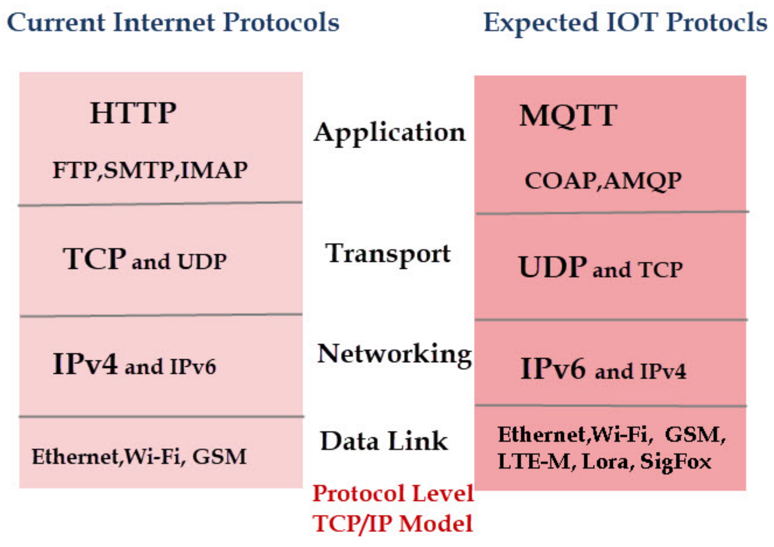
IOT Networking

The IOT will run over the existing TCP/IP network.

The existing TCP/IP networking model uses a 4 layer model with protocols def ined at each level. See understanding the TCP/IP 4 layer model.

The diagram below shows a side by side comparison of the Internet protocols currently in use and those that are likely to be used for the IOT.



IOT and Internet Protocols

Diagram Notes:

1. I have used larger font sizes to depict the protocol popularity. For example IPv4 is larger on the left as it is far more popular on the current Internet. However on the right it is smaller as IPv6 is expected to be more popular in the IOT.

2. Not all protocols are shown.

3. The areas that are showing the greatest activity are the **datalink** (levels 1and 2) and **application layer** (level 4).

4. The network and transport layers are likely to remain unaltered.

Data Link Level Protocols

At this layer you will have a requirement to connect devices that are nearby e.g. on Local networks, and that are more distant e.g. Metropolitan and wide area networks.

Current computer networks use Ethernet and Wi-Fi at this level for home/office networking (LANs) and **3G/4G** for Mobile connections (WANs).

However many IOT devices, like sensors, will be low powered (battery only).

Ethernet isn't suitable for these applications but **low powered W-fi** and **low powered Bluetooth** are.

Although existing wireless technologies (Wi-Fi, Bluetooth,3G/4G) will be used to connect these devices, other newer wireless technology, especially designed for IOT applications, will also need to be considered, and will likely grow in importance.

Among these are:

- **BLE**– Blue tooth Low Energy
- LoRaWAN –
- SigFox
- LTE-M

These are covered in more detail in An overview of IOT wireless technologies

Networking Level or Layer

The protocol that is set to dominate at the networking level in the long term is IPv6.

It is very unlikely that IPv4 will be used, but it may play a role in the initial stages.

For example, most Home IOT devices e.g smart lights currently use IPv4.

Transport Level or Layer

At the transport level **TCP** has dominated the Internet and the web. It is used by **HTTP** and many other popular Internet protocols (SMTP, POP3, IMAP4 etc).

MQTT which I expect to be one of the dominant messaging application protocols currently using **TCP** and is already deployed.

However because of the requirement for low protocol overhead I would expect **UDP** to feature much more in the future IOTs.

MQTT-SN –which runs over **UDP** is likely to see more widespread use in the future.

See TCP vs UDP for more details

Application Level and Messaging Protocols

HTTP is probably the best known protocol at this level as it it the protocol that powers the Web (**WWW**).

HTTP is also going to be important to the Internet of things as it is used for **REST APIs** which are becoming the main mechanism for Web Applications and services to communicate.

However because of the high protocol overhead HTTP is not likely to be a major IOT protocol, but will still enjoy widespread usage on the Internet. See MQTT vs HTTP for IOT.

Application Layer Messaging Protocols

What is a messaging protocol ?– A message protocol defines the rules, formats and functions for transferring messages between the components of a messaging system. –PC Magazine Encyclopedia

IMO Machine to Machine (M2M) Messaging will probably become the Email of the IOT world.

Common IOT Messaging Protocols

There are several messaging protocols currently in use. Many where designed and developed before the IOTs became a buzzword.

However some messaging protocols e.g. **COAP** have been developed for the IOTs.

Which of these protocols will dominate is unsure, but because of the very many different requirements of Internet connected devices there will be more than one.

Important Characteristics for IOT Protocols

- Speed – Amount of data that can be transferred/second
- Latency – amount of time a message takes to be transferred
- Power consumption
- Security
- Availability of software stacks.

Protocols

- MQTT– (**Message Queuing Telemetry Transport**) Uses TCP/IP. Publish subscribe model (P/S)requires a message broker (switch). See an Introduction to MQTT for beginners
- AMQP -(**Advanced Message Queuing Protocol**) Uses TCP/IP. Publish subscribe model and Point to Point .
- COAP-(**Constrained Application Protocol**) -Uses UDP designed specifically for IOT uses request response model like HTTP. RFC 7252
- DDS– (**Data Distribution Service) –**

This article covers the main protocols and their main uses.

The conclusion of this articles seems to be that the IOT will consist of a collection of protocols depending on their intended use.

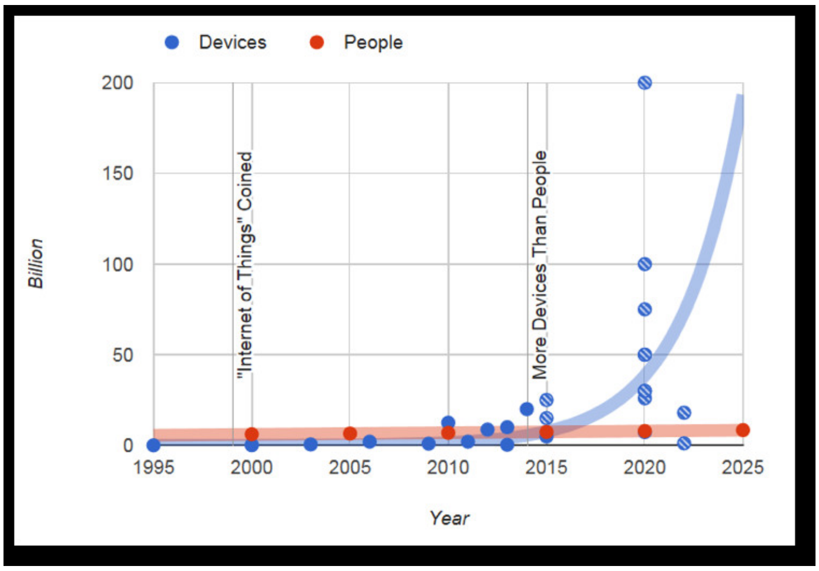
However if you look back to the early days of the Internet **HTTP** was just one of many protocols, but it has become the **dominant one**.

Even though **HTPP** wasn’t designed for file transfer it is used for file transfer. It wasn’t designed for email either, but it is used for email.

IMO IOT messaging protocols will follow a similar pattern with most of the services using 1 dominant protocol.

Messaging protocols for “lightweight” IoT nodes

A fascinating article from Philip N. Howard at George Washington University asserts that based on multiple sources, the number of connected devices surpassed the number of people on the planet in 2014. Further, it estimates that by 2020 we will be approaching 50 billion devices on the Internet of Things (IoT).



Philip N. Howard’s Study of Connected Devices

In other words, while humans will continue to connect their devices to the web in greater numbers, a bigger explosion will come from “things” connecting to the web that weren’t before, or which didn’t exist, or which now use their connection as more of a core feature.

The question is, how will these billions of things communicate between the end node, the cloud, and the service provider?

This article dives into that subject as it relates to a particular class of devices that are very low cost, battery-powered, and which must operate at least seven years without any manual intervention.

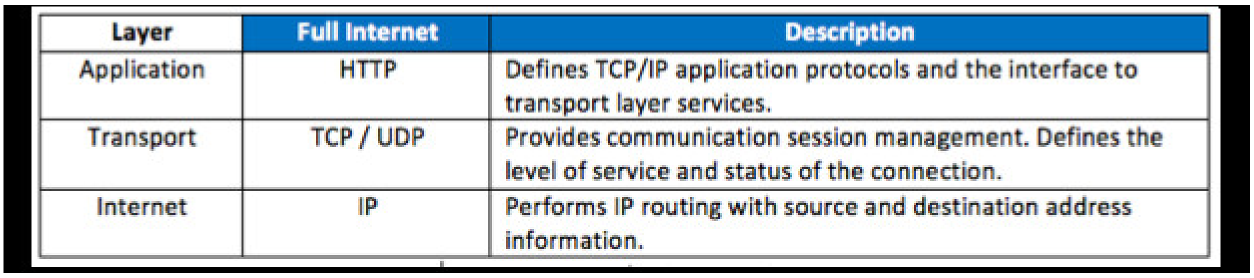
In particular, it looks at two emerging messaging protocols to address the needs of these “lightweight” IoT nodes. The first, MQTT, is very old by today’s standards from way back in 1999. And the second, CoAP, is relatively new but gaining traction.

IoT Communication Protocol Requirements

One definition of IoT is connecting devices to the internet that were not previously connected. A factory owner may connect high-powered lights. A triathlete may connect a battery-powered heartrate monitor. A home or building automation provider may connect a wireless sensor with no line power source.

But the important thing here is that in all the above cases the “Thing” must communicate through the Internet to be considered an “IoT” node.

Since it must use the Internet, it must also adhere to the Internet Engineering Task Force’s (IETF) Internet Protocol Suite. However, the Internet has historically connected resource-rich devices with lots of power, memory and connection options. As such, its protocols have been considered too heavy to apply wholesale for applications in the emerging IoT.



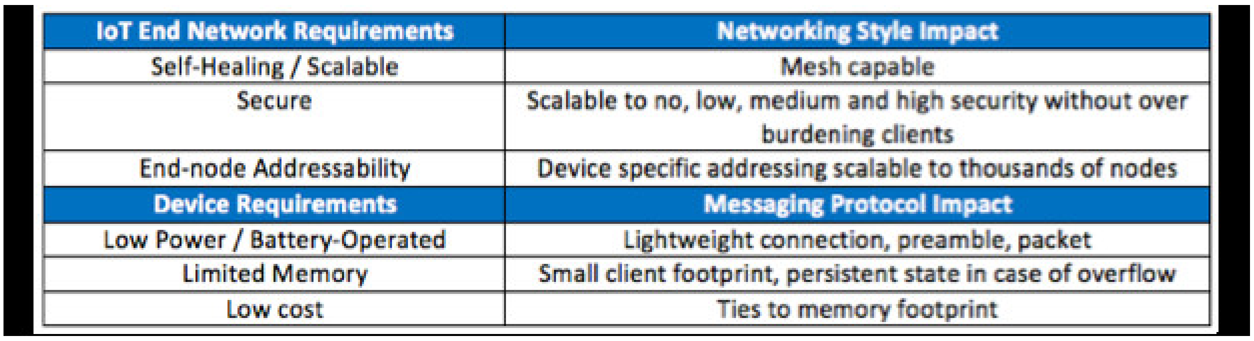
Layer	Full Internet	Description
Application	HTTP	Defines TCP/IP application protocols and the interface to transport layer services.
Transport	TCP / UDP	Provides communication session management. Defines the level of service and status of the connection.
Internet	IP	Performs IP routing with source and destination address information.

Internet Protocol Suite Overview

There are other aspects of the IoT which also drive modifications to IETF’s work. In particular, networks of IoT end nodes will be lossy, and the devices attached to them will be very low power, saddled with constrained resources, and expected to live for years.

The requirements for both the network and its end devices might look like the table below. This new model needs new, lighter weight protocols that don’t require the large amount of resources.

MQTT and CoAP address these needs through small message sizes, message management, and lightweight message overhead. We look at each below.

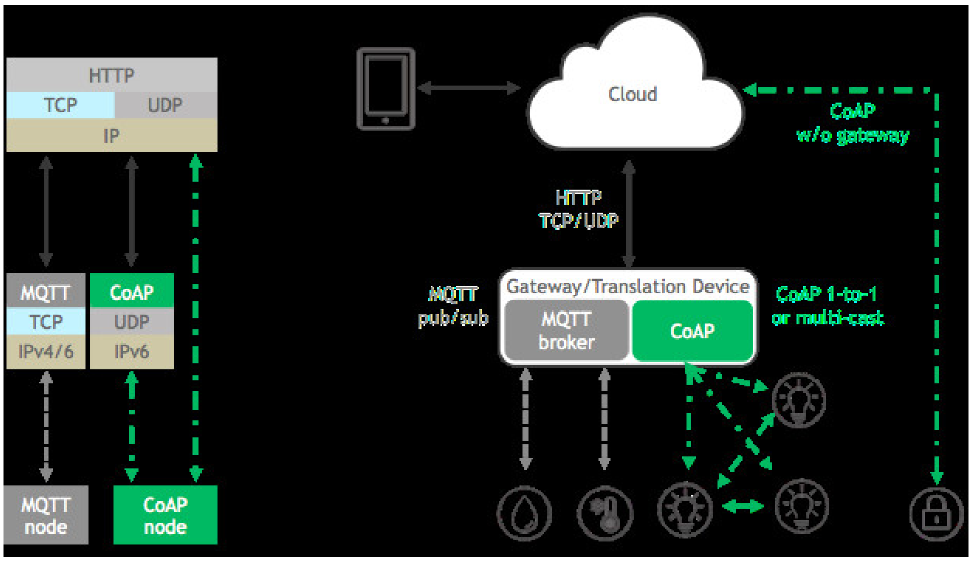


IoT End Network Requirements	Networking Style Impact
Self-Healing / Scalable	Mesh capable
Secure	Scalable to no, low, medium and high security without over burdening clients
End-node Addressability	Device specific addressing scalable to thousands of nodes
Device Requirements	**Messaging Protocol Impact**
Low Power / Battery-Operated	Lightweight connection, preamble, packet
Limited Memory	Small client footprint, persistent state in case of overflow
Low cost	Ties to memory footprint

Requirements for low-cost, power-constrained devices and associated networks

MQTT and CoAP: Lightweight IoT Communications Protocols

MQTT and CoAP allow for communication from Internet-based resource-rich devices to IoT-based resource-constrained devices. Both CoAP and MQTT implement a lightweight application layer, leaving much of the error correction to message retries, simple reliability strategies, or reliance on more resource rich devices for post-processing of raw end-node data.



Conceptual Diagram of MQTT and CoAP Communication to Cloud / Phone

MQTT Overview

IBM invented Message Queuing Telemetry Transport (MQTT) for satellite communications with oil field equipment. It had reliability and low power at its core and so made good sense to be applied to IoT networks.

The MQTT standard has since been adopted by the OASIS open standards society and released as version 3.1.1. It is also supported within the Eclipse community, as well as by many commercial companies who offer open source stacks and consulting.

MQTT uses a “publish/subscribe” model, and requires a central MQTT broker to manage and route messages among an MQTT network’s nodes. Eclipse describes MQTT as “a many-to-many communication protocol for passing messages between multiple clients through a central broker.”

MQTT uses TCP for its transport layer, which is characterized as “reliable, ordered and error-checked.”

MQTT Strengths

Publish / Subscribe Model

MQTT’s “pub/sub” model scales well and can be power efficient. Brokers and nodes publish information and others subscribe according to the message content, type, or subject. (These are MQTT standard terms.) Generally the broker subscribes to all messages and then manages information flow to its nodes.

There are several specific benefits to the Pub/Sub model.

Space decoupling

While the node and the broker need to have each other’s IP address, nodes can publish information and subscribe to other nodes’ published information without any knowledge of each other since everything goes through the central broker. This reduces overhead that can accompany TCP sessions and ports, and allows the end nodes to operate independently of one another.

Time decoupling

A node can publish its information regardless of other nodes’ states. Other nodes can then receive the published information from the broker when they are active. This allows nodes to remain in sleepy states even when other nodes are publishing messages directly relevant to them.

Synchronization decoupling

A node that in the midst of one operation is not interrupted to receive a published message to which it is subscribed. The message is queued by the broker until the receiving node is finished with its existing operation. This saves operating current and reduces repeated operations by avoiding interruptions of on-going operations or sleepy states.

Security

MQTT uses unencrypted TCP and is not “out-of-the-box” secure. But because it uses TCP it can – and should – use TLS/SSL internet security. TLS is a very secure method for encrypting traffic but is also resource intensive for lightweight clients due to its required handshake and increased packet overhead. For networks where energy is a very high priority and security much less so, encrypting just the packet payload may suffice.

MQTT Quality of Service (QoS) levels

The term “QoS” means other things outside of MQTT. In MQTT, “QoS” levels 0, 1 and 2 describe increasing levels of guaranteed message delivery.

MQTT QoS Level 0 (At most once)

This is commonly known as “Fire and forget” and is a single transmit burst with no guarantee of message arrival. This might be used for highly repetitive message types or non-mission critical messages.

MQTT QoS Level 1 (At least once)

This attempts to guarantee a message is received at least once by the intended recipient. Once a published messaged is received and understood by the intended recipient, it acknowledges the message with an acknowledgement message (PUBACK) addressed to the publishing node. Until the PUBACK is received by the publisher, it stores the message and retransmits it periodically. This type of message may be useful for a non-critical node shutdown.

MQTT QoS Level 2 (Exactly once)

This level attempts to guarantee the message is received and decoded by the intended recipient. This is the most secure and reliable MQTT level of QoS. The publisher sends a message announcing it has a QoS level 2 message. Its intended recipient gathers the announcement, decodes it and indicates that it is ready to receive the message. The publisher relays its message. Once the recipient understands the message, it completes the transaction with an acknowledgement. This type of message may be useful for turning on or off lights or alarms in a home.

Last Will and Testament

MQTT provides a "last will and testament (LWT)" message that can be stored in the MQTT broker in case a node is unexpectedly disconnected from the network. This LWT retains the node's state and purpose, including the types of commands it published and its subscriptions. If the node disappears, the broker notifies all subscribers of the node's LWT. And if the node returns, the broker notifies it of its prior state. This feature accommodates lossy networks and scalability nicely.

Flexible topic subscriptions

An MQTT node may subscribe to all messages within a given functionality. For example a kitchen "oven node" may subscribe to all messages for "kitchen/oven/+", with the "+" as a wildcard. This allows for a minimal amount of code (i.e., memory and cost). Another example is if a node in the kitchen is interested in all temperature information regardless of the end node's functionality. In this case, "kitchen/+/temp" will collect any message in the kitchen from any node reporting "temp". There are other equally useful MQTT wildcards for reducing code footprint and therefore memory size and cost.

Issues with MQTT

Central Broker

The use of a central broker can be a drawback for distributed IoT systems. For example, a system may start small with a remote control and window shade, thus requiring no central broker. Then as the system grows, for example adding security sensors, light bulbs, or other window shades, the network naturally grows and expands and may have need of a central broker. However, none of the individual nodes wants to take on the cost and responsibility as it requires resources, software and complexity not core to the end-node function.

In systems that already have a central broker, it can become a single point of failure for the complete network. For example, if the broker is a powered node without a battery back-up, then battery-powered nodes may continue operating during an electrical outage while the broker is off-line, thus rendering the network inoperable.

TCP

TCP was originally designed for devices with more memory and processing resources than may be available in a lightweight IoT-style network. For example, the TCP protocol requires that connections be established in a multi-step handshake process before any messages are exchanged. This drives up wake-up and communication times, and reduces battery life over the long run.

Also in TCP it is ideal for two communicating nodes to hold their TCP sockets open for each other continuously with a persistent session, which again may be difficult with energy- and resource-constrained devices.

Wake-up time

Again, using TCP without session persistence can require incremental transmit time for connection establishment. For nodes with periodic, repetitive traffic, this can lead to lower operating life.

CoAP Overview

With the growing importance of the IoT, the Internet Engineering Task Force (IETF) took on lightweight messaging and defined the Constrained Application Protocol (CoAP). As defined by the IETF, CoAP is for “use with constrained nodes and constrained (e.g., low-power, lossy) networks.” The Eclipse community also supports CoAP as an open standard, and like MQTT, CoAP is commercially supported and growing rapidly with IoT providers.

CoAP is a client/server protocol and provides a one-to-one “request/report” interaction model with accommodations for multi-cast, although multi-cast is still in early stages of IETF standardization. Unlike MQTT, which has been adapted to IoT needs from a decades-old protocol, the IETF specified CoAP from the outset to support IoT with lightweight messaging for constrained devices operating in a constrained environment. CoAP is designed to interoperate with HTTP and the RESTful web through simple proxies, making it natively compatible to the Internet.

Strengths of CoAP

Native UDP

CoAP runs over UDP which is inherently and intentionally less reliable than TCP, depending on repetitive messaging for reliability instead of consistent connections. For example, a temperature sensor may send an update every few seconds even though nothing has changed from one transmission to the next. If a receiving node misses one update, the next will arrive in a few seconds and is likely not much different than the first.

UDP’s connectionless datagrams also allow for faster wake-up and transmit cycles as well as smaller packets with less overhead. This allows devices to remain in a sleepy state for longer periods of time conserving battery power.

Multi-cast Support

A CoAP network is inherently one-to-one; however it allows for one-to-many or many-to-many multi-cast requirements. This is inherent in CoAP because it is built on top of IPv6 which allows for multicast addressing for devices in addition to their normal IPv6 addresses. Note that multicast message delivery to sleeping devices is unreliable or can impact the battery life of the device if it must wake regularly to receive these messages.

Security

CoAP uses DTLS on top of its UDP transport protocol. Like TCP, UDP is unencrypted but can be – and should be – augmented with DTLS.

Resource / Service Discovery

CoAP uses URI to provide a standard presentation and interaction expectations for network nodes. This allows a degree of autonomy in the message packets since the target node's capabilities are partly understood by its URI details. In other words, a battery-powered sensor node may have one type of URI while a line-powered flow control actuator may have another. Nodes communicating to the battery-powered sensor node might be programmed to expect longer response times, more repetitive information, and limited message types. Nodes communicating to the line-powered flow control actuator might be programmed to expect rich, detailed messages, very rapidly.

Asynchronous Communication

Within the CoAP protocol, most messages are sent and received using the request/report model; however, there are other modes of operation that allow nodes to be somewhat decoupled. For example, CoAP has a simplified "observe" mechanism similar to MQTT's pub/sub that allows nodes to observe others without actively engaging them.

As an example of the "observe" mode, node 1 can observe node 2 for specific transmission types, then any time node 2 publishes a relevant message, node 1 receives it when it awakens and queries another node. It's important to note that one of the network nodes must hold messages for observers. This is similar to MQTT's broker model except that there is no broker requirement in CoAP, and therefore no expectation of being able to hold or queue messages for observers.

There are currently draft additions to the standard which may provide a similar CoAP function to MQTT's pub/sub model over the short-to-medium term. The leading candidate today is a draft proposal from Michael Koster, allowing CoAP networks to implement a pub/sub model like MQTT's mentioned above.

Issues with CoAP

Standard Maturity

MQTT is currently a more mature and stable standard than CoAP. It's been Silicon Labs' experience that it is easier to get an MQTT network up and running very quickly than a similar one using CoAP. That said, CoAP has tremendous market momentum and is rapidly evolving to provide a standardized foundation with important add-ons in the ratification pipeline now.

It is likely that CoAP will reach a similar level of stability and maturity as MQTT in the very near term. But the standard is evolving for now, which may present some troubles with interoperability.

Message Reliability (QoS level)

CoAP's "reliability" is MQTT's QoS and provides a very simple method of providing a "confirmable" message and a "non-confirmable" message. The confirmable message is acknowledged with an acknowledgement message (ACK) from the intended recipient. This confirms the message is received but stops short of confirming that its contents were decoded correctly or at all. A non-confirmable message is "fire and forget."

Real-time protocols for IoT apps

Real-time communication technology is an absolute requirement for the development of Internet of things (IoT) applications. Imagine the use case where your phone communicates with your lights. If it takes several seconds before your lights turn on, that's a failed user experience.

The development of real-time communication technologies is a story that can't be explained without mentioning instant messaging. Historically speaking, instant messengers were the original consumer-friendly, Internet-connected real-time communication clients. AOL IM, ICQ, and Jabber are a few examples of instant messenger clients that supported real-time communication. This all happened in the 1990s.

[Learn how to unlock the power of the Internet of things analytics with big data tools in InfoWorld's downloadable Deep Dive. | Explore the current trends and solutions in BI with InfoWorld's Extreme Analytics blog.]

Today, as we move toward developing protocols for communication between IoT devices, we look to lessons learned from building instant messaging solutions. Three major real-time protocols are used by IoT devices today: XMPP, CoAP, and MQTT. Interestingly enough, XMPP started life as Jabber, an open instant messenger protocol.

XMPP

The eXtensible Messaging and Presence Protocol (XMPP) is a TCP communications protocol based on XML that enables near-real-time exchange of structured data between two or more connected entities. Out-of-the-box features of XMPP include presence information and contact list maintenance. While both features were originally designed for instant messaging, they have obvious applications for IoT. Due in part to its open nature and XML foundation, XMPP has been extended for use in publish-subscribe systems -- again, perfect for IoT applications.

There are several major advantages to using XMPP as your IoT communications protocol. The primary advantage is XMPP's decentralized nature. XMPP works similar to email, operating across a distributed network of transfer agents rather than relying on a single, central server or broker (as CoAP and MQTT do). As with email, it's easy for anyone to run their own XMPP server, allowing device manufacturers and API operators to create and manage their own network of devices. And because anyone can run their own server, if security is required, that server could be isolated on a company intranet behind secure authentication protocols using built-in TLS encryption.

Unfortunately, there are a few disadvantages to XMPP as well. One of the largest flaws is the lack of end-to-end encryption. While there are many use cases in which encryption may not yet be necessary, most IoT devices will ultimately need it. The lack of end-to-end encryption is a major downside for IoT manufacturers.

Another downside is the lack of Quality of Service (QoS). Making sure that messages are delivered is even more important in the IoT world than it was in the instant messaging world. If your alarm system doesn't receive the message to turn itself on, then that vacation you've been planning could easily be ruined.

CoAP

The Constrained Application Protocol (CoAP) was specifically developed to allow resource-constrained devices to communicate over the Internet using UDP instead of TCP. Developers can interact with any CoAP-enabled device the same way they would with a device using a traditional REST-based API. CoAP is particularly useful for communicating with low-power sensors and devices that need to be controlled via the Internet.

CoAP is a simple request/response protocol (again, very similar to REST) that follows a traditional client/server model. Clients can make GET, PUT, POST, and DELETE requests to resources. CoAP packets use bitfields to maximize memory efficiency, and they make extensive usage of mappings from strings to integers to keep the data packets small enough to transport and interpret on-device. Aside from the extremely small packet size, another major advantage of CoAP is its usage of UDP; using datagrams allows for CoAP to be run on top of packet-based technologies like SMS.

All CoAP messages can be marked as either "confirmable" or "nonconfirmable," serving as an application-level QoS. While SSL/TLS encryption isn't available over UDP, CoAP makes use of Datagram Transport Layer Security (DTLS), which is analogous to the TCP version of TLS. The default level of encryption is equivalent to a 3,072-bit RSA key. Even with all of this, CoAP is designed to work on microcontrollers with as little as 10KB of RAM.

One of the downsides of CoAP: It's a one-to-one protocol. Though extensions that make group broadcasts possible are available, broadcast capabilities are not inherent to the protocol. Arguably, an even more important disadvantage is the lack of a publish-subscribe message queue.

MQTT

Message Queue Telemetry Transport (MQTT) is a publish-subscribe messaging protocol. Similar to CoAP, it was built with resource-constrained devices in mind. MQTT has a lightweight packet structure designed to conserve both memory usage and power. A connected device subscribes to a topic hosted on an MQTT broker. Every time another device or service publishes data to a topic, all of the devices subscribed to it will automatically get the updated information.

The major advantages of MQTT are the publish-subscribe message queue and the many-to-many broadcast capabilities. Using a long-lived outgoing TCP connection to the MQTT broker, sending messages of limited bandwidth back and forth is simple and straightforward.

The downside of having an always-on connection is that it limits the amount of time the devices can be put to sleep. If the device mostly sleeps, then another MQTT protocol can be used: MQTT-S, which works with UDP instead of TCP.

Another disadvantage of MQTT is the lack of encryption in the base protocol. MQTT was designed to be a lightweight protocol, and incorporating encryption would add a significant amount of overhead to the connection. You can add custom security at the application level, but that may require a significant amount of work.

IoT ABCs

Understanding the underlying protocols will be important for every developer who wants to take full advantage of IoT technology. As the space heats up, there will be more questions about what is required to create efficient communication between devices. More important, for those creating APIs that not only interact with devices but also control them, real-time communications will be an absolute necessity. What started out as a way for people to communicate instantaneously via the Internet has turned into something much more versatile.

As developers, it is our responsibility to explore all of the options. Understanding the protocols and their trade-offs will be key to building a great product. But you should also be aware of options for adding real-time communications to your technology stack that will accelerate your development process. There are a few back-end-as-a-service platforms available that provide real-time communication out of the box. In many cases, using Firebase, Socket.io, or Built.io to build out your IoT platform may save you countless hours of development time. But for those times when we can move faster and still make use of the right technology stack, we should explore all of the options available.

Kurt Collins (@timesnyc) is Director of Evangelism at Built.io. Kurt began his engineering career at Silicon Graphics (SGI), where he worked on debugging its Unix-flavored operating system. Two years ago, Kurt co-founded The Hidden Genius Project along with eight other individuals in response to the urgent need to increase the number of black men in the technology industry.

Mentoring young people interested in the technology industry is a personal passion of his. Today, he continues this work with tech startups and nonprofits alike.