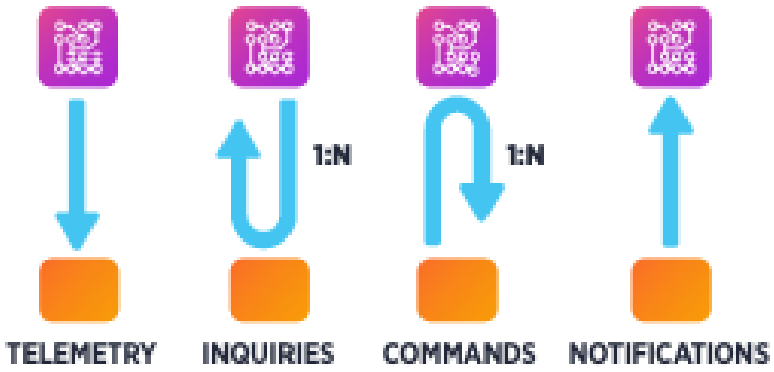
Communication Patterns

The IoT world is all about communication between devices, gateways, and the Cloud; messages are exchanged between all of these parties in order to provide a comprehensive end-to-end solution. It's common to consider the following as the main communication patterns, which are different in terms of how messages flow between parties and the objectives they have:

- **Telemetry**: Data flows in one direction from the device to other systems for conveying status changes in the device itself (i.e. sensors reading, ...)
- **Inquiries**: Requests from the device looking to gather required information or asking to initiate activities
- **Commands**: Commands from other systems sent to a device (or a group of devices) to perform specific activities expecting a result from the command execution, or at least a status for that
- **Notifications**: Information flows in one direction from other systems to a device (or a group of devices) for conveying status changes



Every pattern could have persistence needs, so they'll be using a store and forward mechanism (i.e. using queues, topic/subscriptions in a broker) or relying on direct messaging, where the receiver needs to be online in order to allow the devices to send data. The former is mainly used by the Command pattern, because the device might not be online at the time the command is sent; sometimes a TTL (Time To Live) on the command message is useful in order to avoid the possibility that an offline device will execute an "old" message that is not useful at the time the device comes back online. The direct messaging mechanism is often used for Telemetry even if, in some use cases, storing telemetry data could be useful as well.

Sometimes the Telemetry pattern can evolve into the Event pattern. The main differences are:

- Better QoS (Quality of Service) in terms of delivery. For Telemetry, the "most at once" delivery is enough because even if one data value is lost, a new, updated value will be sent in a very short time. For Event, an "at least once" delivery is needed considering, for example, alarms. In such cases, of course, the destination system should be able to handle the "idempotent" nature of the action related to the received message.

- Message Persistence in order to avoid losing messages if the destination system isn't working properly or isn't online when the message arrives.

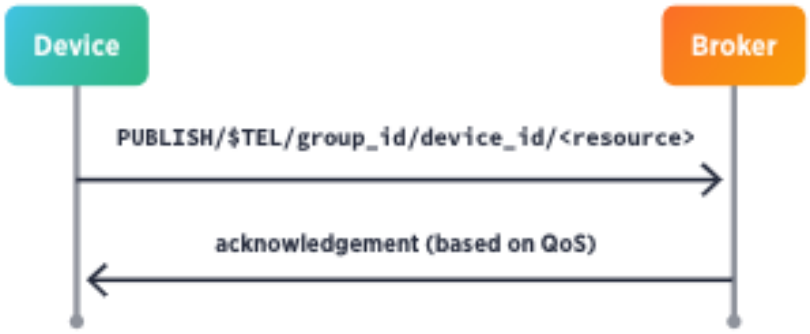
The main protocols used in IoT are really different in the way they implement the patterns above. IoT is more about messaging, which already defines its own patterns, like competing consumers, request/reply, pub/sub, and so on. Protocols provide such patterns in different ways (natively or not).

Telemetry

The HTTP protocol can implement this pattern in two ways: acting as a "client" and sending a PUT/POST request on the resource to update (i.e. representative of a sensor value), or acting as a "server", which receives GET requests from other systems for getting data. In any case, it's all about request/reply. The drawbacks are that HTTP is really verbose because it is "text-based", and it doesn't provide QoS (only based on TCP).

If working in "server" mode, HTTP has networking problems for connecting with the device when there is a NAT or roaming (mobile networks).

The MQTT protocol was born for telemetry, and it natively implements the publish/subscribe pattern. The device acts as a "publisher" for publishing data to a "topic". On the other side, the system acts as a "subscriber", so it's subscribed to that topic for getting messages. MQTT provides all the well-known QoS levels, but no flow control; a broker can be flooded by messages without the possibility of stopping devices from doing that.



The AMQP protocol is well-known because it provides both request/reply and publish/subscribe natively, so it fits great for Telemetry as well having the device send messages to a destination. The big advantage is related to the built-in flow control at two different levels: in terms of bytes exchanged and in terms of messages. In the latter case, there is a "credit-based" mechanism; a sender needs "credits" from the receiver in order to be able to send messages. AMQP provides all QoS levels as well.

Inquiry

Implementing this pattern with HTTP is pretty simple thanks to the request/reply nature of this protocol. It's all about a GET request from a device to a server resource for getting information. The drawback is that all the mechanisms work in a synchronous way.

Inquiry is "more" difficult to implement with MQTT because it needs new semantics at the application level. The device subscribes to a "dedicated" topic (i.e. with a request-id information on it) in order to receive the reply. To send such a request, it has to be a publisher on the inquiry topic putting the request-id inside the published message payload in some way. The request-id is needed by the server to connect to the topic for which a message will be publishing the response. There is the need to create a correlation between request and reply at the application level.

IoT is more about messaging, which already defines its own patterns, like competing consumers, request/reply, pub/sub, and so on. Protocols provide such patterns in different ways (natively or not).

The AMQP protocol fits very well in this case, because an AMQP message provides some useful system properties (as metadata). First of all, every message has an identifier (a message-id) and a reply-to property as well; it means that the devices send an inquiry telling the server where (the address) it wants to receive the response. The server will send the reply setting a correlation-id property (using the message-id of the request). The correlation mechanism is provided for free at the protocol level. Another strength is its asynchronous nature; replies can be provided by the server in any order (if needed) without losing the correlation with the related requests.

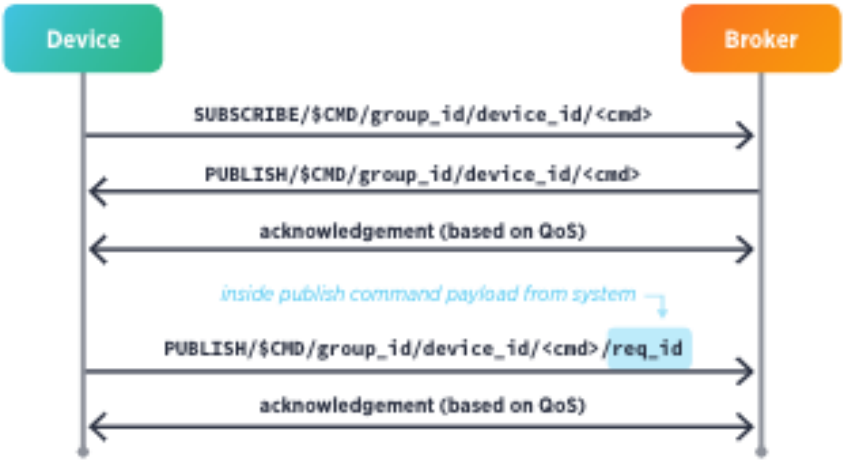
Command

This time, it's not the device that starts the communication, as in the previous patterns, but the external system; it means that the device should be reachable in some other way.

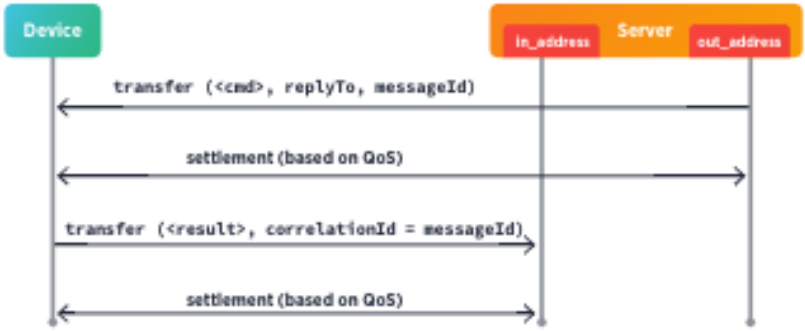
With HTTP, it can be implemented by having the device act as a "server" or "client". If it acts as a "server", the external system sends a POST request, waiting synchronously (long polling) for the result through the related HTTP response. The other possibility is polling the device with subsequent GET requests for getting the command result. In any case, there is always the problem of reaching a device behind NAT or roaming on mobile networks. If it acts as "client", the device sends a GET request to the system, asking if there is a command to execute; the reply can be received through long polling, so that the system doesn't send an HTTP response until a command is available, or with continuous polling. When the command is executed, the device sends another POST request to provide the command result to the server.

The implementation with MQTT is quite similar to the Inquiry. There is no support for request/reply, so it has to be built at the application level using an additional semantic on topics. The device subscribes to a topic for receiving the command, and the server sends such commands as messages, where the payload should contain a sort of request-id. After executing the command, the device sends the result publishing a message to a topic with the above request-id in the name. Even in this case, the command/result correlation is built at the application level.

What happens if the device is offline? Using the “retain” flag, the last command can be delivered when the device comes back online, but commands are queued only if the session feature is used.



AMQP is better than the other protocols from this point of view. The server can send a message command specifying the message-id and the reply-to system properties. After executing the command, the device sends the result as a message with the correlation-id set to the above message-id. This provides a correlation between the command and result; such a message is sent to the reply-to specified address, where the server is waiting for the result. Of course, the exchange happens in a completely asynchronous way; if more commands are not strictly related, the server can send them all together and receive the results asynchronously (eventually out of order). In order to avoid an offline device executing an “old” command when it comes back online, the TTL (Time To Live) property provided in an AMQP message can be used.



Notification

This pattern is quite the opposite of the Telemetry, and the flow is similar to the Command pattern, but without the need for a reply.

With HTTP, the device receives a notification from the external system acting as “server” (sending with a POST request) or acting as a “client” (sending a GET request) on a specific resource. In the

latter case, the server can reply only when the notification is ready (long polling) or immediately (with or without a result): It means that the device has to poll the server additional times.

If HTTP is working in “server” mode, addressing problems related to NAT and roaming are still there.

With MQTT, the device subscribes to a specific topic where the server publishes the notifications. This pattern fits very well with this protocol due to its publish/subscribe nature — with the drawback of not having built-in flow control: It means that the device could be overwhelmed by a lot of notifications.

Finally, with AMQP, the device is able to receive the notification pushed by the server as with MQTT, but with the big strength of built-in flow control: The device can stop the server from sending messages if it isn’t able to process them.