UNIT 4
IP BASED PROTOCOLS FOR IOT

IPv6:
IPv6 basic protocol capabilities include the following:

- Addressing
- Anycast
- Flow Labels
- ICMPv6
- Neighbor discovery (ND)

Like IPv4, IPv6 is a connectionless datagram protocol used primarily for addressing and routing packets between hosts. Connectionless means that a session is not established before exchanging data. “Unreliable” means that delivery is not guaranteed. IPv6 always makes a best-effort attempt to deliver a packet. An IPv6 packet might be lost, delivered out of sequence, duplicated, or delayed. IPv6 per se does not attempt to recover from these types of errors. The acknowledgment of packets delivered and the recovery of lost packets is done by a higher-layer protocol, such as TCP (14). From a packet-forwarding perspective, IPv6 operates in a similar, nearly identical manner to IPv4.

FIGURE IPv6 extension headers.
An IPv6 packet, also known as an IPv6 datagram, consists of an IPv6 header and an IPv6 payload, as shown in Figure 7.3. The IPv6 header consists of two parts, the IPv6 base header and optional extension headers. See Figure 7.4. Functionally, the optional extension headers and upper-layer protocols, for example TCP, are considered part of

<table>
<thead>
<tr>
<th>IPv6 Header Field</th>
<th>Length (bits)</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version</td>
<td>4</td>
<td>Identifies the version of the protocol. For IPv6, the version is 6</td>
</tr>
<tr>
<td>Traffic class</td>
<td>8</td>
<td>Intended for originating nodes and forwarding routers to identify and distinguish between different classes or priorities of IPv6 packets</td>
</tr>
</tbody>
</table>
| Flow label            | 20            | (sometimes referred to as flow ID) Defines how traffic is handled and identified. A flow is a sequence of packets sent either to a unicast or to a multicast destination. This field identifies packets that require special handling by the IPv6 node. The following list shows the ways the field is handled if a host or router does not support flow label field functions:
  - If the packet is being sent, the field is set to zero
  - If the packet is being received, the field is ignored |
| Payload length        | 16            | Identifies the length, in octets, of the payload. This field is a 16-bit unsigned integer. The payload includes the optional extension headers, as well as the upper-layer protocols, for example, TCP |
| Next header           | 8             | Identifies the header immediately following the IPv6 header. The following shows examples of the next header:
  - 00 = Hop-by-hop options
  - 01 = ICMPv4
  - 04 = IP in IP (encapsulation)
  - 06 = TCP
  - 17 = UDP
  - 43 = Routing
  - 44 = Fragment
  - 50 = Encapsulating security payload
  - 51 = Authentication
  - 58 = ICMPv6 |
| Hop limit             | 8             | Identifies the number of network segments, also known as links or subnets, on which the packet is allowed to travel before being discarded by a router. The hop limit is set by the sending host and is used to prevent packets from endlessly circulating on an IPv6 internetwork When forwarding an IPv6 packet, IPv6 routers must decrease the hop limit by 1 and must discard the IPv6 packet when the hop limit is 0 |
| Source address        | 128           | Identifies the IPv6 address of the original source of the IPv6 packet |
| Destination address   | 128           | Identifies the IPv6 address of intermediate or final destination of the IPv6 packet |
the IPv6 payload. Table 7.4 shows the fields in the IPv6 base header. IPv4 headers and IPv6 headers are not directly interoperable: hosts and/or routers must use an implementation of both IPv4 and IPv6 in order to recognize and process both header formats (see Fig. 7.5). This gives rise to a number of complexities in the migration process between the IPv4 and the IPv6 environments. The IP header in IPv6 has been streamlined and defined to be of a fixed length (40 bytes). In IPv6, header fields from the IPv4 header have been removed, renamed, or moved to the new optional IPv6 extension headers. The header length field is no longer needed since the IPv6 header is now a fixed-length entity. The IPv4 “type of service” is equivalent to the IPv6 “traffic class” field. The “total length” field has been replaced with the “payload length” field. Since IPv6 only allows for fragmentation to be performed by the IPv6 source and destination nodes, and not individual routers, the IPv4 segment control fields (identification, flags, and fragment offset fields) have been moved to similar fields within the fragment extension header.

**Comparison between IPv4 and IPv6:**

![Comparison of IPv4 and IPv6 headers.](image)

**FIGURE 7.5** Comparison of IPv4 and IPv6 headers.
The Internet Engineering Task Force (IETF) 6LoWPAN Working Group was formed in 2004 to design an adaptation layer for IPv6 when running over 802.15.4 low-power and lossy networks (LowPAN or LLN). The work included a detailed review of requirements, which were released in 2007 (RFC 4919).

In practice, however, the 6LoWPAN is not restricted to radio links, and the technology can be extended to run over other media, for instance it has been extended to run over low-power CPL (www.watteco.com) or G3 OFDM CPL. IPv6 is also being adapted to other physical layers, independently of 6LoWPAN, for example, for HomePlug CPL. Many fieldbus vendors are now considering an IPv6 adaptation layer for their products.

802.15.4 and most low-power transmission technologies must rely on mesh networking to create large networks. Two techniques may be used:

- "Mesh under": the link layer (layer 2) supporting the IP network takes care of mesh networking and packet forwarding, and the IP layer sees a large subnet. An example of such a mesh under protocol is GeoNET, currently under development to support car to car transmission as part of the ETSI intelligent transport system (ITS) technical committee.

- "Route over": IP level (layer 3) mesh routing. If multiple underlying networking technologies need to be used simultaneously (e.g., wireless 802.15.4 and CPL), or when the underlying networking technology supports only point to point or local broadcast...
link layer communication capabilities, then IP level mesh routing becomes necessary to form the internetwork.

**Overview of the 6LoWPAN Adaptation Layer**

6LoWPAN is designed to work on top of 802.15.4 networks. The optional hop by hop acknowledgment feature of 802.15.4 is used, but the macMaxFrameRetries should be set to a relatively low value (e.g., the default of 3) in order to make sure the 802.15.4 layer will not continue to retry when IP and application-level retransmission mechanisms trigger.

6LoWPAN needs to solve 4 issues:
- **Header compression**: on battery-powered networks, long packet headers is synonymous with energy waste. Native IPv6, with its 40-byte header, was probably one of the worst possible candidates for such networks: without compression, the payload of a single IPv6 UDP packet transmitted over a 802.15.4 link layer would not be able to exceed 53 bytes! In the most favorable case, the LowPAN and UDP compressed headers require just 6 bytes.

- **Packet fragmentation and reassembly**: low-power networks usually provide small MTUs, because transmission uses energy, and transmission time is proportional to the packet size. Also, small packets are less subject to packet loss that may occur over lossy networks such as 802.15.4. For instance, on 802.15.4 networks, the frame size is only 127 bytes, and the MAC level overheads may leave as little as 81 bytes for IP. IPv6 normally requires a MTU of 1280 bytes!
- Adaptation of IPv6 neighbor discovery defined in RFC4861 and 4862.
- Support for “mesh under” layer 2 forwarding.

6LoWPAN currently defines several headers, which appear in the following order when present:
- The mesh addressing header;
- Hop by hop processing header, which encode hop-by-hop options such as BC0 broadcast sequence number;
- Destination processing: for example, the fragment header;
- Payload transport: for example, the IPv6 and UDP compression headers.

**Mesh Addressing Header**
Currently, no “mesh-under” protocol is defined for 802.15.4, so this header is only a facility provided to make it possible in the future. When 802.15.4 mesh-under routing is enabled, the 802.15.4 MAC frame contains the source and destination addresses for each hop, therefore a container is needed for the original and final 802.15.4 addresses. The mesh addressing header provides such a container, and also contains a “HopLeft” counter that should be decremented by each layer2 hop.

Fragment Header

The fragment header for the first fragment specifies the full (reassembled) packet size, and uses a datagram tag common for all fragments of this IP packet, which will be used by the receiver, together with the sender and destination MAC addresses, to identify fragments belonging to the same packets. Subsequent fragments also specify the offset of the fragment in the full IP packet, in multiples of 8 bytes.
RPL:

The IETF Routing Over Low-power and Lossy networks (ROLL) Working Group was formed in 2008 to create an IP level routing protocol adapted to the requirements of mesh networking for the Internet of Things: the first version of RPL (Routing Protocol for Low-power and lossy networks) was finalized in April 2011. RPL specifies a routing protocol specially adapted for the needs of IPv6 communication over “low-power and lossy networks” or LLNs, supporting peer to peer traffic (point to point), communication from a central server to multiple nodes on the LLN (point to multipoint P2MP) and vice versa (multipoint to point MP2P).

The base RPL specification is optimized only for MP2P traffic (upward routing or convergecast used, e.g., in metering networks) or P2MP, and P2P is optimized only through use of additional mechanisms such as draft-ietf-roll-p2p-rpl.

Such LLNs are a constrained environment, which imply specific requirements explored by the IETF ROLL working group in RFC5867, RFC5826, RFC5673, and RFC5548. RPL has been designed according to these LLN specific requirements (typically on networks supporting 6LoWPAN), but is not limited to operation over LLNs. Multiple concurrent instances of RPL may operate in a given network, each RPL instance is characterized by a unique RPLInstanceID. The following sections describe the behavior of an individual RPL instance.

---

objective function (OF, defined by the OCP field of a DIO DODAG configuration option). The objective function is not specified by RPL itself, but in other companion documents according to domain-specific requirements: for the available network metrics, the OF computes the “rank” measuring the “distance” between the node and the DODAG root and also defines the parent node selection policy, for instance an objective function could seek to minimize the expected packet delay, while another might want to avoid routing through any battery-operated node (see [I-D.ietf-roll-routing-metrics]).
RPL Control Messages

RPL routers need to exchange information in order to build the DODAG and populate routing tables. RPL defines a new ICMPv6 (RFC 4443) message, type 155, for this purpose.

RPL defines the following base objects:
- The DODAG information solicitation (DIS) message;
- The DODAG information object (DIO), see Figure 12.8;
- The destination advertisement object (DAO);
- The DAO Ack object;
- The consistency check (CC) object, which is used to check secure message counters and to carry RPL challenges and responses, and is always carried in a secure RPL message.

![Figure 12.8 RPL DIO base object (followed by options).](image)
REST:
“Representational state transfer”, or REST is a distributed software architecture style that was described by Roy Fielding in a thesis presented in 2000. The thesis discusses client server based architectures, analyses the reasons for the success of HTTP and hypertext, and presents a number of constraints that define a RESTful architecture, that is, an architecture that will use the same design principles, and share the same desirable properties as HTTP.

REST Verbs
The paper of Roy Fielding never stated which verbs a REST architecture had to provide, as the central idea was that those verbs should aim at manipulating resources. However, the design principles of HTTP, and its evolution to HTTP 1.1, are discussed at length, and in practice many recent standards start by specifying their HTTP binding, before considering other potential bindings for example, to CoAP or other REST-capable protocols. The exact use of HTTP verbs in a REST context is sometimes ambiguous. Recent standards (ZigBee SEP 2.0, ETSI M2M) have converged on the following usage guidelines:

- GET: request verb for reading a representation of a resource in a safe fashion, that is, the request does not change the state of the resource on the server. If the resource URI represents a collection, a list of URIs of collection members will be returned. Generally, resources should be exposed according to the principle of “gradual reveal”, that is, structured in a way that complex structure subelements will be represented by reference in the representation of the parent resource. Another way to see this is that representations should include references to related representations, or “hypermedia as the engine of application state”.

- PUT: request verb for creating or replacing a resource, in an idempotent fashion, that is, multiple identical requests should have the same effect as a single request. If the URI represents a collection, the entire collection is replaced.

- POST: request verb for appending to a resource or creating a subordinate resource (not safe nor idempotent). For instance, a POST /item would typically result in the creation of /item/<subresource instance number assigned by the server> for example, /item/1. The URI of the created resource is returned in the location header as part of the 201 “created” response.

- DELETE: interface for deleting a resource (idempotent). If the URI represents a collection, the entire collection is replaced.

- HEAD (optional): interface to request metadata regarding a resource, in a safe fashion.

- OPTIONS: interface to request the methods available at the server for a resource, for the authorization level of the client.
There were several recurrent discussions on the Internet and within standard bodies debating whether this or that architecture was RESTful or not. Here are some common topics:

- Are “servers” and “clients” defined at function level or interface by interface? Both ZigBee/Homeplug and ETSI concluded that many “real-world” applications cannot strictly separate functions as server or client. Many functions are both server and client for different interfaces. Therefore, the REST concepts are usually understood and applied on a per interface basis (client and server interfaces).

  What about subscribe/notify? This key functionality appears to be missing in the original REST paper, and is also missing in HTTP, leading to workarounds such as polling for AJAX interfaces. Recent standards reintroduce a subscribe/notify model by defining a dedicated resource to store subscriptions to resource R, and define a resource that need to be implemented by hosts interested in notification related to R, where notifications will be posted. This is a typical example of a case where “clients” of a resource R hosted on a server, will need to implement a server function, while the “server” will act as a client to post notifications.

- Concurrent access control to a resource. When the HTTP binding is used, the etag value is used to prevent simultaneous resource updates race conditions. All resource representations for a given resource are required to have the same etag, and the etag should be changed each time the resource is updated. Clients that want to perform a resource modification based on the assumption that the resource has not been updated since the last time they read it should include a condition based on the etag value (If-Match HTTP header). With these clarifications, it seems that there is now a fairly good agreement among standard bodies on the practical implementation of REST style architectures.
AMQP:

Why AMQP?
AMQP creates full functional interoperability between conforming clients and messaging middleware servers (also called "brokers"). Our goal is to enable the development and industry-wide use of standardised messaging middleware technology that will lower the cost of enterprise and systems integration and provide industrial-grade integration services to a broad audience. It is our aim that through AMQP messaging middleware capabilities may ultimately be driven into the network itself, and that through the pervasive availability of messaging middleware new kinds of useful applications may be developed.

Scope of AMQP
To enable complete interoperability for messaging middleware requires that both the networking protocol and the semantics of the server-side services are sufficiently specified. AMQP, therefore, defines both the network protocol and the server-side services through:

- A defined set of messaging capabilities called the "Advanced Message Queuing Protocol Model" (AMQ model). The AMQ model consists of a set of components that route and store messages within the broker service, plus a set of rules for wiring these components together.
- A network wire-level protocol, AMQP, that lets client applications talk to the server and interact with the AMQ model it implements. One can partially imply the semantics of the server from the AMQP protocol specifications but we believe that an explicit description of these semantics helps the understanding of the protocol.

The Advanced Message Queuing Model (AMQ model)
We define the server's semantics explicitly, since interoperability demands that these be the same in any given server implementation. The AMQ model thus specifies a modular set of components and standard rules for connecting these. There are three main types of component, which are connected into processing chains in the server to create the desired functionality:

- The "exchange" receives messages from publisher applications and routes these to "message queues", based on arbitrary criteria, usually message properties or content.
- The "message queue" stores messages until they can be safely processed by a consuming client application (or multiple applications).
- The "binding" defines the relationship between a message queue and an exchange and provides the message routing criteria.

Using this model we can emulate the classic message-oriented middleware concepts of store-and-forward queues and topic subscriptions trivially. We can also express less trivial concepts such as content-based routing, workload distribution, and on-demand message queues.
The Advanced Message Queuing Protocol (AMQP)
The AMQP protocol is a binary protocol with modern features: it is multi-channel, negotiated, asynchronous, secure, portable, neutral, and efficient. AMQP is usefully split into two layers:

The functional layer defines a set of commands (grouped into logical classes of functionality) that do useful work on behalf of the application. The transport layer that carries these methods from application to server, and back, and which handles channel multiplexing, framing, content encoding, heart-beating, data representation, and error handling. One could replace the transport layer with arbitrary transports without changing the application-visible functionality of the protocol. One could also use the same transport layer for different high-level protocols.

The design of AMQP was driven by these requirements:
- To guarantee interoperability between conforming implementations.
- To provide explicit control over the quality of service.
- To be consistent and explicit in naming.
- To allow complete configuration of server wiring via the protocol.
- To use a command notation that maps easily into application-level API's.
- To be clear, so each operation does exactly one thing.
- The design of AMQP transport layer was driven by these main requirements, in no particular order:
  - To be compact, using a binary encoding that packs and unpacks rapidly.
  - To handle messages of any size without significant limit.
  - To carry multiple channels across a single connection.
  - To be long-lived, with no significant in-built limitations.
  - To allow asynchronous command piping.
AMQ Model Architecture

We can summarise what a middleware server is: it is a data server that accepts messages and does two main things with them, it routes them to different consumers depending on arbitrary criteria, and it buffers them in memory or on disk when consumers are not able to accept them fast enough.

In a pre-AMQP server these tasks are done by monolithic engines that implement specific types of routing and buffering. The AMQ model takes the approach of smaller, modular pieces that can be combined in more diverse and robust ways. It starts by dividing these tasks into two distinct roles:

- The exchange, which accepts messages from producers and routes them message queues.
- The message queue, which stores messages and forwards them to consumer applications.

There is a clear interface between exchange and message queue, called a "binding", which we will come to later.

AMQP provides runtime-programmable semantics, through two main aspects:
1. The ability at runtime via the protocol to create arbitrary exchange and message queue types (some are defined in the standard, but others can be added as server extensions).
2. The ability at runtime via the protocol to wire exchanges and message queues together to create any required message-processing system.

Constrained Application Protocol

CoAP is one of the latest application layer protocol developed by IETF for smart devices to connect to Internet. As many devices exist as components in vehicles and buildings with constrained resources, it leads a lot of variation in power computing, communication bandwidth etc. Thus lightweight protocol CoAP is intended to be used and considered as a replacement of HTTP for being an IoT application layer protocol.

CoAP Features With the completion of the CoAP specification, it is expected that there will be million of devices deployed in various application domains in the future. These applications range from smart energy, smart grid, building control, intelligent lighting control,
industrial control systems, asset tracking, to environment monitoring. CoAP would become the standard protocol to enable interaction between devices and to support IoT applications [S.Keoh13]. The Constrained RESTful Environments (CoRE) is the workgroup in IETF that is designing the CoAP protocol. CoAP needs to consider optimizing length of datagram and satisfying REST protocol to support URI (Uniform Resource Identifier). It also needs to provide dependable communication based on UDP protocol.

Table shows the CoAP features

<table>
<thead>
<tr>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constrained web protocol fulfilling M2M requirements</td>
</tr>
<tr>
<td>Security binding to Datagram Transport Layer Security (DTLS)</td>
</tr>
<tr>
<td>Asynchronous message exchanges</td>
</tr>
<tr>
<td>Low header overhead and parsing complexity.</td>
</tr>
<tr>
<td>URI and Content-type support.</td>
</tr>
<tr>
<td>Simple proxy and caching capabilities</td>
</tr>
<tr>
<td>UDP binding with optional reliability supporting unicast and multicast requests.</td>
</tr>
<tr>
<td>A stateless HTTP mapping, allowing proxies to be built, providing access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP.</td>
</tr>
</tbody>
</table>

CoAP Structure Model
CoAP interactive model is similar to HTTP’s client/server model. Fig 2 shows that CoAP employs a two layers structure. The bottom layer is Message layer that has been designed to deal with UDP and asynchronous switching. The request/response layer concerns communication method and deal with request/response message.

Message Format CoAP is based on the exchange of compact messages that, by default, are transmitted over UDP (i.e. each CoAP message occupies the data section of one UDP datagram). Message of CoAP uses simple binary format. Message= fixed-size 4-byte header plus a variable-length Token plus a sequence of CoAP options plus payload.

The format is shown

```
<table>
<thead>
<tr>
<th>Ver</th>
<th>T</th>
<th>OC</th>
<th>Code</th>
<th>MessageID</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Token (if any, TKL bytes)...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Options (if any)...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>Payload (if any)...</td>
</tr>
</tbody>
</table>
```
MQTT:

1. What is MQTT?
MQTT is a lightweight message queuing and transport protocol.
MQTT, as its name implies, is suited for the transport of telemetry data (sensor and actor data).

MQTT is very lightweight and thus suited for M2M (Mobile to Mobile), WSN (Wireless Sensor Networks) and ultimately IoT (Internet of Things) scenarios where sensor and actor nodes communicate with applications through the MQTT message broker.

Example:
Light sensor continuously sends sensor data to the broker.

Building control application receives sensor data from the broker and decides to activate the blinds.

Application sends a blind activation message to the blind actor node through the broker.

2. MQTT characteristics
MQTT Key features:
• Lightweight message queuing and transport protocol
• Asynchronous communication model with messages (events)
• Low overhead (2 bytes header) for low network bandwidth applications
• Publish / Subscribe (PubSub) model
• Decoupling of data producer (publisher) and data consumer (subscriber) through topics (message queues)
• Simple protocol, aimed at low complexity, low power and low footprint implementations (e.g. WSN - Wireless Sensor Networks)
• Runs on connection-oriented transport (TCP). To be used in conjunction with 6LoWPAN (TCP header compression)
• MQTT caters for (wireless) network disruptions
3. Origins and future of MQTT standard
The past, present and future of MQTT:
MQTT was initially developed by IBM and Eurotech.
The previous protocol version 3.1 was made available under http://mqtt.org/.

In 2014, MQTT was adopted and published as an official standard by OASIS (published V3.1.1).
As such, OASIS has become the new home for the development of MQTT.
The OASIS TC (Technical Committee) is tasked with the further development of MQTT.

Version 3.1.1 of MQTT is backward compatible with 3.1 and brought only minor changes:
• Changes restricted to the CONNECT message
• Clarification of version 3.1 (mostly editorial changes)

4. MQTT model (1/3)
The core elements of MQTT are clients, servers (=brokers), sessions, subscriptions and topics.

---

Prepared by: Prof. D. J. Pawar
4. MQTT model (2/3)

**MQTT client (=publisher, subscriber):**
Clients subscribe to topics to publish and receive messages. Thus subscriber and publisher are special roles of a client.

```
+---+        +---+        +---+
|   |        |   |        |   |
| Client | ----> | Publisher | ----> | Subscriber |
```

**MQTT server (=broker):**
Servers run topics, i.e. receive subscriptions from clients on topics, receive messages from clients and forward these, based on client’s subscriptions, to interested clients.

**Topic:**
Technically, topics are message queues. Topics support the publish/subscribe pattern for clients.
Logically, topics allow clients to exchange information with defined semantics.
Example topic: Temperature sensor data of a building.

```
Publisher ----> Topic ----> Subscriber
```

5. MQTT message format (1/14)

**Message format:**
MQTT messages contain a mandatory fixed-length header (2 bytes) and an optional message-specific variable length header and message payload.

Optional fields usually complicate protocol processing. However, MQTT is optimized for bandwidth constrained and unreliable networks (typically wireless networks), so optional fields are used to reduce data transmissions as much as possible.

**MQTT uses network byte and bit ordering.**

```
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Message Type | DUP | QoS Level | RETAIN
---|---|---|---
Remaining Length (1 - 4 bytes)

Optional: Variable Length Header

Optional: Variable Length Message Payload
```

5. MQTT message format (2/14)

Overview of fixed header fields:

<table>
<thead>
<tr>
<th>Message fixed header field</th>
<th>Description / Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>Message Type</td>
<td></td>
</tr>
<tr>
<td>0: Reserved</td>
<td>8: SUBSCRIBE</td>
</tr>
<tr>
<td>1: CONNECT</td>
<td>9: SUBACK</td>
</tr>
<tr>
<td>2: CONNACK</td>
<td>10: UNSUBSCRIBE</td>
</tr>
<tr>
<td>3: PUBLISH</td>
<td>11: UNSUBACK</td>
</tr>
<tr>
<td>4: PUBACK</td>
<td>12: PINGREQ</td>
</tr>
<tr>
<td>5: PUBREC</td>
<td>13: PINGRESP</td>
</tr>
<tr>
<td>6: PUBREL</td>
<td>14: DISCONNECT</td>
</tr>
<tr>
<td>7: PUBCOMP</td>
<td>15: Reserved</td>
</tr>
<tr>
<td>DUP</td>
<td></td>
</tr>
<tr>
<td>Duplicate message flag, indicates to the receiver that this message may have already been received.</td>
<td></td>
</tr>
<tr>
<td>1: Client or server (broker) re-delivers a PUBLISH, PUBREL, SUBSCRIBE or UNSUBSCRIBE message (duplicate message).</td>
<td></td>
</tr>
<tr>
<td>GoS Level</td>
<td></td>
</tr>
<tr>
<td>Indicates the level of delivery assurance of a PUBLISH message.</td>
<td></td>
</tr>
<tr>
<td>0: At-most-once delivery, no guarantees, &quot;Fire and Forget&quot;.</td>
<td></td>
</tr>
<tr>
<td>1: At-least-once delivery, acknowledged delivery.</td>
<td></td>
</tr>
<tr>
<td>2: Exactly-once delivery.</td>
<td></td>
</tr>
<tr>
<td>RETAIN</td>
<td></td>
</tr>
<tr>
<td>1: Instructs the server to retain the last received PUBLISH message and deliver it as a first message to new subscribers.</td>
<td></td>
</tr>
<tr>
<td>Further details see MQTT GoS.</td>
<td></td>
</tr>
<tr>
<td>Remaining Length</td>
<td></td>
</tr>
<tr>
<td>Indicates the number of remaining bytes in the message, i.e. the length of the (optional) variable length header and (optional) payload.</td>
<td></td>
</tr>
<tr>
<td>Further details see Remaining Length (RL).</td>
<td></td>
</tr>
</tbody>
</table>