Both MQTT and CoAP:

- Are open standards
- Are better suited to constrained environments than HTTP
- Provide mechanisms for asynchronous communication
- Run on IP
- Have a range of implementations

MQTT gives flexibility in communication patterns and acts purely as a pipe for binary data. CoAP is designed for interoperability with the web.

MQTT

MQTT is a publish/subscribe messaging protocol designed for lightweight M2M communications. It was originally developed by IBM and is now an open standard.

Architecture

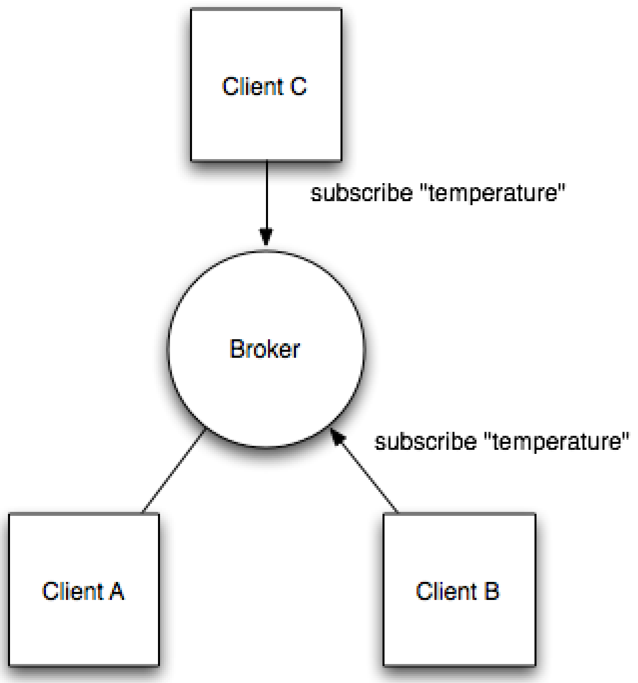
MQTT has a client/server model, where every sensor is a client and connects to a server, known as a broker, over TCP.

MQTT is message oriented. Every message is a discrete chunk of data, opaque to the broker.

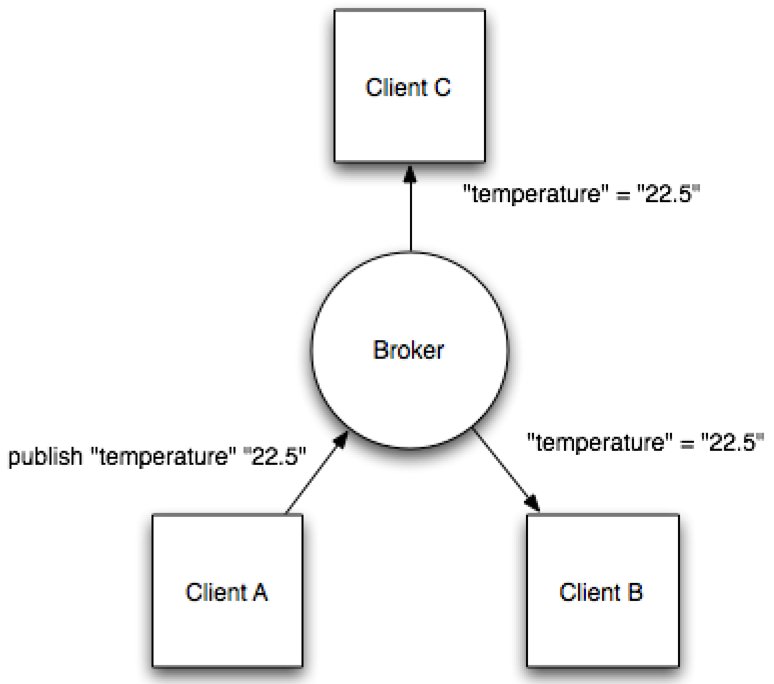
Every message is published to an address, known as a topic. Clients may subscribe to multiple topics. Every client subscribed to a topic receives every message published to the topic.

For example, imagine a simple network with three clients and a central broker.

All three clients open TCP connections with the broker. Clients B and C subscribe to the `topic temperature`.



At a later time, Client A publishes a value of `22.5` for topic `temperature` . The broker forwards the message to all subscribed clients.



The publisher subscriber model allows MQTT clients to communicate one-to-one, one-to-many and many-to-one.

Topic matching

In MQTT, topics are hierarchical, like a filing system (eg. kitchen/oven/temperature). Wildcards are allowed when registering a subscription (but not when publishing) allowing whole hierarchies to be observed by clients.

The wildcard + matches any single directory name, # matches any number of directories of any name.

For example, the topic `kitchen/+/temperature matches kitchen/foo/temperature but not kitchen/foo/bar/temperature`

`kitchen/# matches kitchen/fridge/compressor/valve1/temperature`

Application Level QoS

MQTT supports three quality of service levels, “Fire and forget”, “delivered at least once” and “delivered exactly once”.

Last Will And Testament

MQTT clients can register a custom “last will and testament” message to be sent by the broker if they disconnect. These messages can be used to signal to subscribers when a device disconnects.

Persistence

MQTT has support for persistent messages stored on the broker. When publishing messages, clients may request that the broker persists the message. Only the most recent persistent message is stored. When a client subscribes to a topic, any persisted message will be sent to the client.

Unlike a message queue, MQTT brokers do not allow persisted messages to back up inside the server.

Security

MQTT brokers may require username and password authentication from clients to connect. To ensure privacy, the TCP connection may be encrypted with SSL/TLS.

MQTT-SN

Even though MQTT is designed to be lightweight, it has two drawbacks for very constrained devices.

Every MQTT client must support TCP and will typically hold a connection open to the broker at all times. For some environments where packet loss is high or computing resources are scarce, this is a problem.

MQTT topic names are often long strings which make them impractical for 802.15.4.

Both of these shortcomings are addressed by the MQTT-SN protocol, which defines a UDP mapping of MQTT and adds broker support for indexing topic names.

CoAP

CoAP is the Constrained Application Protocol from the CoRE (Constrained Resource Environments) IETF group.

Architecture

Like HTTP, CoAP is a document transfer protocol. Unlike HTTP, CoAP is designed for the needs of constrained devices.

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.

CoAP is designed to interoperate with HTTP and the RESTful web at large through simple proxies.

Because CoAP is datagram based, it may be used on top of SMS and other packet based communications protocols.

Application Level QoS

Requests and response messages may be marked as “confirmable” or “nonconfirmable”. Confirmable messages must be acknowledged by the receiver with an ack packet.

Nonconfirmable messages are “fire and forget”.

Content Negotiation

Like HTTP, CoAP supports content negotiation. Clients use `Accept` options to express a preferred representation of a resource and servers reply with a `Content-Type` option to tell clients what they’re getting. As with HTTP, this allows client and server to evolve independently, adding new representations without affecting each other.

CoAP requests may use query strings in the form `?a=b&c=d` . These can be used to provide search, paging and other features to clients.

Security

Because CoAP is built on top of UDP not TCP, SSL/TLS are not available to provide security. DTLS, Datagram Transport Layer Security provides the same assurances as TLS but for transfers of data over UDP. Typically, DTLS capable CoAP devices will support RSA and AES or ECC and AES.

Observe

CoAP 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.

Resource Discovery

CoAP defines a standard mechanism for resource discovery. Servers provide a list of their resources (along with metadata about them) at /.well-known/core. These links are in the application/link-format media type and allow a client to discover what resources are provided and what media types they are.

NAT Issues

In CoAP, a sensor node is typically a server, not a client (though it may be both). The sensor (or actuator) provides resources which can be accessed by clients to read or alter the state of the sensor.

As CoAP sensors are servers, they must be able to receive inbound packets. To function properly behind NAT, a device may first send a request out to the server, as is done in LWM2M, allowing the router to associate the two. Although CoAP does not require IPv6, it is easiest used in IP environments where devices are directly routable.

Comparison

MQTT and CoAP are both useful as IoT protocols, but have fundamental differences.

MQTT is a many-to-many communication protocol for passing messages between multiple clients through a central broker. It decouples producer and consumer by letting clients publish and having the broker decide where to route and copy messages. While MQTT has some support for persistence, it does best as a communications bus for live data.

CoAP is, primarily, a one-to-one protocol for transferring state information between client and server. While it has support for observing resources, CoAP is best suited to a state transfer model, not purely event based.

MQTT clients make a long-lived outgoing TCP connection to a broker. This usually presents no problem for devices behind NAT. CoAP clients and servers both send and receive UDP packets. In NAT environments, tunnelling or port forwarding can be used to allow CoAP, or devices may first initiate a connection to the head-end as in LWM2M.

MQTT provides no support for labelling messages with types or other metadata to help clients understand it. MQTT messages can be used for any purpose, but all clients must know the message formats up-front to allow communication. CoAP, conversely, provides inbuilt support for content negotiation and discovery allowing devices to probe each other to find ways of exchanging data.

Both protocols have pros and cons, choosing the right one depends on your application.

Do experiments, build prototypes and deploy test devices on networks.

Toby Jaffey is Founder and CTO at 1248 which provides knowledge and systems to help manufacturers connect their products quickly and effectively, at scale.