# Chapter 6: Cloud Connectivity

## Objective

At the end of Chapter 6 you will understand how to build a complete WICED IoT App using one of the cloud application protocols (MQTT, COAP, AMQP, HTTP or Sockets). In addition, you will have a big picture understanding of each of those protocols.

This section is the prequel to Chapter 6 with the foundation information required to understand the rest of the chapter. At the end of this section you should understand the basics of Encryption, Secure Channels, Sockets and the Application Protocols HTTP, MQTT, AMQP, and COAP.

## Time: 4 Hours

## Fundamentals

### Symmetric and Asymmetric Encryption: A Foundation

When you see “HTTPS” in your browser window, the “S” stands for Secure. The reason it is called Secure is that it uses an encrypted channel for all communication. But how can that be? How do you get a secure channel going? And what does it mean to have a secure channel? What is secure? This could be a very complicated topic as establishing a fundamental mathematical understanding of encryption requires competence in advanced mathematics that is far beyond most everybody on the face of this planet. It is also beyond what there is room to type in this manual. It is also far beyond what I have the ability to explain. But, don’t despair. The practical aspects of getting this going are actually pretty simple.

All encryption does the same thing. It takes un-encrypted data, combines it with a key, and runs it through an encryption algorithm to produce encrypted data. You then transmit the encrypted data over the network. When the other side receives the data it decrypts the encrypted data by combining it with a key, and running the decrypt algorithm.

There are two types of encryption schemes, symmetric and asymmetric.

[Symmetric](https://en.wikipedia.org/wiki/Symmetric-key_algorithm) means that both sides use the same key. That is, the key that you encrypt with is the same as the key you decrypt with. Examples of this type of encryption include [AES](https://en.wikipedia.org/wiki/Advanced_Encryption_Standard) and [DES](https://en.wikipedia.org/wiki/Data_Encryption_Standard). Symmetric encryption is preferred because it is very fast and secure. Unfortunately, both sides need to know the key before you can use it (i.e. the encrypt key is exactly the same as the decrypt key). The problem is, if you have never talked before how do you get both sides to know the key?

[Asymmetric](https://en.wikipedia.org/wiki/Public-key_cryptography), often called Public Key, encryption techniques use two keys that are mathematically related. The keys are often referred to as the “public” and the “private” keys. The private key can be used to decrypt data that the public key encrypted and vice versa. This is super cool because you can give out your public key to everyone, they can encrypt data, then only your private key can be used to decrypt it. What is amazing about Asymmetric encryption is that even knowing the Public key you can’t figure out the private key (there is the math magic). The problem with this encryption technique is that it is slow.

What now? The most common technique to communicate is:

* You open an unencrypted connection to a server
* You give out your public key to the server
* The server then creates a random symmetric key
* The server then encrypts its newly created random symmetric key using your public key and sends it back to you
* You use your private key to decrypt the symmetric key
* You open a new channel using symmetric key encryption



This scheme is completely effective against eavesdropping. What happens if someone eavesdrops the original public key? That is OK because they won’t have the “client private key” required to decrypt the symmetric key. So, what’s the hitch? What this scheme doesn’t work against is called man-in-the-middle (MIM). An MIM attack works by:

* You open an unencrypted connection to a server [but it really turns out that it is a MIM]
* You send your public key to the MIM
* The MIM opens a channel to the server
* The MIM sends its public key to the server
* The Server encrypts a symmetric key using the MIMs public key and send it back to the MIM
* The MIM decrypts the symmetric key using its private key
* The MIM sends you the symmetric key encrypted with your public key
* You unencrypt the MIM symmetric key using your private key
* Then you open new channel to the MIM using the symmetric key
* The MIM opens up a channel to the server using the symmetric key

Once the MIM is in the middle it can read all of the traffic. You are only vulnerable to this attack if the MIM gets in the middle on the first transaction. After that, things are secure.

However, the MIM can easily happen if someone gets control of an intermediate connection point in the network e.g. WiFi Access Point. There are only two ways to protect against MIM attacks:

* Pre Share the key
* Use a [Certificate Authority](https://en.wikipedia.org/wiki/Certificate_authority) (CA)

A CA is a server on the internet that has a huge dictionary of keys. To use a CA, you embed the CAs verified public key in your system (so you can make a secure connection to the CA). Then when you get a key from someone you don’t know, you open a secure connection to the CA and it verifies the key that you have matches the key you were sent.

If the MIM sends you its public key then you check with the CA and find out that the MIM public key does not belong to the server that you are trying to connect to, then you know that you are being subjected to an MIM attack. How do you prevent an MIM when talking to a CA? This is done by building in known valid certificates into your program. This morning when I looked at the certificates on my Mac there were 179 built in, valid certificates.

### [Secure Sockets Layer (SSL) / Transport Layer Security (TLS)](https://en.wikipedia.org/wiki/Transport_Layer_Security)

SSL and TLS are two Application Layer Protocols that handle the key exchange described in the previous section and present an encrypted data pipe to the layer above it - i.e. the Web Browser or the WICED device running MQTT. SSL is a fairly heavy (memory and CPU) protocol and has largely been displaced by the lighter weight and newer TLS.

Both of these protocols are generally ascribed to the Application layer but to me it has always felt like it really belongs between the Application and the Transport Layer. TLS is built into WICED and if you give it the keys (from the DCT) when you initialize a connection its operation appears transparent to the layer above it. Several of the application layer protocols that are discussed in this chapter rest on a TLS connection - i.e. HTTP🡪TLS🡪TCP🡪IP🡪WiFi Datalink 🡪 WiFI 🡪 Router 🡪 Router🡪Server Ethernet🡪Server Datalink🡪Server IP🡪Server TCP🡪TLS🡪HTTP Server

### Sockets – Old School

For Applications, i.e. a web browser, to communicate via the TCP transport layer they need to open a **Socket**. A Socket, or more properly a TCP Socket, is simply a reliable, ordered pipe between two devices on the internet. To open a socket you need to specify the IP Address and [Port](https://en.wikipedia.org/wiki/Port_(computer_networking)) Number (just an unsigned 16-bit integer) on the Server that you are trying to talk to. On the Server there is a program running that listens on that Port for bytes to come through. Sockets are uniquely identified by two tuples (source IP/port) an (destination IP/port) e.g. 192.168.15.8/3287 + 184.27.235.114/80. This one reason why there can be multiple open connections to a webserver running on port 80.

There are a bunch of [standard ports](https://en.wikipedia.org/wiki/List_of_TCP_and_UDP_port_numbers) (which you might recognize) for Applications including:

* HTTP 80
* SMTP 25
* DNS 53
* POP 110
* MQTT 1883

WICED easily supports TCP sockets (wiced\_tcp\_create\_socket) and you could create your own protocol to talk between your IoT device and a server. However, the protocol that you create would be proprietary and probably a bad idea as there are a bunch of very capable protocols for exchanging data (e.g HTTP, MQTT, COAP, AMQP). All of these protocols are built into WICED and sit on top of TCP Sockets. But, sockets are the way that people used to do this kind of task, which is why I called this section Old School.

### [Hyper Text Transfer Protocol (HTTP)](https://en.wikipedia.org/wiki/Hypertext_Transfer_Protocol)

HTTP is an Application Layer Protocol that operates over TCP Sockets. It can perform the following functions:

* GET (retrieve data) from a specific place
* POST (put data) to a specific place
* HEAD, PUT, DELETE, TRACE, OPTIONS, CONNECT, PATH (less commonly used)

To initiate these commands, you open a socket generally to TCP port 80 and send the request in a specific format.

For instance, you can send an HTTP get request to open “/” on example.com:

GET /index.html HTTP/1.1

Host: [www.example.com](http://www.example.com)

Example.com will respond with:

HTTP/1.1 200 OK

Date: Mon, 23 May 2005 22:38:34 GMT

Content-Type: text/html; charset=UTF-8

Content-Encoding: UTF-8

Content-Length: 138

Last-Modified: Wed, 08 Jan 2003 23:11:55 GMT

Server: Apache/1.3.3.7 (Unix) (Red-Hat/Linux)

ETag: "3f80f-1b6-3e1cb03b"

Accept-Ranges: bytes

Connection: close

<html>

<head>

<title>An Example Page</title>

</head>

<body>

Hello World, this is a very simple HTML document.

</body>

</html>

It is possible (and semi-common) to build IoT devices that use HTTP to “PUT” their data to webservers in the cloud and “GET” their instructions/data from webservers. However, HTTP is somewhat heavy and is generally being displaced by other protocols that are more suited to IoT.

### [Message Queueing Telemetry Transport (MQTT)](https://en.wikipedia.org/wiki/MQTT)

MQTT is a lightweight messaging protocol that allows a device to **Publish** messages to a specific **Topic** on a **Message Broker**. The Message Broker will then relay the message to all devices that are **Subscribed** to that topic.

A Topic is simply the name of a message queue - e.g. “mydevice/status” or “mydevice/pressure”.

A Subscription is just the request by a client to have all messages published to a specific topic sent to the client.

Publishing is just the process by which a client sends a blob of data to a specific topic on the message broker.

A Message Broker is just a server that handles the tasks:

* Establishing connections (MQTT Connect)
* Tearing down connections (MQTT Disconnect)
* Accepting subscriptions to a Topic from clients (MQTT Subscribe)
* Turning off subscriptions (MQTT Unsubscribe)
* Accepting messages from clients and pushing them to the subscribers (MQTT Publish)

The format of the messages being sent in MQTT is unspecified. The message broker does not know (or care) anything about the format of the data and it is up to the system designer to specify an overall format of the data. All that being said, [JavaScript Object Notation (JSON)](https://en.wikipedia.org/wiki/JSON) has become the lingua franca of IoT.

Cloud providers that use MQTT include Amazon AWS,

### [Constrained Object Application Protocol (COAP)](https://en.wikipedia.org/wiki/Constrained_Application_Protocol)

Cloud providers that use COAP include Samsung ARTIK

### [Advanced Message Queuing Protocol (AMQP)](https://en.wikipedia.org/wiki/Advanced_Message_Queuing_Protocol)

Cloud providers that use AMQP include

### [JavaScript Object Notation (JSON)](https://en.wikipedia.org/wiki/JSON)

JSON is an open-standard format that uses human-readable text to transmit data consisting of attribute–value pairs. JSON supports the following data types:

* Double precision floating point
* Strings
* Boolean (true or false)
* Array (use “[]” to specify the array)
* Key/Value pairs as “key”:value (use “{}” to specify the keymap)

For example, a legal JSON file looks like this:

{

“astringkey” : “alan”,

“age” : 48,

“badass” : true,

“children”, [“Anna”,”Nicholas”],

“address” : {

“number”:”201”

“street”: “East Main Street”

“city”: “Lexington”,

“state”:”Kentucky”,

“zipcode”:40507

}

}

The WICED-SDK has a JSON parser built in. You can find these functions in the directory “Utilities🡪JSON\_parser”.

## Known Errata + Enhancements + Comments