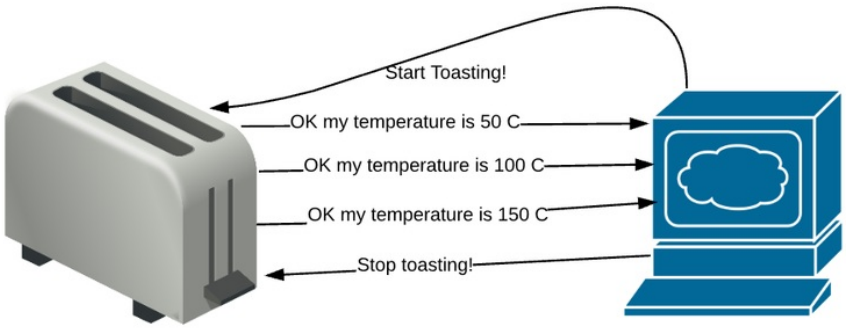
There are two types of players in MQTT, a broker and a client. Compare it with REST we have a 'server' and then 'clients' that connect, the naming is different because the 'server' in MQTT is also managing data between clients. But, it functions in almost-the-same way

There's only **one broker**, but there can be **multiple clients**.

- The broker is considered the 'reliable' one in the relationship - even if the connection goes out, the broker is expected to be available whenever the transport is working.
- The broker gets all the published messages from the clients and stores them in a database or memory.
- The broker then delivers the messages to the subscriber clients.

In general, the MQTT broker is not what we spend a lot of time engineering as there are many great free broker softwares (we're partial to Mosquitto (https://adafru.it/pey)), as well as *dozens* of broker services you can pay for that will do all the management for you. For example, Microsoft Azure (https://adafru.it/Cis), AWS IoT (https://adafru.it/Cit), even our very own adafruit.io (https://adafru.it/fsU)

The part you will be most involved in is the client - that's part of your application, whether it be a sensor, robot, or... toaster!



Broker & Clients Example

Here's an example showing the way brokers and clients interact

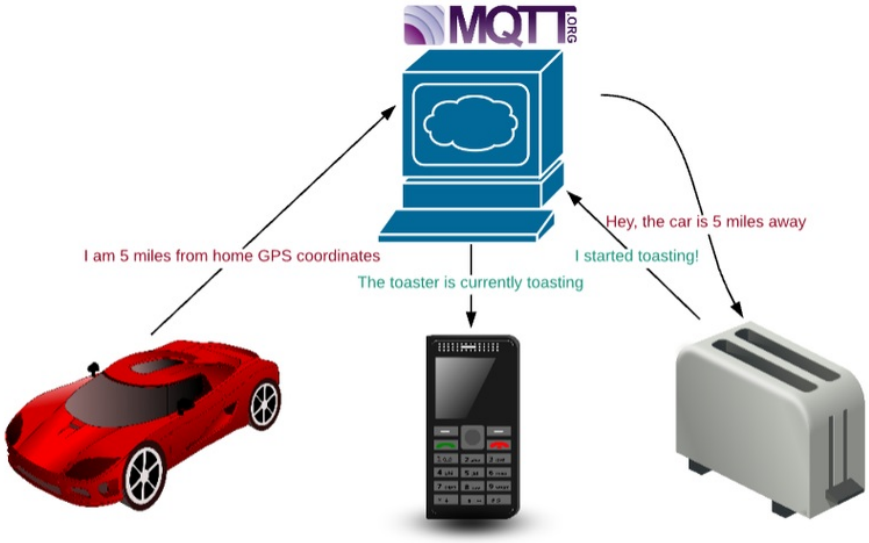
Let's say you have your internet of toaster at home (it has a WiFi chip, and is on your home network). And you have a geotracker in your car (a cellular+gps connection).

There's 2 ways you could have the two devices 'talk' to each other

1. Have one of the devices run as a 'server' with an IP address, so that the other sensor can connect to it at any time
2. Have a third 'server' somewhere, both toaster and car connect to that computer server and the computer sends messages back and forth.

Option #1 is in a sense, the 'least expensive' because no extra computer is needed. However, it's crazy difficult to pull off because the toaster or car have to be constantly waiting for a connection. And the other device needs to know the IP address of the listener. And then, what happens if a third device is involved?

Thus, we go with option #2 - the server that handles the messages? That's the **MQTT Broker** and the car and toaster are both **MQTT Clients**. You can now easily add more clients to add or monitor data, such as your mobile app (CyberToastApp, now available at the iTunes store?)



MQTT Topics

OK so now you know you need a broker, and you've got a few clients to connect up. You've loaded up your MQTT client software. Now what? How do you actually send/receive data?

Remember we mentioned that the whole point of protocols was to take the pile o' bits from the transport layer and organize them! With REST, the data is organized into URLs, but with MQTT we use topics.

Much like REST's **GET**/**POST** commands that reference a URL, MQTT clients **publish** and **subscribe** to a topic. And, like URLs, topics use / delimeters and ascii-text categorization.

We'll cover wildcards later, suffice to say that while REST strongly recommends, but does not require, true hierarchical URLs, because there's often meta-data in a REST call that can give more information, MQTT only has the topic name. So it's *really important* to have and consider MQTT topics a proper hierarchy.

Topic Hierarchy

Much like you would sort your files like

Documents/School/English/midterm.doc

or

Documents/Work/Presentation/Q3Sales.ppt

topics have a hierarchy.

We'll cover wildcards later, suffice to say that while REST strongly recommends, but does not require, true hierarchical URLs (because there's often meta-data in a REST call that can give more information) MQTT only has the topic name. So it's *really important* to have and consider MQTT topics a proper hierarchy.

Factory Temperature Network

Lets say you have a factory and you wanted to make temperature sensors placed on the HVAC units on each floor in your building publish their temperature, you could set up your topics like so:

`sensors/`*`FLOOR`*`/temperature/`*`HVAC_UNIT_ID`*

So, when we setup the topics, they'll look like this:

- The third floor main AC would be `sensors/fl3/temperature/AC`
- The tenth floor boiler would be `sensors/fl10/temperature/boiler`
- The fourth floor heater would be `sensors/fl4/temperature/heater`
- etc.

You don't have to organize the factory exactly like so, you have flexibility on how many layers of hierarchy you want, and what would be considered a super or sub category. But, you'll want to figure it out fast because its not easy to change once you've started pushing and storing data!

Dynamic or Pre-defined Topics?

Most brokers require you to set up your topics ahead of time and will not let you publish or subscribe to non-existent

topics (this is good for avoiding typos and corrupted data).

Others are flexible - whenever a publication comes in for a topic, that topic will be automatically created. This is good when you have a dynamic sensor network with new nodes joining without warning, so that you don't have to log into the server to tell it "hey we added another temperature sensor on the roof"

However, its much more likely that when a client subscribes to a topic, the brokers will send an error if that topic does not exist yet.

Publications to Subscriptions

By default, MQTT brokers were not designed to store or log data (they can, of course, but it's not a requirement of protocol)

The broker's job is quite simple. Whenever a client publishes a message to the topic on the broker, the broker will immediately send the message to all the clients that have subscribed to that topic.

That's it!

Using Topics For Configuration Changes

Using the topic model, you can do a lot with a little. For example, you can develop clients to only use specific topics to publish data to, and subscribe to a topic for configuration changes.

For example, say you have a remote cellular temperature sensor that publishes data to your broker under the **sensors/outpost/temperature**. Every temperature measurement requires power and data, so you don't want to transmit more than you need to. You could try to hard-code an algorithm for how often to measure (maybe more during the day than during night) *or* you could have a configuration client publish to the **sensors/output/temperature_frequency** topic. Then the sensor will be able to get updates at the next connection, and know to change the frequency of measurements. This kind of thing is *possible* with REST, but its not as elegant with polling as with a subscription-push.

Data Formatting & Typing

One thing to note is that topics don't have strict typing built in. HTTP for example, has data encoding on transfer in the headers. And if you remember in the REST JSON example, each element of the JSON collection had a *type* like a string, boolean or url - That's because its ascii-encoded.

With MQTT, to keep things light, we don't have any existing structure. There's nothing stopping a client in your factory's MQTT network from uploading a binary cat photo to the `temperature` topic. This is unlike REST where you have more control over data checking. Now, you *can* add code to your MQTT broker that will scrub bad data out for you, but it's not built in! So either be careful with your topics and do the data checking on the client pub/sub or perhaps have another client that subscribes to all topics and looks for bad data to clean up.

Topic Wildcards

There is one clever trick that the broker can do with topics, that you **cannot** do with REST: wildcards! With wildcards, any client can subscribe to any *part* of that topic, rather than a fully qualified topic.

So, given our earlier factory example, maybe you want to subscribe to everything at at your location, or maybe only sensors on the 3rd floor, or maybe just a specific temperature sensor. Especially if you have new clients joining in, wildcards will take care of the groupings for you.

There are two wildcards available, # and +

+ wildcard is used to get a **single level** of hierarchy**.** Here's an example wildcard topic:

`sensors/fl3/temperature/+`

This topic would let you monitor **all** of the temperature sensors on the 3rd floor.

You can get even more advanced with wildcard subscriptions with something like:

`sensors/+/temperature/+`

Note that there are two + wildcards here! This topic would get *all* the temperature data on *all* floors, that are publishing to a temperature topic. And you'd get updates *any* time one of those sensors were published.

wildcard can be used as a match for all remaining levels of hierarchy.

For example if you wanted all to subscribe to *everything* going on all *floors*, you'd use:

`sensors/#`

Or maybe you want to listen in on all 10th floor sensors:

`sensors/fl10/#`

As you can see, thinking through your topic hierarchy is required if you want to take advantage of the + and # wildcards!

MQTT QoS

MQTT QoS (Quality of Service)

MQTT has some basic Quality of Service (QoS) capability built in. Basically, say you were using MQTT over a radio, and your toaster is sending radio signals to some base station...there's a chance those messages won't arrive.

Depending on your type of data and the underlying transport, you may have different requirements for how reliable you want MQTT to be. **More reliability means more data transmissions, and more code management**. This is mostly managed by the broker, but the client can request certain QoS

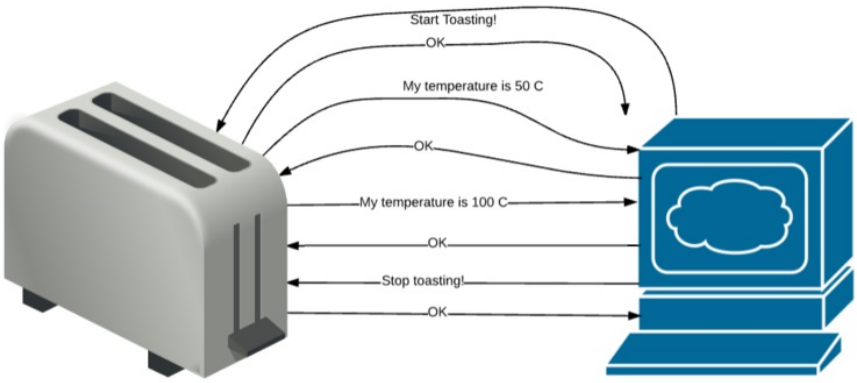
QoS 0 - Fire and Forget

The simplest QoS is called 0 - in this QoS, the broker/client will deliver the message once, with no confirmation. This is best used when your underlying transport is very reliable, say TCP/IP, or when you want to conserve power and data at the trade of consistancy

For example, say you have a GPS vehicle tracker that sends location data over cellular every 30 minutes. If your tracker is running over a long period of time it might be OK if once in a while, a GPS location is 'dropped'. You just want to get most messages so you have an idea of where your truck is. It's more important that you conserve battery because the cellular network has reliability and re-transmission built in - if you are outside range you don't want to keep re-trying, you'll use up your battery!

QoS 1 - Delivered at least once

The next step up is QoS 1. In this QoS, the broker/client will deliver the message at least once, with confirmation required. These are called **ACK**s and you'll get an **ACK** for every publication, subscription, etc. You'll need to send an ACK from the broker *and* the client!



For example, you have an IoT AC unit that is monitored and controlled remotely. When you send a message for it to turn on you want to make sure it turns on! If you send a message and the transport fails, you can just keep sending messages over until you get a reply from the broker.

This QoS is what most people use. It's a good balance!

QoS 2 - Deliver only once

Finally, you have QoS 2, the greatest of the QoS's! In this QoS, the broker/client will deliver the message **exactly once**

by using a four step handshake.

This one is not as common because it's very hard to really guarantee single-delivery without a rock solid transport (https://adafru.it/Civ). So you will want to consider a feedback system using different topics and sensors instead!

For example, you have a robotic tractor that you send commands like "Move forward 50 meters" - you want to make sure that you don't accidentally send two messages and it turns out it moved 100 meters! Since QoS 2 is a bit of a pain, you may want to change this to, instead, use two topics: one that sends the movement command from controller to tractor, and then another one from the tractor to controller that gives the current location.

MQTT Extras

There's three more little extras that you can use with MQTT

Retained Messages

Technically there is no base requirement for the MQTT broker to store your data once the subscriptions have been sent. But you can request or configure **retained messages** with your broker!

If you've enabled retained messages, the broker will **keep** a copy of published messages, even after they are sent to all the subscribers. If a new subscription is made by a client after the publication, and there is a previously sent and retained message for that topic, the retained message will be sent immediately.

The “last known good” retained message can be handy for remote configurations, when a topic isn’t updated that often, but you are adding new clients that are subscribing to the topic.

For example, say you have a remote controlled light bulb. The light bulb can be ON or OFF and receives message from other clients via the broker over WiFi. But, oh no! The WiFi router went down. When the WiFi comes back on, the light bulb may not know if there was an ON/OFF request that came through while the WiFi was down. It can connect and re-subscribe to get that retained message which will let it know what the last state was set to.

Clean Session / Durable Connections

A client can set a “clean session” flag - that’s like a clean start if you want to start from scratch or not.

If it’s set to **false** the connected client is considered “durable” this means when the client disconnects, all the subscriptions store will remain and any QoS messages will be stored until it connects again.

If the session is clean and marked as **true**, all subscriptions will be removed then the client disconnects.

This is useful if you want to minimize the setup of clients in an oft-disconnected transport, the client is already identified to the broker by the CONNECT packet so it avoids having to re-send all the SUBSCRIBE requests on every connect.

Last Will

The Last Will is sort of the opposite of the Retained Message. Whereas retained messages allow a client to find out if they missed a message on connection, Last Will lets a client send a final message on unexpected disconnection. Last Wills have a topic & QoS - it’s basically just a pre-set publication.

Say you have a piece of industrial equipment that is controlled by a large relay, the relay is connected over MQTT as a client to the **factory/machine/relay** topic that will let it know whether the relay should be ON or OFF.

Another client is the controller, it sends the commands to the relay. There's a risk that the controller may fall off the network somehow, and the relay will be operating without a controller. In this case, the 'safest' setting for the machine is to have the relay be off. So, the controller client sends a **Last Will** to the **factory/machine/onoffrelay** topic with the **OFF** message. If the broker doesn't hear from the controller in a pre-set amount of time, it will send send the last will to all subscribers.

CoAP

The last protocol we'll cover is CoAP (https://adafru.it/Ciw) - **Co**nstrained **A**pplication **P**rotocol

CoAP is a bit like REST but pared down to be as light as MQTT.

REST-isms

Like REST, it has GET/PUT/POST/DELETE to URLs behavior. But unlike REST, it has a very light and simple packet structure that is designed to be as minimal as possible, with binary data representation for commands and data whenever possible.

Stateless **and** Sessionless

HTTP/REST uses sessions but is stateless, you're expected to disconnect after data is transmitted. MQTT uses the idea of a 'session' or continuous connection (e.g. one socket, one session).

CoAP is not only stateless (per connection), it's sessionless: data is sent and requested at *any* time, somewhat like if you had MQTT but without a connection state.

That means you could run CoAP on a transport like UDP, SMS, packet radio or satellite where it's hard to get immediate responses!

Server/Client one-to-one model

Like REST, the server doesn't manage many-client message routing. Each client connects to the server and sends/requests data. BUT like MQTT, the client can become an 'observer' to get frequent asynchronous updates to a topic of interest (that way it's not polling like traditional REST)

Watch your Transport!

About that stateless connectivity - on the Internet usually UDP is used. But that can be a problem with firewalls that often block individual 'random' packets and only permit outgoing TCP connections (you can run CoAP over TCP like Particle but it isn't as common) but then you sort of lose the benefit of UDP.

This is most relevant when running on cellular & WiFi where you've got a direct Internet connection. Another downside is that since each message is stand-alone you cannot have any fragmentation - each message must fit in a single ZigBee, UDP, sigfox, etc. packet.

REST Interoperability

The biggest benefit to CoAP is it's similar enough to REST that you can use it to easily transfer data between web applications because *in theory* its designed to be interoperable. And, there's more stuff built into the protocol for data formatting and resource querying. But we don't see CoAP used very often so it's hard to say if people are really taking advantage of it!

What to Use?

There's a few other protocols that you might bump into on your IoT journey but we covered the big Three: HTTP/REST, MQTT and CoAP.

Even though protocols and transports are two layers of your IoT network, they aren't completely independent.

- In particular, you'll likely be using **REST** whenever you have a transport that has **high bandwidth and speed** - like Ethernet, when your transport is Internet-connected and talking to online services.
- **MQTT** is most often used when your transport has **battery, bandwidth or packet size restrictions**, like WiFi or Cellular.
- Finally, **CoAP** is there when you have **very-low bandwidth stateless connections** like LoRa, ZigBee or Bluetooth.

Once you've picked your transport and protocol, you can take the next step and start sketching out your IoT network. That will mean determining what services you will be using.

Until then, arrivederci, auf wiedersehen and goodbye!

Demo Time!