18CSE345T/ IoT Architecture And Protocols

Unit 4: Transport and Session Layer Protocols

Syllabus

Transport Layer Protocols –Introduction, TCP, MPTCP, UDP, DCCP, SCT, TLS, DTLS, Session Layer-HTTP, CoAP, Implementation demo of CoAP, MQTT, Implementation demo of MQTT, MQTT-SN, Implementation demo of MQTT-SN, XMPP, AMQP, Introduction to Contiki- Practical demo

Transport layer

The OSI Transport layer protocol (ISO-TP) manages end-to-end control and error checking to ensure complete data transfer. It performs transport address to network address mapping, makes multiplexing and splitting of transport connections, and also provides functions such as Sequencing, Flow Control and Error detection and recover Five transport layer protocols exist in the ISO-TP, ranging from Transport Protocol Class 0 through Transport Protocol Class 4 (TP0, TP1, TP2, TP3 & TP4).

The protocols increase in complexity from 0-4. TP0-3 works only with connection-oriented communications, in which a session connection must be established before any data is sent; TP4 also works with both connection-oriented and connectionless communications.

Transport Protocol Class 0 (TP0) performs segmentation (fragmentation) and reassembly functions. TP0 discerns the size of the smallest maximum protocol data unit (PDU) supported by any of the underlying networks, and segments the packets accordingly. The packet segments are reassembled at the receiver.

Transport Protocol Class 1 (TP1) performs segmentation (fragmentation) and reassembly, plus error recovery. TP1 sequences protocol data units (PDUs) and will retransmit PDUs or reinitiate the connection if an excessive number of PDUs are unacknowledged.

Transport Protocol Class 2 (TP2) performs segmentation and reassembly, as well as multiplexing and demultiplexing of data streams over a single virtual circuit.

Transport Protocol Class 3 (TP3) offers error recovery, segmentation and reassembly, and multiplexing and demultiplexing of data streams over a single virtual circuit. TP3 also sequences PDUs and retransmits them or reinitiates the connection if an excessive number are unacknowledged.

Transport Protocol Class 4 (TP4) offers error recovery, performs segmentation and reassembly, and supplies multiplexing and demultiplexing of data streams over a single virtual circuit. TP4 sequences PDUs and retransmits them or reinitiates the connection if an excessive number are unacknowledged. TP4 provides reliable transport service and functions with either connection-oriented or connectionless network service. TP4 is the most commonly used of all the OSI transport protocols and is similar to the Transmission Control Protocol (TCP) in the TCP/IP suite.

Both TP4 and TCP are built to provide a reliable connection oriented end-to-end transport service on top of an unreliable network service. The network service may loose packets, store them, deliver them in the wrong order or even duplicate packets. Both protocols have to be able to deal with the most severe problems e.g. a subnetwork stores valid packets and sends them at a later date. TP4 and TCP have a connect, transfer and a disconnect phase. The principles of doing this are also quite similar.

One difference between TP4 and TCP that should be mentioned is that TP4 uses ten different TPDU (Transport Protocol Data Unit) types whereas TCP knows only one. This makes TCP simple but every TCP header has to have all possible fields and therefore the TCP header is at least 20 bytes long whereas the TP4 header maybe as little as 5 bytes. Another difference is the way both protocols react in case of a call collision. TP4 opens two bidirectional connections between the TSAPs whereas TCP opens just one connection. TP4 uses a different flow control mechanism for its messages. It also provides means for quality of service measurement.

Protocol Structure:

The OSI transport protocols are quite complicated in terms of their structure, which has 10 different types, each with its own header and PDU structure. The ten types are:

CR - Connection Request. The header of this type of message has 7 bytes and the length of the entire TPDU is a variable.

CC- Connection Confirm. The header of this type of message has 7 bytes and the length of the entire TPDU is a variable.

DR – Disconnect Request. The header of this type of message has 7 bytes and the length of the entire TPDU is a variable.

DC – Disconnect Confirm. The header of this type of message has 6 bytes and the length of the entire TPDU is a variable.
DT – Data TPDU. The header of this type of message has 3 bytes and the length of the entire TPDU is a variable.
ED – Expedited Data TPDU. The header of this type of message has 5 bytes and the length of the entire TPDU is a variable.
DA – Data Acknowledgement TPDU. The header of this type of message has 5 bytes and the length of the entire TPDU is a variable.
EA – Expedited Data Acknowledgement TPDU. The header of this type of message has 5 bytes and the length of the entire TPDU is a variable.
RT – Reject TPDU. The header of this type of message has 5 bytes.
ER – Error TPDU. The header of this type of message has 5

TCP

Transmission Control Protocol (TCP) is the transport layer protocol in the TCP/IP suite, which provides a reliable stream delivery and virtual connection service to applications through the use of sequenced acknowledgment with retransmission of packets when necessary. Along with the Internet Protocol (IP), TCP represents the heart of the Internet protocols.

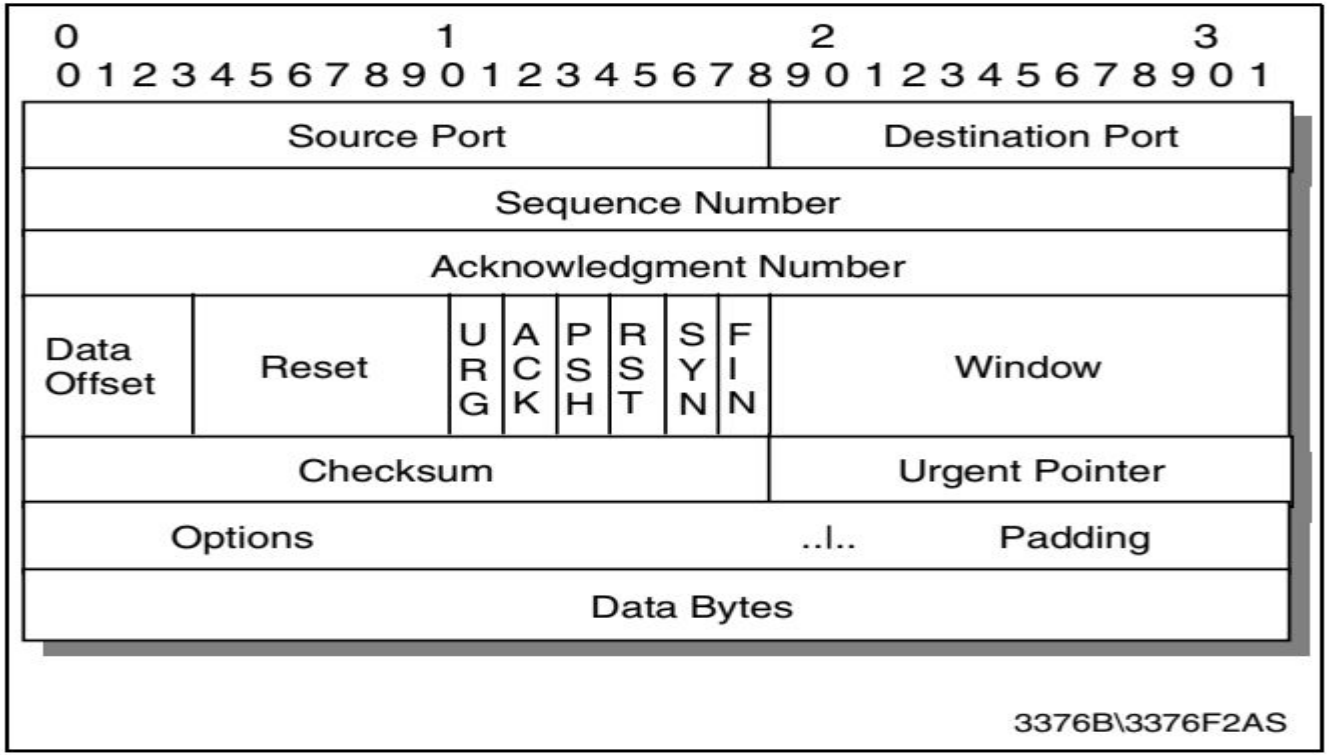
Since many network applications may be running on the same machine, computers need something to make sure the correct software application on the destination computer gets the data packets from the source machine, and some way to make sure replies get routed to the correct application on the source computer. This is accomplished through the use of the TCP "port numbers". The combination of IP address of a network station and its port number is known as a "socket" or an "endpoint". TCP establishes connections or virtual circuits between two "endpoints" for reliable communications.

Among the services TCP provides are stream data transfer, reliability, efficient flow control, full-duplex operation, and multiplexing.

With stream data transfer, TCP delivers an unstructured stream of bytes identified by sequence numbers. This service benefits applications because the application does not have to chop data into blocks before handing it off to TCP. TCP can group bytes into segments and pass them to IP for delivery.

- TCP offers reliability by providing connection-oriented, end-to-end reliable packet delivery. It does this by sequencing bytes with a forwarding acknowledgment number that indicates to the destination the next byte the source expects to receive. Bytes not acknowledged within a specified time period are retransmitted.
- **The reliability mechanism of TCP** allows devices to deal with lost, delayed, duplicate, or misread packets. A time-out mechanism allows devices to detect lost packets and request retransmission.
- TCP offers efficient **flow control** - When sending acknowledgments back to the source, the receiving TCP process indicates the highest sequence number it can receive without overflowing its internal buffers.
- **Full-duplex operation**: TCP processes can both send and receive packets at the same time.
- **Multiplexing in TCP**: Numerous simultaneous upper-layer conversations can be multiplexed over a single connection.

TCP


0 1 2 3
0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
Source Port
Destination Port
Sequence Number
Acknowledgment Number
Data Offset
Reset
URG
ACK
PSH
RST
SYN
FIN
Window
Checksum
Urgent Pointer
Options
..I..
Padding
Data Bytes
3376B\3376F2AS

Where:

• **Source Port**: The 16-bit source port number, used by the receiver to reply.

• **Destination Port**: The 16-bit destination port number.

• **Sequence Number**: The sequence number of the first data byte in this segment. If the SYN control bit is set, the sequence number is the initial sequence number (n) and the first data byte is n+1.

• **Acknowledgment Number**: If the ACK control bit is set, this field contains the value of the next sequence number that the receiver is expecting to receive.

• **Data Offset**: The number of 32-bit words in the TCP header. It indicates where the data begins.

• Reserved: Six bits reserved for future use; must be zero.

• **URG**: Indicates that the urgent pointer field is significant in this segment.

• ACK: Indicates that the acknowledgment field is significant in this segment.

• PSH: Push function.

• RST: Resets the connection

- SYN: Synchronizes the sequence numbers.
- FIN: No more data from sender.
- Window: Used in ACK segments. It specifies the number of data bytes, beginning with the one indicated in the acknowledgment number field that the receiver (= the sender of this segment) is willing to accept.
- Checksum: The 16-bit one's complement of the one's complement sum of all 16-bit words in a pseudo-header, the TCP header, and the TCP data. While computing the checksum, the checksum field itself is considered zero

MPTCP

- **Multipath TCP** (**MPTCP**) is an ongoing effort of the Internet Engineering Task Force's (IETF) Multipath TCP working group, that aims at allowing a Transmission Control Protocol (TCP) connection to use multiple paths to maximize resource usage and increase redundancy
- Mobile user
  - WiFi and cellular at the same time
- High-end servers
  - Multiple Ethernet cards
- Data centers
  - Rich topologies with many paths
- Benefits of multipath
  - Higher throughput
  - Failover from one path to another
  - Seamless mobility

Overview of MPTCP Operation

- The design of Multipath TCP has been influenced by many requirements, but there are two that stand out: application compatibility and network compatibility.
- Application compatibility implies that applications that today run over TCP should work without any change over Multipath TCP.
- Next, Multipath TCP must operate over any Internet path where TCP operates.
- Many paths on today's Internet include middleboxes, such as Network Address Translators, firewalls, and various kinds of transparent proxies. Unlike IP routers, all these devices do know about the TCP connections they forward and affect them in special ways.
- Designing TCP extensions that can safely traverse all these middleboxes has proven to be challenging. Before diving into the details of Multipath TCP, let's recap the basic operation of normal TCP.
- A connection can be divided into three phases:
  connection establishment
  data transfer
  connection release

- A TCP connection starts with a three-way handshake. To open a TCP connection, the client sends a SYN (for "synchronize") packet to the port on which the server is listening.
- The SYN packet contains the source port and initial sequence number chosen by the client, and it may contain TCP options that are used to negotiate the use of TCP extensions.
- The server replies with a SYN+ACK packet, acknowledging the SYN and providing the server's initial sequence number and the options that it supports.
- The client acknowledges the SYN+ACK, and the connection is now fully established. All subsequent packets in the connection use the IP addresses and ports used for the initial handshake. They compose the tuple that uniquely identifies the connection.
- After the handshake, the client and the server can send data packets (segments, in TCP terminology). The sequence number is used to delineate the data in the different segments, reorder them, and detect losses. The TCP header also contains a cumulative acknowledgment, essentially a number that acknowledges received data by telling the sender which is the next byte expected by the receiver. Various techniques are used by TCP to retransmit the lost segments.
- After the data transfer is over, the TCP connection must be closed. A TCP connection can be closed abruptly if one of the hosts sends a Reset (RST) packet, but the usual way to terminate a connection is by using FIN packets. These FIN packets indicate the sequence number of the last byte sent.
- The connection is terminated after the FIN segments have been acknowledged in both directions

Multipath TCP allows multiple subflows to be set up for a single MPTCP session. An MPTCP session starts with an initial subflow, which is similar to a regular TCP connection as described above.

After the first MPTCP subflow is set up, additional subflows can be established. Each additional subflow also looks similar to a regular TCP connection, complete with SYN handshake and FIN tear-down, but rather than being a separate connection, the subflow is bound into an existing MPTCP session.

Data for the connection can then be sent over any of the active subflows that has the capacity to take it.

An important point about Multipath TCP, especially in the context of smartphones, is that the set of subflows that are associated with a Multipath TCP connection is not fixed.

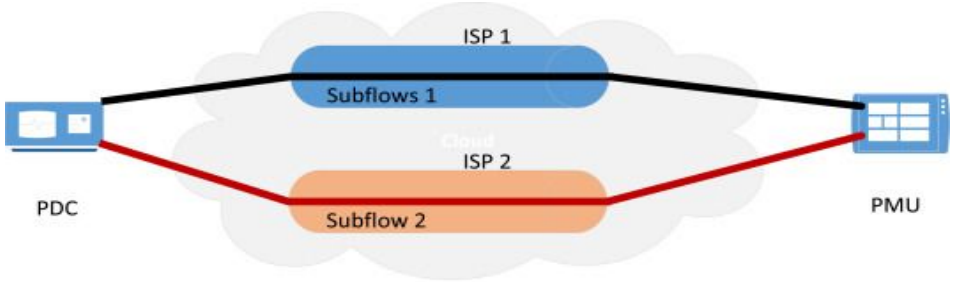
Subflows can be dynamically added and removed from a Multipath TCP connection throughout its lifetime, without affecting the byte-stream transported on behalf of the application.

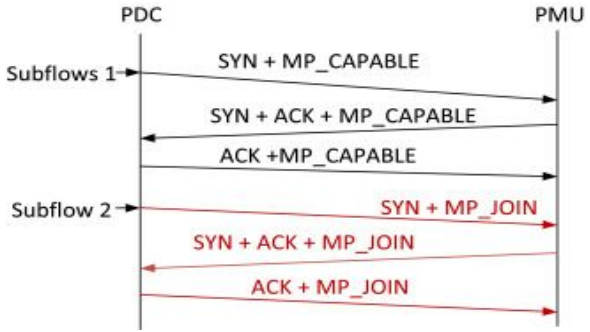
Multipath TCP also implements mechanisms that allow adding and removing new addresses even when an endpoint operates behind a NAT, but we will not detail them here.

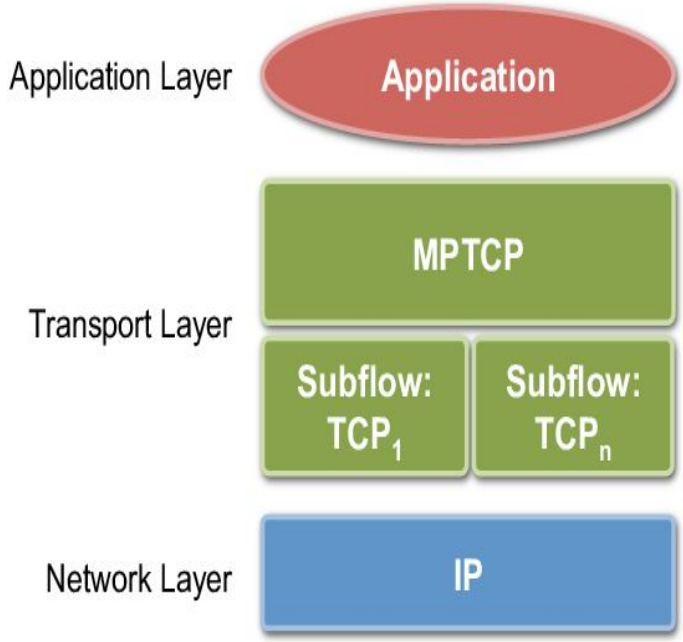
If the smartphone moves to another WiFi network, it will receive a new IP address. At that time, it will open a new subflow using its newly allocated address and tell the server that its old address is not usable anymore. The server will now send data towards the new address.

These options allow smartphones to easily move through different wireless connections without breaking their Multipath TCP connections

Multipath TCP uses its own sequence numbering space. Each segment sent by Multipath TCP contains two sequence numbers: the subflow sequence number inside the regular TCP header, and an additional data sequence number (DSN) carried inside a TCP option. This solution ensures that the segments sent on any given subflow have consecutive sequence numbers and do not upset middleboxes. Multipath TCP can then send some data sequence numbers on one path and the remainder on the other path; old middleboxes will ignore the DSN option, and it will be used by the Multipath TCP receiver to reorder the bytestream before it is given to the receiving application.



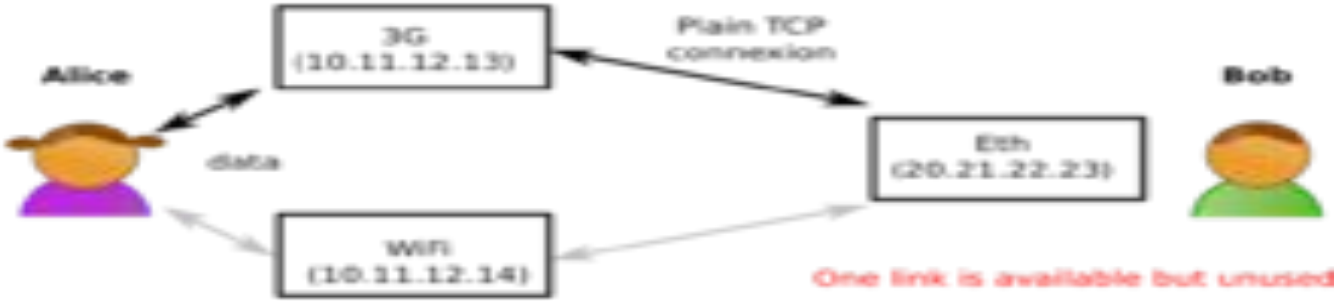




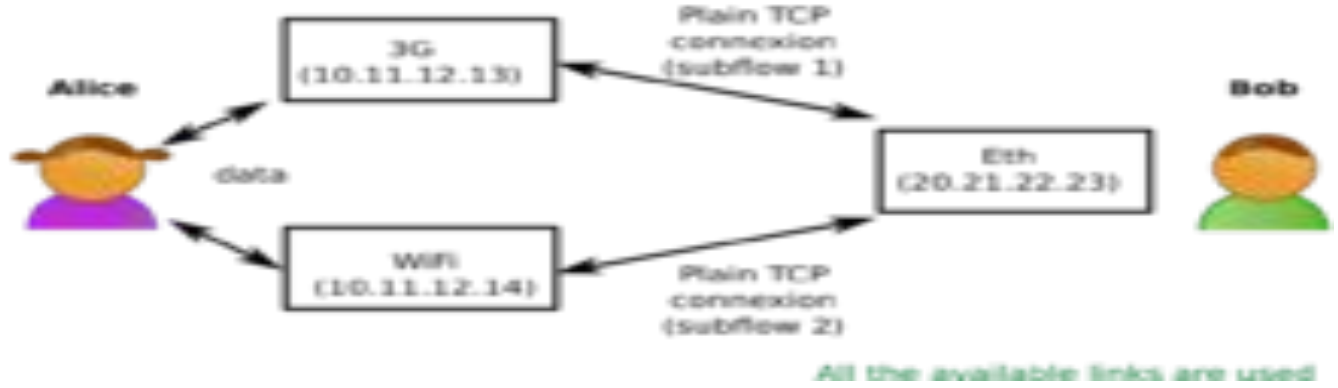
The Multipath TCP option has the Kind (30), length (variable) and the remainder of the content begins with a 4-bit subtype field, for which IANA has created and will maintain a sub-registry entitled "MPTCP Option Subtypes" under the "Transmission Control Protocol (TCP) Parameters" registry. Those subtype fields are defined as follows: Values 0x8 through 0xe are currently unassigned.

Value	Symbol	Name
0x0	MP_CAPABLE	Multipath Capable
0x1	MP_JOIN	Join Connection
0x2	DSS	Data Sequence Signal (Data ACK and data sequence mapping)
0x3	ADD_ADDR	Add Address
0x4	REMOVE_ADDR	Remove Address
0x5	MP_PRIO	Change Subflow Priority
0x6	MP_FAIL	Fallback
0x7	MP_FASTCLOSE	Fast Close
0xf	(PRIVATE)	Private Use within controlled testbeds

Data transmission with plain TCP



Data transmission with MPTCP



The purpose of the different protocol operations are:

- to handle when and how to add/remove paths (for instance if there's a connection lost of some congestion control)
- to be compatible with legacy TCP hardware (such as some firewalls that can automatically reject TCP connections if the sequence number aren't successive)
- to define a fair congestion control strategy between the different links and the different hosts (especially with those that doesn't support MPTCP)

- Multipath TCP adds new mechanisms to TCP transmissions:

- The subflow system, used to gather multiple standard TCP connections (the paths from one host to another). Subflows are identified during the TCP three-way handshake. After the handshake, an application can add or remove some subflows (subtypes 0x3 and 0x4).
- The MPTCP DSS option contains a data sequence number and an acknowledgement number. These allow receiving data from multiple subflows in the original order, without any corruption (message subtype 0x2)
- A modified retransmission protocol handles congestion control and reliability.

UDP

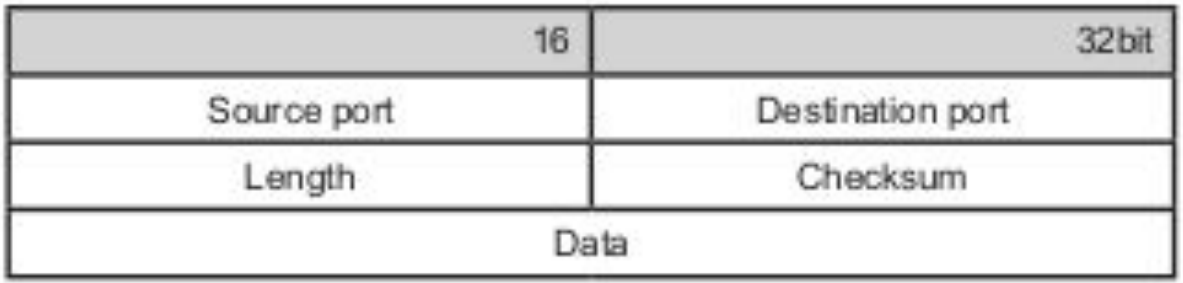
UDP is a connectionless transport layer (layer 4) protocol in the OSI model which provides a simple and unreliable message service for transaction-oriented services. UDP is basically an interface between IP and upper-layer processes. UDP protocol ports distinguish multiple applications running on a single device from one another.

Since many network applications may be running on the same machine, computers need something to make sure the correct software application on the destination computer gets the data packets from the source machine and some way to make sure replies get routed to the correct application on the source computer. This is accomplished through the use of the UDP “port numbers”.

For example, if a station wished to use a Domain Name System (DNS) on the station 128.1.123.1, it would address the packet to station 128.1.123.1 and insert destination port number 53 in the UDP header. The source port number identifies the application on the local station that requested domain name server, and all response packets generated by the destination station should be addressed to that port number on the source station. Details of UDP port numbers can be found in the reference.

Unlike TCP, UDP adds no reliability, flow-control, or error-recovery functions to IP. Because of UDP’s simplicity, UDP headers contain fewer bytes and consume less network overhead than TCP.

UDP is useful in situations where the reliability mechanisms of TCP are not necessary, such as in cases where a higher-layer protocol or application might provide error and flow control.
UDP is the transport protocol for several well-known application-layer protocols, including Network File System (NFS), Simple Network Management Protocol (SNMP), Domain Name System (DNS), and Trivial File Transfer Protocol (TFTP).



Datagram Congestion Control Protocol (DCCP)

- DCCP which implements a congestion-controlled, unreliable flow of datagrams suitable for use by applications such as streaming media.

Introduction

DCCP provides the following features, among others:

- An unreliable flow of datagrams, with acknowledgements.
- A reliable handshake for connection setup and teardown.
- Reliable negotiation of features.
- A choice of TCP-friendly congestion control mechanisms
- Options that tell the sender, with high reliability, which packets reached the receiver, and whether those packets were ECN marked, corrupted, or dropped in the receive buffer.
- Congestion control incorporating Explicit Congestion Notification (ECN) and the ECN Nonce.
- Mechanisms allowing a server to avoid holding any state for unacknowledged connection attempts or already-finished connections.
- Path MTU discovery

Datagram Congestion Control Protocol (DCCP)

- DCCP is intended for applications that require the flow-based semantics of TCP, but have a preference for delivery of timely data over in-order delivery or reliability, or which would like different congestion control dynamics than TCP.

Important Differences from TCP

- **Packet stream.** DCCP is a packet stream protocol, not a byte stream protocol. The application is responsible for framing.
- **Unreliability.** DCCP will never retransmit a datagram. Options are retransmitted as required to make feature negotiation and ack information reliable.
- **Packet sequence numbers.** Sequence numbers refer to packets, not bytes. Every packet sent by a DCCP endpoint gets a new sequence number, even including pure acknowledgements.
- **Copious space for options** (up to 1020 bytes).
- **Feature negotiation.** This is a generic mechanism by which endpoints can agree on the values of "features", or properties of the connection.
- **Choice of congestion control.** One such feature is the congestion control mechanism to use for the connection. In fact, the two endpoints can use different congestion control mechanisms for their data packets:

Datagram Congestion Control Protocol (DCCP)

- **Different acknowledgement formats.** The CCID for a connection determines how much ack information needs to be transmitted.
- **No receive window.** DCCP is a congestion control protocol, not a flow control protocol.
- **Distinguishing different kinds of loss.** A Data Dropped option lets one endpoint declare that a packet was dropped because of corruption, because of receive buffer overflow, and so on. This facilitates research into more appropriate rate-control responses for these non-network-congestion losses (although currently all losses will cause a congestion response).
- **Definition of acknowledgement.** In TCP, a packet is acknowledged only when the data is queued for delivery to the application. This does not make sense in DCCP, where an application might request a drop-from-front receive buffer, for example. We acknowledge a packet when its options have been processed. The Data Dropped option may later say that the packet's payload was discarded.
- **Integrated support for mobility.**
- **No simultaneous open.**

Datagram Congestion Control Protocol (DCCP)

- **Concepts and Terminology**
  Each DCCP connection runs between two endpoints, which we often name DCCP A and DCCP B. Data may pass over the connection in either or both directions.
- We often consider a subset of the connection, namely a *half-connection*, which consists of the data packets sent in one direction, plus the corresponding acknowledgements sent in the other direction. In the context of a single half-connection, the HC-Sender is the endpoint sending data, while the HC-Receiver is the endpoint sending acknowledgements.
- DCCP uses a generic mechanism to negotiate connection properties, such as the CCIDs active on the two half-connections. These properties are called features. The Change, Prefer, and Confirm options negotiate feature values.
- **DCCP Packets**
  DCCP has nine different packet types: DCCP-Request, DCCP-Response, DCCP-Data, DCCP-Ack, DCCP-DataAck, DCCP-CloseReq, DCCP-Close, DCCP-Reset, and DCCPMove

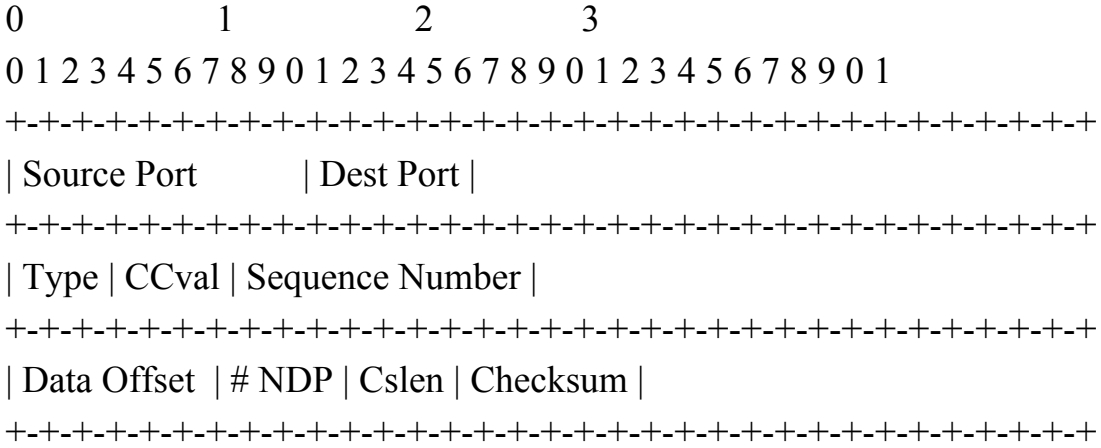
DCCP Packets

- The client sends the server **a DCCP-Request packet** specifying the client and server ports, the service being requested, and any features being negotiated, including the CCID that the client would like the server to use.
- The server sends the client a **DCCP-Response packet** indicating that it is willing to communicate with the client. The response indicates any features and options that the server agrees to, begins or continues other feature negotiations if desired
- The client sends the server a **DCCP-Ack packet** that acknowledges the DCCP Response packet. This acknowledges the server's initial sequence number and returns the Init Cookie if there was one in the
- Next comes zero or more DCCP-Ack exchanges as required to finalize feature negotiation. The client may piggyback an application-level request on its final ack, producing **a DCCP-Data Ack packet**.
- The server and client then exchange **DCCP-Data packets**, DCCP-Ack packets acknowledging that data, and, optionally, **DCCP-DataAck packets** containing piggybacked data and acknowledgements. If the client has no data to send, then the server will send DCCP-Data and DCCP-DataAck packets, while the client will send DCCP-Acks exclusively.
- The server sends **a DCCP-CloseReq packet** requesting a close.
- The client sends a **DCCP-Close packet** acknowledging the close.
- The server sends a **DCCP-Reset packet** whose Reason field is set to "Closed", and clears its connection state.
- The client receives the DCCP-Reset packet and holds state for a reasonable interval of time to allow any remaining packets to clear the network.

Packet Formats

Generic Packet Header

All DCCP packets begin with a generic DCCP packet header:



Options and Features

All DCCP packets may contain options, which occupy space at the end of the DCCP header and are a multiple of 8 bits in length. All options are always included in the checksum. An option may begin on any byte boundary.

The following options are currently defined:

Option Type	Length	Meaning
0	1	Padding
2	1	Slow Receiver
32	3-4	Ignored
33	variable	Change
34	variable	Prefer
35	variable	Confirm
36	variable	Init Cookie
37	variable	Ack Vector [Nonce 0]
38	variable	Ack Vector [Nonce 1]
39	variable	Data Dropped
40	6	Timestamp
41	6-10	Timestamp Echo
42	Variable	Identification
44	variable	Challenge
45	4	Payload Checksum
46	4-6	Elapsed Time
128-255	variable	CCID-specific options

Feature Numbers

The first data byte of every Change, Prefer, or Confirm option is a feature number, defining the type of feature being negotiated. The remainder of the data gives one or more values for the feature, and is interpreted according to the feature. The current set of feature numbers is as follows:

Number	Meaning	Neg.?
1	Congestion Control (CC)	Y
2	ECN Capable	Y
3	Ack Ratio	N
4	Use Ack Vector	Y
5	Mobility Capable	Y
6	Loss Window	N
7	Connection Nonce	N
8	Identification Regime	Y
128-255	CCID-Specific Features	?

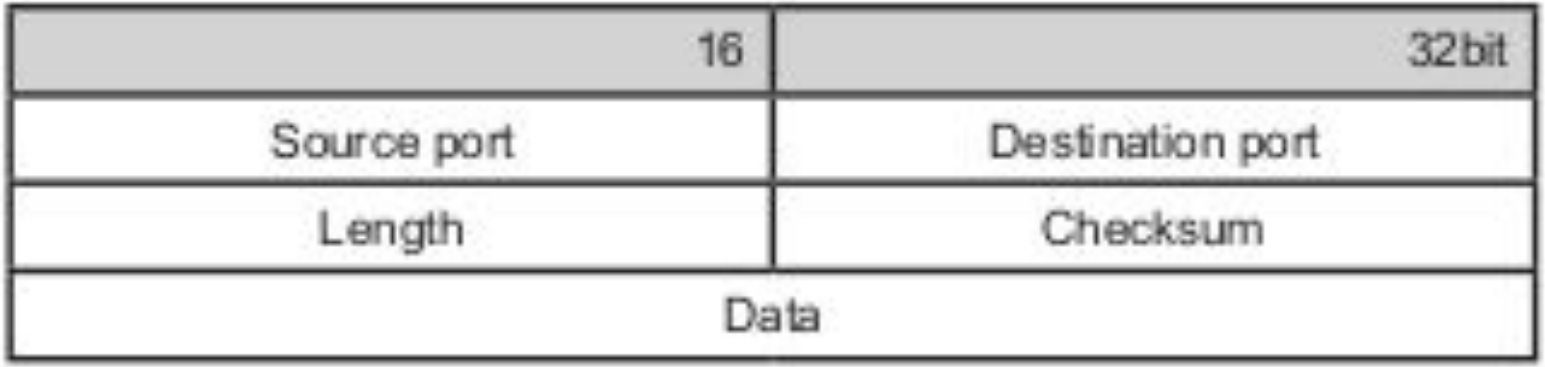
SCTP

- Stream Control Transmission Protocol (SCTP) is designed to transport PSTN signalling messages (SS7/C7) over IP networks, but is capable of broader applications. SCTP is a reliable transport protocol operating on top of a connectionless packet network such as IP.
- SCTP is designed to address the limitations and complexity of TCP while transporting real time signaling and data such as PSTN signaling over an IP network. SCTP can also run on top of the UDP layer.

SCTP offers the following services:

- acknowledged error-free non-duplicated transfer of user data;
- data fragmentation to conform to discovered path MTU size;
- sequenced delivery of user messages within multiple streams, with an option for order-of-arrival delivery of individual user messages;
- optional bundling of multiple user messages into a single SCTP packet; and network-level fault tolerance through supporting of multihoming at either or both ends of an association.

- The design of SCTP includes appropriate congestion avoidance behavior and resistance to flooding and masquerade attacks. The SCTP datagram is comprised of a common header and chunks. The chunks contain either control information or user data.


16
32bit
Source port
Destination port
Length
Checksum
Data

Transport Layer Security

- Transport Layer Security (TLS) Protocol is to provide privacy and data integrity between two communicating applications.

- The protocol is composed of two layers: the TLS Record Protocol and the TLS Handshake Protocol. At the lowest level, layered on top of some reliable transport protocol (TCP) is the TLS Record Protocol.
- **TLS Record Protocol**
  - provides data confidentiality using symmetric key cryptography
  - provides data integrity using a keyed message authentication checksum (MAC)
- The keys are generated uniquely for each session based on the security parameters agreed during the TLS handshake
- The TLS Record Protocol provides connection security that has two basic properties:

- Private - Symmetric cryptography is used for data encryption (DES, RC4, etc.) The keys for this symmetric encryption are generated uniquely for each connection and are based on a secret negotiated by another protocol (such as the TLS Handshake Protocol). The Record Protocol can also be used without encryption.

- Reliable - Message transport includes a message integrity check using a keyed MAC. Secure hash functions (SHA, MD5, etc.) are used for MAC computations. The Record Protocol can operate without a MAC, but is generally only used in this mode while another protocol is using the Record Protocol as a transport for negotiating security parameters.

The TLS Record Protocol is used for encapsulation of various higher level protocols. One such encapsulated protocol, the TLS Handshake Protocol, allows the server and client to authenticate each other and to negotiate an encryption algorithm and cryptographic keys before the application protocol transmits or receives its first byte of data.

Basic operation of the TLS Record Protocol

- read messages for transmit
- fragment messages into manageable chunks of data
- compress the data, if compression is required and enabled
- calculate a MAC
- encrypt the data
- transmit the resulting data to the peer

At the opposite end of the TLS connection, the basic operation of the sender is replicated, but in the reverse order

- read received data from the peer
- decrypt the data
- verify the MAC
- decompress the data, if compression is required and enabled
- reassemble the message fragments
- deliver the message to upper protocol layers

- **TLS Handshake Protocol** is layered on top of the TLS Record Protocol
- TLS Handshake Protocol is used to
  - authenticate the client and the server
  - exchange cryptographic keys
  - negotiate the used encryption and data integrity algorithms before the applications start to communicate with each other

Figure illustrates the actual handshake message flow

- **[Step1]**
- the client and server exchange Hello messages
- the client sends a ClientHello message, which is followed by the server sending a ServerHello message
- these two messages establish the TLS protocol version, the compression mechanism used, the cipher suite used, and possibly the TLS session ID
- additionally, both a random client nonce and a random server nonce are exchanged that are used in the handshake later on

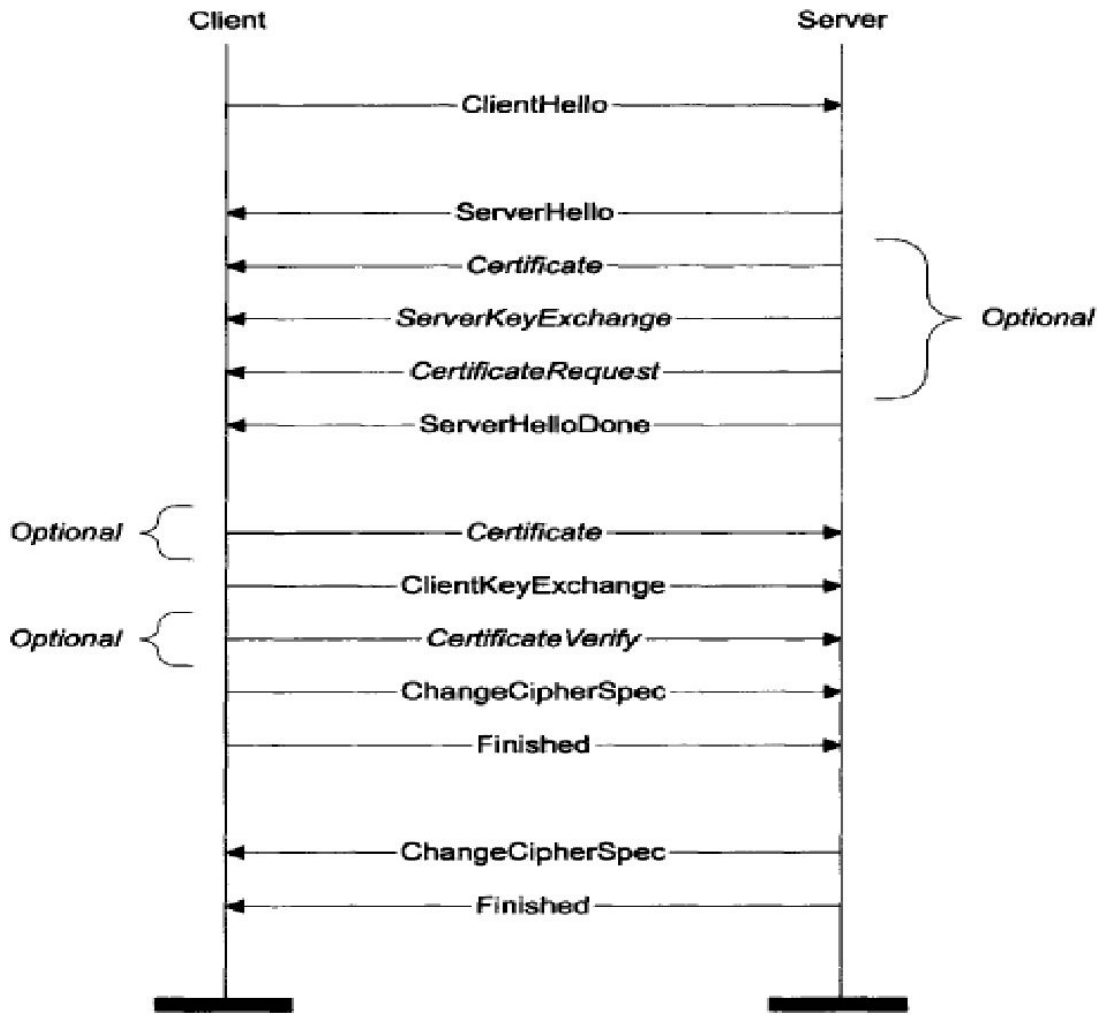


Figure 14.1 The TLS handshake.

[Step2]

- the server may send any messages associated with the ServerHello
- depending on the selected cipher suite, it will send its certificate for authentication
- the server may also send a key exchange message and a certificate request message to the client, depending on the selected cipher suite
- to mark the end of the ServerHello and the Hello message exchange, the server sends a ServerHelloDone message

[Step3]

- next, if requested, the client will send its certificate to the server
- in any case, the client will then send a key exchange message that sets the pre-master secret between the client and the server
- optionally, the client may also send a Certificate Verify message to explicitly verify the certificate that the server requested

[Step4]

- then, both the client and the server send the ChangeCipherSpec messages and enable the newly negotiated cipher spec
- the first message passed in each direction using the new algorithms, keys and secrets is the Finished message, which includes a digest of all the handshake messages
- each end inspects the Finished message to verify that the handshake was not tampered with

- **Digest of all the handshake messages**
  - means the results of applying a one-way hash function to the handshake messages
- It also includes functionality for client and server authentication using public key cryptography
- **Advantage of TLS**
  - applications can use it transparently to securely communicate with each other
  - TLS is visible to applications, making them aware of the cipher suites and authentication certificates negotiated during the set-up phases of a TLS session
- TLS is based on the Secure Socket Layer (SSL), a protocol originally created by Netscape. One advantage of TLS is that it is application protocol independent. The TLS protocol runs above TCP/IP and below application protocols such as HTTP or IMAP. The HTTP running on top of TLS or SSL is often called HTTPS.

- The TLS standard does not specify how protocols add security with TLS; the decisions on how to initiate TLS handshaking and how to interpret the authentication certificates exchanged are left up to the judgment of the designers and implementers of protocols which run on top of TLS.

The TLS Handshake Protocol provides connection security that has three basic properties:

1. The peer's identity can be authenticated using asymmetric, or public key, cryptography (RSA, DSS, etc.). This authentication can be made optional, but is generally required for at least one of the peers.

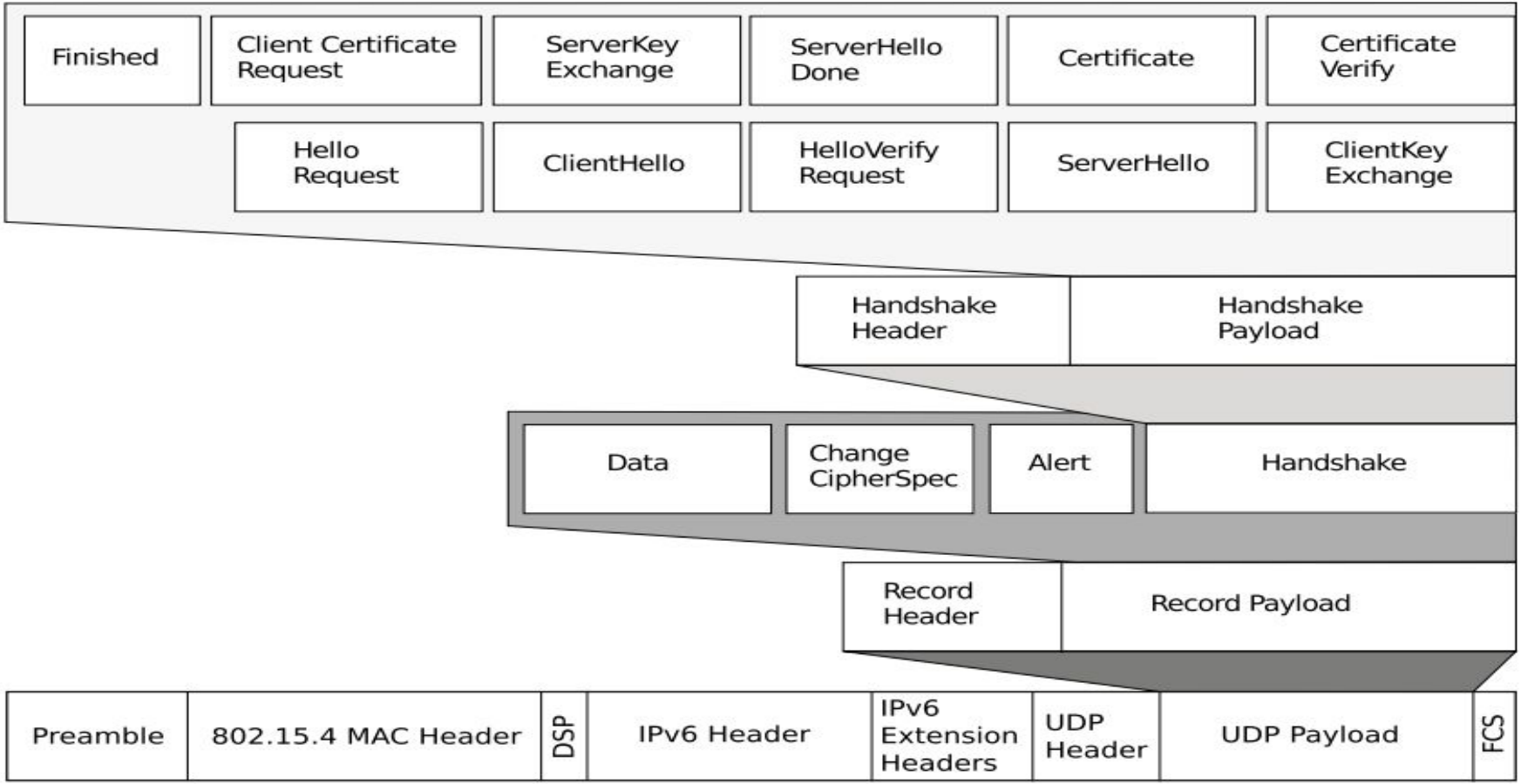
2. The negotiation of a shared secret is secure: The negotiated secret is unavailable to eavesdroppers, and for any authenticated connection the secret cannot be obtained, even by an attacker who can place himself in the middle of the connection.

3. The negotiation is reliable: no attacker can modify the negotiation communication without being detected by the parties to the communication.

Datagram Transport Layer Security (DTLS)

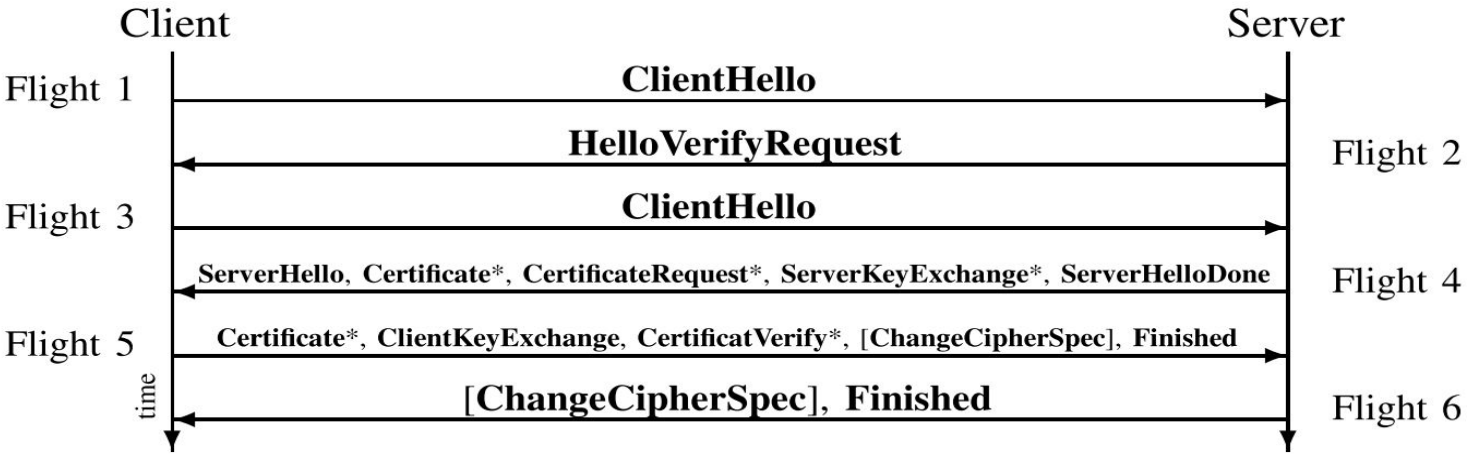
- DTLS (Datagram Transport Layer Security) is used by CoAP as the security protocol
  - For key management and data encryption and integrity protection.
- DTLS consists of two sublayers:
  - Upper layer contains:
    - ***Handshake***, *Alert* and *ChangeCipherSpec* protocols
    - Or ***application data***.
  - Lower layer contains the Record protocol
    - Carrier for the upper layer protocols
    - Record header contains content type and fragment fields.
- DTLS is between Application layer and Transport Layer

Layout of a packet secured with DTLS



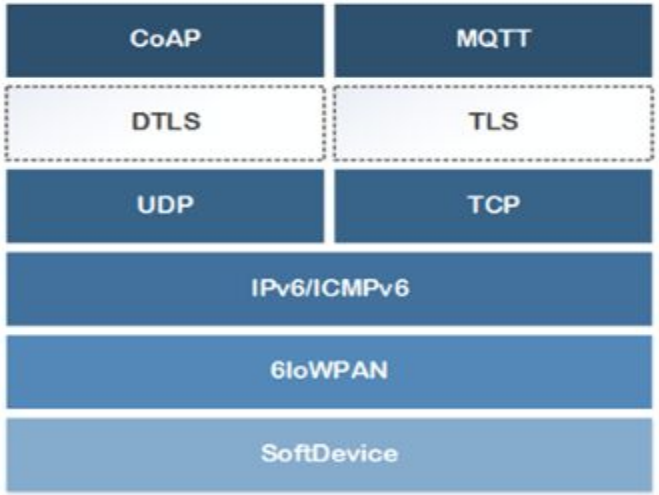
DTLS-Handshake Process

- The handshake messages are used to negotiate security keys, cipher suites and compressing methods.
- This paper is limited to the header compression process only.
- During the handshake process the ClientHello message is sent twice.
  - Without cookie
  - With the server's cookie



DTLS handshake protocol. * means optional.

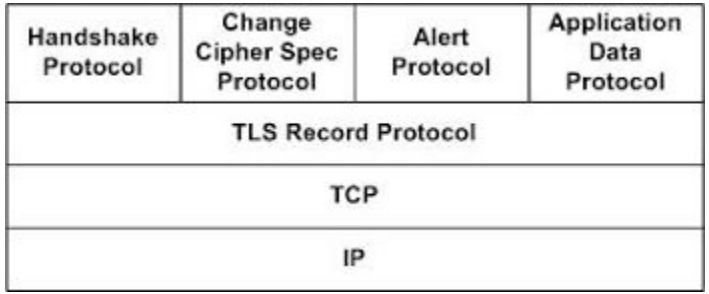
- DTLS is similar to TLS intentionally except that DTLS has to solve two problems: packet lost and reordering.
  DTLS implements
  - packet retransmission
  - assigning sequence number within the handshake
  - replay detection.
- Datagram Transport Layer Security (DTLS) is a communications protocol designed to protect data privacy and preventing eavesdropping and tampering. It is based on the Transport Layer Security (TLS) protocol, which is a protocol that provides security to computer-based communications networks. The main difference between DTSL and TLS is that DTLS uses UDP and TLS uses TCP. It is used across web browsing, mail, instant messaging and VoIP.



- **Handshake comparison**

Field	TLS	DTLS
Message type	1	1
Message length	3	3
Message sequence number	Doesn't exist	2
Fragment offset	Doesn't exist	3
Fragment length	Doesn't exist	3
Total	**4**	**12**

- **TLS protocol Architecture**



Session Layer

- The OSI Session Layer Protocol (ISO-SP) provides session management, e.g. opening and closing of sessions. In case of a connection loss it tries to recover the connection. If a connection is not used for a longer period, the session layer may close it down and re-open it for next use.

- This happens transparently to the higher layers. The Session layer provides synchronization points in the stream of exchanged packets.

- The Session Protocol Machine (SPM), an abstract machine that carries out the procedures specified in the session layer protocol, communicates with the session service user (SS-user) through an session-service-access-point (SSAP) by means of the service primitives.

- Service primitives will cause or be the result of session protocol data unit exchanges between the peer SPMs using a transport connection. These protocol exchanges are effected using the services of the transport layer.

- Session connection endpoints are identified in end systems by an internal, implementation dependent, mechanism so that the SS-user and the SPM can refer to each session connection.

The functions in the Session Layer are those necessary to bridge the gap between the services available from the Transport Layer and those offered to the SS-users.

The functions in the Session Layer are concerned with dialogue management, data flow synchronization, and data flow resynchronization.

These functions are described below; the descriptions are grouped into those concerned with the connection establishment phase, the data transfer phase, and the release phase.

Functional unit	SPDU code	SPDU name
Kernel	CN	CONNECT
	OA	OVERFLOW ACCEPT
	CDO	CONNECT DATA OVERFLOW
	AC	ACCEPT
	RF	REFUSE
	FN	FINISH
	DN	DISCONNECT
	AB	ABORT
	AA	ABORT ACCEPT
	DT	DATA TRANSFER
	PR	PREPARE

Negotiated release	NF GT PT	NOT FINISHED GIVE TOKENS PLEASE TOKENS
Half-duplex	GT PT	GIVE TOKENS PLEASE TOKENS
Duplex		No additional associated SPDUs
Expedited data	EX	EXPEDITED DATA
Typed data	TD	TYPED DATA
Capability data exchange	CD CDA	CAPABILITY DATA CAPABILITY DATA ACK
Minor synchronize	MIP MIA GT PT	MINOR SYNC POINT MINOR SYNC ACK GIVE TOKENS PLEASE TOKENS

Symmetric synchro-nize	MIP MIA	MINOR SYNC POINT MINOR SYNC ACK
Data separation		No additional associated SPDUs
Major synchronize	MAP MAA PR GT PT	MAJOR SYNC POINT MAJOR SYNC ACK PREPARE GIVE TOKENS PLEASE TOKENS
Resynchronize	RS RA PR	RESYNCHRONIZE RESYNCHRONIZE ACK PREPARE
Exceptions	ER ED	EXCEPTION REPORT EXCEPTION DATA
Activity management	AS AR AI AIA AD ADA AE AEA PR GT PT GTC GTA	ACTIVITY START ACTIVITY RESUME ACTIVITY INTERRUPT ACTIVITY INTERRUPT ACK ACTIVITY DISCARD ACTIVITY DISCARD ACK ACTIVITY END ACTIVITY END ACK PREPARE GIVE TOKENS PLEASE TOKENS GIVE TOKENS CONFIRM GIVE TOKENS ACK

HTTP

- The Hypertext Transfer Protocol (HTTP) is an application-level protocol with the lightness and speed necessary for distributed, collaborative, hypermedia information systems. HTTP has been in use by the World-Wide Web global information initiative since 1990.

- HTTP allows an open-ended set of methods to be used to indicate the purpose of a request. It builds on the discipline of reference provided by the Uniform Resource Identifier (URI), as a location (URL) or name (URN), for indicating the resource on which a method is to be applied. Messages are passed in a format similar to that used by Internet Mail and the Multipurpose Internet Mail Extensions (MIME).

- HTTP is also used as a generic protocol for communication between user agents and proxies/gateways to other Internet protocols, such as SMTP, NNTP, FTP, Gopher and WAIS, allowing basic hypermedia access to resources available from diverse applications and simplifying the implementation of user agents.

- The HTTP protocol is a request/response protocol. A client sends a request to the server in the form of a request method, URI, and protocol version, followed by a MIME-like message containing request modifiers, client information, and possible body content over a connection with a server. The server responds with a status line, including the message's protocol version and a success or error code, followed by a MIME-like message containing server information, entity meta information, and possible entity-body content.

The first version of HTTP, referred to as HTTP/0.9, was a simple protocol for raw data transfer across the Internet. HTTP/1.0, as defined by RFC 1945, improved the protocol by allowing messages to be in the format of MIME-like messages, containing meta information about the data transferred and modifiers on the request/response semantics.

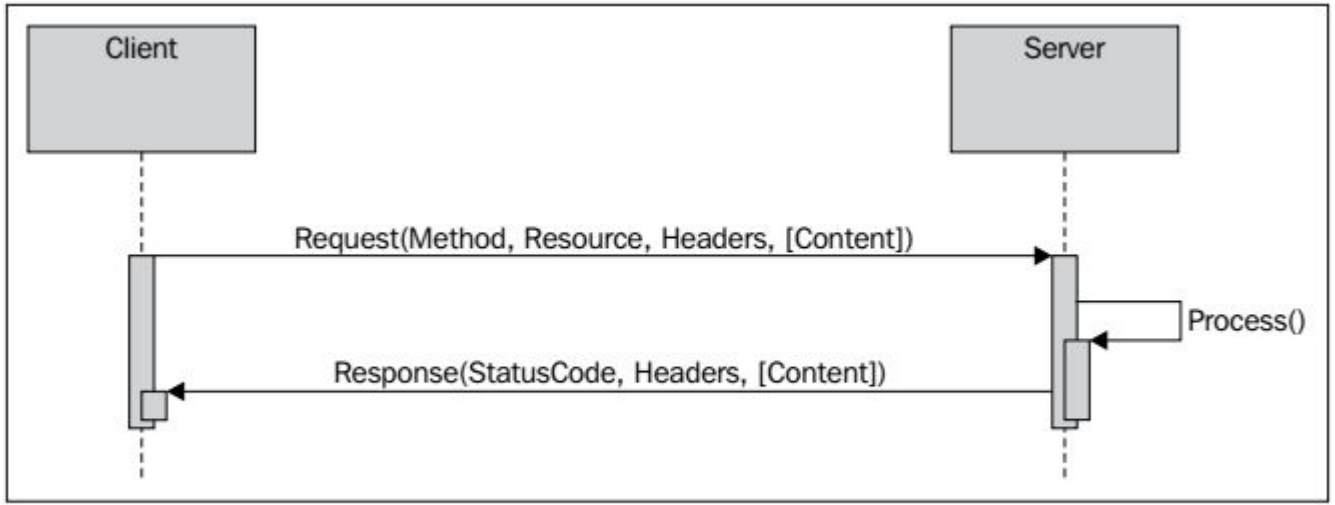
However, HTTP/1.0 does not sufficiently take into consideration the effects of hierarchical proxies, caching, the need for persistent connections, or virtual hosts. “HTTP/1.1” includes more stringent requirements than HTTP/1.0 in order to ensure reliable implementation of its features. There is a secure version of HTTP (S-HTTP) specification, which will be discussed in a separate document.

It’s a study the request/response pattern and the ways to handle events, user authentication, and web services.

some of the basic concepts used in HTTP which we will be looking at:

- The basics of HTTP
- How to add HTTP support to the sensor, actuator, and controller projects
- How common communication patterns such as request/response and event subscription can be utilized using HTTP

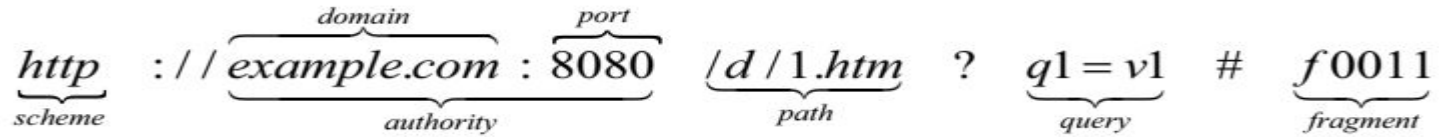
HTTP is a stateless request/response protocol where clients request information from a server and the server responds to these requests accordingly. A request is basically made up of a method, a resource, some headers, and some optional content. A response is made up of a three-digit status code, some headers and some optional content. This can be observed in the following diagram



HTTP request/response pattern

In the structure of the URL presented next, the resource is identifed by the path and the server by the authority portions of the URL. The PUT and DELETE methods allow clients to upload and remove content from the server, while the POST method allows them to send data to a resource on the server, for instance, in a web form.

The structure of a URL is shown in the following diagram:



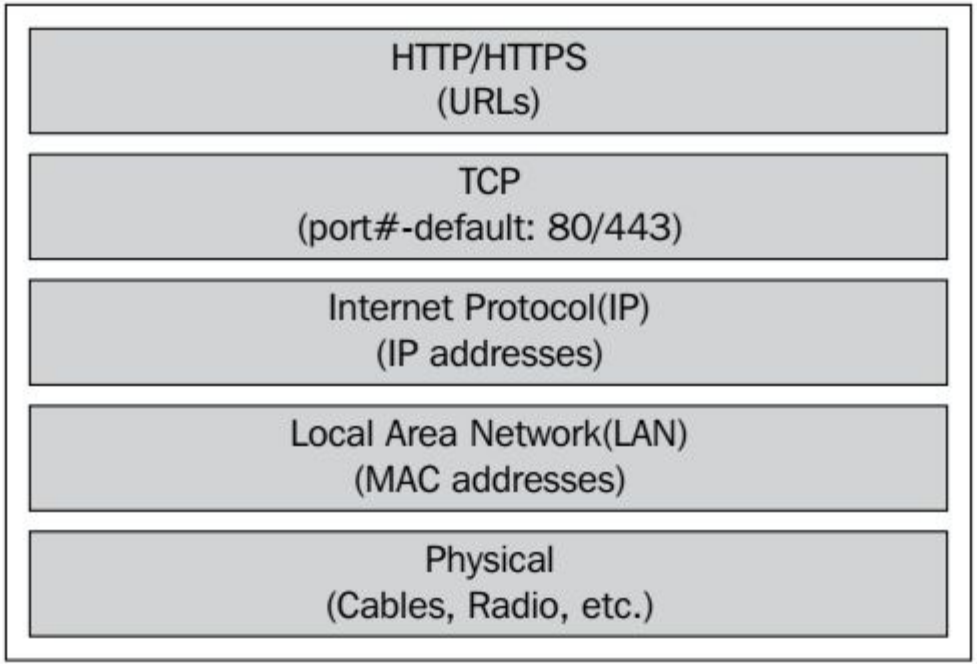
Structure of a Uniform Resource Locator (URL)

H sent over the network. These headers are human readable key - value text pairs that contain information about how content is encoded, for how long it is valid, what type of content is desired, and so on.

The type of content is identifed by a Content-Type header, which identifes the type of content that is being transmitted. Headers also provide a means to authenticate clients to the server and a mechanism to introduce states in HTTP.

By introducing cookies, which are text strings, the servers can ask the client to remember the cookies, which the client can then add to each request that is made to the server.

- HTTP works on top of the **Internet Protocol** (**IP**). In this protocol, machines are addressed using an IP address, which makes it possible to communicate between different **local area networks** (**LANs**) that might use different addressing schemes, even though the most common ones are Ethernet-type networks that use **media access control** (**MAC**) addresses.
- Communication in HTTP is then done over a **Transmission Control Protocol** (**TCP**) connection between the client and the server. The TCP connection makes sure that the packets are not lost and are received in the same order in which they were sent. The connection endpoints are defned by the corresponding IP addresses and a corresponding port number.
- The assigned default port number for HTTP is 80, but other port numbers can also be used; the alternative HTTP port 8080 is common.
- **To simplify communication, Domain Name System (DNS) servers provide a mechanism of using host names instead of IP addresses when referencing a machine on the IP network**
- Encryption can done through the use of Secure Sockets Layer (SSL) or Transport Layer Security (TLS). When this is done, the protocol is normally named Hypertext Transfer Protocol Secure (HTTPS) and the communication is performed on a separate port, normally 443. In this case, most commonly the server, but also the client, can be authenticated using X.509 certifcates that are based on a Public Key Infrastructure (PKI), where anybody with access to the public part of the certificate can encrypt data meant for the holder of the private part of the certifcate. The private part is required to decrypt the information. These certifcates allow the validation of the domain of the server or the identity of the client.


HTTP/HTTPS
(URLs)
TCP
(port#-default: 80/443)
Internet Protocol(IP)
(IP addresses)
Local Area Network(LAN)
(MAC addresses)
Physical
(Cables, Radio, etc.)

COAP

- The Constrained Application Protocol (CoAP) is another session layer protocol designed by IETF Constrained RESTful Environment (Core) working group to provide lightweight RESTful (HTTP) interface. Representational State Transfer (REST) is the standard interface between HTTP client and servers. However, for lightweight applications such as IoT, REST could result in significant overhead and power consumption.
- CoAP is designed to enable low-power sensors to use RESTful services while meeting their power constrains. It is built over UDP, instead of TCP commonly used in HTTP and has a light mechanism to provide reliability.
- CoAP architecture is divided into two main sublayers: messaging and request/response.
- The messaging sublayer is responsible for reliability and duplication of messages while the request/response sublayer is responsible for communication.
- As shown in Figure , CoAP has four messaging modes: confirmable, non- confirmable, piggyback and separate. Confirmable and non- confirmable modes represent the reliable and unreliable transmissions, respectively while the other modes are used for request/response.
- Piggyback is used for client/server direct communication where the server sends its response directly after receiving the message, i.e., within the acknowledgment message. On the other hand, the separate mode is used when the server response comes in a message separate from the acknowledgment, and may take some time to be sent by the server.
- As in HTTP, CoAP utilizes GET, PUT, PUSH, DELETE messages requests to retrieve, create, update, and delete, respectively

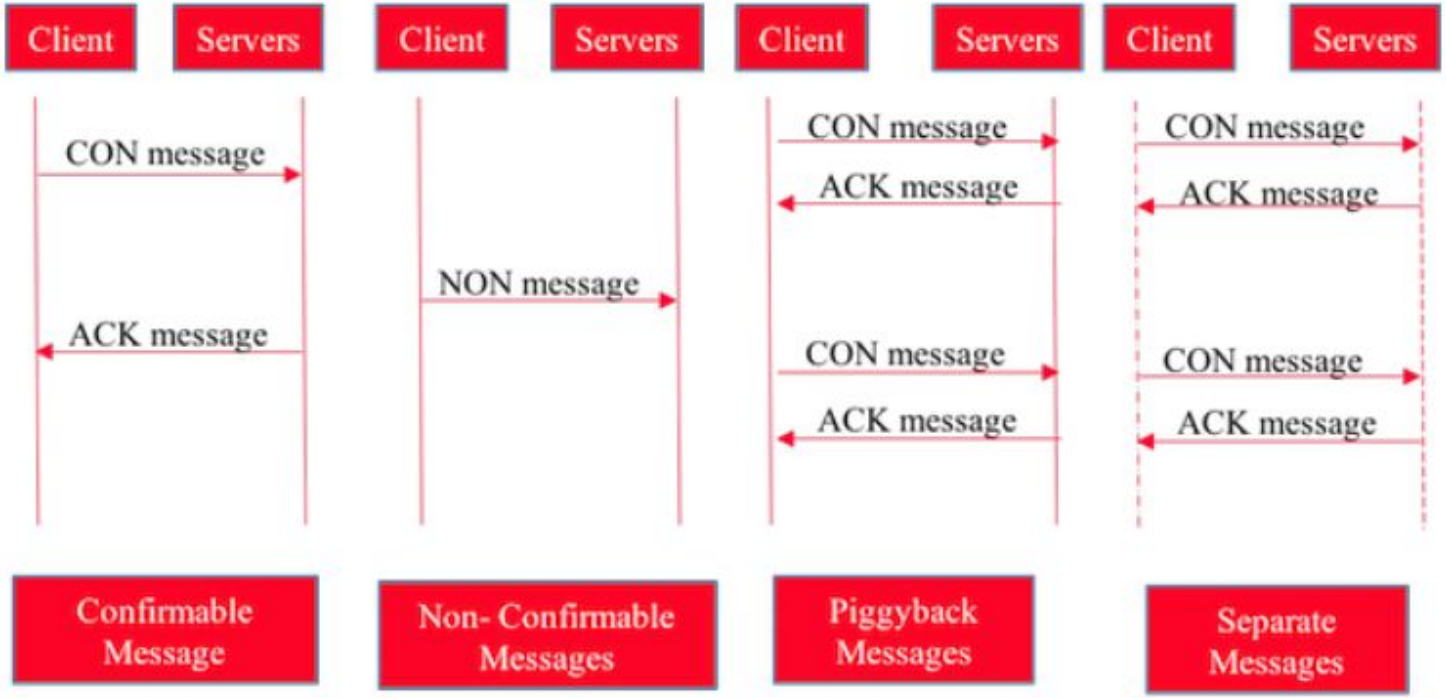


Figure 7: CoAP Messages.

- The CoAP messaging model is primarily designed to facilitate the exchange of messages over UDP between endpoints, including the secure transport protocol Datagram Transport Layer Security (DTLS).
- CoAP over Short Message Service (SMS) as defined in Open Mobile Alliance for Lightweight Machine-to-Machine (LWM2M) for IoT device management is also being considered.
- From a formatting perspective, a CoAP message is composed of a short fixed-length Header field (4 bytes), a variable-length but mandatory Token field (0–8 bytes), Options fields if necessary, and the Payload field. Figure 6-7 details the CoAP message format, which delivers low overhead while decreasing parsing complexity.
- As you can see in Figure 6-7, the CoAP message format is relatively simple and flexible. It allows CoAP to deliver low overhead, which is critical for constrained networks, while also being easy to parse and process for constrained devices.

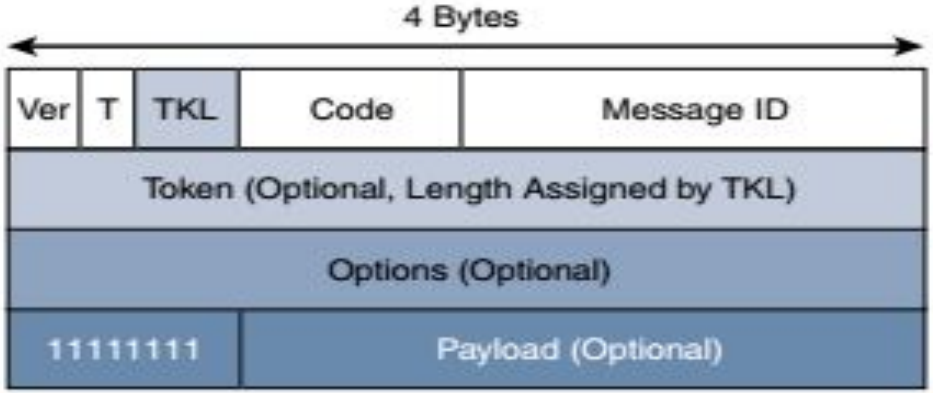


Figure 6-7 *CoAP Message Format*

Table 6-1 *CoAP Message Fields*

CoAP Message Field	Description
Ver (Version)	Identifies the CoAP version.
T (Type)	Defines one of the following four message types: Confirmable (CON), Non-confirmable (NON), Acknowledgement (ACK), or Reset (RST). CON and ACK are highlighted in more detail in Figure 6-9.
TKL (Token Length)	Specifies the size (0–8 Bytes) of the Token field.
Code	Indicates the request method for a request message and a response code for a response message. For example, in Figure 6-9, GET is the request method, and 2.05 is the response code. For a complete list of values for this field, refer to RFC 7252.
Message ID	Detects message duplication and used to match ACK and RST message types to Con and NON message types.
Token	With a length specified by TKL, correlates requests and responses.
Options	Specifies option number, length, and option value. Capabilities provided by the Options field include specifying the target resource of a request and proxy functions.
Payload	Carries the CoAP application data. This field is optional, but when it is present, a single byte of all 1s (0xFF) precedes the payload. The purpose of this byte is to delineate the end of the Options field and the beginning of Payload.

- CoAP can run over IPv4 or IPv6. However, it is recommended that the message fit within a single IP packet and UDP payload to avoid fragmentation.
- For IPv6, with the default MTU size being 1280 bytes and allowing for no fragmentation across nodes, the maximum CoAP message size could be up to 1152 bytes, including 1024 bytes for the payload. In the case of IPv4, as IP fragmentation may exist across the network, implementations should limit themselves to more conservative values and set the IPv4 Don't Fragment (DF) bit.
- While most sensor and actuator traffic utilizes small-packet payloads, some use cases, such as firmware upgrades, require the capability to send larger payloads. CoAP doesn't rely on IP fragmentation but defines (in RFC 7959) a pair of Block options for transferring multiple blocks of information from a resource representation in multiple request/ response pairs.
- The CoAP request/response semantics include the methods GET, POST, PUT, and DELETE.

Example 6-2 *CoAP URI format*

```
coap-URI = "coap:" "//" host [":" port] path-abempty ["?" query]
coaps-URI = "coaps:" "//" host [":" port] path-abempty  ["?" query]
```

CoAP defines four types of messages: confirmable, non-confirmable, acknowledgement, and reset. Method codes and response codes included in some of these messages make them carry requests or responses. CoAP code, method and response codes, option numbers, and content format have been assigned by IANA as Constrained RESTful Environments (CoRE) parameters.

While running over UDP, CoAP offers a reliable transmission of messages when a CoAP header is marked as "confirmable." In addition, CoAP supports basic congestion control with a default time-out, simple stop and wait retransmission with exponential back-off mechanism, and detection of duplicate messages through a message ID. If a request or response is tagged as confirmable, the recipient must explicitly either acknowledge or reject the message, using the same message ID

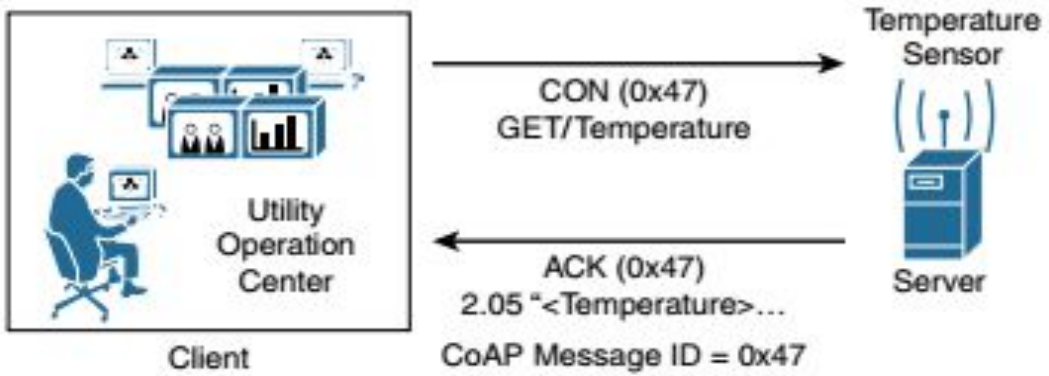


Figure 6-9 *CoAP Reliable Transmission Example*

MQTT

- Message Queue Telemetry Transport (MQTT) was introduced by IBM in 1999 and standardized by OASIS in 2013. It is designed to provide embedded connectivity between applications and middleware's on one side and networks and communications on the other side.
- It follows a publish/subscribe architecture, as shown in Figure, where the system consists of three main components: publishers, subscribers, and a broker.
- From IoT point of view, publishers are basically the lightweight sensors that connect to the broker to send their data and go back to sleep whenever possible. Subscribers are applications that are interested in a certain topic, or sensory data, so they connect to brokers to be informed whenever new data are received.
- The brokers classify sensory data in topics and send them to subscribers interested in the topics.

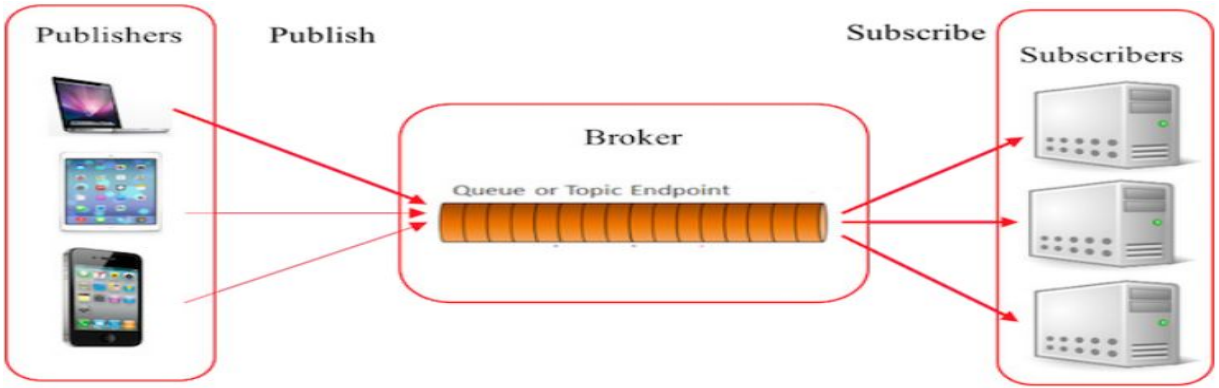


Figure 5: MQTT Architecture

- Considering the harsh environments in the oil and gas industries, an extremely simple protocol with only a few options was designed, with considerations for constrained nodes, unreliable WAN backhaul communications, and bandwidth constraints with variable latencies.
- A great example is the collaboration and social networking application Twitter.

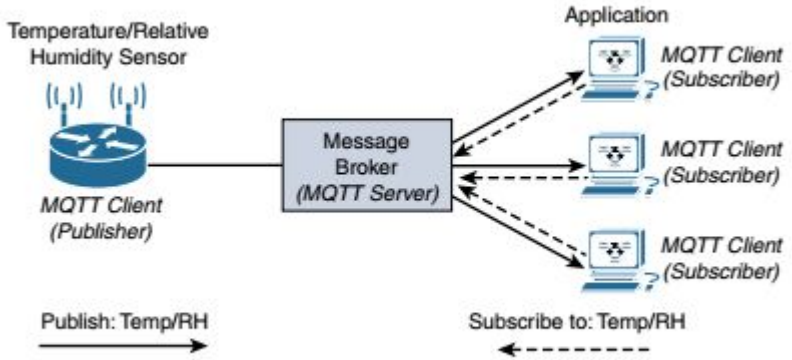


Figure 6-10 *MQTT Publish/Subscribe Framework*

With MQTT, clients can subscribe to all data (using a wildcard character) or specific data from the information tree of a publisher. In addition, the presence of a message broker in MQTT decouples the data transmission between clients acting as publishers and subscribers.

In fact, publishers and subscribers do not even know (or need to know) about each other. A benefit of having this decoupling is that the MQTT message broker ensures that information can be buffered and cached in case of network failures. This also means that publishers and subscribers do not have to be online at the same time.

MQTT control packets run over a TCP transport using port 1883. TCP ensures an ordered, lossless stream of bytes between the MQTT client and the MQTT server. Optionally, MQTT can be secured using TLS on port 8883, and WebSocket (defined in RFC 6455) can also be used.

MQTT is a lightweight protocol because each control packet consists of a 2-byte fixed header with optional variable header fields and optional payload. You should note that a control packet can contain a payload up to 256 MB. Figure 6-11 provides an overview of the MQTT message format.

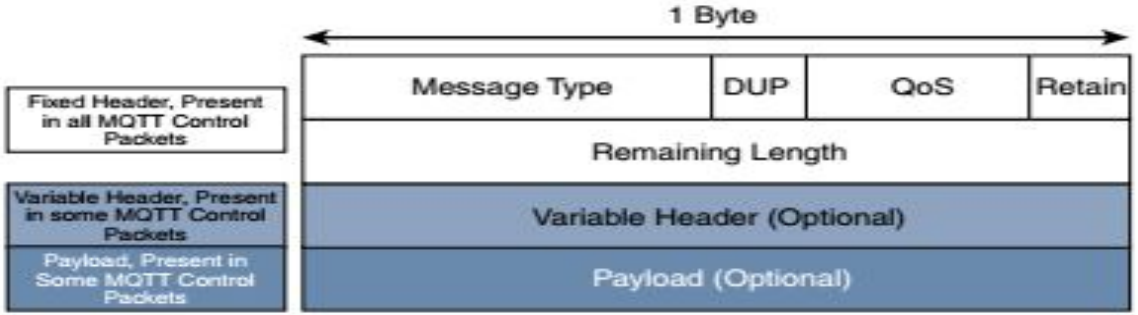


Figure 6-11 *MQTT Message Format*

Compared to the CoAP message format in Figure 6-7, you can see that MQTT contains a smaller header of 2 bytes compared to 4 bytes for CoAP. The first MQTT field in the header is Message Type, which identifies the kind of MQTT packet within a message. Fourteen different types of control packets are specified in MQTT version 3.1.1. Each of them has a unique value that is coded into the Message Type field. Note that values 0 and 15 are reserved. MQTT message types are summarized in Table 6-2.

Table 6-2 *MQTT Message Types*

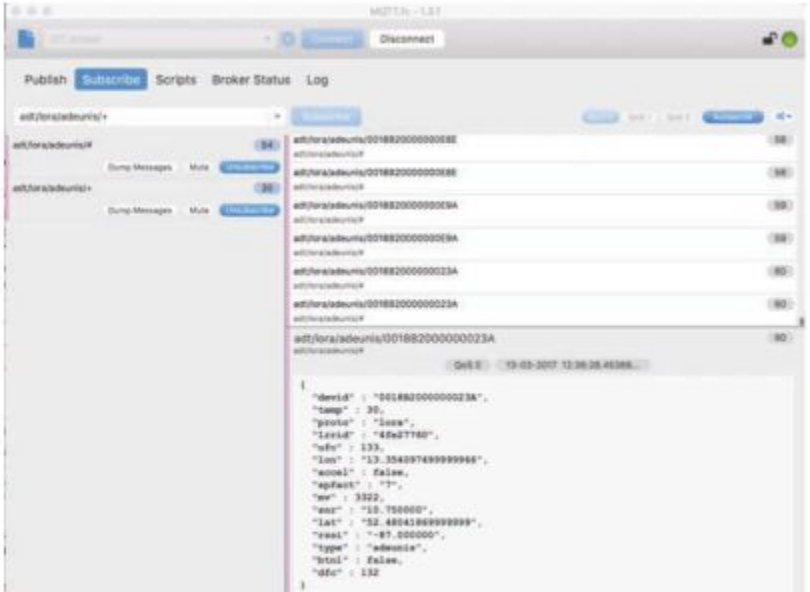
Message Type	Value	Flow	Description
CONNECT	1	Client to server	Request to connect
CONNACK	2	Server to client	Connect acknowledgement
PUBLISH	3	Client to server Server to client	Publish message
PUBACK	4	Client to server Server to client	Publish acknowledgement
PUBREC	5	Client to server Server to client	Publish received
PUBREL	6	Client to server Server to client	Publish release
PUBCOMP	7	Client to server Server to client	Publish complete
SUBSCRIBE	8	Client to server	Subscribe request
SUBACK	9	Server to client	Subscribe acknowledgement
UNSUBSCRIBE	10	Client to server	Unsubscribe request

Message Type	Value	Flow	Description
UNSUBACK	11	Server to client	Unsubscribe acknowledgement
PINGREQ	12	Client to server	Ping request
PINGRESP	13	Server to client	Ping response
DISCONNECT	14	Client to server	Client disconnecting

- The next field in the MQTT header is DUP (Duplication Flag). This flag, when set, allows the client to notate that the packet has been sent previously, but an acknowledgement was not received.
- The QoS header field allows for the selection of three different QoS levels.
- The next field is the Retain flag. Only found in a PUBLISH message (refer to Table 6-2), the Retain flag notifies the server to hold onto the message data. This allows new subscribers to instantly receive the last known value without having to wait for the next update from the publisher.
- The last mandatory field in the MQTT message header is Remaining Length. This field specifies the number of bytes in the MQTT packet following this field.
- MQTT sessions between each client and server consist of four phases: session establishment, authentication, data exchange, and session termination.
- Each client connecting to a server has a unique client ID, which allows the identification of the MQTT session between both parties. When the server is delivering an application message to more than one client, each client is treated independently

Subscriptions to resources generate SUBSCRIBE/SUBACK control packets, while unsubscription is performed through the exchange of UNSUBSCRIBE/UNSUBACK control packets. Graceful termination of a connection is done through a DISCONNECT control packet, which also offers the capability for a client to reconnect by re-sending its client ID to resume the operations.

A message broker uses a topic string or topic name to filter messages for its subscribers. When subscribing to a resource, the subscriber indicates the one or more topic levels that are used to structure the topic name. The forward slash (/) in an MQTT topic name is used to separate each level within the topic tree and provide a hierarchical structure to the topic names. Figure 6-12 illustrates these concepts with **adt/lora.adeunis** being a topic level and **adt/lora/adeunis/0018B2000000023A** being an example of a topic name.



- Wide flexibility is available to clients subscribing to a topic name. An exact topic can be subscribed to, or multiple topics can be subscribed to at once, through the use of wildcard characters. A subscription can contain one of the wildcard characters to allow subscription to multiple topics at once.
- The pound sign (#) is a wildcard character that matches any number of levels within a topic. The multilevel wildcard represents the parent and any number of child levels. For example, subscribing to **adt/lora/adeunis/#** enables the reception of the whole subtree, which could include topic names such as the following:
  - **adt/lora/adeunis/0018B20000000E9E**
  - **adt/lora/adeunis/0018B20000000E8E**
  - **adt/lora/adeunis/0018B20000000E9A**
- The plus sign (+) is a wildcard character that matches only one topic level. For example, **adt/lora/+** allows access to **adt/lora/adeunis/** and **adt/lora/abeeway** but not to **adt/lora/ adeunis/0018B20000000E9E**.
- Topic names beginning with the dollar sign ($) must be excluded by the server when subscriptions start with wildcard characters (# or +). Often, these types of topic names are utilized for message broker internal statistics. So messages cannot be published to these topics by clients.
- For example, a subscription to +/monitor/Temp does not receive any messages published to $SYS/monitor/Temp. This topic could be the control channel for this temperature sensor.

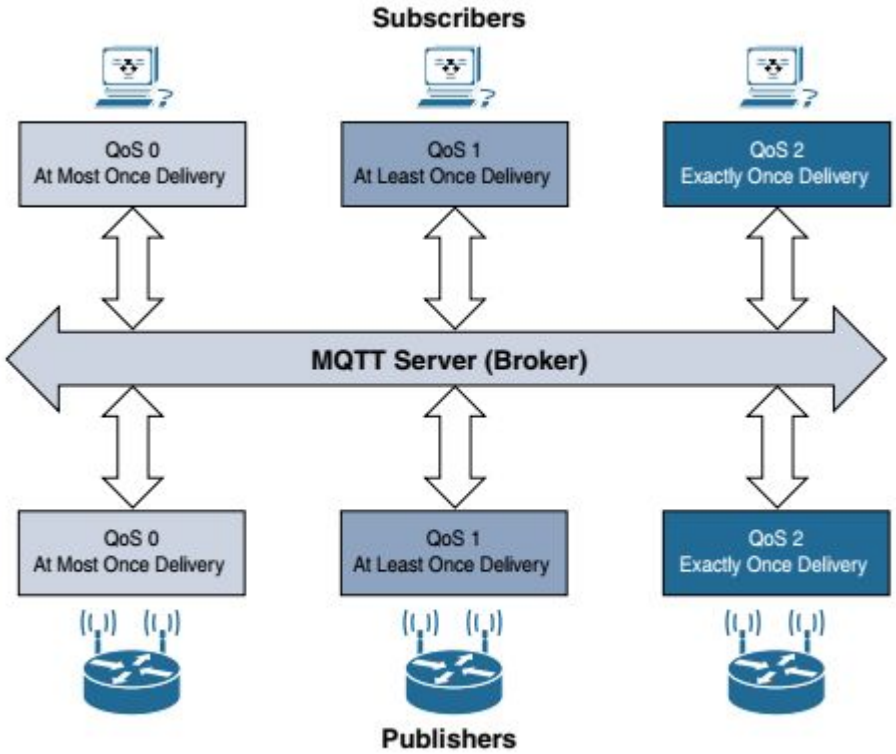
- PINGREQ/PINGRESP control packets are used to validate the connections between the client and server. Similar to ICMP pings that are part of IP, they are a sort of keepalive that helps to maintain and check the TCP session.
- Securing MQTT connections through TLS is considered optional because it calls for more resources on constrained nodes. When TLS is not used, the client sends a clear-text username and password during the connection initiation. MQTT server implementations may also accept anonymous client connections (with the username/password being “blank”). When TLS is implemented, a client must validate the server certificate for proper authentication. Client authentication can also be performed through certificate exchanges with the server, depending on the server configuration.
- The MQTT protocol offers three levels of quality of service (QoS). QoS for MQTT is implemented when exchanging application messages with publishers or subscribers, and it is different from the IP QoS that most people are familiar with. The delivery protocol is symmetric. This means the client and server can each take the role of either sender or receiver. The delivery protocol is concerned solely with the delivery of an application message from a single sender to a single receiver.

These are the three levels of MQTT QoS:

- **QoS 0:** This is a best-effort and unacknowledged data service referred to as “at most once” delivery. The publisher sends its message one time to a server, which transmits it once to the subscribers. No response is sent by the receiver, and no retry is per formed by the sender. The message arrives at the receiver either once or not at all.
- **QoS 1:** This QoS level ensures that the message delivery between the publisher and server and then between the server and subscribers occurs at least once. In PUBLISH and PUBACK packets, a packet identifier is included in the variable header. If the message is not acknowledged by a PUBACK packet, it is sent again. This level guarantees “at least once” delivery.
- **QoS 2:** This is the highest QoS level, used when neither loss nor duplication of mes sages is acceptable. There is an increased overhead associated with this QoS level because each packet contains an optional variable header with a packet identifier. Confirming the receipt of a PUBLISH message requires a two-step acknowledgement

process. The first step is done through the PUBLISH/PUBREC packet pair, and the second is achieved with the PUBREL/PUBCOMP packet pair. This level provides a “guaranteed service” known as “exactly once” delivery, with no consideration for the number of retries as long as the message is delivered once.

As mentioned earlier, the QoS process is symmetric in regard to the roles of sender and receiver, but two separate transactions exist. One transaction occurs between the publishing client and the MQTT server, and the other transaction happens between the MQTT server and the subscribing client. Figure 6-13 provides an overview of the MQTT QoS flows for the three different levels.



- As with CoAP, a wide range of MQTT implementations are now available. They are either published as open source licenses or integrated into vendors' solutions, such as Facebook Messenger. For more information on MQTT implementations, see either the older MQTT.org site, at http://mqtt.org, or check out the MQTT community wiki, at https://github.com/mqtt/mqtt.github.io/wiki.

Note A free tool for working and experimenting with MQTT is MQTT.fx (shown in Figure 6-12). For more information on MQTT.fx, see www.mqttfx.org.

- In summary, MQTT is different from the "one-to-one" CoAP model in its "many-to-many" subscription framework, which can make it a better option for some deployments.
- MQTT is TCP-based, and it ensures an ordered and lossless connection. It has a low overhead when optionally paired with UDP and flexible message format, supports TLS for security, and provides for three levels of QoS.
- This makes MQTT a key application layer protocol for the successful adoption and growth of the Internet of Things.

Table 6-3 *Comparison Between CoAP and MQTT*

Factor	CoAP	MQTT
Main transport protocol	UDP	TCP
Typical messaging	Request/response	Publish/subscribe
Effectiveness in LLNs	Excellent	Low/fair (Implementations pairing UDP with MQTT are better for LLNs.)
Security	DTLS	SSL/TLS
Communication model	One-to-one	many-to-many
Strengths	Lightweight and fast, with low overhead, and suitable for constrained networks; uses a RESTful model that is easy to code to; easy to parse and process for constrained devices; support for multicasting; asynchronous and synchronous messages	TCP and multiple QoS options provide robust communications; simple management and scalability using a broker architecture
Weaknesses	Not as reliable as TCP-based MQTT, so the application must ensure reliability.	Higher overhead for constrained devices and networks; TCP connections can drain low-power devices; no multicasting support

MQTT-SN (MQTT for Sensor Networks)

- Due to the large number of sensors in IOT Sensor networks these sensors will be mainly wireless.
- The main characteristics of these networks that drove the design are:
- Low Power battery operated sensors with very limited processing power and storage.
- Limited payload size
- Not always on (sleeping)
- **MQTT-SN (MQTT for Sensor networks) was designed specifically to work on wireless networks,** and , as far as possible, to work in the same way as MQTT.
- It uses the same publish/subscribe model and can be considered as a version of MQTT.

MQTT-SN vs MQTT

- The main differences involve:
- **Reducing the size of the message payload**
- Removing the need for a permanent connection by using **UDP** as the transport protocol.

The **MQTT-SN** specification lists these differences.

- Connect message split into three messages two are optional and are used for the will message
- Topic id’s used in place of topic names.
- Short Topic names
- Pre-defined topics.
- Discovery process to let clients discover the Gateway
- Will Topic and messages can be changed during the session
- Off line keep alive procedure for **sleeping clients.**

Architecture and Components

- The specification lists three components:
- MQTT-SN client
- MQTT-SN Gateway
- MQTT-SN forwarder.
- What seems to be missing is a broker/server as in the MQTT sense.
- However the architectural diagram in the specification does appear to show one and the RSMB server does implement one.

4 MQTT-SN Architecture

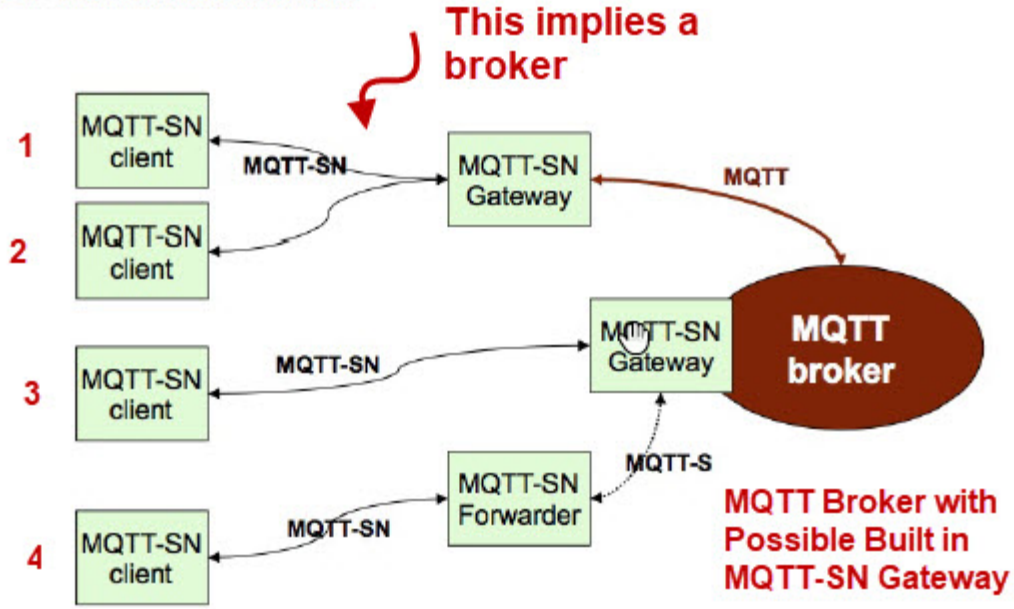


Figure 1: MQTT-SN Architecture

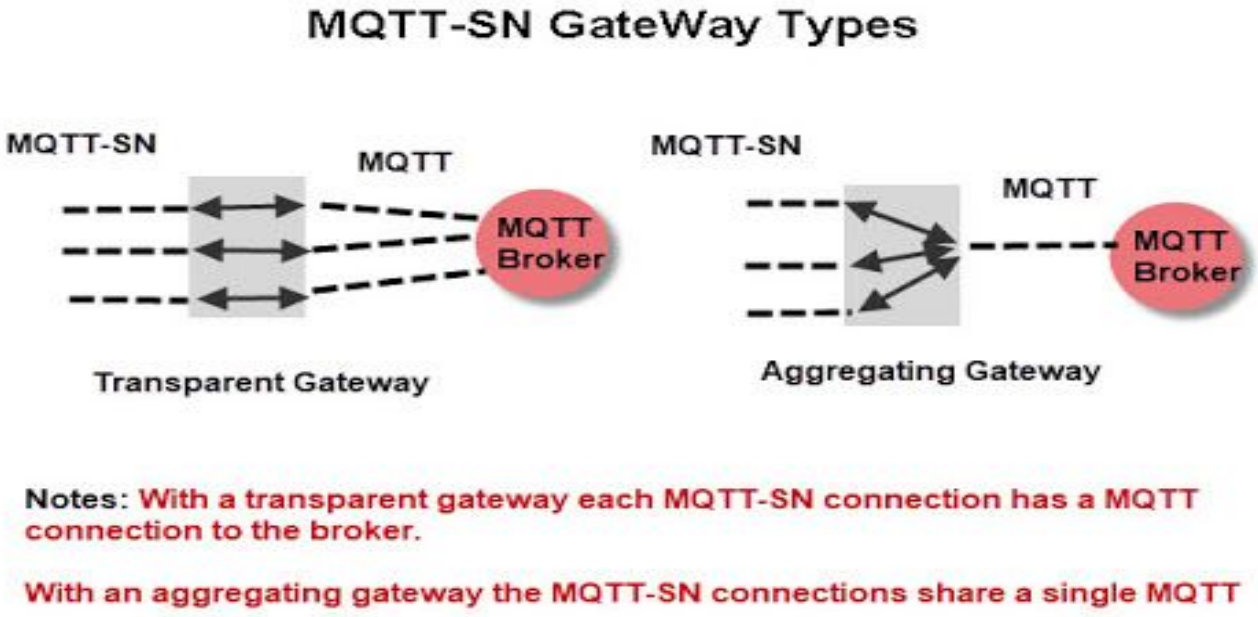
Is The connection between client 1 and client 2 MQTT-SN? or MQTT-SN>MQTT>MQTT-SN ?

GateWay Types

The specification defines two gateway types.

A **transparent gateway** were each **MQTT-SN** connection has a corresponding **MQTT** connection. This is the easiest type to implement.

An **aggregating gateway** were multiple MQTT-SN connections share a single **MQTT** connection



MQTT-SN Operation

- Although **MQTT-SN** uses **UDP** as the transport protocol and not TCP it is designed, as far as possible., to work in the same way as MQTT.
- In that regard MQTT-SN usually requires a connection to the broker before it can send and receive messages.
- This connection is, in effect, a **virtual connection**.
- However there is a mode of operation that doesn't require a connection, but this doesn't work with pure Gateways. (see below).

QOS Levels

- **MQTT-SN** supports QOS 0,1,2 as per MQTT, but it also supports a special publish QOS of 3 or -1.
- **Note:** it is known as QOS -1 but the QOS flag in the message is set to 11 or decimal 3.
- Publishing messages with a QOS of -1 or 3 doesn't require an initial connection to have been set up, and requires the use of short topic names or pre-defined topic ids.

Subscribing to MQTT-SN Topics

- You can **subscribe to a topics** using 3 different formats:
- A long topic name as per MQTT e.g. **house/sensor1**
- A short topic name of 2 characters only e.g. **s1**
- A pre-defined **topic id** (integer) e.g. 1
- **Wildcards** can be used as per MQTT, but they only make sense for long topic names.
- **Note:** Predefined topics are defined on the Gateway and client using a list.

Publishing Messages With an Established Connection

- You can publish a message using:
- A topic ID
- A short topic name- 2 characters
- You can a get a **topic ID** by either:
- Subscribing to the long topic name.
- Registering the Long topic name.
- Using a pre-defined topic-id
- The Subscribe and Register functions both return a **topic ID** that you use in place of the long topic name when publishing.
- **Topic ids** are assigned to each client, and they are not broker wide. For example:
- A client may subscribe to topic **house/bulb1** and get a **topic ID** of 1.
- A second client may subscribe to topic house/bulb2 and get a **topic ID** of 1.
- Topic id 1 for client 2 refers to **house/bulb2,** and for client1 topic id 1 refers to **house/bulb1**
- **Note**: When using QOS of 1 or 2 you need to wait for a PUBACK message before you publish a new message.

- So the format is
- PUB
- Wait PUBACK
- PUB
- etc
- **In MQTT_SN you can get a PUBACK even to a QOS level 0 message.**
- This is because it is used to return a error if the publish was rejected.

Publishing Messages Without a Connection

- You can publish messages without first creating a connection by using **QOS of -1 or 3**.
- You can publish a message without registering a topic or subscribing to a topic using:
- A pre configured topic ID
- A short topic name – 2 characters.
- Published messages aren't acknowledged.
- This mode of publish is ideal for simple sensors.

Gateway Discovery

- MQTT-SN clients have the ability to discover brokers/gateways.
- There are two mechanisms used:
- Advertising by a broker or Gateway
- A Search by the client
- Both methods use a **multicast packet**.

How it Works

- The broker of gateway advertises on a multicast address.
- In the packet is the name of the advertising Gateway the advertising duration and the Gateway number.
- The client maintains a list of active gateways and can use the duration to determine if a gateway is still active.
- The client must be listening on the multicast address.
- Alternatively the client can search for a gateway by sending a **search packet** on a multicast address.
- This can be answered by either another client or a Gateway.
- However for some reason the Gateway response doesn't contain the Gateway address, but a response from another client does.

Topic Registration

- A client can register a topic with a broker and a broker can also register a topic with a client.

Client Registration

- The client registers a long topic name with the broker and the broker returns a **topic ID** that the client uses to refers to that topic name when it publishes messages.

Broker or Gateway Registration

- If a client subscribes to a wildcard topic e.g. house/#
- What happens when a broker receives a publish for topic **house/bulb2**, as the client doesn't have a **topic ID** for **house/bulb2**.
- In the case the broker assigns a **topic ID** and notifies the client using topic registration.

MQTT-SN Brokers and Gateways

- To test **MQTT-SN** you will need a broker. There currently aren't many NQTT-SN brokers available.
- The RSMB broker developed by Ian Craggs of IBM was the first and was the basis for the Mosquitto MQTT broker.
- This broker hasn't been actively developed for many years but I'm hopeful it will be soon.
- At the moment predefined topics and sleeping clients aren't implemented.
- The broker functions as a **MQTT-SN broker** and MQTT-SN to MQTT Gateway.

Paho Eclipse Gateway

- This started I believe as a fork of RSMB and works as a pure Gateway.

XMPP

- Extensible Messaging and Presence Protocol (XMPP) is a messaging protocol that was designed originally for chatting and message exchange applications. It was standardized by IETF more than a decade ago. Hence, it is well known and has proven to be highly efficient over the internet. Recently, it has been reused for IoT applications as well as a protocol for SDN. This reusing of the same standard is due to its use of XML which makes it easily extensible.

- XMPP supports both publish/ subscribe and request/ response architecture and it is up to the application developer to choose which architecture to use. It is designed for near real-time applications and, thus, efficiently supports low-latency small messages.

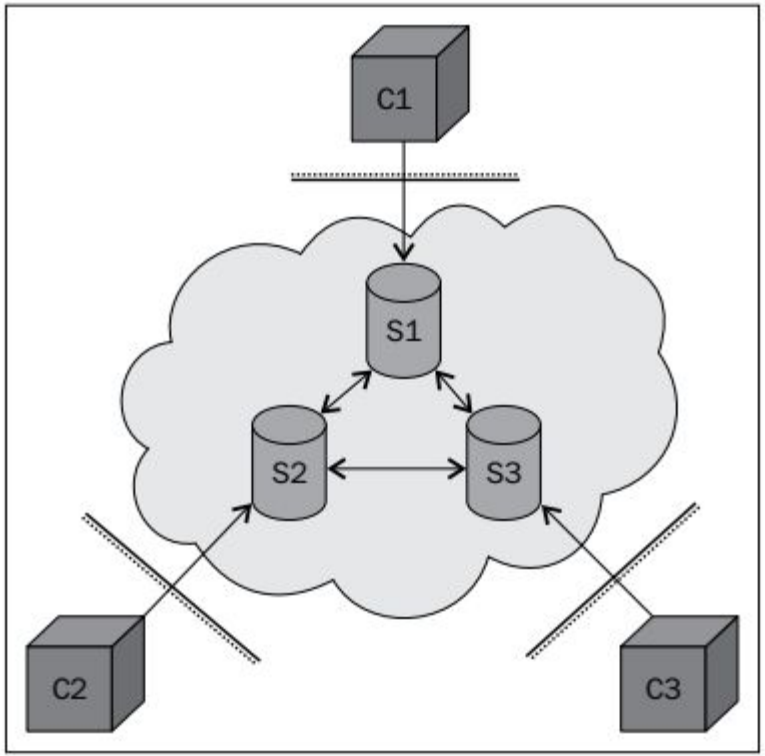
- It does not provide any quality of service guarantees and, hence, is not practical for M2M communications. Moreover, XML messages create additional overhead due to lots of headers and tag formats which increase the power consumption that is critical for IoT application. Hence, XMPP is rarely used in IoT but has gained some interest for enhancing its architecture in order to support IoT applications.

- The XMPP protocol also uses message brokers to bypass firewall barriers. But apart from the publish/subscribe pattern, it also supports other communication patterns, such as point-to-point request/response and asynchronous messaging, that allow you to have a richer communication experience.

Federating for global scalability

- The XMPP architecture builds on the tremendous success and global scalability of the **Simple Mail Transfer Protocol** (**SMTP**). The difference is that XMPP is designed for real-time instantaneous messaging applications, where smaller messages are sent with as little latency as possible and without any persistence.
- XMPP uses a federated network of XMPP servers as message brokers to allow clients behind separate firewalls to communicate with each other.
- Each server controls its own domain and authenticates users on that domain.
- Clients can communicate with clients on other domains through the use of federation where the servers create connections between themselves in a secure manner to interchange messages between their domains. It is this federation that allows you to have a globally scalable architecture.
- All of this happens at the server level, so there is nothing that clients need to worry about. They only need to ensure that they maintain the connection with their respective servers, and through the servers, each of them will have the possibility to send messages to any other client in the federated network.
- It is this architecture of federation that makes XMPP scalable and allows you to make billions of devices communicate with each other in the same federated network.

The following illustration shows how clients (**C1**, **C2**, and **C3**) behind firewalls connect to different servers (**S1**, **S2**, and **S3**) in a federated network to exchange messages:



A small federated XMPP network

Providing a global identity

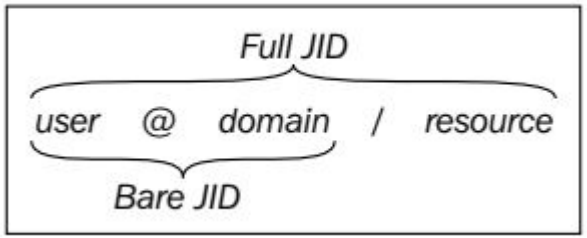
XMPP servers do more than relay messages between clients. They also provide each client with an authenticated identity. If the server is a public server in the global federated network of XMPP servers, it is a global identity.

When clients connect, the servers make sure the clients authenticate themselves by providing their corresponding client credentials, which would consist of a username and password. This authentication is done securely using an extensible architecture based on **Simple Authentication and Security Layer** (**SASL**).

The connection can also be switched over to **Transport Layer Security** (**TLS**) through negotiation between the client and the server, encrypting the communication between them. The identity of the client is often called XMPP address or **Jabber ID** (**JID**). The XMPP protocol was developed in a project named Jabber.

For this reason, many terminologies retain this name.

Each connection is also bound to a specific resource, which is normally a random string. Together, the username, domain name, and resource constitute the full JID of a connection, while only the username and domain name constitute the bare JID of an account.



Authorizing communication

Another important reason for using XMPP servers to relay communication instead of serverless peer-to-peer technologies is to assure the clients that only authorized communication will be relayed. This feature comes in handy, especially for small devices with limited decision-making capabilities. The server does so by ensuring that the full JID identifier instead of only the bare JID identifier is used to communicate with the application or device behind it. The reason is twofold:

First, multiple clients might use the same account at the same time. You need to provide the resource part of the full JID for the XMPP Server to be able to determine which connection the corresponding message should be forwarded to. Only this connection will receive the message. This enables the actual clients to have direct communication between them.

Second, only trusted parties (or friends) are given access to the resource part once the thing or application is connected. This means that, in turn, only friends can send messages between each other, as long as the resource parts are sufficiently long and random so they cannot be guessed and the resource part is kept hidden and not published somewhere else.

Sensing online presence

- To learn about the resource part of a corresponding client, you send a presence subscription to its bare JID. If accepted by the remote client, you will receive presence messages every time the state of the contact is changed, informing you whether it is online, offline, away, busy, and so on.
- In this presence message, you will also receive the full JID of the contact. Once a presence subscription has been accepted by the remote device, it might send you a presence subscription of its own, which you can either accept or reject.
- If both the parties accept it and subscribe to the presence from each other, then parties are said to be friends.
- XMPP servers maintain lists of contacts for each account and their corresponding presence subscription statuses. These lists are called **rosters**.
- The client only needs to connect and then receive its roster from the server. This makes it possible to move the application between physical platforms and unnecessary to store contact information in the physical device.

Using XML

XMPP communication consists of bidirectional streams of XML fragments. The reason for using XML has been debated since it affects message sizes negatively when compared to binary alternatives, but it has many positive implications as well. These can be listed as follows:

• Having a fixed content format makes the interchange and reuse of data simpler

• XML is simple to encode, decode, and parse, making data telegrams well-defined

• Using a text format makes telegrams readable by humans, which makes documentation and debugging simpler

• XML has standard tools for searching validation and transformation, which permits advanced operations and analysis to be performed on data without previous knowledge about message formats

• Through the use of XML namespaces, messages can be separated between protocol extensions and versioning is supported

[In cases where the message size is important, there are methods in XMPP that help compress XML to very efficient binary messages using Efficient XML Interchange (EXI).]

Communication patterns

XMPP supports a rich set of communication patterns. It does this by providing three communication primitives called **stanzas.**

We've already presented the first of these, **the presence stanza**. This is used to send information about oneself to interested and authorized parties.

The second is the **message stanza**. This is used to send asynchronous messages to a given receiver.

The third is the **iq stanza**, short for information/query. This stanza is used to provide a request/response communication pattern. A request is sent to a given receiver, which returns a response or an error, as appropriate.

There are four different kinds of receivers of stanzas.

First, you have the peer. To communicate with a peer, you provide the full JID of the peer connection as the destination address of your stanza.

Then you have a server. To communicate with a server directly, you use the domain name of the server as the destination address.

A server might host server components of various kinds. These might be internal or external components hosted by external applications. These components are addressed using a corresponding subdomain name and can be dynamically discovered using simple requests to the server.

Finally, you have a contact. To communicate with a contact, which is implicitly handled by your server and the server handling the contact, depending on the type of message, you need to use the base JID of the contact as the address.

- **XMPP Extension Protocols** (**XEPs**)
- An alternative way to connect to an XMPP server is by using **Bidirectional streams Over Synchronous HTTP** (**BOSH**). This allows clients with access to only the HTTP protocol to use XMPP as well. Some servers also publish XMPP over web socket interfaces. This makes it possible to access the XMPP network for clients, such as web browsers and so on.

AMQP

- The Advanced Message Queuing Protocol (AMQP) is another session layer protocol that was designed for financial industry. It runs over TCP and provides a publish/ subscribe architecture which is similar to that of MQTT.
- The difference is that the broker is divided into two main components: exchange and queues, as shown in Figure
- The exchange is responsible for receiving publisher messages and distributing them to queues based on pre-defined roles and conditions. Queues basically represent the topics and subscribed by subscribers which will get the sensory data whenever they are available in the queue.

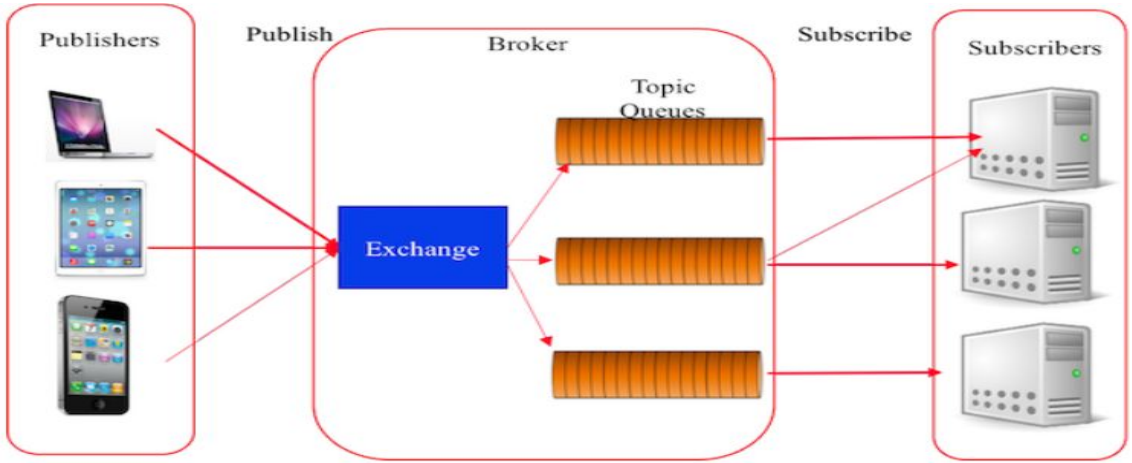


Figure 6: AMQP Architecture

- Sits over TCP for reliable point-to-point connection.
- Endpoints acknowledge acceptance for each message.
- Focus on tracking of all messages and ensuring that each is delivered as intended, regardless of failure or reboot.
- Mostly used in business messaging.
- In addition to being a common messaging protocol between systems, AMQP is designed with a few key features in mind:
- Security
- Reliability
- Interoperability
- Standard
- Open

These key features of AMQP provide a few benefits to business who utilize this protocol with their systems. Some of these benefits include:

- Lower development costs when integrating software systems due to increased interoperability
- Higher message transport security without relying on the developers to “get it right.”
- Lower chance of vendor lock-in
- Broader availability of tools and libraries that support the messaging protocol

- A single common messaging protocol allows for greater system integration across development platforms and frameworks. Here's a list of some libraries for different programming languages that support the AMQP protocol:
- C# – AMQP .NET Lite
- Java – Apache Qpid Java Message Service client, and IIT Software SwiftMQ Java client
- Python – Apache Qpid Proton-Python
- PHP – Apache Qpid Proton-PHP
- C – Apache Qpid Proton-C
- In addition to the libraries listed above, a quick online search yields many open source projects in various programming languages and platforms that are built to communicate using AMQP.

Introduction to Contiki- Practical demo

- **Contiki** is an operating system is an operating system for networked, memory-constrained systems with a focus on low-power wireless Internet of Things is an operating system for networked, memory-constrained systems with a focus on low-power wireless Internet of Things devices. Extant uses for Contiki include systems for street lighting, sound monitoring for smart cities, radiation monitoring, and alarms. It is open-source software is an operating system for networked, memory-constrained systems with a focus on low-power wireless Internet of Things devices. Extant uses for Contiki include systems for street lighting, sound monitoring for smart cities, radiation monitoring, and alarms. It is open-source software released under a BSD license.
- Contiki was created by Adam DunkelsDunkels in 2002 and has been further developed by a worldwide team of developers from Texas Instruments, Atmel, Cisco, ENEADunkels in 2002 and has been further developed by a worldwide team of developers from Texas Instruments, Atmel, Cisco, ENEA, ETH ZurichDunkels in 2002 and has been further developed by a worldwide team of developers from Texas Instruments, Atmel, Cisco, ENEA, ETH Zurich, Redwire, RWTH Aachen UniversityDunkels in 2002 and has been further developed by a worldwide team of developers from Texas Instruments, Atmel, Cisco, ENEA, ETH Zurich, Redwire, RWTH Aachen University, Oxford University, SAP, Sensinode, Swedish Institute of Computer Science, ST Microelectronics, Zolertia, and many others. Contiki gained popularity because of its built in

A full installation of Contiki includes the following features:

Multitasking kernel

Optional per-application preemptive multithreading

Protothreads

Internet Protocol SuiteInternet Protocol Suite (TCP/IP) networkingInternet Protocol Suite (TCP/IP) networking, including IPv6

Windowing system and GUI

Networked remote display using Virtual Network Computing

A web browser (claimed to be the world's smallest)

Personal web server

Simple telnet client

Screensaver

Contiki is supported by popular SSL/TLSContiki is supported by popular SSL/TLS libraries such as wolfSSL, which includes a port in its 3.15.5 release.

Installation

1. Install Linux OS (Ubuntu recommended). You can install either the 32 bit or 64 bit

In this “How to” post I’m specifically using Ubuntu 18.04 (64 bit)

2. Open up your terminal (Ctrl + Alt + T) and run the following commands

wget https://github.com/contiki-os/contiki/archive/3.0.zip

unzip 3.0.zip

mv contiki-3.0 Contiki

3. Install gcc-arm-none-eabi

sudo apt-get install gcc-arm-none-eabi

sudo apt-get install build-essential binutils-msp430 gcc-msp430 msp430-libc msp430mcu mspdebug openjdk-8-jdk openjdk-8-jre ant libncurses5-dev lib32ncurses5 gdb-multiarch