Internet of Things Protocol Comparison

Approved – 21 Jan 2020

Open Mobile Alliance

OMA-WP-Protocol_Comparison-V1_0-20200121-A

Internet of Things Protocol Comparison

Editors:

Tim Spets, Senior Standards Architect, Greenwave Systems

Hannes Tschofenig, Senior Principal Engineer, Arm Limited

Contributors:

Christian Legare, SiLabs

Michel Kohanim, Universal Devices

James Milne, Universal Devices

Bill Silverajan, Tampere University of Technology

Romedius Weiss, Innsbruck University/Arm Limited

Thomas Hardjono, Massachusetts Institute of Technology

Hasan Derhamy, Lulea University of Technology

David Decker, Landis+Gyr

Mark Baugher, Consultant

Ari Keränen, Ericsson

Matthew Gillmore, Itron

January 2020

Contents

1. REFERENCES 5
2. INTRODUCTION 6
3. SUMMARY 7
3.1 HANDSHAKE REQUIREMENTS/INTERACTION PATTERNS 7
3.2 SESSION BASED, PUBLISH-SUBSCRIBE VS. REQUEST-RESPONSE 7
3.3 SECURITY 7
3.4 HEADER PAYLOAD 8
3.5 MATURITY 8
3.6 QUIC PROTOCOL 13
3.7 CONCLUSION 13
4. HIGH-LEVEL DESCRIPTION OF PROTOCOLS CONSIDERED 15
4.1 HTTP 15
4.2 WEBSOCKET 15
4.3 COAP 16
4.4 XMPP 17
4.5 MQTT 19
4.5.1 MQTT-SN 19
5. SECURITY 21
APPENDIX A. CHANGE HISTORY (INFORMATIVE) 22

Figures

Figure 1 : HTTP GET message sent from the client to the server 15
Figure 2 : WebSocket - All traffic between the client and server 16
Figure 3 : Exchange with a Confirmable (CON) message (GET) being sent and an Acknowledgement message (ACK) in reply 17
Figure 4 : XMPP bi-directional communication between clients via a central server 18
Figure 5 : MQTT bi-directional communication between clients via the central server 19
Figure 6 : Translation between MQTT and MQTT-SN 20
Figure 7 : Transport Layer Security (TLS) over TCP or Datagram Transport Layer Security (DTLS) over UDP 21

Tables

Table 1 - Protocol Category 6
Table 2 - Protocol Comparison Table 12

1. References

[RFC5246] T. Dierks, E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, August 2008, URL: https://www.rfc-editor.org/info/rfc5246.

[RFC6120] Saint-Andre, P., "Extensible Messaging and Presence Protocol (XMPP): Core", RFC 6120, March 2011, URL: http://www.rfc-editor.org/info/rfc6120.

[RFC6347] E. Rescorla, N,. Modadugu "Datagram Transport Layer Security Version 1.2", RFC 6347, January 2012, URL: https://www.rfc-editor.org/info/rfc6347.

[RFC6455] Fette, I. and A. Melnikov, "The WebSocket Protocol", RFC 6455, December 2011. URL: http://www.rfc-editor.org/info/rfc6455.

[RFC7230] Fielding, R., and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, June 2014, URL: http://www.rfc-editor.org/info/rfc7230.

[RFC7231] Fielding, R., and J. Reschke, "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, June 2014, URL:http://www.rfc-editor.org/info/rfc7231.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, June 2014, URL: http://www.rfc-editor.org/info/rfc7252.

[RFC7390] A. Rahman, E. Dijk, "Group Communication for the Constrained Application Protocol (CoAP)", RFC 7390, October 2014. URL: https://tools.ietf.org/html/rfc7390.

[RFC7540] Belshe, M., Belshe, M., and M. Thomson, "Hypertext Transfer Protocol Version 2 (HTTP/2)", RFC 7540, May 2015. URL: http://www.rfc-editor.org/info/rfc7540.

[RFC7541] R. Peon, et al., "HPACK: Header Compression for HTTP/2", RFC 7541, May 2015, URL: https://tools.ietf.org/html/rfc7541.

[RFC7641] Kartke, K., "Observing Resources in the Constrained Application Protocol (CoAP)", RFC 7641, September 2015, URL: http://www.rfc-editor.org/info/rfc7641.

[RFC768] J. Postel ISI "User Datagram Protocol", 1980, URL: https://www.rfc-editor.org/info/rfc768.

[RFC7925] H. Tschofenig, et al., "Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things", RFC 7925, July 2016, URL: https://tools.ietf.org/html/rfc7925.

[RFC793] Information Sciences Institute University of Southern California "Transmission Control Protocol", RFC 793, September 1981, URL: https://www.rfc-editor.org/info/rfc793.

[RFC7959] Bormann, C. and Z. Shelby, "Block-Wise Transfers in the Constrained Application Protocol (CoAP)", RFC 7959, August 2016, URL: http://www.rfc-editor.org/info/rfc7959.

[RFC8095] G. Fairhurst, et al., "Services Provided by IETF Transport Protocols and Congestion Control Mechanisms", RFC 8095, March 2017, URL: https://tools.ietf.org/html/rfc8095.

[RFC8132] P. van der Stok, et al., "PATCH and FETCH Methods for the Constrained Application Protocol (CoAP)", RFC 8132, April 2017, URL: https://tools.ietf.org/html/rfc8132.

[RFC8323] C. Bormann, et al., "CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets", RFC8323), February 2018, URL: https://tools.ietf.org/html/rfc8323.

[I-D.ietf-core-coap-pubsub] M. Koster, et al., "Publish-Subscribe Broker for the Constrained Application Protocol (CoAP)", draft-ietf-core-coap-pubsub-05 (work in progress), July 2018, URL: https://tools.ietf.org/html/draft-ietf-core-coap-pubsub-05.

[MQTT] Banks, A., and R. Gupta, "OASIS Standard MQTT Version 3.1.1 Plus Errata 01", 2015, URL: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.

[MQTT-SN] Andy Stanford-Clark and Hong Linh Truong, “MQTT For Sensor Networks (MQTT-SN) “URL: http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf.

[QUIC] J. Iyengar et al., "QUIC: A UDP-Based Multiplexed and Secure Transport" (work in progress), August 2018. URL: https://tools.ietf.org/html/draft-ietf-quic-transport-14.

2. Introduction

Many standards-developing organizations have contributed various technological building blocks to make IoT deployments more robust and secure. A popular debate among technologists working on IoT deployments is about the best choice of protocols for getting data from and to IoT devices. In this article, members of the IPSO Working Group considered at six standardized protocols (HTTP, HTTP/2, WebSockets, XMPP, MQTT, CoAP), and refer to them as "transfer protocols." This is not a comprehensive list of protocols in use in IoT, but it represents an example of each of the different constructs, reliable, unreliable, REST, publish/subscribe, chat, point to point, client/server, extended services etc.

This whitepaper compares the differences between these six transfer protocols as used with IoT devices. The purpose is to provide technical and product personnel a way to assess the impact of each protocol and what they provide with regard to their IoT products. IoT products will likely require a suite of standard protocols to support the many different configurations and requirements of the systems/services that they are deployed in. Eight functional areas that represent the needs of IoT deployments are provided for a more concise comparison between the transfer protocols:

- Handshake requirements: What are the communication basics and the overall architecture? Does the transfer protocol require a reliable transport?
- For session-based transfer protocols: How is communication interaction achieved? Is it a request/response, publish/subscribe, or peer-to-peer pattern?
- Native security support
- Header/payload structure: What is the per-message overhead? What options are there?
- TCP/UDP support
- Maturity
- Target application space
- Additional features

This whitepaper also provides a high-level overview of each transfer protocol and offers background for the more detailed comparison.

Complete systems typically employ technologies in a hierarchical manner, often organized in the form of a protocol stack, as shown in the table below. This whitepaper focuses only on transfer protocols.

Protocol Category	Examples
Frameworks Use transfer protocols to connect endpoints, define common messaging and data model to support IoT communications.	Lightweight M2M, TR-069, OCF
Transfer Protocols IP-based protocols used to transfer application data.	HTTP, HTTP/2, WebSockets, XMPP, MQTT, CoAP
Transport Protocols Provide end-to-end service to an application by the transport (see also RFC 8095).	Reliable transport (such as TCP), unreliable transport (such as UDP), and pseudo-transports offering security features (such as TLS and DTLS).
Physical and Data Link Layer 2 Protocols Provide the physical and data link layer functionality, as defined by the ISO OSI model.	Ethernet, Wi-Fi

Table 1 - Protocol Category

3. Summary

3.1 Handshake requirements/interaction patterns

Point to point protocols, including HTTP, HTTP/2, WebSockets and CoAP, require a direct IP connectivity between endpoints. Initiation of the connection requires the endpoint to provide a server (listening on a specific TCP or UDP port). HTTP was built for Internet communication and is the basis for REST-based communication. There are many implementations and well-known resources for HTTP.

When deploying devices in environments where there is no Internet connection (typical with a home network running on a Local Area Network behind a firewall) HTTP, WebSockets, and CoAP can be used to connect the device to the user or to other devices, such as gateways, that provide access beyond the LAN.

When deploying devices that have constrained resources (CPU, RAM, network bandwidth), CoAP provides a smaller footprint than HTTP/1.1. There are two main reasons for the smaller footprint: first, a CoAP implementation is more lightweight thanks to its use of UDP, since many of the complex features of TCP (such as the sophisticated congestion control mechanism) are not needed. Second, HTTP provides additional features that are often not required on low-end IoT devices. Furthermore, CoAP was designed to enable simple implementations and low resource consumption both in the network and endpoints.

A major drawback to HTTP/1.1 for IoT is the header format, which is verbose and not optimized for constrained environments.

HTTP/2 improves on HTTP by adding bi-directional communication and push messages. HTTP/2 uses TCP and TLS; additionally a new header compression technique (called HPACK, see RFC 7541) has been introduced. WebSockets improve on HTTP/1.1 by providing a mechanism to keep the connections open and exchanging arbitrary data between endpoints. CoAP provides similar REST services as HTTP, but with a reduced footprint.

For group communication, XMPP and MQTT provide a single connection to a server/broker where all endpoints can communicate with each other using an application-layer addressing style. XMPP uses a JIDs (Jabber IDs) and MQTT uses topics to convey messages between endpoints.

MQTT provides a lighter client implementation than XMPP, but does not provide presence notifications. XMPP is XML-based communication, and therefore more verbose than MQTT.

Publish/subscribe support is available for CoAP with [I-D.ietf-core-coap-pubsub], and CoAP may become an alternative to XMPP, and a more standardized alternative to MQTT. CoAP is more lightweight than XMPP since it inherits the efficient on-the-wire encoding of CoAP. CoAP also standardizes the resource model based on the RESTful design pattern, unlike MQTT, which leaves the structure of topics unspecified.

3.2 Session based, publish-subscribe vs. request-response

HTTP/1.1 provides unidirectional client-to-server communication, and relies on TCP. CoAP provides a similar service to HTTP/1.1, but CoAP can communicate over unreliable transport with UDP.However, CoAP can also utilize additional transports (e.g. TCP, SMS, LoRaWAN, etc.). HTTP/2 adds a server push to HTTP/1.1 providing server-to-client messaging. WebSockets provide a full-duplex communication protocol over TCP. XMPP provides peer-to-peer communication, where each endpoint may transmit messages at any time. XMPP clients utilize XMPP servers to relay request and response messages. XMPP provides a single connection to an XMPP server to communicate with any number of XMPP clients.

3.3 Security

Each analyzed transfer protocol relies on either DTLS or TLS as the underlying communication security protocol.

For point to point transfer protocols (HTTP/1.1, HTTP/2, WebSockets, and CoAP), DTLS/TLS provides end to end security.

For transfer protocols that use intermediate servers (such as XMPP and MQTT), TLS offers communication security only on each leg. This means that you may need a higher-layer security mechanism in addition to TLS when end-to-end security is needed. Furthermore, an access control mechanism may be needed for these transfer protocols to determine which client can see which communication interaction.

3.4 Header Payload

HTTP/1.1 is used for web services, and has more overhead than CoAP. WebSockets, after initial setup, has a reduced header overhead at the loss of a standardized protocol format.

XMPP has the largest header overhead to the use of XML. The MQTT header has low overhead due to its binary encoding (similar to CoAP).

3.5 Maturity

HTTP is the most mature candidate in the list of transfer protocols analyzed. HTTP/2, WebSockets and CoAP are more recent developments. XMPP has existed for over 10 years. MQTT 3.1.1 was released in 2014.

Features	HTTP/1.1	HTTP/2	WebSockets	XMPP	MQTT	CoAP
	Handshake requirements/interaction patterns (reliability)					
General	HTTP uses TCP as the underlying transport protocol. The interaction pattern is request/response with pipelining and keep-alive. Data transmission is reliable thanks to the properties of TCP. HTTP is a point to point protocol (no multicast support). HTTP uses methods (such as GET, POST, PUT, DELETE) to indicate the desired action on the identified resource. Status codes (200 OK, etc.) indicate the success or failure condition of the request.	Similar to HTTP with three major additions: HTTP/2 allows multiple concurrent exchanges on the same connection (via so-called streams). It also introduces unsolicited push of representations from servers to clients. Finally, HTTP/2 introduces compression of headers.	Uses HTTP for WebSockets negotiation and then TCP for regular data communication. WebSockets support new header fields: Upgrade, Connection:upgrade, Sec-WebSocket-Key, Sec-WebSocket-Protocol, and Sec-WebSocket-Version. Confirmation at handshake layer is through HTTP headers: Sec-WebSocket-Accept and Sec-WebSocket-Protocol The main goal of WebSockets is to keep the TCP connection open for extended durations, providing bi-directional communication.	XMPP uses a distributed client-server architecture, wherein a client must connect to a server to gain access to other endpoints in the network. Clients are then allowed to exchange XML messages (called stanzas) with other clients, which can be associated with other servers. XMPP provides a near-real-time exchange of structured, yet extensible, data between any two or more network entities. XMPP allows for asynchronous communication and does not support the REST methodology. Instead, the communication uses a stream of XML stanzas.	MQTT provides a messaging transport that is agnostic to the content of the payload. MQTT provides bi-directional communication between clients via the central server. It uses a publish/subscribe message pattern that provides one-to-many message distribution and decoupling of applications.	CoAP supports reliable and unreliable delivery of data. Reliable delivery implies that the sender marks a request as “Confirmable” and expects an answer from the receiver. If that answer does not arrive the client retransmits the request. CoAP provides two layers of communication: a reliability layer and a REST layer, which mimics HTTP with methods such as POST, PUT, GET, DELETE. PATCH and FETCH was added to CoAP in RFC 8132.
Transport	TCP	TCP	TCP	TCP	TCP UDP with MQTT-SN	UDP with RFC 7252 (TCP support available with [RFC8323], which is used to ease Firewall traversal)
Security	TLS	TLS	TLS	TLS	TLS	DTLS with RFC 7252 TLS with [RFC8323]
Guaranteed delivery?	Yes	Yes	Yes	Yes (for hop-by-hop) and No (for end-to-end)	Yes (depending on the QoS settings)	Optional (depending on the selected delivery option). With [RFC8323] guaranteed delivery is ensured.
REST supported	Yes	Yes	No	No	No	Yes
Pub/Sub supported	No	No	No	Yes	Yes	RFC 7461 is an extension for CoAP that enables CoAP clients to retrieve a representation of a resource and keep this representation updated by the server over a period of time. Draft-ietf-core-pubsub defines CoAP pub/sub broker.
Group Communication	No	No	No	Yes (using pub/sub at the application layer)	Yes (using pub/sub at the application layer)	Yes (using IP multicast).

HTTP/1.1	HTTP/2	WebSockets	XMPP	MQTT	CoAP
Session based, publish-subscribe vs request-response					
Request/Response Session semantics needs to be added by the application itself (e.g., by encoding in the URL, stored in cookies) since HTTP does not provide this functionality. HTTP does not support pub/sub, but support for it may be added at the application layer.	Semantics are the same as HTTP with exception of HTTP/2 Server Push, in which a web server sends resources to a web browser before the browser requests them. This can be useful when the server knows the client will need to have certain specific resources available to fully process the original request.	WebSockets provides a persistent TCP connection. The session is tied to Sec-WebSocket-Key and stays alive as long as the connection remains alive. The connection itself is exposed via the "onmessage" and "send" functions defined by the WebSocket interface. Both endpoints may send and receive messages (bidirectionally) after the WebSocket has been established.	The JID Domain uses a FQDN for the client to connect to the server using a TCP/TLS long-lived socket connection. XMPP provides a bi-directional stream-based communication The client authenticates using SASL, and the server binds a resource to a stream.	MQTT provides three qualities of service for message delivery: **1) At most once.** Messages are delivered with best effort, but message loss can occur. **2) At least once.** Messages are assured to arrive, but duplicates may occur. **3) Exactly once.** Messages are assured to arrive exactly once. This level could be used, for example, with billing systems where duplicate or lost messages could lead to incorrect charges being applied. MQTT is connection-oriented. MQTT requires the client to have a priori knowledge of the topics.	CoAP messages are exchanged asynchronously between CoAP endpoints. They are used to transport CoAP requests and responses For asynchronous notifications, CoAP has the ability to “observe” resources (see RFC 7641). This allows an observer to register interest in specific resource. Upon changes to the subscribed resource, the observer will be notified.
Native Security Support (Click here for OMA IPSO Security White Papers)					
HTTP relies on TLS to provide communication security. The use of application layer security on top of HTTP (for example, using OAuth) is common, but it is outside the scope of HTTP itself. TLS is the de facto security mechanism for everything on the Internet.	Similar to HTTP.	The WebSockets application layer uses Sec-WebSocket-Key, Origin, and Version for initial connection. Apart from that, security is identical to HTTP (TLS/TCP secure transport).	XMPP clients use TLS for communication security and SASL for users to login into the server. For end-to-end security (from one client to another via the XMPP server), additional security mechanisms have been used, such as “off-the-record messaging”. Messages between two clients may traverse multiple servers. XMPP servers may provide services such as rosters to provide access control to clients.	MQTT supports user names and passwords in connection requests to servers. MQTT relies on TLS communication security between the client and the server. Messages between two clients may traverse multiple servers. There is no access control mechanism like rosters available for XMPP.	CoAP uses (D)TLS to offer communication security.

HTTP/1.1	HTTP/2	WebSockets	XMPP	MQTT	CoAP
Session based, publish-subscribe vs request-response					
Extremely mature and widely used. Many commercial and open source implementations are available. Many tools available. Note: Availability for web-based applications does not always translate to embedded environment.	Many implementations exist for desktop operating systems. The support on IoT OSs is rather limited.	Relatively new, but widely supported by most browsers, servers, and application frameworks, such as Node.JS.	Originally RFC 3920 in 2004. Based on Jabber implementations. High stability.	MQTT v3.1.1 OASIS was standardized in 2014 and originally developed by IBM in 1999.	CoAP was published as an RFC in June 2014. CoAP has been selected as an IoT protocol of choice by other standardization organizations (such as the OMA, OCF, oneM2M). Many implementations are available, and an overview of some of the implementations can be found at http://coap.technology/ Many of the implementations are targeting the use in embedded environments.
HTTP	**HTTP/2**	**WebSockets**	**XMPP**	**MQTT**	**CoAP**
Header/Payload structure					
Header overhead is relatively high since the header uses human-readable plaintext rather than binary encoding. Furthermore, URI parameters need to be encoded in Base64. With pipelining and keep-alive, the TCP socket can remain open, reducing the number of TCP messages for setting up and tearing down the TCP connection.	HTTP/2 introduced binary encoding and compression of message headers (RFC 7541). This greatly reduces the message size. With the help of streams, the need for opening multiple concurrent TCP connections is reduced. This may be less applicable to most IoT scenarios, since most IoT devices require only a simple communication interaction.	WebSockets uses a special framing structure. There is an initial overhead to establish a WebSockets communication, but once established it allows bi-directional data transfer over a single TCP connection. CoAP over WebSockets (see [RFC8323]) allows a standardized protocol (namely CoAP) to be used over WebSockets, which helps to increase interoperability.	Three types of messages “stanzas” are supported, 1) Message stanza, which includes a ‘to’ attribute for the recipient. 2) Presence stanza, which includes information about network availability. 3) IQ stanza, which is a request/response mechanism that enables an entity to make a request of, and receive a response from, another entity. Due to the use of XML, the communication overhead is large.	MQTT has a small protocol overhead (the fixed-length header is just 2 bytes).	CoAP has a variable length header structure, which is at minimum 4 bytes long. CoAP uses a binary encoding for the header and the options carried in the header. The encoding of the data in the body of the message depends on the application data being exchanged. For CoAP over TCP (see [RFC8323]), the efficient encoding of the header has been maintained although slightly changed to elide two header fields and to add length field of a variable-size.
TCP/UDP Support					
TCP only	TCP only	TCP only	TCP only	TCP only	CoAP was developed for use over UDP. CoAP over TCP is defined in [RFC8323].
Original target application space					
Any type of interaction model that does not require datagram transport.	HTTP/2 is meant to replace the use of HTTP on the web and particularly for the mobile environment.	Publishing real-time events for web applications (chat, device status updates, etc.)	Instant Messaging /Chat applications	Constrained environments such as for communication in Machine to Machine (M2M) and Internet of Things (IoT) contexts where a small code footprint is required and/or network bandwidth is at a premium.	CoAP is a specialized web transfer protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks.
Additional features					

HTTP/1.1	HTTP/2	WebSockets	XMPP	MQTT	CoAP
Session based, publish-subscribe vs request-response					
			Presence: provide information about the network availability of a contact (know who is online). Subscription (pub/sub): presence information provided only to contacts that are authorized. Detailed requirements in RFC 2779.	In addition to MQTT, which relies on TCP, MQTT-SN was designed to be used by sensor networks and uses UDP as a transport protocol.	
Organization owner					
IETF	IETF	IETF	IETF	OASIS	IETF

Table 2 - Protocol Comparison Table

1. For IoT devices that are connected using low-power radio technologies, protocols with compact encoding such as CoAP are the better choice. CoAP is primarily designed for small data transmissions over UDP. It can also be used for multicast communication, which is useful for both device discovery and group communication.
2. For IoT devices that require group communication, XMPP, MQTT or CoAP with extensions are all suitable. Only CoAP can utilize IP multicast. (RFC 7390).
3. Reliable delivery is not critical for some IoT scenarios. For example, an IoT device frequently sending sensor data can transmit via an unreliable CoAP message, reducing transmission overhead. In the unlikely case of a message loss, the server will not miss one sensor reading.
4. All these protocols make use of TLS/DTLS-based communication security.
5. HTTP is widely implemented on the Web and smart phones. It can be used with IoT devices that have robust network connections, since HTTP has higher overhead than other protocols due to the ASCII encoding of the header.
6. While HTTP/2 provides header compression and other performance-enhancing features, it is still used less frequently in the embedded environment due to the lack of open source code available for use in IoT operating systems.
7. Over time, CoAP and MQTT have become more similar. Both protocols are still in active development in the IETF CoRE working group (for CoAP) and in OASIS (for MQTT).
8. More research is needed to compare the footprint (code size), RAM utilization and performance of the presented transfer protocols for IoT scenarios with actual traffic patterns.

3.6 QUIC Protocol

Although the QUIC protocol [QUIC] was not included in this comparison, there are valuable attributes worth examining in the future including:

1) Moving the congestion avoidance algorithms to the application layer (UDP transport)
2) Reducing “round-trips” on mesh networks
    a. Optimization of encryption setup
3) Ability to handle packet loss
4) Can fall back to TCP if UDP traffic is perceived to be blocked

3.7 Conclusion

Any of these transfer protocols can be a good fit for an IoT device with sufficient resources (Flash memory, RAM, power, good Internet connectivity). When resources are scarce, then some of these protocols become less suitable. A few high-level observations can be made:

1. For IoT devices that are connected using low-power radio technologies, protocols with compact encoding such as CoAP are the better choice. CoAP is primarily designed for small data transmissions over UDP. It can also be used for multicast communication, which is useful for both device discovery and group communication.
2. For IoT devices that require group communication, XMPP, MQTT or CoAP with extensions are all suitable. Only CoAP can utilize IP multicast. (RFC 7390).
3. Reliable delivery is not critical for some IoT scenarios. For example, an IoT device frequently sending sensor data can transmit via an unreliable CoAP message, reducing transmission overhead. In the unlikely case of a message loss, the server will not miss one sensor reading.
4. All these protocols make use of TLS/DTLS-based communication security.

5. HTTP is widely implemented on the Web and smart phones. It can be used with IoT devices that have robust network connections, since HTTP has higher overhead than other protocols due to the ASCII encoding of the header.
6. While HTTP/2 provides header compression and other performance-enhancing features, it is still used less frequently in the embedded environment due to the lack of open source code available for use in IoT operating systems.
7. Over time, CoAP and MQTT have become more similar. Both protocols are still in active development in the IETF CoRE working group (for CoAP) and in OASIS (for MQTT).
8. More research is needed to compare the footprint (code size), RAM utilization and performance of the presented transfer protocols for IoT scenarios with actual traffic patterns.

4. High-Level Description of Protocols Considered

4.1 HTTP

The Hypertext Transfer Protocol (HTTP) is an application protocol is the foundation of data communication for the World Wide Web and is also used in Internet of Things environments. HTTP is transferred over TCP port 80 (for plaintext) and over port 443 (for TLS protected communication). It offers stateless operation and in the original design the TCP connection was closed after every request/response. With HTTP/1.1, a keep-alive technique was added to create the illusion of a persistent connection and to re-use the same connection for multiple request/response interactions. HTTP uses so-called methods to indicate the desired action on the identified resource. These methods are GET, POST, PUT, etc., as described in RFC 7231 and RFC 5789 (for PATCH).

The diagram below shows an example with an HTTP GET message sent from the client to the server, followed by a response (200 OK) from the server with the requested data.

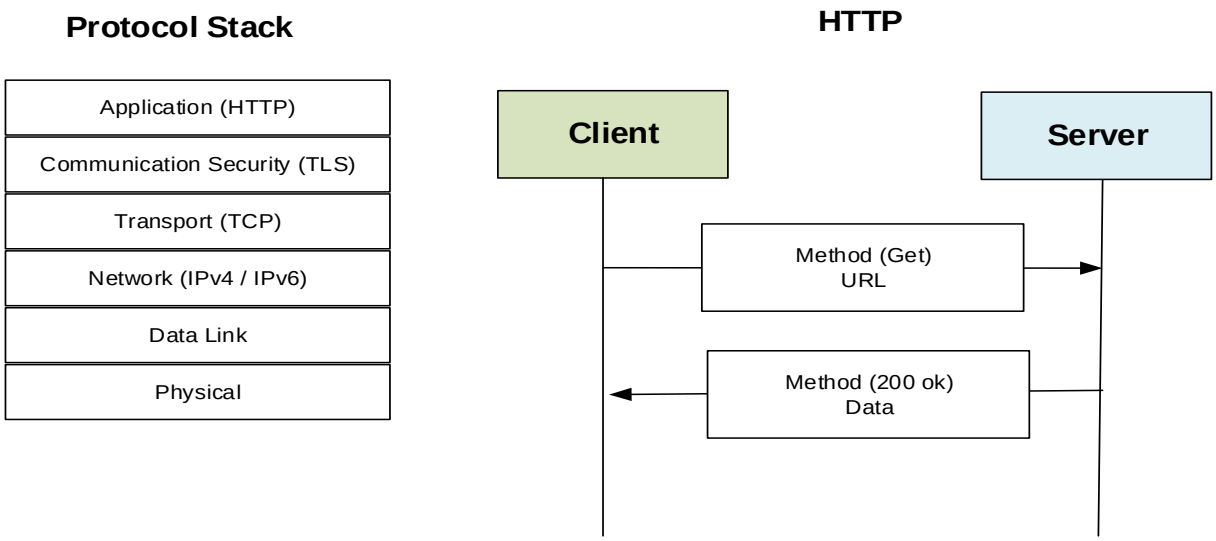

Figure 1 : HTTP GET message sent from the client to the server

4.2 WebSocket

The WebSocket protocol allows web applications to exchange data in both directions over a single TCP connection. It provides an alternative to repeatedly polling a server via HTTP to accomplish real-time interaction. The diagram below summarizes the WebSockets integration into HTTP. After the HTTP communication is established, the client sends an Upgrade request to the server to initiate the WebSocket interaction. When the server responds with a "101 Switching Protocols" response, the two parties are now able to use the WebSockets functionality. The server keeps the socket connection open until the client closes it. Once initiated, all traffic between the client and server is bi-directional, allowing for either end to send WebSocket frames.

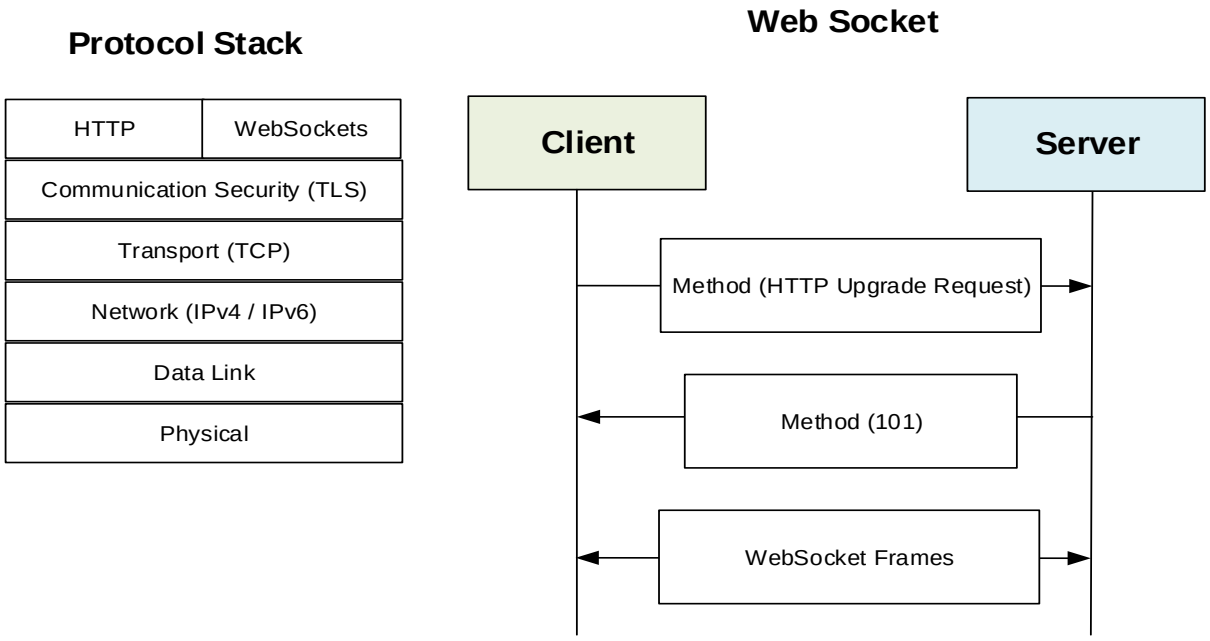

Figure 2 : WebSocket - All traffic between the client and server

4.3 CoAP

The Constrained Application Protocol (CoAP) can communicate over UDP/TCP on port 5683 and over DTLS/TLS on port 5684. CoAP supports offers messages exchanges with optional reliability. Both unicast and multicast communication support is offered. The design of CoAP re-uses HTTP-like methods (POST, PUT, GET, etc.) and for cases where large amounts of data (such as firmware images) must be transferred, the block-wise transfer extension is available. Block-wise transfer chunks the larger data item into smaller blocks that are then transmitted individually. Block-wise transfer is described in RFC 7959.

The diagram below shows an example exchange with a Confirmable (CON) message (GET) being sent and an Acknowledgement message (ACK) in reply. Later, a response message with data utilizes a Non-confirmable message (NON).

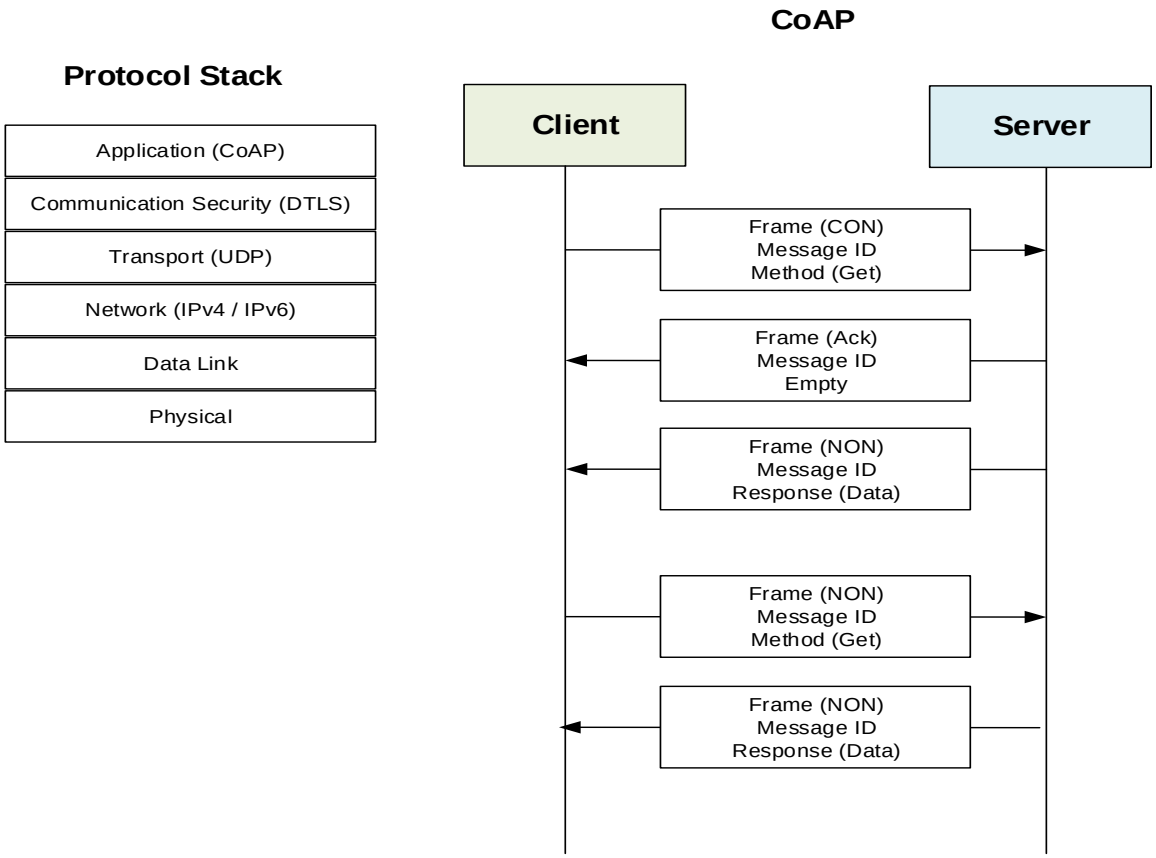

Figure 3 : Exchange with a Confirmable (CON) message (GET) being sent and an Acknowledgement message (ACK) in reply

4.4 XMPP

XMPP (Extensible Messaging and Presence Protocol) provides bi-directional communication between clients via a central server. XMPP is the standardized protocol behind Jabber, a protocol developed for instant messaging (IM). Each client connects to the server with a TCP/TLS socket. Once connected, any client can communicate with any other client, and the server is responsible for message routing. To identify clients, XMPP uses a Jabber Identifier in the form of node@domain/resource. The identifier consists of three components: The “Node” uniquely identifies a single entity, and the domain refers to the “home” domain the client connects to. The resource is optional and uniquely identifies a connection.

There may be multiple XMPP servers connected to provide communications across domains. XMPP servers typically provide a so-called Roster Service that allows clients to indicate to each other that they are connected.

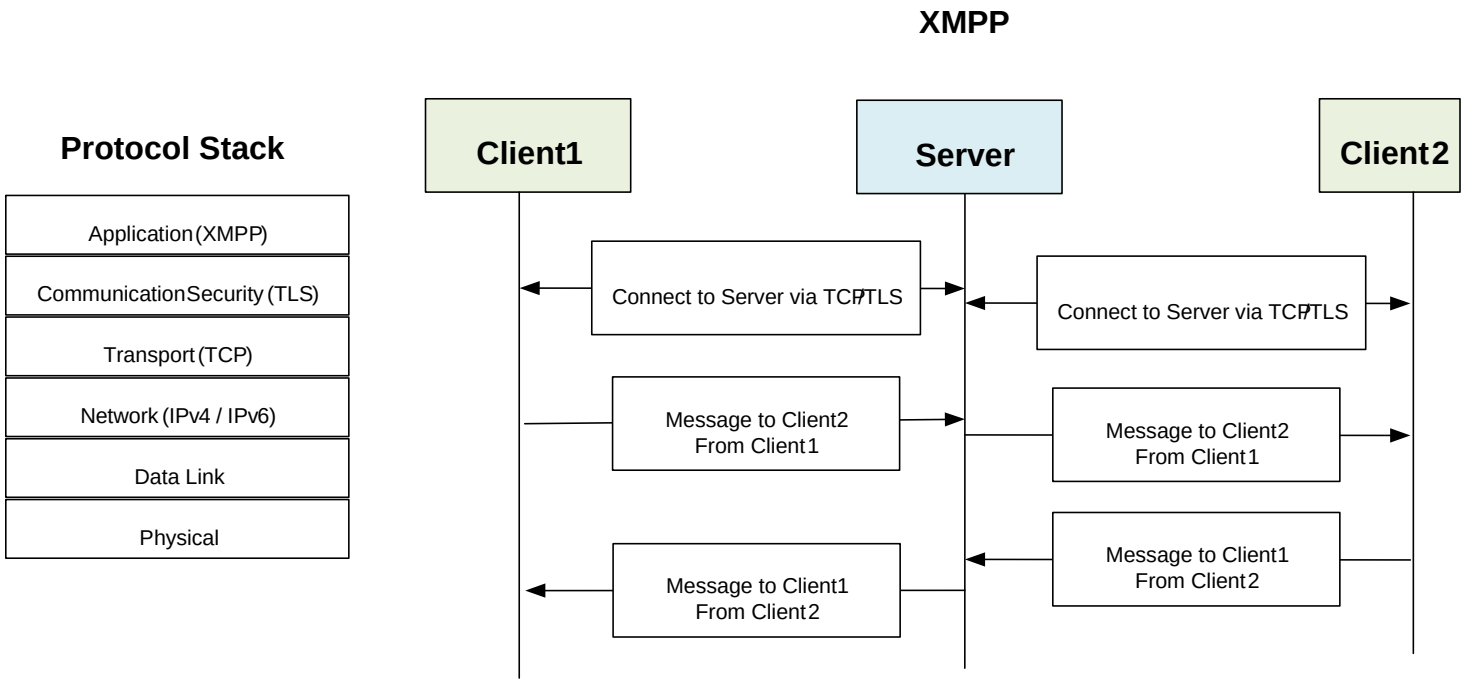

Figure 4 : XMPP bi-directional communication between clients via a central server

4.5 MQTT

MQTT (Message Queue Telemetry Transport) requires that clients connect to a central server (also known as a broker). From there, endpoints may publish (write) or subscribe (read on notification) to "topics".

A PUBLISH message is sent from a client to a server for distribution to subscribers. Each PUBLISH message is associated with a topic name. This is a hierarchical name space that defines a taxonomy of information sources to which subscribers can register. A message that is published to a specific topic name is delivered to all connected subscribers for that topic. "Wildcards" can be used in the hierarchical topic space.

The SUBSCRIBE message allows a client to register an interest in one or more topic names with the server. Messages published to these topics are delivered from the server to the client as PUBLISH messages. The SUBSCRIBE message also specifies the QoS level at which the subscriber wants to receive published messages. The topic names are treated by MQTT as opaque strings without any meaning. No topic names have been standardized.

MQTT provides bi-directional communication between clients via the central server. Each client connects to the server with a TCP/TLS socket. Once connected, any client can publish and subscribe to topics. In the diagram below, clients use topics for bi-directional communication.

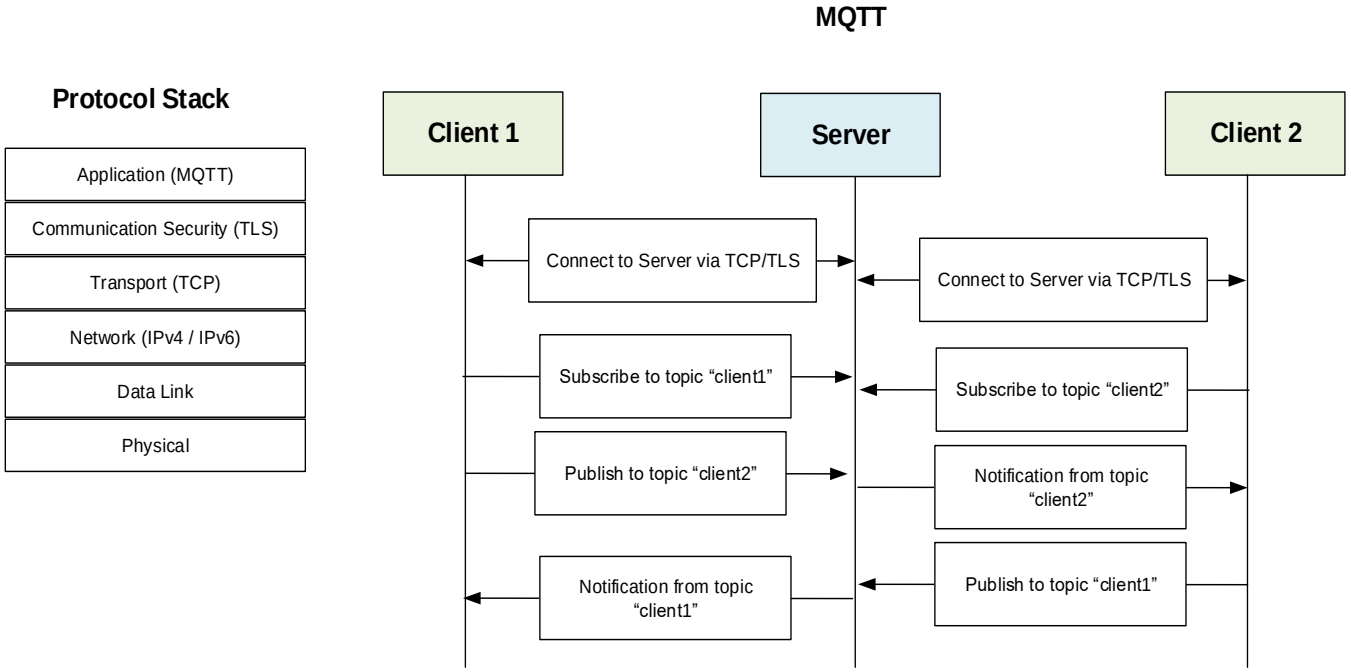

Figure 5 : MQTT bi-directional communication between clients via the central server

4.5.1 MQTT-SN

MQTT-SN is designed to be as close as possible to MQTT, but is adapted to the peculiarities of a wireless communication environment such as low bandwidth, high link failures, short message length, etc. It is also optimized for the implementation on low-cost, battery-operated devices with limited processing and storage resources. MQTT-SN uses UDP for transport whereas MQTT uses TCP. There are three kinds of MQTT-SN components, MQTT-SN clients, MQTT-SN gateways (GW), and MQTT-SN forwarders. MQTT-SN clients connect themselves to a MQTT server via a MQTT-SN GW using the MQTT-SN protocol. A MQTT-SN GW may or may not be integrated with a MQTT server. In case of a stand-alone GW the MQTT protocol is used between the MQTT server and the MQTT-SN GW. Its main function is the translation between MQTT and MQTT-SN.

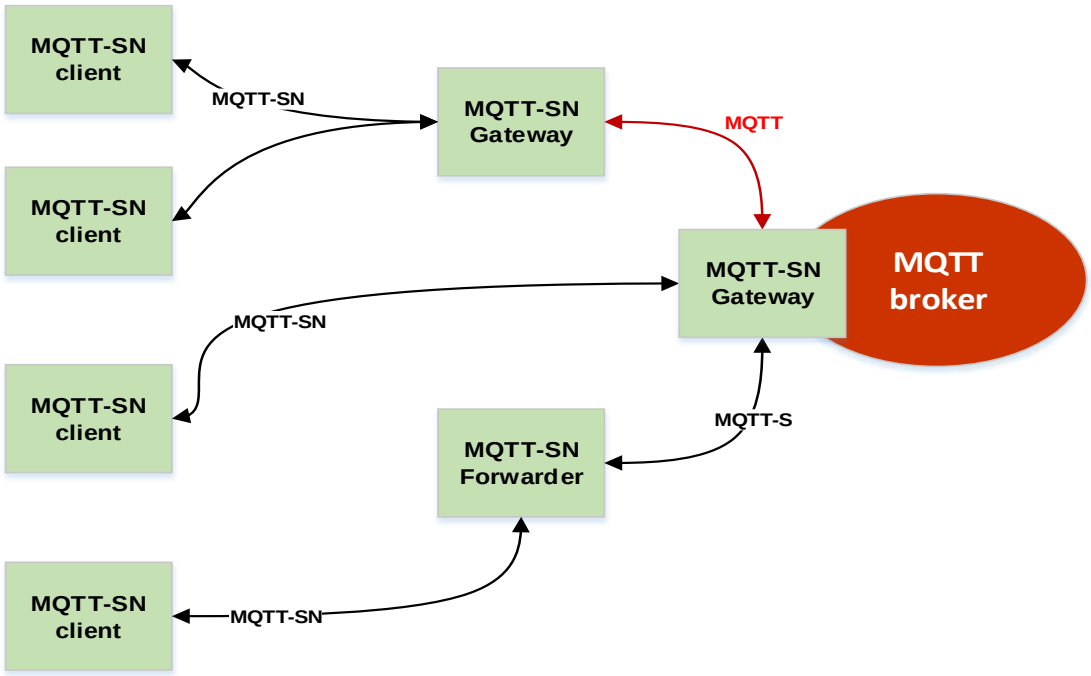

Figure 6 : Translation between MQTT and MQTT-SN

5. Security

Each of the transfer protocols discussed above uses either Transport Layer Security (TLS) over TCP or Datagram Transport Layer Security (DTLS) over UDP. TLS and DTLS are widely used thanks to the fact that they are open-source protocols that contain crypto-security details with an industry-tested API.

DTLS uses the same handshake messages and flows as TLS, with three principal differences:

- A stateless cookie exchange has been added to prevent denial-of-service attacks. This ensures that the client is reachable at the given IP address. To this end, the HelloVerifyRequest message has been added to DTLS 1.0.
- Enhancements have been made to the DTLS handshake header to handle message loss, reordering, and fragmentation. Retransmission timers were added to the specification to detect the loss of handshake messages.
- Extensions were made to the record layer to allow for independent decryption of individual records. This required a sequence number to be added, along with an epoch field to deal with rekeying.

With these exceptions, the DTLS message formats, flows, and logic are the same as those of TLS 1.2. The subsequent figure illustrates the initial (cold-start) TLS and DTLS exchanges. These handshakes are computationally demanding, but occur only when the session is initiated. Once the handshake has been completed, application data can be protected for integrity and confidentiality using highly-efficient symmetric key cryptography. Note that the details of the handshake vary, and some of the shown messages are even optional. The details of the exchanges depend on the selected cipher suite and the negotiated extensions. More details about TLS and DTLS profiles for IoT can be found in RFC 7925.

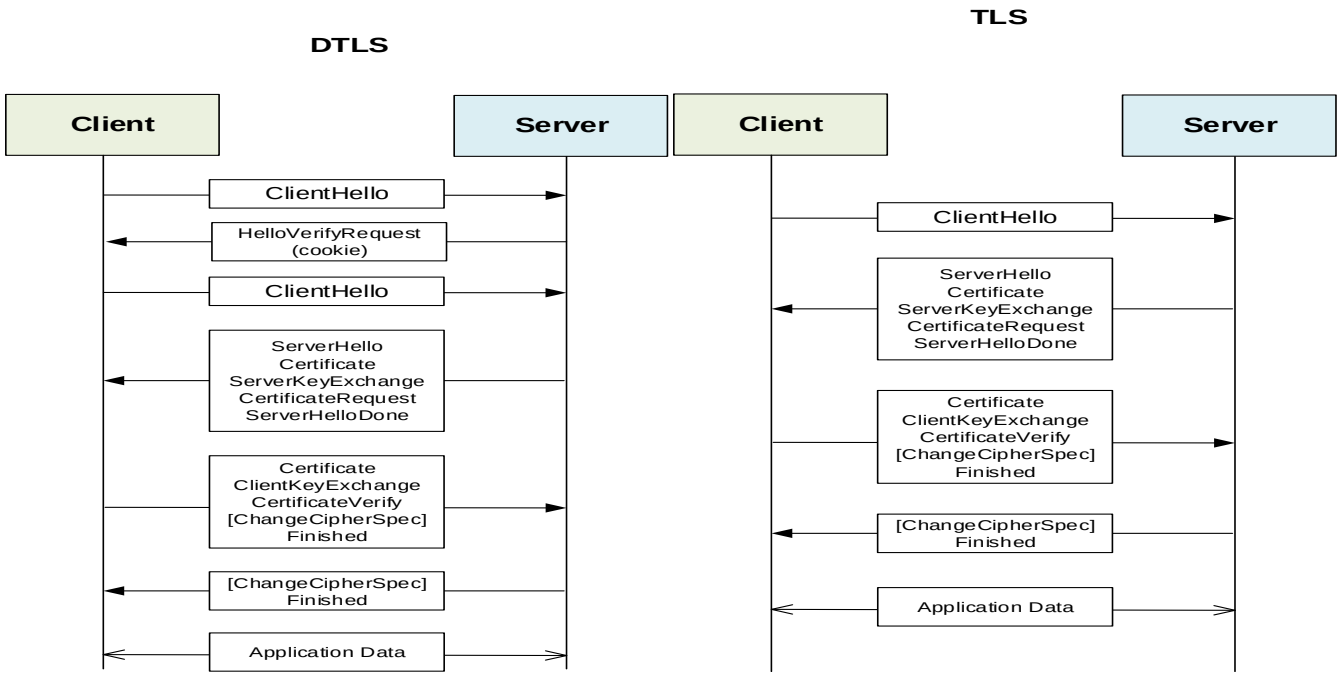

Figure 7 : Transport Layer Security (TLS) over TCP or Datagram Transport Layer Security (DTLS) over UDP

Appendix A. Approved History (Informative)

Document Identifier	Date	Sections	Description
OMA-WP-Protocol_Comparison-V1_0-20200121-A	21 Jan 2020	all	Status changed to Approved by IPSO Ref: # OMA-IPSO-2020-0005-INP_OMA_WP_Protocol_Comparison_V1_0_for_Final_Approval