Which application/messaging protocol is right for me?

Building a connected device solution calls for several design and architectural decisions. Which protocol(s) should you use to connect your devices and their applications?

Which application/messaging protocol is right for me?

Building a connected device solution (IoT/Internet of Things) calls for several design and architectural decisions. One of these includes how your fleet of devices and its various applications will communicate with one another. In order words, which protocol(s) should you use to connect your devices and their applications?

We will cover different aspects and criteria to consider when deciding on the protocol(s) for your project running higher-end embedded Linux embedded devices. For the sake of focus, we will be intentionally omitting protocols for lower-end devices such as sensors.

Fit for purpose

There is no one-size fits all protocol. The best choice depends on your current and anticipated use cases. There may even be more than one protocol depending on the architecture, type of traffic, and payload.

Factors to consider include:

- Will the traffic be public, or internal only?
- How important is resource consumption on the devices?
- What level of scale do you have to support and do you need synchronous and/or asynchronous communication?
- Quality of service (QoS)
- How critical is it for the traffic to be encrypted?
- Data integrity and privacy
- How important is ease-of-use and time to market over technical superiority?
- Interoperability with other protocols essential
- Do you even consider developing your own protocol (hope not), want to try an open protocol that potentially can crumble (but maintained by yourself), or go for a safer more common widely supported and standardized protocol in use by large organizations and popular applications, whether open source or not?

The stack of questions quickly adds up while navigating to the best solution for your connected device project. When faced with a large number of choices, one recommendation would be to develop a set of criteria and align with your team on what is deemed most important for the specific project.

Criteria

At Mender.io, we recently went through this exercise. In addition to discussing and ensuring we considered the most obvious aspects, we ended up with a short list of priorities perceived to be most important to our specific scenario. Beside obvious aspects such as scalability, security, and avoiding reinvention, we concluded with the following list for our first implementation:

Heterogeneous support

How easy will it be to support various types of devices? Mender is a general-purpose solution that needs to support a wide variety of devices today and in the future.

Development speed

How hard will it be to implement the protocol? Do libraries exist in, for us, relevant languages? We value standardized and well-tested solutions.

Popularity and standardization

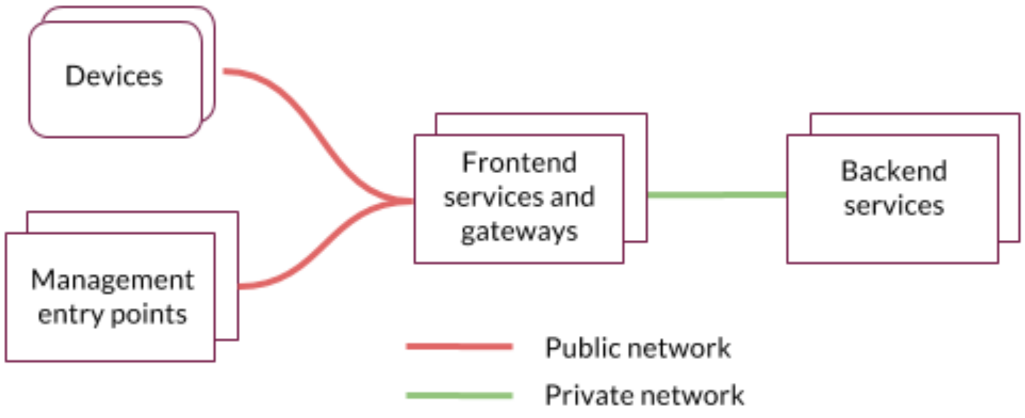
We do not consider the protocol to be the layer where our level of innovation and differentiation should reside. We prefer something common, standard, battle-tested, and what will represent a path of low resistance among users and the ecosystem as a whole.

For your project, you might end up with a different list of criteria, but the point of having such a list lies in helping you narrow your options and eventually lead to a good decision.

Solutions to consider

For the rest of this article we will share some insights into common solutions used in the market today, and briefly touch upon some of their pros and cons.

The presented solution should be viewed in light of the architecture below, assuming the protocols will sit above TCP/IP. The architecture consists of devices deployed in the field, some public entry points, frontend services and gateways that eventually communicate with a set of backend services. The backend services follow a microservices principle.



Devices: Out in the field and in public networks. There will be devices that are connected to the public network. These devices will run an agent that can execute various tasks and services based on input from the user.

Management entry points: The devices might be manageable or accessible by third party vendors through entry points (for instance web browser, CLI, API, etc.)

Front-end services and gateways: Various gateways can filter traffic from devices and the entry points on public networks to the appropriate backend services that run on a private network. Also in this tier, there can be various services exposed that are needed for your application (serving js-files, etc.)

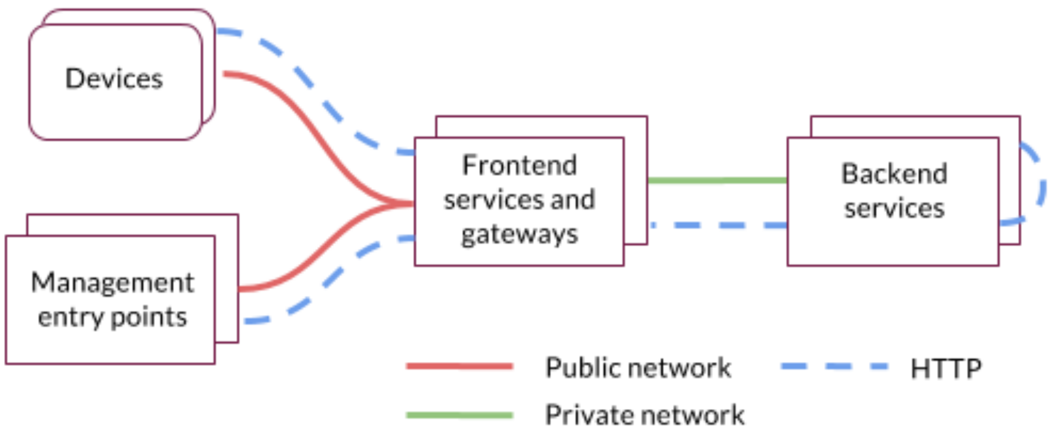
Backend services: The business logic runs on the on the backend, where there can also be various operational services like logging, notification services, and application specific services, all of which run on a private network and might need to speak to one other internally. These services are architected as microservices leading to lots of inter service traffic.

Some protocols to be aware of are HTTP, MQTT, AMQP, CoAP, OMA LWM2M, Thread, NATS, Kafka and XMPP. In this article, we will restrict ourselves to solutions built on HTTP, MQTT and AMQP.

Solution 1: HTTP/REST only

In this model all traffic, independent of sender and receiver and on public or private network, is using HTTP.

HTTP is probably the most common and widely used protocol on the Internet today, and the foundation for www. HTTP also has strong adoption within projects running higher-end Linux devices. As the market becomes more and more standardized (moving away from the legacy read-only embedded/M2M world filled with exotic and proprietary solutions), HTTP appears to be a good choice when development speed and compatibility ranks high.



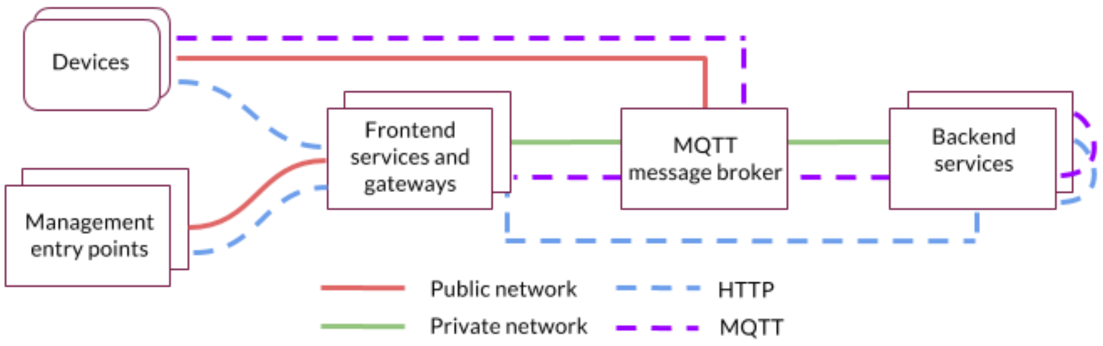
The latest and second major version of HTTP, version HTTP/2 was published as RFC 7540 standard in 2015. An important aspect for embedded devices is that the standard includes more efficient bandwidth utilization by supporting header compression, server push, and multiplexing. The protocol can easily be encrypted using, for instance, TLS.

REST is a popular architectural style that uses simple HTTP verbs to send and receive data and instructions. Most popular websites today operate in this manner, so HTTP is an extensively tested protocol with great library support. In reliable, bigger capacity networks with more powerful devices, HTTP/REST can be a simple and safe solution. If your application consists of multiple busy services, this model might cause stress on your environment as each service needs to post a new poll every time they are doing a transaction/executing to ensure they operate on the latest data. In this case, it might make sense to combine or replace HTTP with an asynchronous message queuing protocol as described in other solutions below. HTTP does not support any QoS found in other protocols.

PROS	CONS
• Open standard • Well-known protocol • Simple implementation • Easy to integrate with • Simple - one and same protocol for all communication • Works with "any" language	• Can lead to frequent polling throughout the network due to inter-service update requests • Message duplication could be required (if same message needs to be delivered to more than one service) • No native QoS

Solution 2: HTTP for frontend, and MQTT elsewhere

If HTTP is too heavyweight for your devices and/or your distributed microservice-oriented backend drives a flood of requests, there are other approaches. You could combine HTTP and a messaging protocol where services simply subscribe to the information needed and automatically gets it without stressing the network. In the following solution, devices only use the MQTT protocol. The entry points use HTTP to communicate with frontend services and gateways. Backend services use a combination of MQTT and HTTP to communicate with each other. The reason for using the asynchronous MQTT protocol would be used to ensure that all services have the latest information the need (subscribed to) at the cheapest possible cost (reducing need for frequent polling of updates). HTTP will be used for synchronous communication. As for the devices, MQTT is a more lightweight solution than HTTP and in many cases more suitable for connected devices than HTTP.



MQTT is a lightweight protocol originally developed by IBM in 1999. Its goal was to address their use case for a protocol reducing battery loss with minimal bandwidth with connected oil pipelines over satellite connection. Its latest version is 3.1.1 is ISO/IEC 20922 certified and an OASIS standard. Facebook uses MQTT for their mobile messenger application along with the IoT offering on Amazon Web Services.

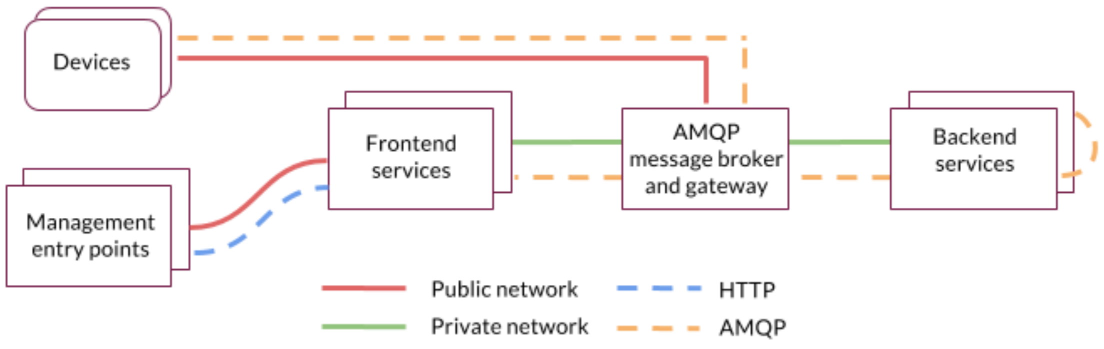
MQTT appears to be one of the more popular protocols for connected devices that require low memory and battery consumption. Implementation is rather simple and offers solid/extensive library support. It can be encrypted using TLS and offers QoS functionality making it suitable for less reliable network settings.

PROS	CONS
• Open standard • Both synchronous (HTTP) and asynchronous (MQTT) communication • Fairly easy implementation (lots of libraries and well-tested client brokers) • Native QoS (MQTT) • Optimized for more lightweight Linux devices	• Two protocols means more complexity

Solution 3: mostly AMQP and some HTTP for frontend

Another interesting model combines the messaging AMQP protocol and HTTP.

In this model, HTTP would be used for web traffic between entry points and frontend services on the public network, while backend services will only communicate using an AMQP message broker. Devices use only AMQP, as it offers synchronous communication like HTTP and there is no need for two synchronous protocols. The AMQP message broker in this case acts as gateway for all traffic between public and private network.



Advanced Message Queuing Protocol 1.0, released in 2011, is an ISO/IEC 19464 standard.

Created by John O'Hara and the security team at JPMorgan Chase in 2003, the goal was to find a messaging solution that provides durability, the capability of handling very high volumes and a high degree of interoperability. AMQP is an asynchronous (access controlled) queuing system where messages are published and subscribed to. It supports large scale setups where billions of messages can reliably (thanks to QoS) be delivered. The protocol can be integrated with TLS and is used for synchronous traffic, thus can fully replace HTTP. Although it can work nicely for your connected device project, it was originally built for inter-application server communication inside the datacenter.

AMQP is being used and supported by major organizations such as Bank of America, Goldman Sachs, Huawei, Kaazing, JPMorgan Chase, Microsoft, Red Hat, and the US Department of Homeland Security. The AMQP specification is extensive.

AMQP is lightweight and thereby more suitable for devices with less resources or power consumption limitations. The QoS functionality of AMQP fits in nicely with less reliable networks. AMQP brings great interoperability capabilities, but the depth of this protocol makes it the hardest to implement, especially if you cannot find a library/client for your needs.

PROS	CONS
• Open standards • Synchronous and asynchronous communication • AMQP broker could work as a gateway (devices could be behind the NAT) • Server and client verification through certificates (e.g. RabbitMQ) • Good libraries for Go, Java and Erlang. Well-tested client brokers • Native QoS	• Poor support for C • Not as widely used or known • More complex to implement

Summary

A first step in evaluating your options could be to socialize and gain alignment with your team on the acceptance criteria. Use the list above to build and test various solutions to evaluate which protocol will best meet your needs today and in the future.

In this article, we focused on three protocols that are compatible with higher-end devices using TCP/IP. There is no one-size fits all, but hopefully the information shared will provide more insight into your options and choices and clarify what your key criteria will be.