

# Internet of Things Security

## Lecture 4: Application Protocols and Security Challenges

Mehmoona Jabeen

[Mehmoona.jabeen@au.edu.pk](mailto:Mehmoona.jabeen@au.edu.pk)

Department of Cyber Security, Air University

# Lecture Outlines

- Introduction to Application Protocols
- COAP
- Security Challenges in COAP
- MQTT
- COAP vs MQTT and HTTP

# Need of IoT Application Protocols

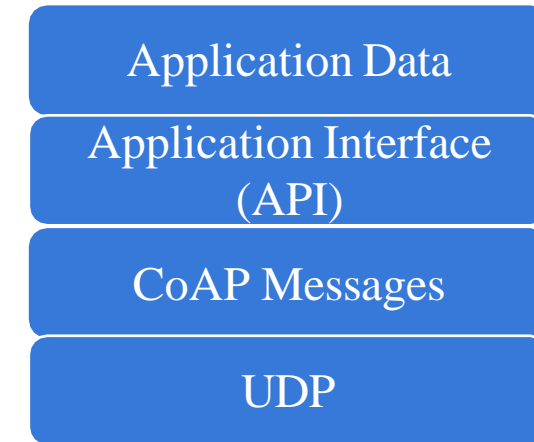
- To enable web-based services in constrained wireless
- Networks in
  - 8-bit micro-controllers
  - limited memory
  - low-power networks
- Problem:
  - WEB solutions like HTTP are hardly applicable
- Solution:
  - re-design web-based services for constrained networks
    - Request-Response Model
    - Publish-Subscribe Model

# Application Protocols

1. Constrained Application Protocol (CoAP)
  - It is designed for low-power devices and is usually paired with UDP, which makes it highly efficient.
2. Message Queuing Telemetry Transport (MQTT)
  - It's ideal for remote environments or applications with limited bandwidth.
  - MQTT uses a connection oriented publish/subscribe architecture, where MQTT applications can either publish (transmit) or subscribe to (receive) topics.
3. Advanced Message Queuing Protocol (AMQP)
  - AMQP is an open source protocol for Message-Oriented Middleware (MOM) and designed to facilitate communications between systems, devices, and applications from multiple vendors.
  - Opposed to MQTT it supports more routing options than just publish / subscribe on topics
  - but this flexibility comes with complexity on application setup and additional protocol overhead.
4. Extensive Messaging and Presence Protocol (XMPP)
  - XMPP is built on XML and was initially designed for instant messaging (IM).
  - It comes with an overhead for the exchange of presence information and is not optimized for memory constrained devices.

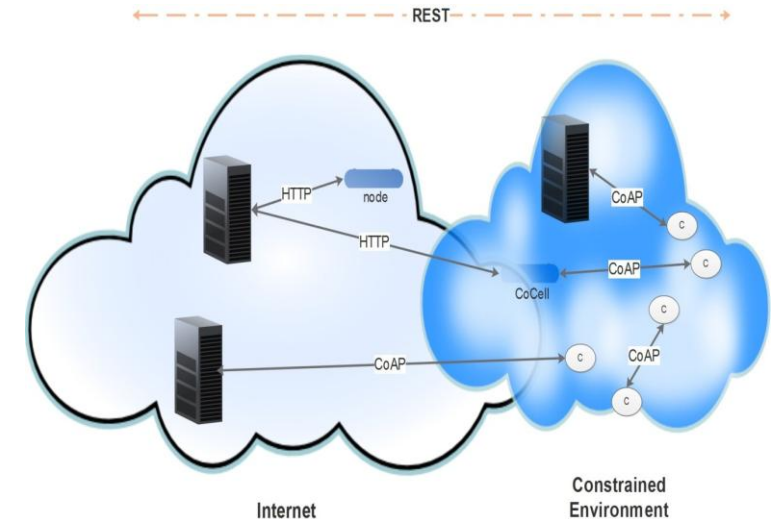
# Constrained Application Protocol (CoAP)

- CoAP is an IETF standard and the core protocol is defined in IETF RFC 7252.
- It is well suited for nodes that run on simple microcontrollers, with limited ROM and RAM.
- It works at the application layer of the TCP/IP stack and utilizes UDP as the underlying transport protocol.
- Features
  - It provides a simple discovery mechanism
  - Integration with Web is easy
  - It provides asynchronous message exchange
  - Uses URIs to define resources/services
  - Uses REST-like request/response model



# CoAP Features

- Web protocol fulfilling M2M requirements in constrained environments
- UDP binding with optional reliability supporting unicast and multicast requests.
- Asynchronous message exchanges.
- Low header overhead and parsing complexity.
- URI and Content-type support allowing devices to specify the format of the payload data exchange between them.
- Simple proxy and caching capabilities.



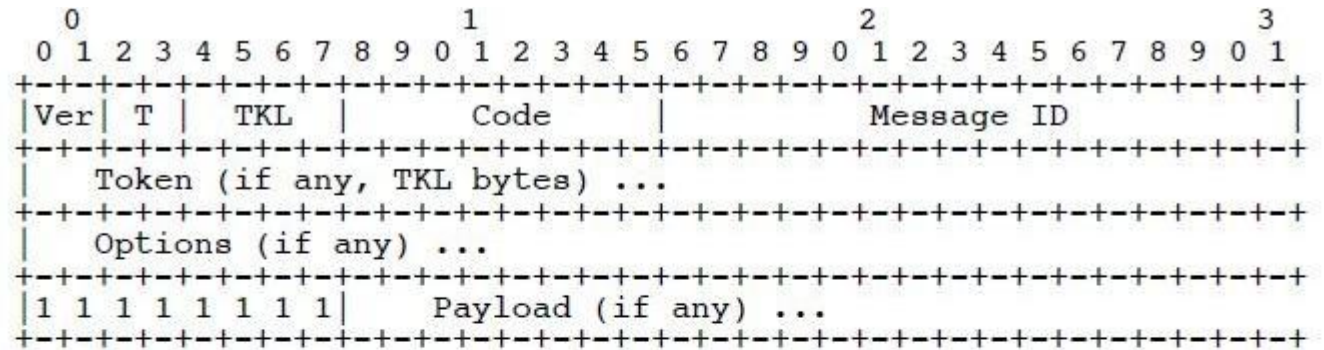
# CoAP Messages

The CoAP standard defines 4 different types of Messages:

1. Confirmable Message (CON) : This type of message requires that the receiver send an Acknowledgment message back to the sender for confirmation of the receipt of this message
2. Non-Confirmable Message (NON): Does not require any ACK. This is used when no Acknowledgment is required i.e. reliability is not important.
3. Acknowledgment Message (ACK) : This message is sent, by a receiver, as an Acknowledgment for a Confirmable message that is received from the sender
4. Reset Message (RST): This message is typically sent when the receiver is not able to process a Confirmable or Non-Confirmable message due to some error.

# CoAP Message Format

- CoAP messages are encoded in a simple binary format.
- The message format starts with a fixed-size 4-byte header.
- Followed by a variable-length Token value, which can be between 0 and 8 bytes long.



**Ver** - Version (1)

**T** - Message Type (Confirmable, Non-Confirmable, Acknowledgement, Reset)

**TKL**- Token Length, if any, the number of Token bytes after this header

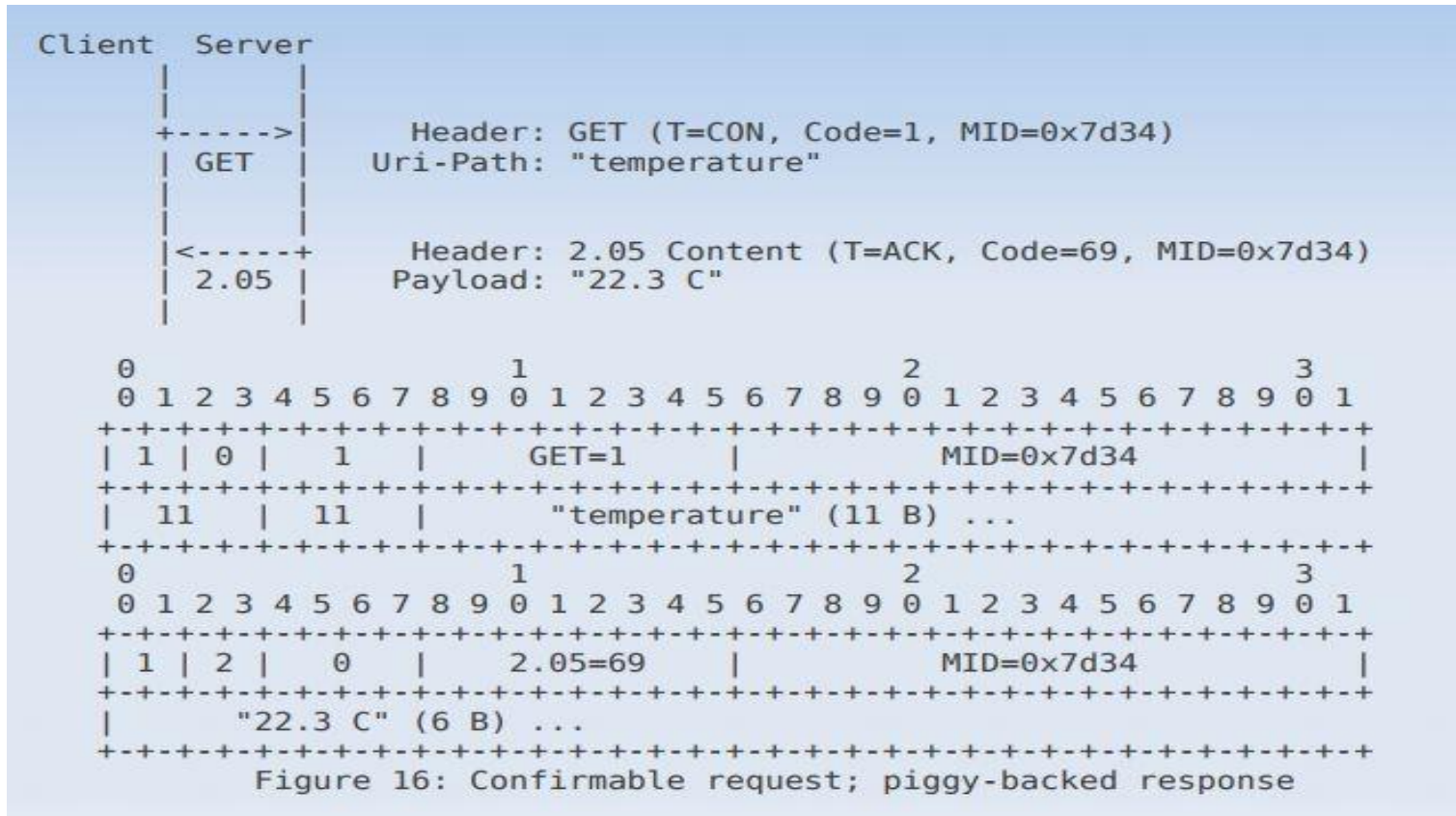
**Code** - Request Method (1-10) or Response Code (40-255)

**Message ID** - 16-bit identifier for matching responses

**Token** - Optional response matching token



# CoAP Message Format

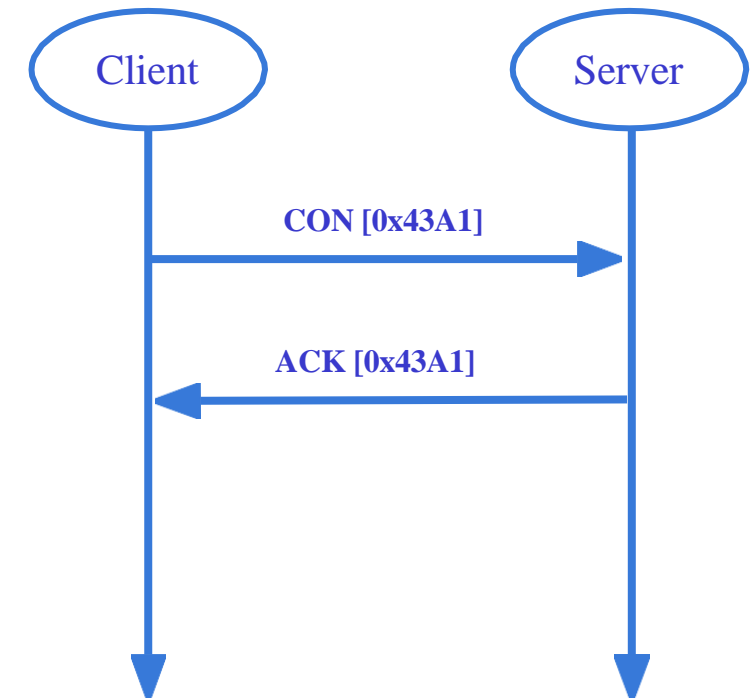


# CoAP Model

- Requests use Method Codes and Responses use response codes similar to HTTP but defined in binary format.
- Separate from the message type, a message may carry a request, a response, or be Empty.
- CoAP standard defines four Request methods - GET, PUT, POST, and DELETE.
- CoAP standard defines Response code similar to HTTP - 2.xx for success, 4.xx for client error, and 5.xx for server error.
- Unique 0-8 byte Tokens are used for identifying, mapping request and response much the same way Message-IDs are used to identify messages and map ACKs.
- A response to a request sent in a CON message can be piggy-backed in the ACK message or may be sent as a separate CON or NON message.

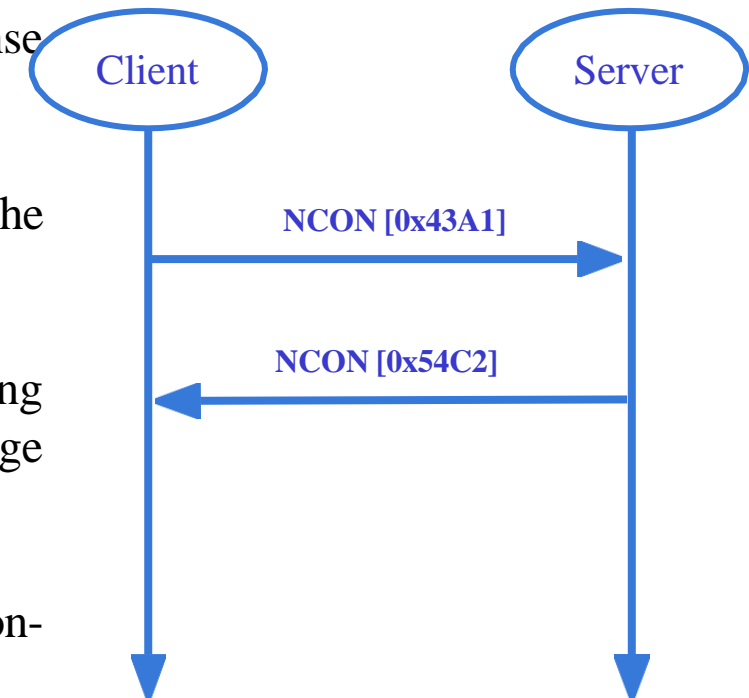
# CoAP Reliable Messaging

- The reliable transmission of a message is initiated by marking the message as Confirmable in the CoAP header.
- A Confirmable message always carries either a request or response, unless it is used only to elicit a Reset message, in which case it is Empty.
- A recipient **MUST** either (a) acknowledge a Confirmable message with an Acknowledgement message or (b) reject the message
- The sender retransmits the Confirmable message at exponentially increasing intervals, until it receives an acknowledgement (or Reset message) or runs out of attempts.

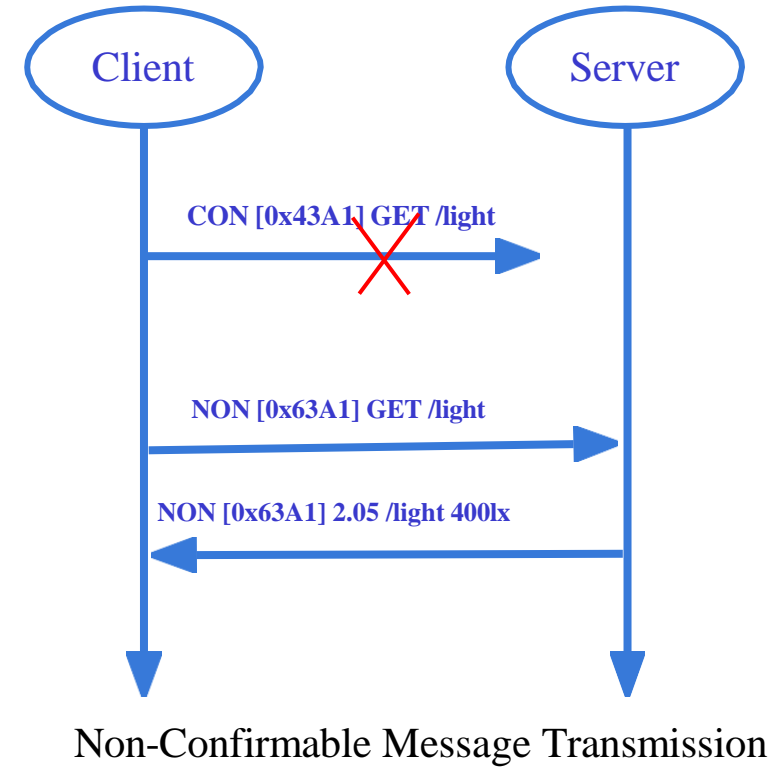
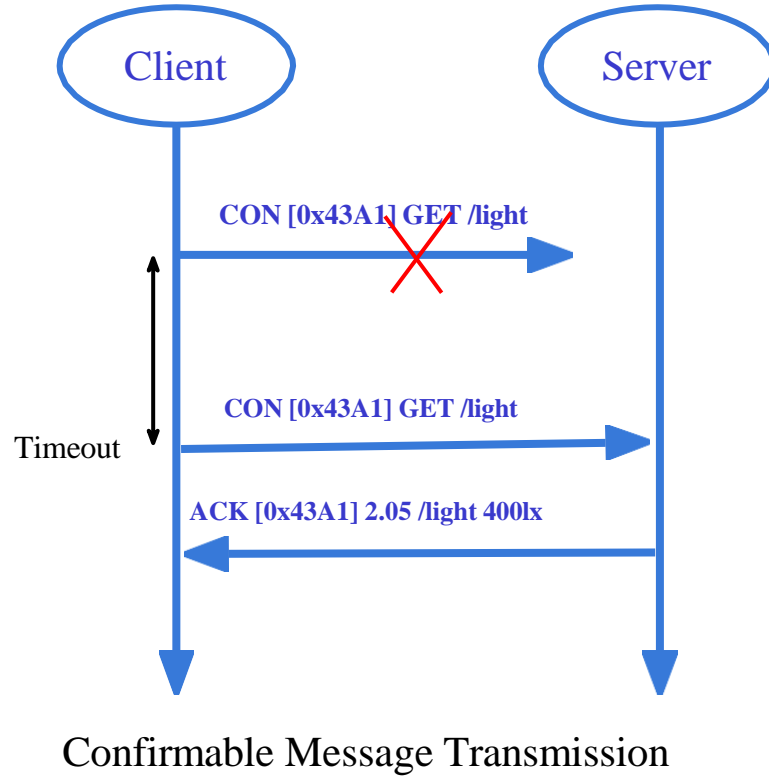


# CoAP Unreliable transmission

- ❑ A message can be transmitted less reliably by marking the message as Non-confirmable.
- ❑ A Non- confirmable message always carries either a request or response and MUST NOT be Empty.
- ❑ A Non-confirmable message MUST NOT be acknowledged by the recipient.
- ❑ Rejecting a Non- confirmable message MAY involve sending a matching Reset message, and apart from the Reset message the rejected message MUST be silently ignored.
- ❑ At the CoAP level, there is no way for the sender to detect if a Non-confirmable message was received or not.
- ❑ To enable the receiver to act only once on the message, Non-confirmable messages specify a Message ID as well.



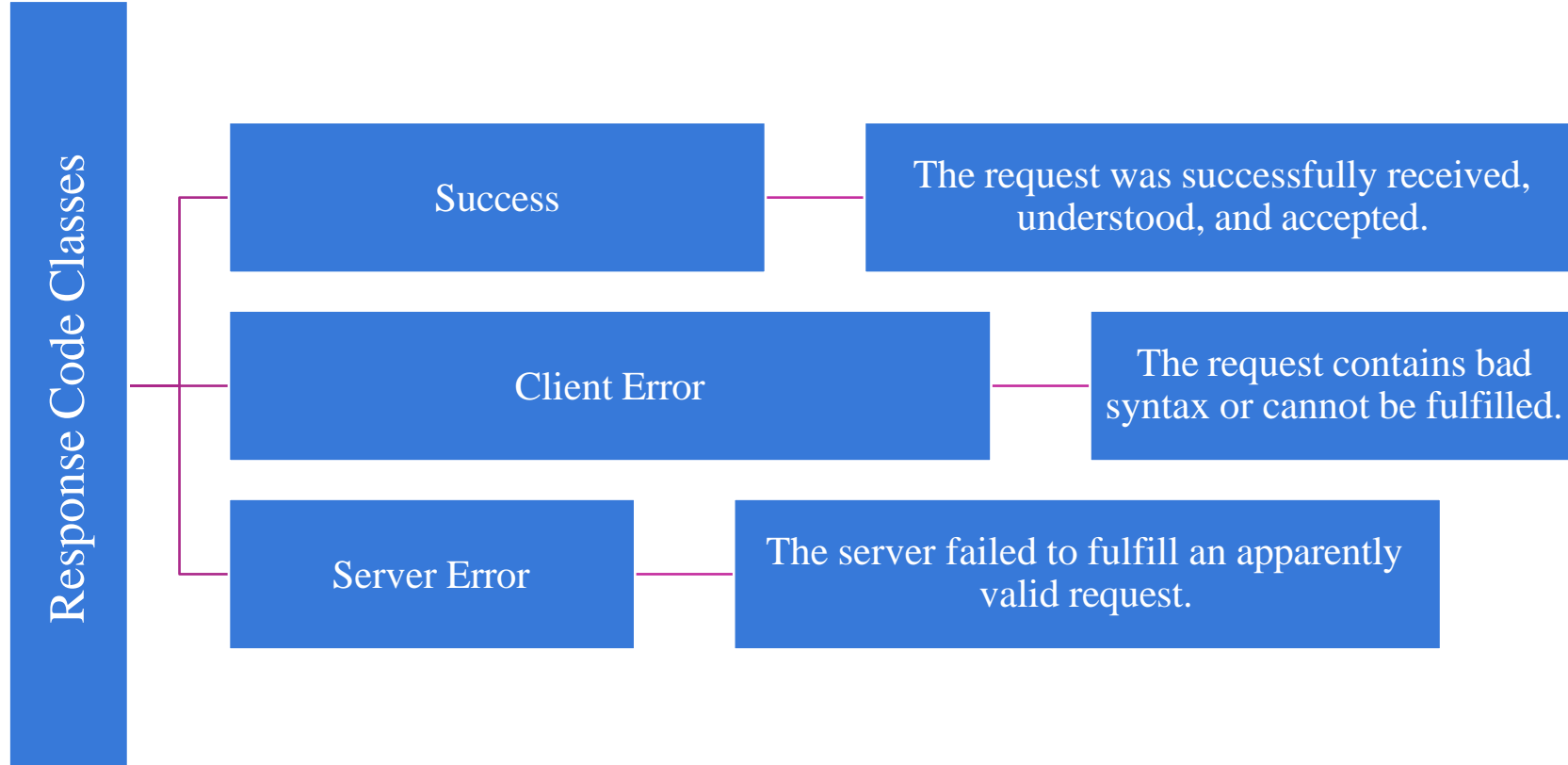
# Packet Loss



# CoAP Semantics

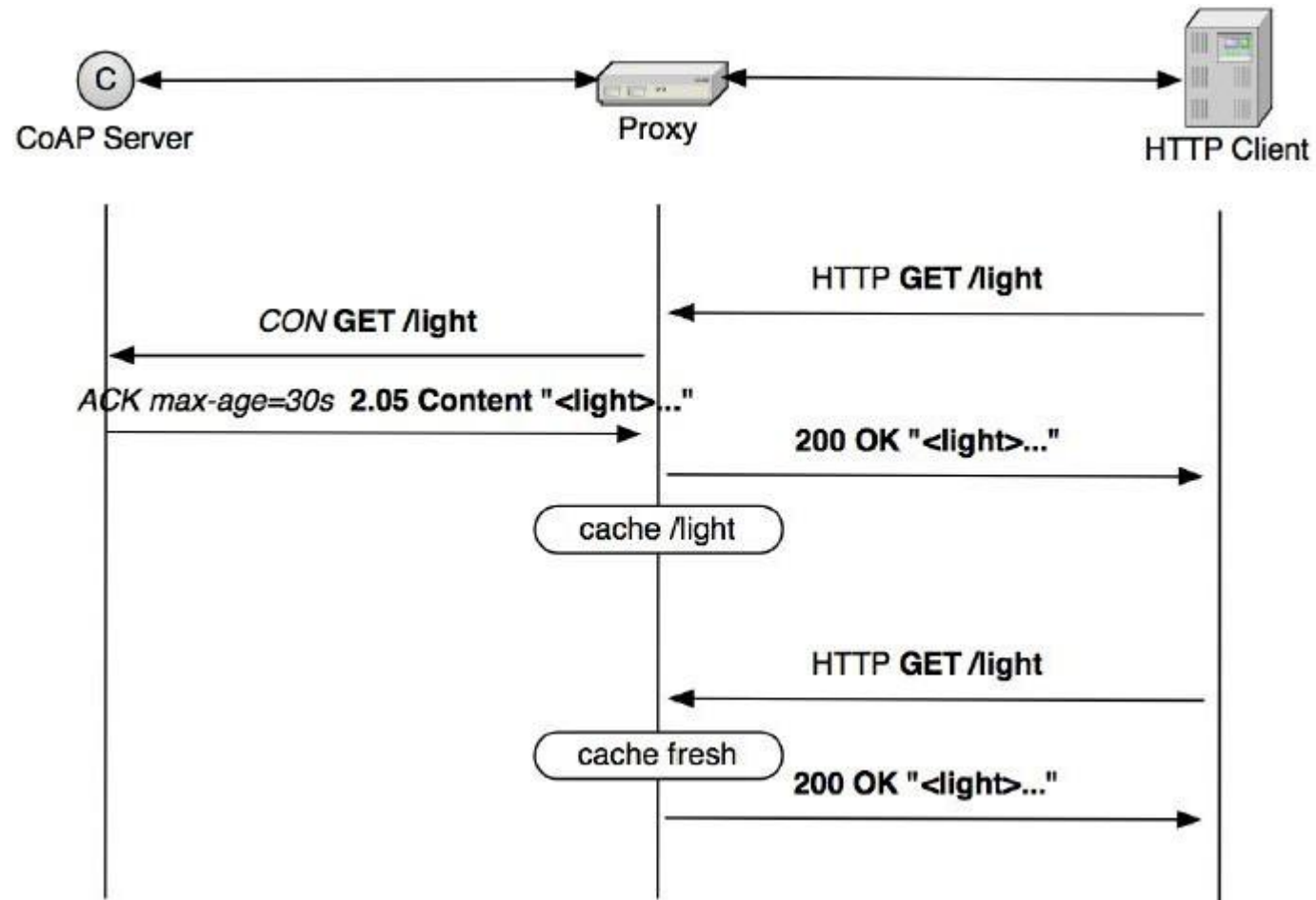
- After receiving and interpreting a request, a server responds with a CoAP response that is matched to the request by means of a client- generated token.
- A response is identified by the Code field in the CoAP header being set to a Response Code.
- Similar to the HTTP Status Code, the CoAP Response Code indicates the result of the attempt to understand and satisfy the request.
- The Response Code numbers to be set in the Code field of the CoAP header are maintained in the CoAP Response Code Registry.

# CoAP Semantics



Code	Description
2.01	Created
2.02	Deleted
2.03	Valid
2.04	Changed
2.05	Content
4.00	Bad Request
4.01	Unauthorized
4.02	Bad Option
4.03	Forbidden
4.04	Not Found
4.05	Method Not Allowed
4.06	Not Acceptable

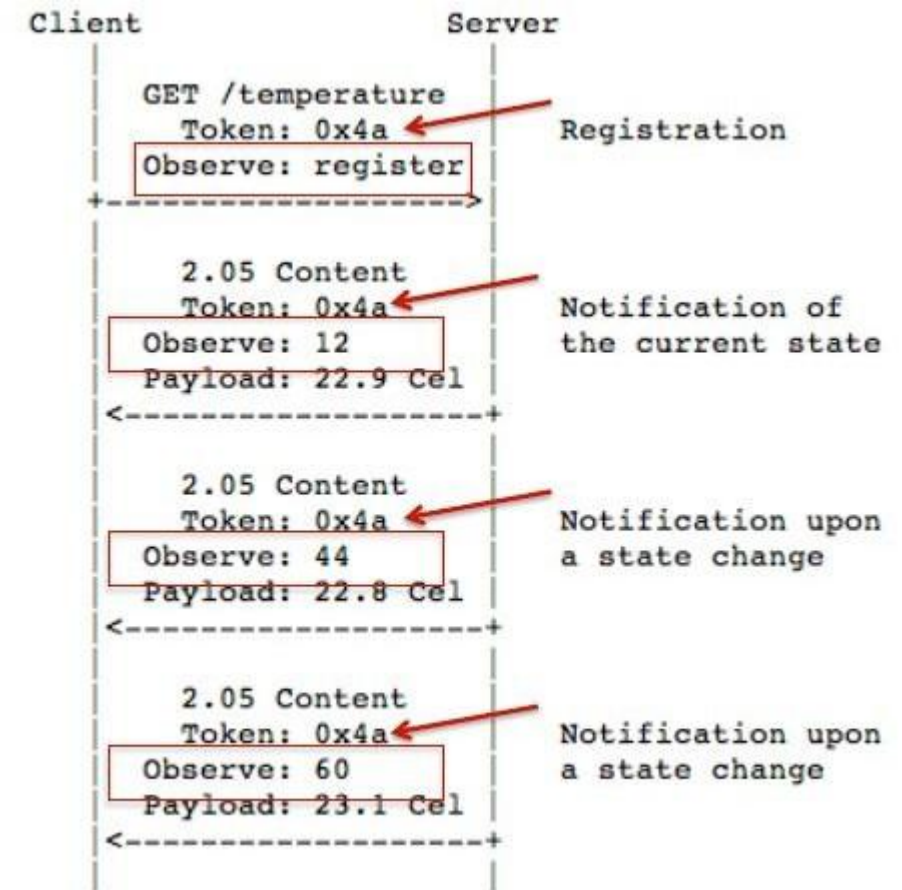
# Proxy and caching





# CoAP Observation

- Problem with REST
  - REST paradigm is often **PULL** type in which data is retrieved by issuing an explicit request.
  - Information in IoT may often be **periodic** or **event driven**, where a client may not know about the change in resource.
- Solution
  - Use Observation on CoAP resources
  - RFC 7641 defines an extension to CoAP protocol
  - Client may register for a particular resource to intimate in the state change of a resource
  - Server works as the observer and a change in resource state is sent to client without any request.



# CoAP vs HTTP

## HTTP

- a. HTTP is content oriented
- b. HTTP is based on TCP protocol that is not suitable for notification push services.
- c. As HTTP can use small script to integrate various resources and services.
- d. Interoperation provided by HTTP is the key point, for this, HTTP is employed in application level.
- e. Default port is 80

## CoAP

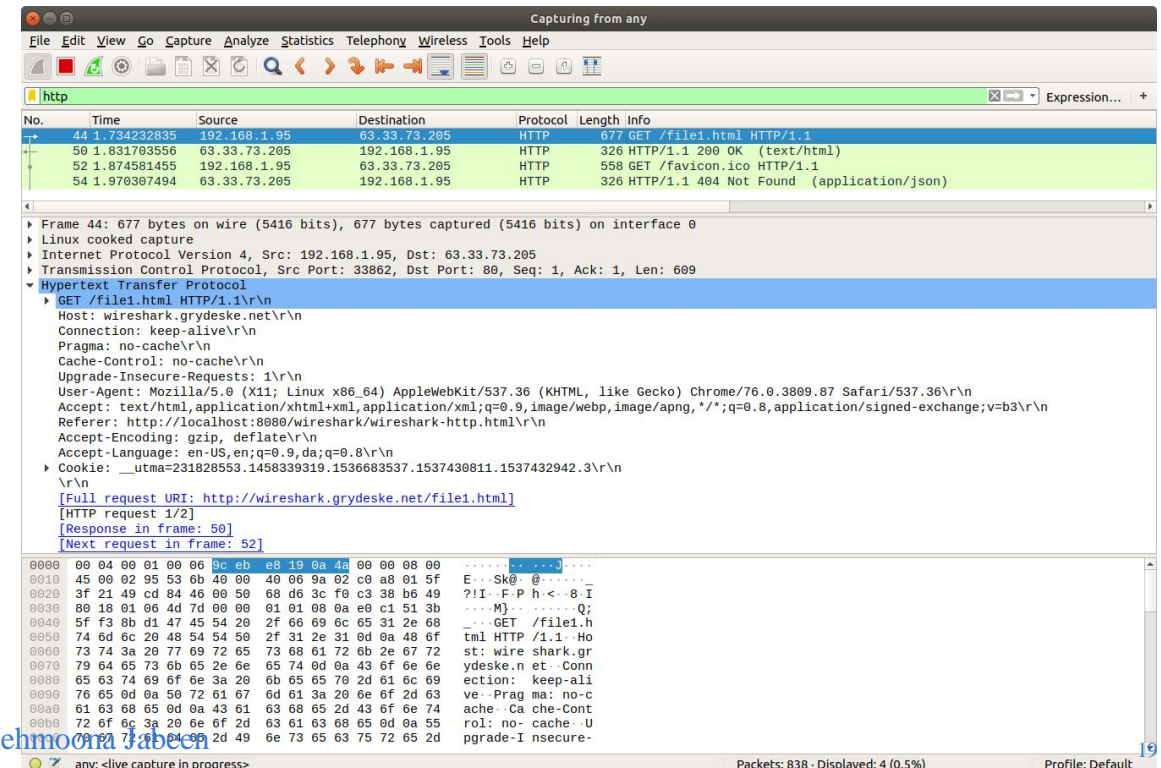
- a. CoAP is network-oriented protocol
- b. CoAP operates over UDP instead of TCP.
- c. CoAP provides URI, REST method such as GET, POST, PUT, and DELETE.
- d. CoAP allows IP multicast, which satisfies group communication for IoT.
- e. CoAP defines a retransmission mechanism and provides resource discovery
- f. Default port is 5683, 5684 (secure)

# Headers

## COAP

	Protocol	Length	Info
	CoAP	61	CON, MID:56717, POST, TKN:a8 6c 2b f2
	CoAP	54	ACK, MID:56717, 2.04 Changed, TKN:a8
	CoAP	62	CON, MID:56718, GET, TKN:6b e9 4d ec
	CoAP	54	ACK, MID:56718, 4.06 Not Acceptable,
	CoAP	64	CON, MID:56719, GET, TKN:0b d5 23 6c
	CoAP	54	ACK, MID:56719, 4.06 Not Acceptable,

## HTTP



Wireshark capture showing HTTP traffic. The packet list shows a GET request for /file1.html and a 200 OK response. The packet details pane shows the full request and response headers.

**Packet List:**

No.	Time	Source	Destination	Protocol	Length	Info
44	1.734232835	192.168.1.95	63.33.73.205	HTTP	677	GET /file1.html HTTP/1.1
50	1.831703556	63.33.73.205	192.168.1.95	HTTP	326	HTTP/1.1 200 OK (text/html)
52	1.874581455	192.168.1.95	63.33.73.205	HTTP	558	GET /favicon.ico HTTP/1.1
54	1.978397494	63.33.73.205	192.168.1.95	HTTP	326	HTTP/1.1 404 Not Found (application/json)

**Packet Details (Frame 44):**

- Frame 44: 677 bytes on wire (5416 bits), 677 bytes captured (5416 bits) on interface 0
- Linux cooked capture
- Internet Protocol Version 4, Src: 192.168.1.95, Dst: 63.33.73.205
- Transmission Control Protocol, Src Port: 33862, Dst Port: 80, Seq: 1, Ack: 1, Len: 609
- Hypertext Transfer Protocol
  - GET /file1.html HTTP/1.1\r\n
  - Host: wireshark.grydeske.net\r\n
  - Connection: keep-alive\r\n
  - Pragma: no-cache\r\n
  - Cache-Control: no-cache\r\n
  - Upgrade-Insecure-Requests: 1\r\n
  - User-Agent: Mozilla/5.0 (X11; Linux x86\_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/76.0.3809.87 Safari/537.36\r\n
  - Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,image/apng,\*/\*;q=0.8,application/signed-exchange;v=b3\r\n
  - Referer: http://localhost:8080/wireshark/wireshark-http.html\r\n
  - Accept-Encoding: gzip, deflate\r\n
  - Accept-Language: en-US,en;q=0.9,da;q=0.8\r\n
  - Cookie: \_\_utma=231828553.1458339319.1536683537.1537430811.1537432942.3\r\n

**Full request URI:** <http://wireshark.grydeske.net/file1.html>

**Response in frame: 50**

**Next request in frame: 52**

# Security Challenges in CoAP

- CoAP itself does not provide protocol primitives for authentication or authorization; where this is required, it can either be provided by communication security (i.e., IPsec or DTLS) or by object security.
- Path Traversal
  - Few CoAP implementations are not (or incorrectly) parsing or ignoring double dots “..” in the URI. The issue here is the same as with Web, for example, if the URI is bound to a file, there are chances of breaking out of CoAP root and accessing or manipulating other files.
- Cross Protocol Attacks
  - The CoAP protocol has tight coupling with HTTP in many fronts including request/response format, caching, proxying, etc.
  - This also gives rise to opportunities for HTTP-to-CoAP translation/proxying and vice versa.
  - Due to the similarity, some of the traditional attacks that affect HTTP today could very well apply to CoAP now or in the future.

# Security Challenges in CoAP

- Attacking the Application via Malicious Input

In the CoAP application architecture, typically the server resides on the sensors and you can discover the resources/services available on the server and send ill-formatted request by using fuzzer to turn to DoS mode.

- Accessing and Manipulating Devices Via Requests

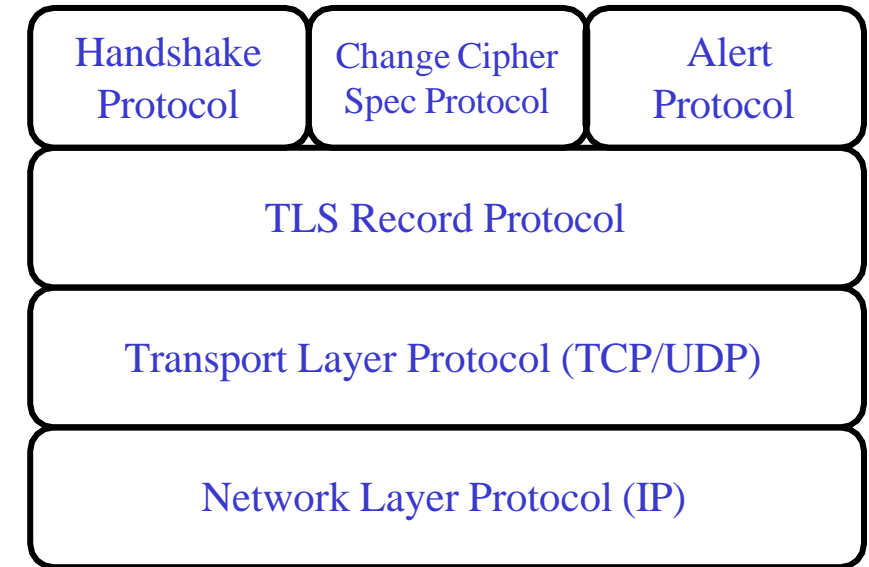
Using various techniques mentioned above, if you can read sensitive data and/or can write data of your choice to sensitive resources by any of the request methods, you have complete control on the device.

# Mitigation

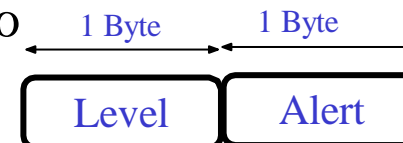
- The vulnerabilities and issues in CoAP are quite similar to other protocols, lack of authentication, access control. There are a few things that one should consider when trying to implement a secure CoAP service:
  - Use DTLS.
  - Prefer not to implement a custom authentication/encryption mechanism.
  - Prefer to utilize all 8-byte to generate random Tokens for Requests. It's good to utilize.
  - Filter double and single dots (".." and ".") in the Uri-Path to prevent against path traversal attacks.
  - Secure the keying material/Certificates etc properly on the devices.
  - Filter input (Request Application payload) on the Server (Device) side.
  - Filter input (Response Application payload) on the cloud side.
  - Implement proper access control and Auth mechanism for accessing resources that perform a critical operation or provide critical information.
  - Log all Activity.
  - Have a way to notify cloud/user about unauthenticated/malicious looking requests.
  - Don't hardcode credentials/sensitive information in the device firmware.

# Transport Layer Security (TLS)

- The SSL Record Protocol provides three services:
  - Client and server authentication (PKI)
  - Confidentiality: The Handshake Protocol defines a shared secret key that is used for conventional encryption of SSL payloads.
  - Message Integrity: The Handshake Protocol also defines a shared secret key that is used to form a message authentication code (MAC).
- The Change Cipher Spec Protocol is used to copy the pending state into the current state, which updates the cipher suite to be used on this connection.
- The Alert Protocol is used to convey SSL-related alerts to the peer entity.



TLS Architecture





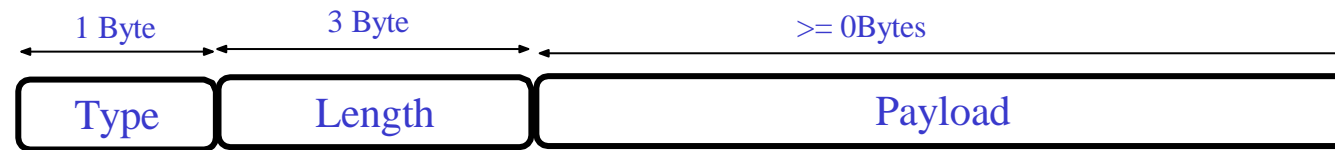
# Alert Messages

Alert Code	Alert Message	Description
0	<b>close_notify</b>	Notifies the recipient that the sender will not send any more messages on this connection.
10	<b>unexpected_message</b>	Received an inappropriate message This alert should never be observed in communication between proper implementations. This message is always fatal.
20	<b>bad_record_mac</b>	Received a record with an incorrect <b>MAC</b> . This message is always fatal.
21	<b>decryption_failed</b>	Decryption of a TLSCiphertext record is decrypted in an invalid way: either it was not an even multiple of the block length or its padding values, when checked, were not correct. This message is always fatal.
22	<b>record_overflow</b>	Received a TLSCiphertext record which had a length more than <b>2<sup>14</sup>+2048</b> bytes, or a record decrypted to a TLSCompressed record with more than <b>2<sup>14</sup>+1024</b> bytes. This message is always fatal.
30	<b>decompression_failure</b>	Received improper input, such as data that would expand to excessive length, from the decompression function. This message is always fatal.
40	<b>handshake_failure</b>	Indicates that the sender was unable to negotiate an acceptable set of security parameters given the options available. This is a fatal error.
42	<b>bad_certificate</b>	There is a problem with the certificate, for example, a certificate is corrupt, or a certificate contains signatures that cannot be verified.
43	<b>unsupported_certificate</b>	Received an unsupported certificate type.
44	<b>certificate_revoked</b>	Received a certificate that was revoked by its signer.
45	<b>certificate_expired</b>	Received a certificate has expired or is not currently valid.
46	<b>certificate_unknown</b>	An unspecified issue took place while processing the certificate that made it



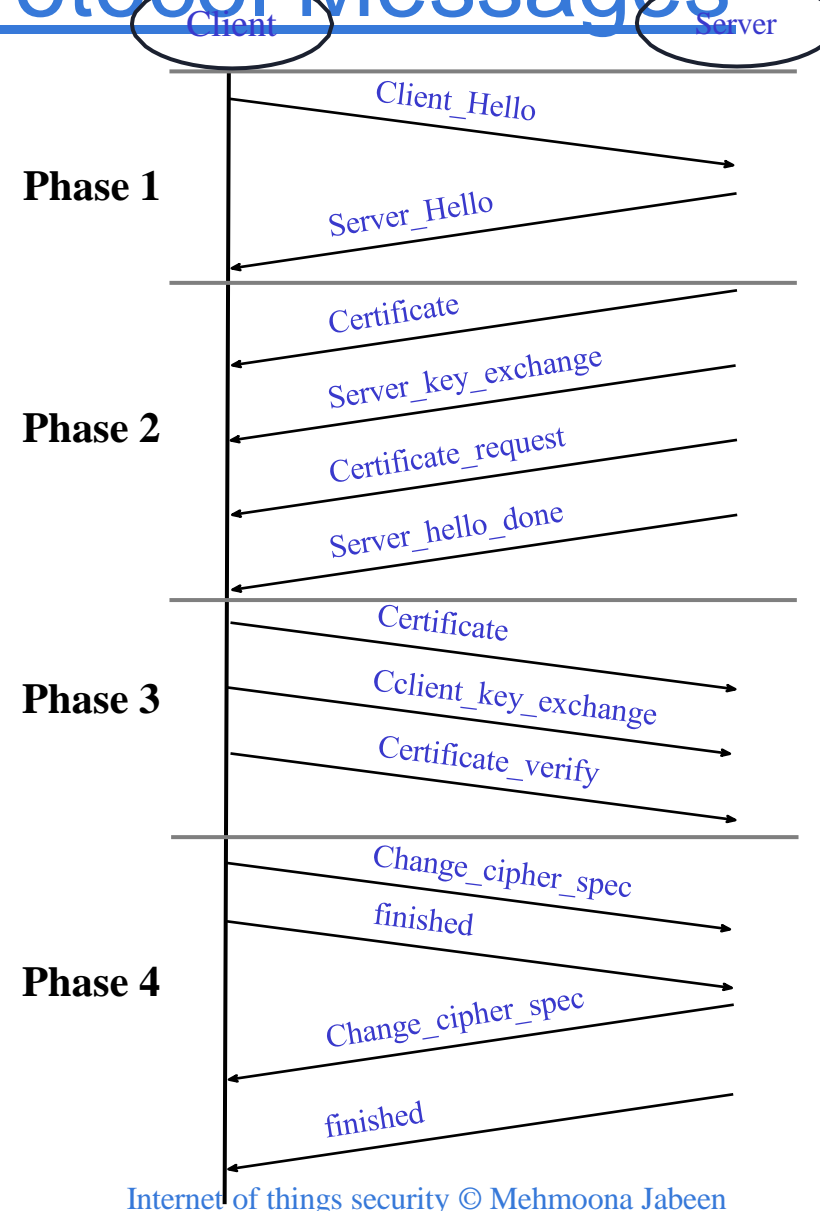
# Handshake Protocol

- The Handshake Protocol consists of a series of messages exchanged by client and server.
  - Type (1 byte): Indicates one of ten messages.
  - Length (3 bytes): The length of the message in bytes.
  - Content (# 0 bytes): The parameters associated with this message



- Consists of four phases:
  - Phase 1: Establish security capabilities, cipher suit, compression method and initial random numbers
  - Phase 2: Server authentication and key exchange
  - Phase 3: Client authentication and key exchange
  - Phase 4: Change cipher suit and finish handshake

# Handshake Protocol Messages



# Message Contents

- Client Hello Message contents:
  - Number of highest SSL understood by the client
  - Client's random structure (32-bit timestamp and 28-byte pseudorandom number)
  - Session ID client wishes to use (ID is empty for existing sessions)
  - List of cipher suits the client supports
    - $\text{SSL\_NULL\_WITH\_NULL\_NULL} = \{ 0, 0 \}$ ,  $\text{SSL\_RSA\_WITH\_NULL\_MD5} = \{ 0, 1 \}$
    - $\text{SSL\_RSA\_WITH\_NULL\_SHA} = \{ 0, 2 \}$ ,  $\text{SSL\_RSA\_EXPORT\_WITH\_RC4\_40\_MD5} = \{ 0, 3 \}$
    - $\text{SSL\_RSA\_WITH\_RC4\_128\_MD5} = \{ 0, 4 \}$ ,  $\text{SSL\_RSA\_WITH\_RC4\_128\_SHA} = \{ 0, 5 \}$
    - $\text{SSL\_RSA\_EXPORT\_WITH\_RC2\_CBC\_40\_MD5} = \{ 0, 6 \}$ ,  $\text{SSL\_RSA\_WITH\_IDEA\_CBC\_SHA} = \{ 0, 7 \}$
    - $\text{SSL\_RSA\_EXPORT\_WITH\_DES40\_CBC\_SHA} = \{ 0, 8 \}$ ,  $\text{SSL\_RSA\_WITH\_DES\_CBC\_SHA} = \{ 0, 9 \}$
    - $\text{SSL\_RSA\_WITH\_3DES\_EDE\_CBC\_SHA} = \{ 0, 10 \}$
  - List of compression methods the client supports

# Message Contents

- Server Respond to client with SERVER HELLO message:
  - Server version number: lower version of that suggested by the client and the highest supported by the server
  - Server's random structure: 32-bit timestamp and 28-byte pseudorandom number
  - Session ID: corresponding to this connection
  - Cipher suite: selected by the server for client's list
  - Compression method: selected by the server from client's list

# TLS Session State Parameters

- **Session Identifier:** An arbitrary byte sequence chosen by the server to identify an active or resumable session state.
- **Peer Certificate:** An X509.v3 certificate of the peer. This element of the state may be null.
- **Compression method:** The algorithm used to compress data prior to encryption.
- **Cipher spec:** Specifies the bulk data encryption algorithm (such as null, AES, etc.) and a hash algorithm (such as MD5 or SHA-1) used for MAC calculation.
- **Master secret:** 48-byte secret shared between the client and the server.
- **Is resumable:** A flag indicating whether the session can be used to initiate new connections.
- **Server and Client Random:** Byte sequences that are chosen by the server and client for each connection.
- **Server write MAC secret:** The secret key used in MAC operations on data.
- **Client write MAC secret:** The secret key used in MAC operations on data sent by the client.
- **Server write key:** The secret encryption key for data encrypted by the server and decrypted by the client.
- **Client write key:** The symmetric encryption key for data encrypted by the client and decrypted by the server.
- **Initialization vectors:** When a block cipher in CBC mode is used, an initialization vector (IV) is maintained for each key.
- **Sequence numbers:** Each party maintains separate sequence numbers for transmitted and received

# Datagram Transport Layer Security (DTLS)

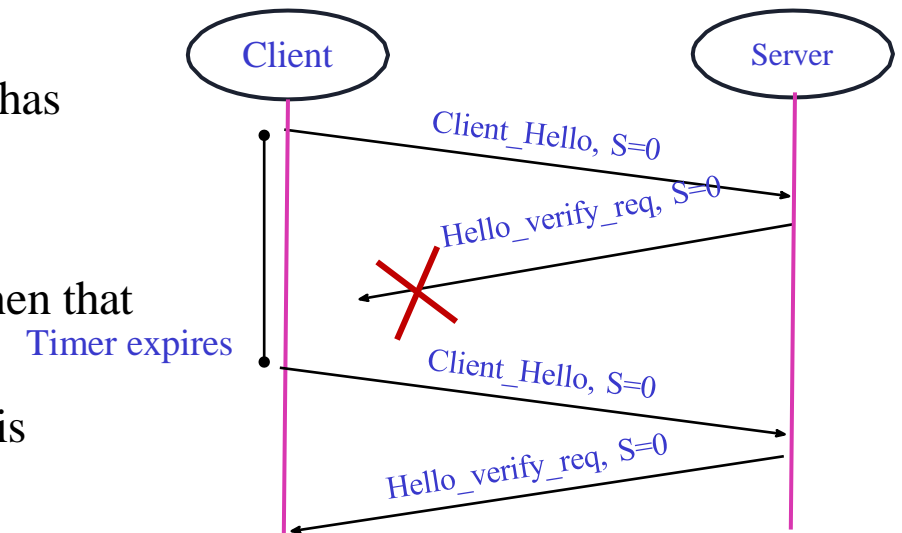
- To construct TLS over datagram transport (UDP), defined in RFC6347.
- The reason that TLS cannot be used directly in datagram environments is
  - Packets may be lost or reordered in UDP.
  - TLS has no internal mechanism to handle this kind of unreliability; therefore, TLS implementations break when rehosted on datagram transport.
- The purpose of DTLS is to make only the minimal changes to TLS required to fix this problem.
- To the greatest extent possible, DTLS is identical to TLS.
- Unreliability creates problems for TLS at two levels:
  1. TLS does not allow independent decryption of individual records.
    - Because the integrity check depends on the sequence number, if record N is not received, then the integrity check on record N+1 will be based on the wrong sequence number and thus will fail.
  2. The TLS handshake layer assumes that handshake messages are delivered reliably and breaks if those messages are lost.

# TLS Challenges over UDP

- Loss-Insensitive Messaging
  - In TLS's traffic encryption layer (called the TLS Record Layer), records are not independent.
- There are two kinds of inter-record dependency:
  - Cryptographic context (stream cipher key stream) is retained between records.
  - Anti-replay and message reordering protection are provided by a MAC that includes a sequence number, but the sequence numbers are implicit in the records.
- DTLS solves the first problem by banning stream ciphers.
- DTLS solves the second problem by adding explicit sequence numbers.
- Providing Reliability for Handshake
  - The TLS handshake is a lockstep cryptographic handshake.
  - Messages must be transmitted and received in a defined order; any other order is an error.
- In addition, TLS handshake messages are potentially larger than any given datagram, thus creating the problem of IP fragmentation. DTLS must provide fixes for both of these problems

# Providing Reliability in Handshake

- Packet Loss
  - DTLS uses a simple retransmission timer to handle packet loss.
  - Client knows that either the ClientHello or the HelloVerifyRequest has been lost and retransmits.
  - When the server receives the retransmission, it knows to retransmit.
  - The server also maintains a retransmission timer and retransmits when that timer expires.
  - Retransmission do not apply to the HelloVerifyRequest, because this would require creating state on the server.
  - Note that **Alert messages are not retransmitted** at all, even when they occur in the context of a handshake.
  - However, a DTLS implementation which would ordinarily issue an alert **SHOULD** generate a new alert message if the offending record is received again





# Providing Reliability in Handshake

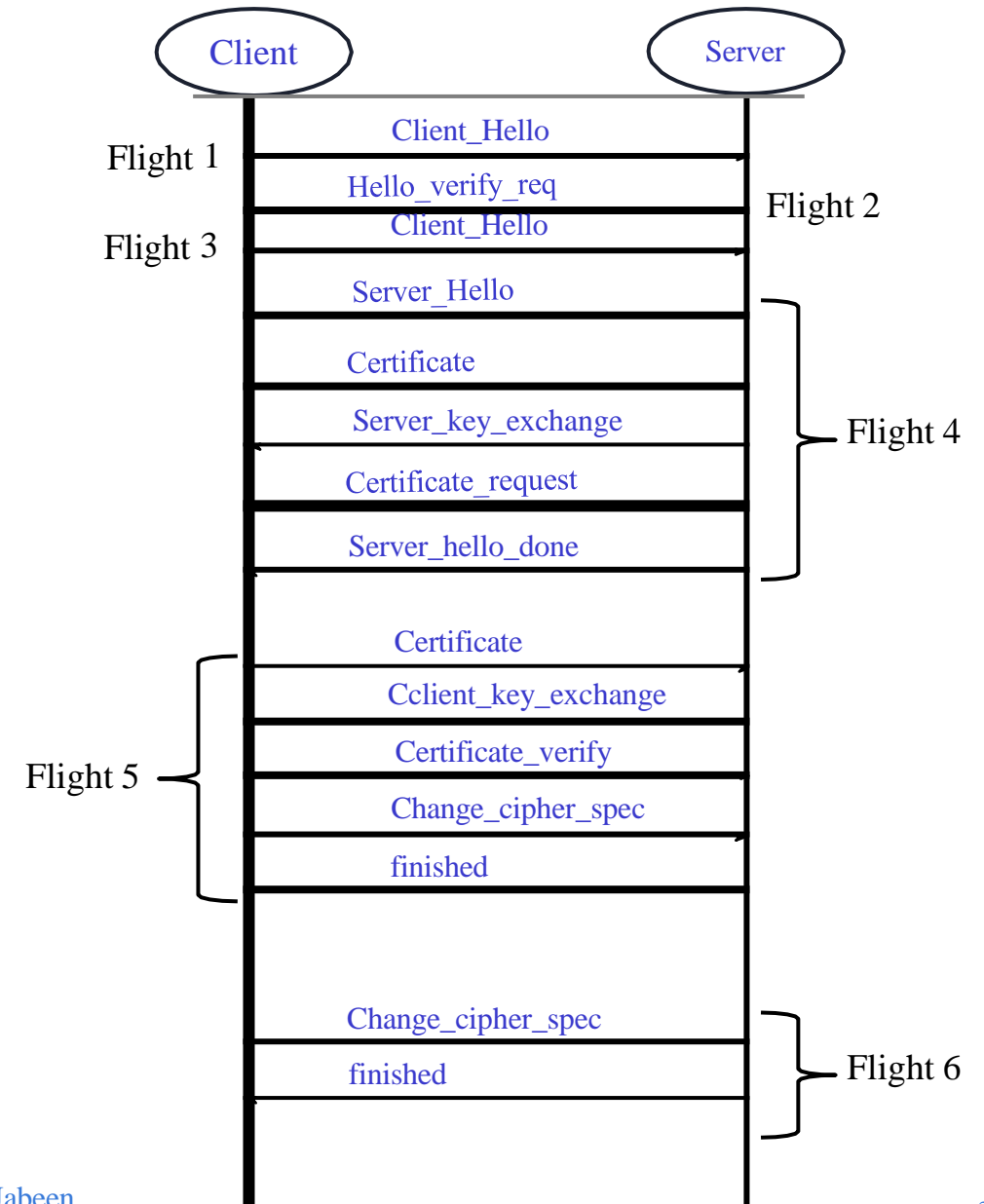
- Reordering
  - In DTLS, each handshake message is assigned a specific sequence number within that handshake.
  - When a peer receives a handshake message, if it is in order then it processes it. Otherwise, it queues it for future handling once all previous messages have been received.
- Message Size
  - UDP datagrams are often limited to <1500 bytes
  - Each DTLS handshake message may be fragmented over several DTLS records, each of which is intended to fit in a single IP datagram.
  - Each DTLS handshake message contains both a fragment offset and a fragment length.
- Replay detection
  - DTLS optionally supports record replay detection.
  - By maintaining a bitmap window of received records, Records that are too old to fit in the window and records that have previously been received are silently discarded.

# Difference from TLS

- The DTLS record layer is extremely similar to that of TLS 1.2. The only change is the inclusion of an explicit **sequence number** in the record.
- DTLS uses all of the same handshake messages and flows as TLS, with three principal changes:
  1. A stateless cookie exchange has been added to prevent denial-of-service attacks.
    - When the client sends its ClientHello message to the server, the server MAY respond with HelloVerifyRequest message. This message contains a stateless cookie generated using the technique of [PHOTURIS]. The client MUST retransmit the ClientHello with the cookie added. The server then verifies the cookie and proceeds with the handshake only if it is valid.
  2. Modifications to the handshake header to handle message loss, reordering, and DTLS message fragmentation (in order to avoid IP fragmentation).
  3. Retransmission timers to handle message loss.

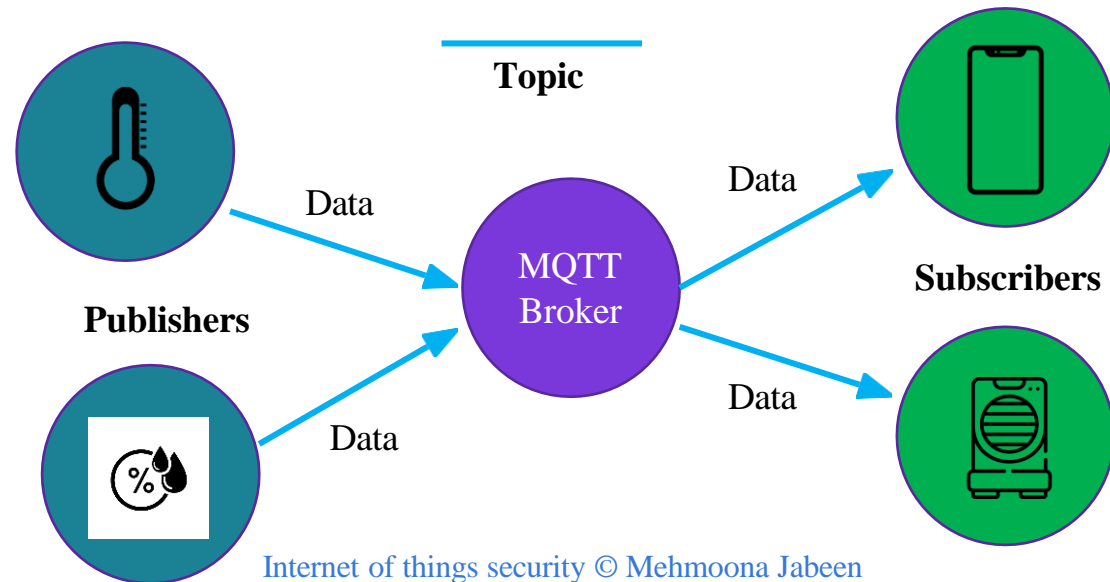
# DTLS Handshake

- DTLS messages are grouped into a series of message flights.
- Each flight of messages may consist of a number of messages, they should be viewed as monolithic for the purpose of timeout and retransmission.



# Message Queuing Telemetry Transport (MQTT)

- MQTT is a machine-to-machine (M2M) protocol designed as an extremely **lightweight publish/subscribe** messaging transport.
- MQTT is an ISO standard (ISO/IEC PRF 20922) messaging protocol for use on top of the TCP/IP protocol and originally developed by IBM for service industry.
- It is designed for connections with remote locations where a "small code footprint" is required or the network bandwidth is limited.



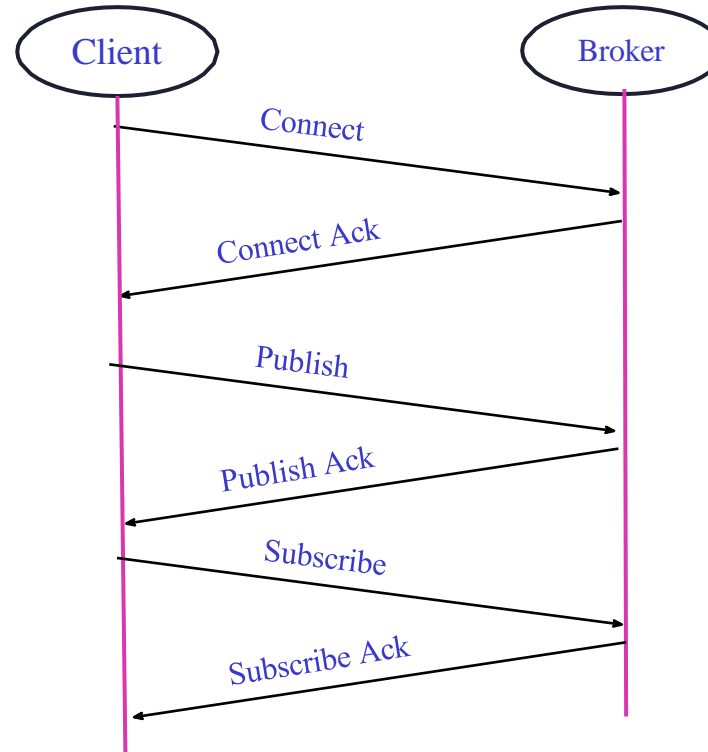
# MQTT Architecture

- Facebook messenger uses MQTT for efficient low overhead communication to save device battery.
- Virtual channels are defined as topics and devices register for the topics to publish and subscribe data.
- Data producers register their topics with brokers and publish data with those topics.
- Clients can subscribe to a topic or a set of related topics
- Publish/Subscribe: Clients can subscribe to topics or publish topics at the same time.
- Many open source implementations of clients and brokers are available
  - Really small message broker (RSMB)
  - Mosquitto
  - Micro broker: Java based for PDAs, notebooks

# MQTT Protocol

Topic names, Client ID, User names and Passwords are encoded as UTF-8 strings

Timer expires



# MQTT Header Format

- The MQTT packet or message format consists of a 2 bytes fixed header (always present) + Variable-header (not always present)+ payload (not always present).
- Fixed Header (Control field + Length): CONNACK
- Fixed Header (Control field + Length) + Variable Header: PUBACK
- Fixed Header (Control field + Length) + Variable Header + payload: CONNECT

Fixed header in all MQTT Control packets

Variable header in some MQTT Control packets

Payload in some MQTT Control packets

bit	7	6	5	4	3	2	1	0
byte 1	Message Type				DUP	QoS		Retain
byte 2	Remaining length (i.e. length of option + payload)							
byte 3	Variable Header Component							
byte n								
byte m	Payload							
byte n								

- The minimum size of the packet length field is 1 byte which is for messages with a total length less than 127 bytes.(not including control and length fields).
- The maximum packet size is 256MB. Small packets less than 127 bytes have a 1 byte packet length field.
- Packets larger than 127 and less than 16383 will use 2 bytes. etc.
- **Note:** 7 bits are used with the 8th bit being a continuation bit.
- The duplicate flag is used when re-publishing a message with QoS or 1 or 2
- The publisher can inform the broker using RETAIN to keep the last message on that topic

Bit position	Name	Description
3	DUP	Duplicate delivery
2-1	QoS	Quality of Service
0	RETAIN	RETAIN flag



# MQTT Message Types

Mnemonic	Enumeration	Description
Reserved	0	Reserved
CONNECT	1	Client request to connect to Server
CONNACK	2	Connect Acknowledgment
PUBLISH	3	Publish message
PUBACK	4	Publish Acknowledgment
PUBREC	5	Publish Received (assured delivery part 1)
PUBREL	6	Publish Release (assured delivery part 2)
PUBCOMP	7	Publish Complete (assured delivery part 3)
SUBSCRIBE	8	Client Subscribe request
SUBACK	9	Subscribe Acknowledgment
UNSUBSCRIBE	10	Client Unsubscribe request
UNSUBACK	11	Unsubscribe Acknowledgment
PINGREQ	12	PING Request
PINGRESP	13	PING Response
DISCONNECT	14	Client is Disconnecting
Reserved	15	Reserved

# MQTT CONNECT message

- **Protocol name** is present in the variable header of a CONNECT message as UTF-encoded string .
- The **protocol version** is present in the variable header of a CONNECT message as an 8-bit unsigned value.
- **Connect Flags** are the Clean Session, Will, Will QoS, and Retain Flags.

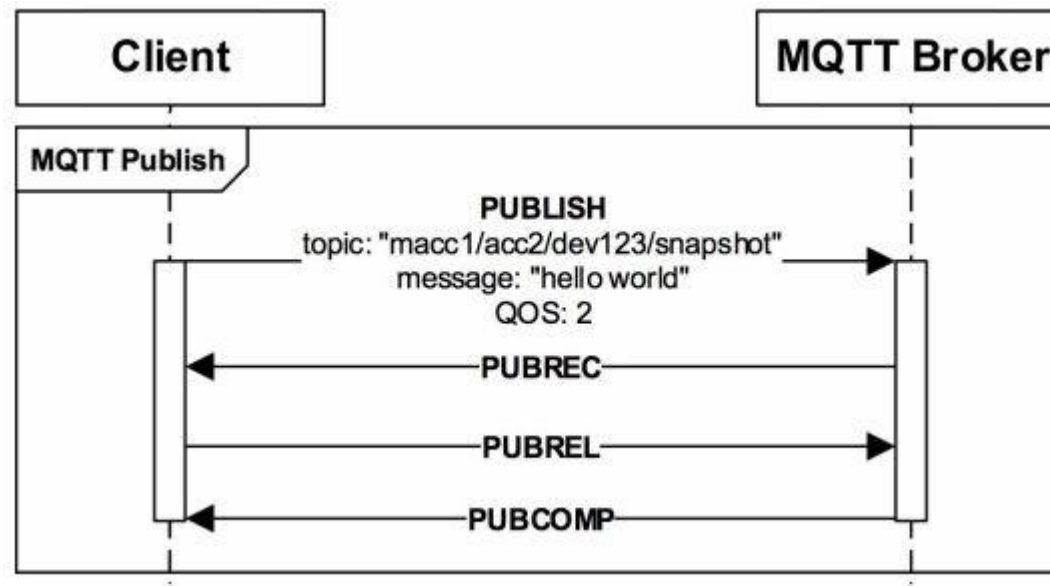
bit	7	6	5	4	3	2	1	0
	User Name Flag	Password Flag	Will Retain	Will QoS	Will Flag	Clean Session	Reserved	
	x	x	x	x	a x	x m	x	

- d treat the connection as "clean"
- The Will message defines that a message is published on behalf of the client by the server
- A connecting client can specify a **user name and a password**, and setting the flag bits signifies that a User Name, and optionally a password, are included in the payload of a CONNECT message.

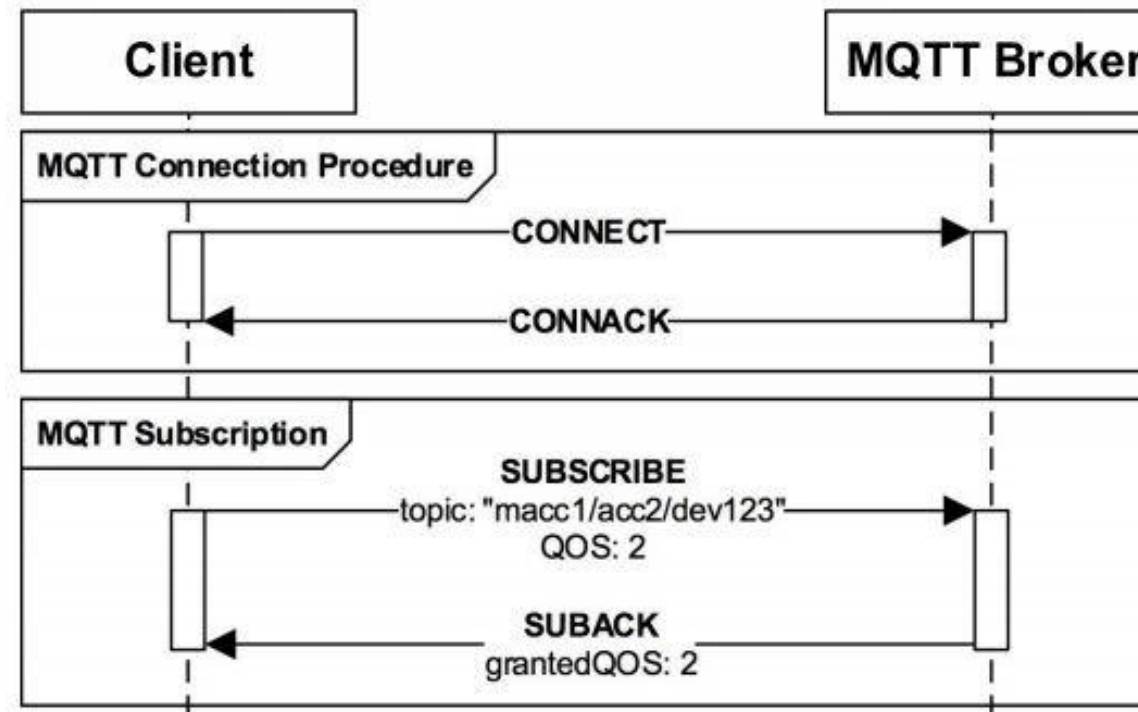
# MQTT Topics

- UTF-8 string that the broker uses to filter messages for each connected client.
- The topic consists of one or more topic levels. Each topic level is separated by a forward slash
- For example: `myhome/firstfloor/livingroom/temperature`
- Each topic must contain at least 1 character and that the topic string permits empty spaces. Topics are **case-sensitive**.
- For example, `myhome/temperature` and `MyHome/Temperature` are two **different** topics. Additionally, the forward slash alone is a valid topic.
- Topic Wildcard characters
- + shows single word: `myhome/firstfloor+/temperature`
- # multi level replace: `myhome/firstfloor/#`
- Best practices
  - Never use a leading forward slash
  - Don't subscribe to #
  - Keep the MQTT topic short and concise

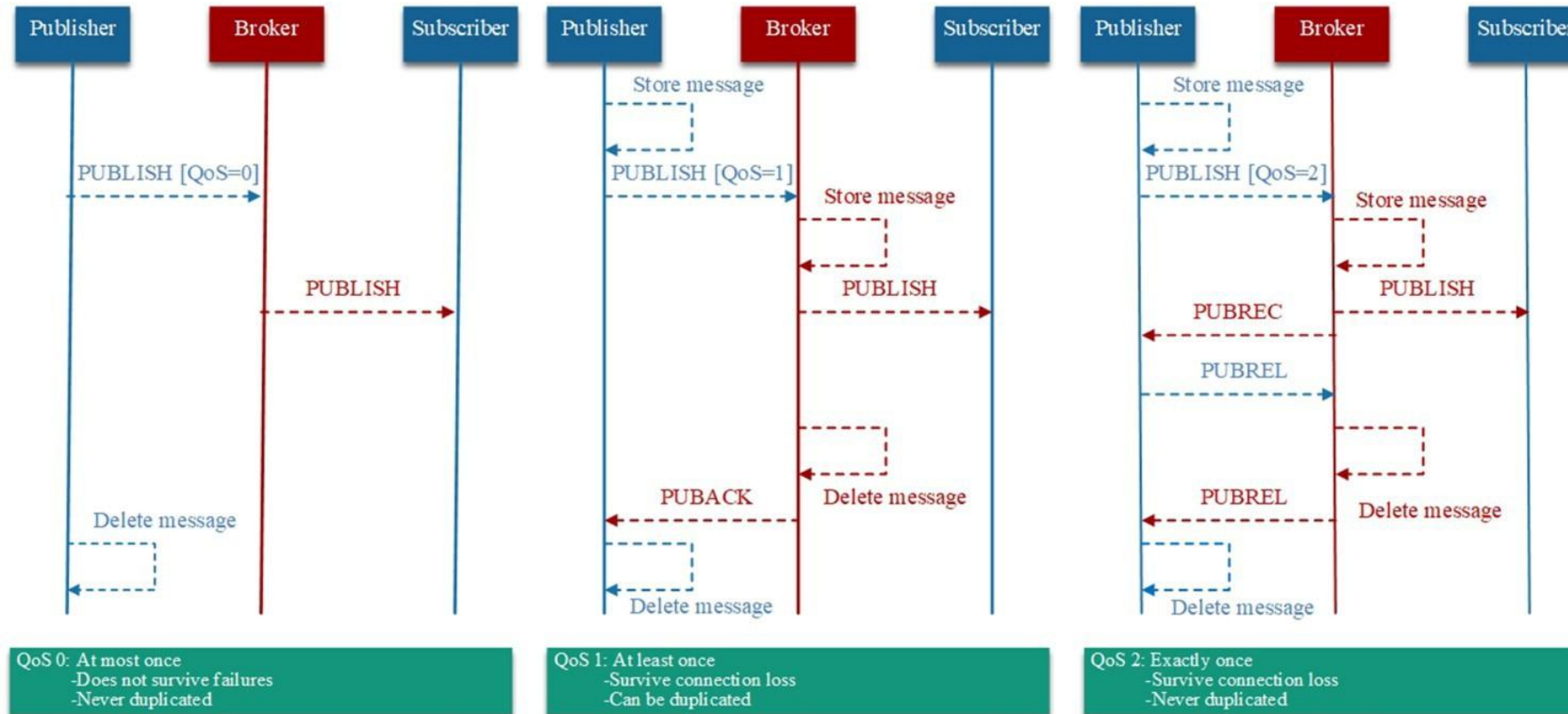
# MQTT Publish



# MQTT Subscribe



# QoS in MQTT



# MQTT vs CoAP vs HTTP

- Common features
  - Aim for low data overhead and little computing efforts
  - Promise to work even in restricted network environments
- Differences
  - MQTT is publish-subscribe oriented, CoAP is request-response oriented
  - MQTT on top of TCP, CoAP on top of UDP → reliability effects

	MQTT	HTTP
Design	Data centric	Document centric
Pattern	Publish/Subscribe	Request /Response
Complexity	Simple	More Complex
Message Size	Small. Binary with 2B header	Large. ASCII
Service Levels	Three	One
Libraries	30kB C and 100 kB Java	Large
Data Distribution	1 to zero, one, or n	1 to 1 only

# MQTT Security

- **Pre-shared key attack:** Security protocols rely on pre-shared keys in some IoT implementations, including the MQTT protocol.
- **Sniffing attack:** Most MQTT protocol implementation has no security mechanism by default which makes the sensitive data sniffed by the attacker via sniffer applications or even monitoring the traffic of the MQTT network.
- **Man-in-the middle (MITM):** The authentication process of MQTT dispatching the credentials as plaintext without utilizing any encryption method
- **Buffer overflow:** This type of attack could be occurring as the result of the opening port of MQTT protocol where inadequate mechanisms of the authorization/authentication, as well as insufficient validation/parsing of messages, leads to a lot of vulnerabilities in the MQTT protocol



# MQTT Security

- CVE-2021-0229
  - An uncontrolled resource consumption vulnerability in Message Queue Telemetry Transport (MQTT) server of Juniper Networks Junos OS allows an attacker to cause MQTT server to crash and restart leading to a Denial of Service (DoS) by sending a stream of specific packets.
- CVE-2021-28166
  - In Eclipse Mosquitto version 2.0.0 to 2.0.9, if an authenticated client that had connected with MQTT v5 sent a crafted CONNACK message to the broker, a NULL pointer dereference would occur.
- CVE-2020-13849
  - The MQTT protocol 3.1.1 requires a server to set a timeout value of 1.5 times the Keep-Alive value specified by a client, which allows remote attackers to cause a denial of service (loss of the ability to establish new connections), as demonstrated by SlowITe.

