**ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ**

**ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ**

**ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ**

# Πτυχιακή Εργασία

| Τίτλος Πτυχιακής Εργασίας | ***Ανάλυση και σύγκριση πρωτοκόλλων επικοινωνίας επιπέδου εφαρμογής (HTTP3, MQTT, CoAP) και υλοποίηση συνδέσεων***<br><br>***Analysis and Comparison of Application Layer Communication Protocols (HTTP3, MQTT, CoAP) and Implementation of Connections*** |
|---|---|
| Ονοματεπώνυμο Φοιτητή | Γιώργος Δημητρακόπουλος |
| Πατρώνυμο | Θεόδωρος |
| Αριθμός Μητρώου | Π/ 14031 |
| Επιβλέπων | Π. Κοτζανικολάου, Καθηγητής |

Ημερομηνία Παράδοσης Σεπτέμβριος 2024

# Copyright ©

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν αποκλειστικά τον συγγραφέα και δεν αντιπροσωπεύουν τις επίσημες θέσεις του Πανεπιστημίου Πειραιώς.

Ως συγγραφέας της παρούσας εργασίας δηλώνω πως η παρούσα εργασία δεν αποτελεί προϊόν λογοκλοπής και δεν περιέχει υλικό από μη αναφερόμενες πηγές.

## Επιτελική Σύνοψη

Η πτυχιακή αυτή εξετάζει τρία από τα πιο διαδεδομένα πρωτόκολλα επικοινωνίας τα HTTP, MQTT, και CoAP, με τα δύο τελευταία να είναι τα πιο διαδεδομένα στο IoT ενώ το πρώτο λειτουργεί ως βάση σύγκρισης. Αναλύονται οι μηχανισμοί και οι ιδιαιτερότητες του κάθε πρωτοκόλλου από μεριά σχεδιασμού, και με περιληπτικό αλλά περιεκτικό τρόπο. Γίνεται αναφορά βιβλιογραφίας για τη σύγκριση και των τριών ως προς διάφορες πτυχές τους. Η σύγκριση επεκτείνεται για τα MQTT-CoAP υλοποιώντας πρόγραμμα προσομοίωσης για μέτρηση χρήσης πόρων επεξεργαστή, ως προς τα δύο πιο συχνά μοντέλα επικοινωνίας σε εφαρμογή IoT δηλαδή το Publish-Subscribe και το Request-Response.

## Abstract

This dissertation examines three of the most widespread communication protocols, namely HTTP MQTT and CoAP, with the latter two being the most widespread in IoT while the former acts as a baseline. The mechanisms and workings of each protocol are analyzed from a design point of view, and in a concise but comprehensive way. A literature review is made for the comparison of all three protocols through various aspects. The comparison is extended for MQTT-CoAP by implementing simulation programs to measure the usage of processing resources, in terms of the two most used communication models in IoT applications namely the Publish-Subscribe model and Request-Response model.

## Ευχαριστίες

Θα ήθελα να ευχαριστήσω πρώτα τους γονείς μου για τη συνεχής συναισθηματική υποστήριξη και υπομονή τους κατά τους μήνες γραφής της πτυχιακής. Τον υποψήφιο διδάκτωρ Δ.Κούτρα για τις άμεσες συμβουλές που μου παρείχε όταν τις χρειάστηκα. Και τέλος, τον επιβλέπων καθηγητή Π.Κοτζανικολάου που μου έδωσε την ευκαιρία να εκπονήσω τη πτυχιακή μία ανάσα στο τέλος των σπουδών μου.

Ιούλιος 2024

Δημητρακόπουλος Γιώργος

# Πίνακας Περιεχομένων

1 Εισαγωγή ... 1
2 Ανάλυση Πρωτοκόλλων ... 3
*2.1* Το πρωτόκολλο HTTP ... 3
*2.2* Το πρωτόκολλο MQTT ... 8
*2.3* Το πρωτόκολλο CoAP ... 12
3 Σύγκριση Πρωτοκόλλων ... 17
*3.1* Απόδοση ... 17
*3.2* Ασφάλεια ... 18
*3.3* Ευκολία μάθησης και ανάπτυξης-χρήσης ... 18
*3.4* Πεδία εφαρμογής ... 18
*3.5* Συγκριτικός Πίνακας ... 19
4 Υλοποίηση συνδέσεων ... 20
*4.1* Μεθοδολογία πειράματος ... 20
*4.2* Publish-Subscribe ... 20
*4.3* Request-Response ... 28
5 Συμπεράσματα ... 33
6 Βιβλιογραφικές Πηγές ... 34

# 1 Εισαγωγή

Το *διαδίκτυο των πραγμάτων* (*Internet of Things, IoT*) είναι ένας συνδυασμός τεχνολογικών εξελίξεων και ανθρώπινων αναγκών για όλο και αυξανόμενη συνδεσιμότητα με οτιδήποτε συμβαίνει στο άμεσο και ευρύτερο περιβάλλον [3]. Στοχεύει στη βελτίωση της απόδοσης των επιχειρήσεων που το εφαρμόζουν δίνοντας τη δυνατότητα να κάνουν πράγματα χωρίς ανθρώπινη παρέμβαση, και επίσης διευκολύνει τη καθημερινότητα των χρηστών του [4]. Αποτελείται από ένα δίκτυο των λεγόμενων *έξυπνων* συσκευών οι οποίες επικοινωνούν μεταξύ τους και συνεργάζονται για την επίτευξη ενός συγκεκριμένου στόχου. Τέτοιες συσκευές είναι εφοπλισμένες με *αισθητήρες* (*sensors*) για να αντιλαμβάνονται το φυσικό περιβάλλον γύρω τους ή/και *ενεργοποιητές* (*actuators*) ώστε να αλληλοεπιδρούν κάνοντας διάφορες συγκεκριμένες ενέργειες πάνω σε αυτό, αλλά και με μικροελεγκτές (*microcontrollers*) ή οποιαδήποτε άλλη μορφή επεξεργαστικής μονάδας.

Καθώς το IoT είναι μια γενική έννοια είναι λογικό να περιμένει κανείς ότι οι εφαρμογές του θα είναι ποικίλες και θα βρίσκονται σε πολλούς τομείς της ζωής μας και των βιομηχανιών. Πράγματι, μερικά παραδείγματα εφαρμογής του είναι τα ακόλουθα: [2][3]

- Στα έξυπνα σπίτια, έξυπνες συσκευές γενικά στοχεύουν σε τρία πράγματα: μέτρηση κατανάλωσης δαπανών όπως νερό και ενέργεια, ενίσχυση ασφάλειας με έξυπνους συναγερμούς και συστήματα ασφάλειας που ελέγχονται από απόσταση, αλλά και οτιδήποτε συνεισφέρει στην άνεση ζωής του κατοίκου, παράδειγμα όπως αυτόματος φωτισμός, παράθυρα, έξυπνα ψυγεία και τηλεοράσεις, οικιακές συσκευές κ.α.
- Στην φροντίδα υγείας και ιατρικές υπηρεσίες, έξυπνες συσκευές μπορούν να χρησιμοποιηθούν για να παρακολουθούν συνεχόμενα τη κατάσταση του ασθενούς από απόσταση, συλλέγοντας πληροφορίες όπως καρδιακοί παλμοί, οξυγόνο, πίεση κ.α., αναγνωρίζοντας πιθανόν επικίνδυνων μοτίβων. Επίσης μπορούν να βοηθάνε στη αυτόματη καταγραφή απαραίτητου ιατρικού εξοπλισμού σε μονάδες υγείας.
- Στις κατασκευαστικές, ειδικοί αισθητήρες μπορούν να παρακολουθούν τη κατάσταση και απόδοση μηχανημάτων, αναγνωρίζοντας σφάλματα λειτουργίας τους ώστε να επισκευαστούν γρηγορότερα αποφεύγοντας έτσι εργατικά ατυχήματα αλλά και μειώνοντας το μέσο χρόνο εκτός λειτουργίας. Επίσης βοηθούν στη καταγραφή υλικών και εμπορευμάτων.
- Στις λιανικές επιχειρήσεις, αισθητήρες μπορούν να καταγράφουν τα επίπεδα εμπορευμάτων αυτόματα αναγνωρίζοντας και ειδοποιώντας σε περίπτωση έλλειψης τους. Επίσης παρατηρώντας τη συμπεριφορά των πελατών ως ακολουθία ενεργειών βοηθάνε στη λήψη αποφάσεων για καλύτερη τοποθέτηση προϊόντων και διευκόλυνση των πελατών.
- Στη γεωργία, συσκευές μπορούν να μετράνε διάφορα χαρακτηριστικά στο χώμα και την ανάπτυξη των καλλιεργειών ώστε να βελτιώνουν το σχέδιο ύδρευσής τους. Ενώ στη βιομηχανία κρεάτων μπορούν να παρακολουθούν την υγεία των ζώων και στη διαχείριση του εξοπλισμού και υλικών στην εφοδιαστική αλυσίδα.

- Στις μεταφορές, παρακολουθώντας την απόδοση οχημάτων βοηθούν στη βελτιστοποίηση διαδρομών και παρακολούθηση της κατάστασης των αποστολών/παραγγελιών. Επίσης βοηθούν να ληφθούν καλύτερες αποφάσεις για τη μείωση καυσίμων.

Ορισμένα χαρακτηριστικά του IoT είναι ότι οι έξυπνες συσκευές συνήθως είναι ετερογενείς, έχουν διαφορετικό σχεδιασμό και δυνατότητες, δηλαδή διαφορετικού τύπου αισθητήρες και ενεργοποιητές επικοινωνούν με πιο δυνατές μονάδες επεξεργασίας για την εκτέλεση υπολογισμών. Επίσης να είναι κατά προτίμηση αποδοτικές ως προς τη κατανάλωση, ώστε να διαρκεί για μεγάλο χρονικό διάστημα η μπαταρία τους σε περίπτωση που είναι ασύρματες και απομακρυσμένες. Επιπλέον κάθε συσκευή πρέπει να έχει μοναδική ταυτότητα για να ξεχωρίζεται από τις υπόλοιπες στο ίδιο δίκτυο.

Σε μια υποδομή IoT όμως το κυριότερο χαρακτηριστικό είναι η δια-συνδεσιμότητα δηλαδή ότι οι έξυπνες συσκευές πρέπει να μπορούν και να επικοινωνούν μεταξύ τους. Αυτό γίνεται με τη χρήση πρωτοκόλλων δικτύου για ασύρματα ή ενσύρματα δίκτυα, επιλεγμένα ανάλογα με το πεδίο εφαρμογής, (όπως 6LoWPAN, Zigbee, DDS κ.α.) [4]. Πάνω από αυτά τα πρωτόκολλα βρίσκονται τα πρωτόκολλα επιπέδου εφαρμογής και επικοινωνίας και αυτό είναι το κομμάτι του IoT που εστιάζει η παρούσα πτυχιακή.

Η δομή της πτυχιακής είναι η εξής: στο κεφάλαιο 1 παρουσιάστηκε μια σύντομη περιγραφή και εισαγωγή του IoT, στο κεφάλαιο 2 αναλύονται τα πιο ευρέως χρησιμοποιημένα πρωτόκολλα επικοινωνίας, στο κεφάλαιο 3 γίνεται σύγκριση αυτών ως προς διάφορα κριτήρια, στο κεφάλαιο 4 υλοποιούνται συνδέσεις και ένας server σε ένα από τα επιλεγμένα πρωτόκολλα, και τέλος ακολουθούν τα συμπεράσματα από την όλη μελέτη της πτυχιακής.

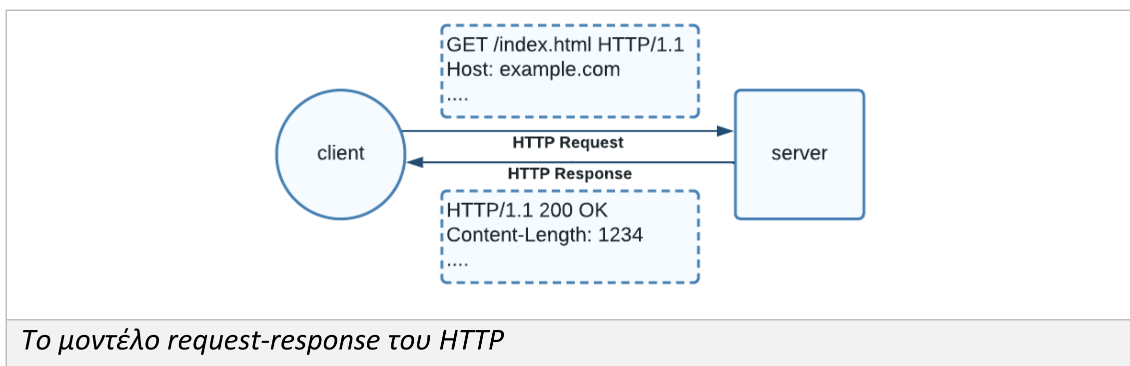
# 2 Ανάλυση Πρωτοκόλλων

Σε αυτό το κεφάλαιο αναλύονται τα πιο ευρέως χρησιμοποιούμενα πρωτόκολλα επικοινωνίας σε περιβάλλον IoT, καθώς και μερικά ιστορικά τους στοιχεία. Ένα κοινό σημείο όλων αυτών των πρωτοκόλλων είναι ότι με βάση το μοντέλο αναφοράς Ανοικτής Διασύνδεσης Συστημάτων (μοντέλο αναφοράς OSI) βρίσκονται στο επίπεδο εφαρμογής, που είναι και το ανώτερο αφαιρετικά επίπεδο από τα 7 συνολικά [1]. Αξίζει να σημειωθεί ότι σε μια εφαρμογή IoT τα παρακάτω επιλεγμένα πρωτόκολλα δεν λειτουργούν μόνα τους αλλά χρησιμοποιούν και άλλα πρωτόκολλα χαμηλότερου επιπέδου με βάση το OSI τα οποία όμως είναι εκτός πεδίου της πτυχιακής και δε θα αναλυθούν παρά μόνο αναφορικά όπου κριθεί απαραίτητο.

## *2.1* Το πρωτόκολλο HTTP

Το *πρωτόκολλο μεταφοράς υπερκειμένου* (*Hypertext Transfer Protocol, HTTP*) αρχικά σχεδιάστηκε για την επικοινωνία μεταξύ του *φυλλομετρητή* (*browser*) και του *εξυπηρετητή ιστού* (*web server*) με σκοπό τη μεταφορά εγγράφων με πολυμεσικό περιεχόμενο [5] όπως είναι οι ιστοσελίδες στο διαδίκτυο, πλέον όμως χρησιμοποιείται και για την επικοινωνία μεταξύ πολλών ειδών εφαρμογών και προγραμμάτων. Είναι ίσως από τα πιο παλαιά και σημαντικά πρωτόκολλα επικοινωνίας στο διαδίκτυο εφόσον και χρονολογείται ίσα με αυτό. Αρχικά προτάθηκε και σχεδιάστηκε από τον Tim Berners-Lee και τη CERN το 1989 μαζί με την τότε πρωτοποριακή ιδέα *WorldWideWeb* [6]. Όμως η πρώτη του κανονικοποιημένη έκδοση *HTTP/1.1,* που εξακολουθεί να χρησιμοποιείται και σήμερα εισάχθηκε έπειτα από οχτώ χρόνια το 1997, ενώ το 2015 ήρθε η έκδοση *HTTP/2* και μόλις το 2022 η έκδοση *HTTP/3* [6]. Σε επόμενη ενότητα θα αναλυθούν πολύ συνοπτικά οι διαφορές τους.

Το HTTP είναι πρωτόκολλο τύπου client-server όπου ο client (και μόνο αυτός) ξεκινάει μια σύνδεση με τον server για την ανταλλαγή μηνυμάτων. Στη πιο απλοϊκή μορφή του μια τυπική συνεδρία αποτελείται από ένα αίτημα (*request*) από τον client προς τον server και αφού εκείνος το αναγνωρίσει και το επεξεργαστεί στέλνει μία απάντηση (*response*). Είναι ένα πρωτόκολλο χωρίς κατάσταση (*stateless*) διότι κάθε request είναι ανεξάρτητο από τα προηγούμενα και επηρεάζει μόνο το αμέσως επόμενο response του.




*Το μοντέλο request-response του HTTP*

### 2.1.1 Δομή μηνύματος

Όλα τα μηνύματα HTTP έχουν συγκεκριμένη δομή, είναι σε μορφή κειμένου με χαρακτήρες ASCII και καταλαμβάνουν πολλές γραμμές. Παρακάτω ακολουθεί ένα τυπικό request και response και έπειτα η περιγραφή των τμημάτων του.

```
POST /test?key1=value1&key2=value2 HTTP/1.1
Host: example.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:120.0)
Content-Type: application/x-www-form-urlencoded
Content-Length: 27
Accept-Language: en-us
Accept-Encoding: gzip, deflate
Connection: keep-alive
```

*Παράδειγμα HTTP request*

```
HTTP/1.1 200 OK
Date: Mon, 27 Jul 2009 12:28:53 GMT
Server: Apache/2.2.14 (Win32)
Content-Length: 46
Set-Cookie: foo=bar,xyz=123;domain=example.com;
Content-Type: text/html
Connection: close
```

*Παράδειγμα HTTP response*

Όπως φαίνεται παραπάνω ένα μήνυμα HTTP είτε είναι request είτε response αποτελείται από τρία τμήματα. Το πρώτο τμήμα καταλαμβάνει μόνο μία γραμμή και είναι διαφορετική μεταξύ request και response. Για το request, η πρώτη λέξη βρίσκεται η μέθοδος αιτήματός του (ή αλλιώς λέγεται και ρήμα *HTTP method/verb*), ακολουθεί ο πόρος-στόχος του αιτήματος ως τμήμα ενός URL και μετά η έκδοση πρωτοκόλλου. Για το response, η πρώτη λέξη είναι η έκδοση και ακολουθείται με το κωδικό κατάστασης της απάντησης (*status code*). Στο δεύτερο τμήμα του μηνύματος βρίσκονται οι επικεφαλίδες, μία σε κάθε γραμμή με πρώτα το όνομα ακολουθούμενο από το διαχωριστικό “*:* “ και μετά τη τιμή της. Το τρίτο τμήμα, που περιέχει το σώμα του μηνύματος (*body* ή αλλιώς *payload*) βρίσκεται μετά από τη πρώτη κενή γραμμή μέχρι το τέλος του μηνύματος, και είναι προαιρετικό καθώς ανάλογα με τη μέθοδο αιτήματος μπορεί να λείπει. Επίσης μόνο το payload μπορεί να περιέχει binary data.

### 2.1.2 Uniform Resource Identifier (URI)

Το ενιαίο αναγνωριστικό πόρου (*Uniform Resource Identifier URI*) είναι μία μοναδική ακολουθία χαρακτήρων που αναγνωρίζει ένα λογικό ή φυσικό πόρο σε τεχνολογία ιστού [7]. Στο HTTP ένα *URL* (*Uniform Resource Locator*) ακολουθάει τη σύνταξη ενός URI και περιλαμβάνει το σχήμα ή πρωτόκολλο, τη διεύθυνση προαιρετικά μαζί με τη θύρα κ.α., όμως στο μήνυμα ενός HTTP εμφανίζεται μόνο ένα κομμάτι αυτού, το μονοπάτι *path* και το ερώτημα *query*. Το path αποτελείται από τμήματα χωριζόμενα με *slash* “*/*“ και καθορίζει το πόρο που αναφέρεται το αίτημα HTTP. Μετά το ερωτηματικό “*?*” βρίσκεται το query, που αποτελείται από παραμέτρους ερωτήματος (*query parameters*) *με* το καθένα να έχει ένα όνομα και μία τιμή χωριζόμενα με *“=”* ενώ μεταξύ τους χωρίζονται με “*&*”.

Σε περίπτωση που υπάρχουν μη ASCII χαρακτήρες στο path, query ή στις τιμές των επικεφαλίδων, ή/και μη έγκυροι χαρακτήρες στο εκάστοτε πεδίο (π.χ. τιμή παραμέτρου με "&") τότε αυτοί μετατρέπονται σε σημασιολογικά ισοδύναμες ακολουθίες χαρακτήρων με τη κωδικοποίηση *URL-Encoding*. Κάθε χαρακτήρας κωδικοποιείται στην ακολουθία bytes του με τη κωδικοποίηση *UTF-8*, και έπειτα κάθε byte του τυπώνεται στη δεκαεξαδική του μορφή με το "*%*" μπροστά. Για παράδειγμα στη συμβολοσειρά "/path?x=123&y=/αβ" οι τρεις τελευταίοι χαρακτήρες "/αβ" μετατρέπονται σε "%2F%CE%B1%CE%B2" διότι το *slash* αναπαρίσταται με 0x2F κατά ASCII ενώ το "α" με τα bytes "0xCE 0xB1" κατά UTF-8.

### 2.1.3 Μέθοδοι αιτήματος

Η πρώτη λέξη που εμφανίζεται στο request είναι η μέθοδος αιτήματος και καθορίζει την ενέργεια που πρόκειται να πραγματοποιηθεί σε ένα πόρο. Η τιμή του μπορεί να είναι μία από τις ακόλουθες [8]. Σημειώνεται ότι η σημασιολογία ή σκοπός τους αποτελεί γενικό κανόνα και το αν θα τηρηθεί κρίνεται από τον web server και την εφαρμογή.

- *GET*: χρησιμοποιείται για τη μεταφορά ενός πόρου από το server στο client.
- *POST*: όταν ο client επιθυμεί να υποβάλει στοιχεία (π.χ. HTML form).
- *PUT*: όταν ο client επιθυμεί να αντικαταστήσει ένα πόρο.
- *HEAD*: ίδιο με το GET, χωρίς σώμα στην απάντηση (δηλ. μόνο επικεφαλίδες).
- *OPTIONS*: για τη ζήτηση των αποδεκτών μεθόδων και επικεφαλίδων πριν ένα request.
- *DELETE*: για τη διαγραφή ενός πόρου.
- *CONNECT*: σύνδεση με Proxy server για το άνοιγμα διαύλου επικοινωνίας με server.
- *TRACE*: για την αποσφαλμάτωση όπου στέλνεται πίσω το request που έλαβε ο server.

### 2.1.4 Κωδικοί κατάστασης

Όπως αναφέρθηκε και προηγουμένως, στη πρώτη γραμμή του response βρίσκεται ο κωδικός κατάστασης (*status code*) ο οποίος είναι ένας τριψήφιος αριθμός στο εύρος 100-599 και δηλώνει αν το request ήταν επιτυχές ή σε περίπτωση σφάλματος το είδος του [9]. Στο προηγούμενο παράδειγμα response ο κωδικός "200 OK" δηλώνει ότι το request πραγματοποιήθηκε με αναμενόμενο τρόπο. Το πρώτο ψηφίο του κωδικού ορίζει σε ποια από τις 5 κατηγορίες βρίσκεται. Επίσης το πρωτόκολλο ορίζει ότι οι κωδικοί είναι επεκτάσιμοι και ότι μια υλοποίηση του δεν υποχρεούται να αναγνωρίζει όλους τους κωδικούς αλλά πρέπει τη κατηγορία του. Ακολουθεί συνοπτική απαρίθμηση των 5 κατηγοριών τους.

- *1xx Κωδικοί πληροφόρησης*. Το request λήφθηκε και κατανοήθηκε αλλά εκκρεμεί κάποια ενέργεια για την ολοκλήρωσή του.
- *2xx Κωδικοί επιτυχίας*. Το request λήφθηκε, κατανοήθηκε και εκτελέσθηκε με επιτυχία.
- *3xx Κωδικοί ανακατεύθυνσης*. Χρησιμοποιείται στην ανακατεύθυνση URL (*redirection*) και συνήθως απαιτείται επιπλέον ενέργεια από το client.
- *4xx Κωδικοί σφάλματος client*. Δηλώνουν ότι υπάρχει κάπου λάθος στο request που έστειλε η μεριά του client και πρέπει να ενημερωθεί ο χρήστης.
- *5xx Κωδικοί σφάλματος server*. Δηλώνουν ότι ο server έλαβε και κατανόησε το request αλλά απέτυχε να πραγματοποιήσει.

### 2.1.5 Επικεφαλίδες

Οι επικεφαλίδες (*headers*) επιτρέπουν στο client ή server να στέλνουν επιπλέον πληροφορία μαζί με το περιεχόμενο του μηνύματος στο request ή response αντίστοιχα. Κάθε επικεφαλίδα έχει ένα όνομα και μία ή περισσότερες τιμές χωρισμένες με κόμμα. Λόγω της ευρείας χρήσης και εφαρμογών του πρωτοκόλλου συχνά χρειάζονταν νέες επικεφαλίδες οι οποίες κατά κανόνα ξεκινούν με το προθεματικό "*x-*". Αυτές που ορίζει το πρωτόκολλο κατατάσσονται σε 4 ομάδες ανάλογα με το σκοπό τους [10].

Στη πρώτη κατηγορία βρίσκονται οι επικεφαλίδες αναπαράστασης (*representation headers*) και περιγράφουν την αναπαράσταση του πόρου στο σώμα του μηνύματος. Αυτές είναι οι *Content-Type, Content-Encoding, Content-Language* και *Content-Location*. Για παράδειγμα ένα συγκεκριμένο περιεχόμενο μπορεί να αναπαρασταθεί με πολλές μορφές όπως *XML* ή *JSON*, γραμμένο σε άλλη γλώσσα, με συμπίεση ή χωρίς, ή με άλλη μορφή κωδικοποίησης. Ο client δηλώνει τη προτίμησή του στη μορφή περιεχομένου στέλνοντας επικεφαλίδα τύπου *Accept-**, έπειτα ο server στέλνει επικεφαλίδα αναπαράστασης μαζί με το περιεχόμενο. Στην επικεφαλίδα *Content-Type* εκφράζεται ο τύπος περιεχομένου χρησιμοποιώντας το γνωστό *MIME-type*.

Στη δεύτερη κατηγορία βρίσκονται οι επικεφαλίδες που αφορούν τη μεταφορά του περιεχομένου (*payload headers*). Αυτές είναι οι *Content-Length, Content-Range, Trailer* και *Transfer-Encoding*. Περιλαμβάνουν πληροφορίες όπως το μήκος του περιεχομένου σε bytes, ή αν πρόκειται για πολλά κομμάτια τότε ποιο από αυτά στέλνεται και με ποια συμπίεση ή άλλα δεδομένα απαραίτητα για τη μεταφορά.

Οι υπόλοιπες επικεφαλίδες χωρίζονται σε δύο κατηγορίες request/response headers ανάλογα με το που εμφανίζονται. Επειδή το πλήθος αυτών είναι τεράστιο η ανάλυση της κάθε μίας από αυτές είναι εκτός πεδίου μελέτης. Αναφορικά μπορεί να αφορούν [10]: την επαλήθευση ταυτότητας του client (*authentication* ή *authorization*), διαχείριση ταχείας μνήμης για απόδοση (*caching*), πληροφορίες του χρήστη (*user-agent*), διαπραγμάτευση για τη μορφή περιεχομένου, διαχείριση σύνδεσης, μπισκότα (*cookies*), κανόνες πρόσβασης για τη χρήση ξένων πόρων (*CORS*), σχετικά με διαμεσολαβητές (*proxies*) κ.α.

### 2.1.6 Συνεδρίες και Cookies

Το πρωτόκολλο HTTP είναι *stateless* πρωτόκολλο χωρίς κατάσταση, μπορούν όμως να επιτευχθούν συνεδρίες (*sessions*) με τη χρήση των cookies. Με τον όρο session εννοούμε διατήρηση κάποιων δεδομένων σε μια σειρά από requests και responses. Για παράδειγμα μπορούν να χρησιμοποιηθούν cookies για την υλοποίηση ενός login για να θυμάται ο server τον client ώστε εκείνος να μην στέλνει συνεχώς τα στοιχεία αυθεντικοποίησης του. Άλλα παραδείγματα συνεδριών μπορεί να είναι η προσωποποίηση της εφαρμογής για το χρήστη ή η ιχνηλάτηση των ενεργειών του στην εφαρμογή για σκοπούς μάρκετινγκ, το λεγόμενο *analytics*.

Τα cookies είναι δεδομένα που στέλνονται στο response από το server με την επικεφαλίδα *Set-Cookie* και αφού τα λάβει ο client τα αποθηκεύει σε ασφαλές σημείο και στα επόμενα requests προς τον ίδιο server του τα επιστρέφει με την επικεφαλίδα *Cookie* [11]. Κατά προτίμηση πρέπει να είναι μικρά σε μέγεθος διότι στέλνονται με κάθε request. Προαιρετικά μαζί με κάθε cookie τίθεται η ημερομηνία λήξης του με την ιδιότητα *Expires*

αλλιώς διατηρείται μέχρι το κλείσιμο της σύνδεσης. Με τις ιδιότητες *Domain* και *Path* καθορίζεται η ορατότητά του, δηλαδή για το ίδιο domain σε πια URLs ισχύει.

### 2.1.7 HTTP vs HTTPS

Το *Hypertext Transfer Protocol Secure* (*HTTPS*) είναι μία επέκταση του HTTP και όχι ξεχωριστό πρωτόκολλο [12]. Ουσιαστικά είναι η εφαρμογή του πρωτοκόλλου κρυπτογράφησης *Transport Layer Security* (*TLS*) πάνω στο HTTP. Σε μία σύνδεση TCP για HTTP η προκαθορισμένη θύρα, δηλαδή εκείνη που χρησιμοποιείται όταν δε την αναγράφουμε στο URL, είναι η 80 ενώ για το HTTPS είναι η 443. Σε ένα μη ασφαλές δίκτυο όπως είναι το ίντερνετ ή δημόσια σημεία πρόσβασης Wi-Fi, η κρυπτογράφηση της επικοινωνίας είναι αναγκαία όταν μεταφέρονται ευαίσθητα δεδομένα, ώστε να μην υποκλαπούν ή αλλοιωθούν από κακόβουλους χρήστες με επιθέσεις man-in-the-middle. Ακόμα και όταν τα δεδομένα δε θεωρούνται ευαίσθητα απαιτείται ασφάλεια για την αποφυγή εισαγωγής ξένων πακέτων με κακό σκοπό.

Ένας web server για να χρησιμοποιήσει HTTPS θα πρέπει ο διαχειριστής του να εκδώσει ένα ψηφιακό πιστοποιητικό (*digital certificate*) για το δημόσιο κλειδί που θα χρησιμοποιεί στη κρυπτογράφηση καθώς και με άλλες πληροφορίες για αυτόν όπως domain κτλ. Έπειτα το πιστοποιητικό θα πρέπει να υπογραφεί από μία αρχή πιστοποίησης (*certificate authority, CA*) ώστε να φαίνεται έγκυρο. Έτσι όταν ένας χρήστης λαμβάνει το πιστοποιητικό επικοινωνεί με μία αλυσίδα από CA που εκείνος εμπιστεύεται ώστε να βεβαιωθεί πως είναι έγκυρο, ότι δεν έχει λήξει και ότι πρόκειται όντως για τον server που προσπαθεί να επικοινωνήσει.

### 2.1.8 Διαφορές εκδόσεων HTTP 1.1/2/3

Σε αυτή την υποενότητα αναφέρονται συνοπτικά οι αλλαγές που φέρνει μία έκδοση καθώς και τους περιορισμούς της προηγούμενής της. Παλιότερα στο HTTP/1.0 κάθε σύνδεση TCP ήταν μόνο για ένα request και response. Το HTTP/1.1 προσθέτει καινούργιες επικεφαλίδες, επαναχρησιμοποιεί τη σύνδεση και εφαρμόζει τεχνική *pipelining* για την αποστολή πολλαπλών requests πριν τη λήψη των responses [13]. Ένα επιπλέον χαρακτηριστικό του είναι η χρήση προσωρινής μνήμης (*cache*) ώστε ο client να αποθηκεύει απαντήσεις σε συχνά αιτήματα με σταθερό περιεχόμενο για ένα συγκεκριμένο χρονικό διάστημα χωρίς έτσι να ξοδεύονται άσκοπα πόροι επικοινωνίας με τον server.

Πριν το HTTP/2, τα μηνύματα του πρωτοκόλλου ήταν σε μορφή κειμένου και έτσι τα binary data στο σώμα μηνύματος έπρεπε να μετατραπούν σε κείμενο π.χ. με κωδικοποίηση *Base64*, καταναλώνοντας τελικά περισσότερα bytes. Επιπλέον το HTTP/2 προσφέρει πολυπλεξία αιτημάτων (*multiplexing*) μαζί με προτεραιότητες. Η επικοινωνία μεταξύ client και server λαμβάνει χώρα σε μία σύνδεση TCP και χωρίζεται σε πολλαπλές ροές [13]. Κάθε ροή έχει ένα *ID* και διαιρείται σε πολλαπλά πλαίσια (*frames*) τύπου *DATA* ή *HEADER*. Στη ροή με ID=0 μεταφέρονται πλαίσια ελέγχου. Για λόγους αλληλεξαρτήσεων μεταξύ ροών ή δικαιοσύνης κάθε ροή έχει και μια προτεραιότητα. Επίσης επειδή οι επικεφαλίδες είναι συχνά επαναλαμβανόμενες και καταλαμβάνουν πολλούς χαρακτήρες, το HTTP/2 εισάγει τον αλγόριθμο συμπίεσης *HPACK* για τη συμπίεση τους.

Ένα μειονέκτημα του HTTP/2 είναι ότι επειδή χρησιμοποιεί μόνο μία σύνδεση TCP είναι επιρρεπής στο πρόβλημα *Head of Line Blocking* (*HoLB*). Αυτό σημαίνει ότι αν η ροή που είναι η σειρά της καθυστερεί λόγω απώλειας πακέτων, τότε καθυστερούν όλες οι ροές.

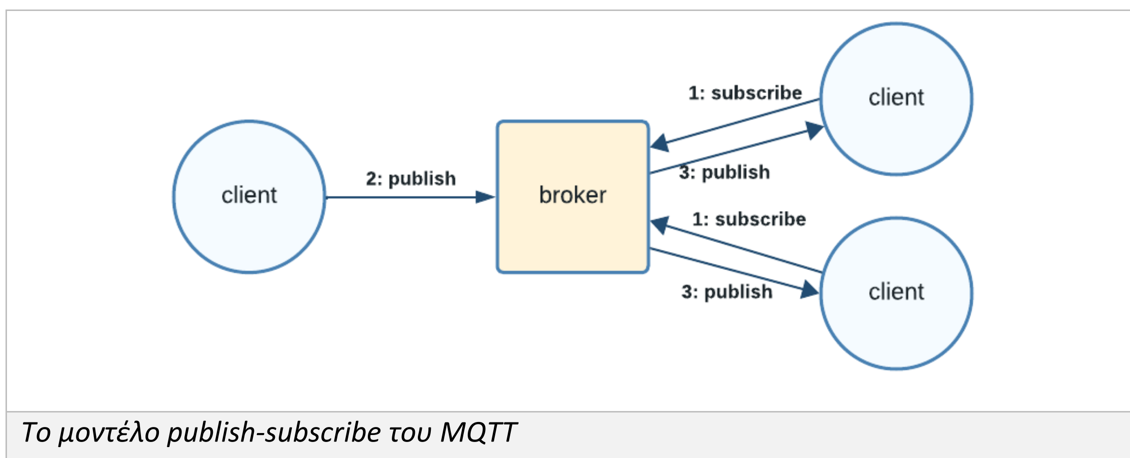
Ένα μικρότερο μειονέκτημα είναι ότι υπάρχει ένα κόστος για τη χειραψία στο TCP και TLS. Περισσότερες πληροφορίες για το HTTP/2 υπάρχουν στο άρθρο [14].

Το HTTP/3 λειτουργεί πάνω στο πρωτόκολλο μεταφοράς *QUIC* της Google που είναι τύπου UDP σκοπεύοντας έτσι τη μείωση καθυστέρησης, ενώ τα προηγούμενα βασιζόντουσαν στο TCP, περιλαμβάνει δυνατότητες TLS/1.3. Χρησιμοποιεί το *QPACK* για τη συμπίεση επικεφαλίδων. Λύνει το πρόβλημα HoLB καθώς έχει σχεδιαστεί με την υπόθεση ότι στο QUIC τα πακέτα διαφορετικών ροών φτάνουν με διαφορετική σειρά, χωρίς να μπλοκάρεται έτσι η σύνδεση. Μειώνει το χρόνο μετάβασης μετ' επιστροφής (*Round-Trip Time*) ενώνοντας τη χειραψία κρυπτογράφησης και εκείνη της μεταφοράς. Περισσότερες πληροφορίες για το HTTP/3 υπάρχουν στη σειρά άρθρων [15].

## *2.2* Το πρωτόκολλο MQTT

Το πρωτόκολλο *MQ Telemetry Transport* (*MQTT*) είναι ένα ελαφρύ πρωτόκολλο μεταφοράς μηνυμάτων σχεδιασμένο για περιβάλλοντα χαμηλού εύρους ζώνης όπως το IoT [16]. Σχεδιάστηκε το 1999 από τους Andy Stanford-Clark και Arlen Nipper για χρήση στη βιομηχανία πετρελαίου και φυσικού αερίου όπου χρειάζονταν αισθητήρες χαμηλής κατανάλωσης μπαταρίας και επικοινωνίας μέσω δορυφόρου. Όσο για την ονομασία, επειδή ανήκε στη σειρά προϊόντων MQSeries της IBM το MQ σήμαινε *λανθασμένα* Message Queuing διότι το πρωτόκολλο δεν αφορά ουρές μηνυμάτων αλλά έχει ως βασικό στοιχείο το μηχανισμό publish-subscribe, ο οποίος περιγράφεται στην επόμενη υποενότητα.

Η έκδοση MQTT 3.1.1 είναι η πρώτη κανονικοποιημένη έκδοση υπό τον οργανισμό OASIS Standards και χρονολογείται το 2014, ενώ η πιο πρόσφατη έκδοση MQTT 5 το 2019 [16]. Η κανονική έκδοση του χρησιμοποιεί το πρωτόκολλο μεταφοράς TCP/IP (πρόσφατα και το QUIC) με θύρα συνήθως την 1883 ή με TLS την 8883. Για εφαρμογή του σε browsers χρησιμοποιεί WebSockets (MQTT over WSS). Η παραλλαγή του *MQTT-SN* αφορά δίκτυα αισθητήρων (sensor networks) χωρίς δυνατότητες TCP/IP χρησιμοποιώντας διαφορετικά πρωτόκολλα μεταφοράς ή τεχνολογίες όπως Zigbee, UDP και Bluetooth.




*Το μοντέλο publish-subscribe του MQTT*

### 2.2.1 Μηχανισμός Publish-Subscribe

Ο μηχανισμός publish-subscribe βοηθάει στην επίτευξη των κύριων στόχων του πρωτοκόλλου που είναι η απόδοση σε εύρος ζώνης, η χαμηλή κατανάλωση ενέργειας και η επεκτασιμότητά του. Το πρωτόκολλο ορίζει δύο ειδών οντοτήτων: έναν *message broker*

και ένα πλήθος από *clients.* Ένας broker είναι ένα κομμάτι λογισμικού και μπορεί να τρέχει στο τοπικό δίκτυο ή απομακρυσμένα στο cloud είτε self-hosted είτε επί πληρωμή. Εδώ client θεωρείται οποιαδήποτε συσκευή τρέχει το πρωτόκολλο MQTT και συνδέεται σε έναν broker. Κάθε client έχει και ένα μοναδικό όνομα ή ID (όσο αφορά τον ίδιο broker).

Στο μοντέλο publish-subscribe οι clients δεν επικοινωνούν απευθείας μεταξύ τους αλλά μέσω του broker με μορφή *δημοσίευσης* και *συνδρομής*. Όταν ένας client θέλει να στείλει ένα μήνυμα με δεδομένα το στέλνει στον broker δημοσιεύοντάς το (*publish*) με συγκεκριμένο *topic*. Έπειτα εκείνος το στέλνει σε όλους τους clients οι οποίοι είχανε κάνει εγγραφή (*subscribe*) στο συγκεκριμένο topic ή σε ένα pattern που ταιριάζει σε αυτό.

Πριν όμως γίνει οποιοδήποτε publish-subscribe θα πρέπει να πραγματοποιηθεί σύνδεση στέλνοντας ένα πακέτο *Connect* και ο broker απαντώντας με πακέτο *CONNACK*, ομοίως για *Subscribe-SUBACK* και *Publish-PUBACK*. Προκειμένου να βεβαιωθεί ο client ότι μία σύνδεση που δεν έχει κίνηση για ένα χρονικό διάστημα (περίοδος *Keep-Alive* π.χ. 1 λεπτό) είναι όντως ενεργή, στέλνει ένα πακέτο *PINGREQ* και αναμένει απάντηση *PINGRESP*. Επίσης αν ο broker μετά από μιάμιση περίοδο δε λάβει PINGREQ τότε θεωρεί τον client εκτός και τερματίζει τη σύνδεση.

### 2.2.2 Topics και Wildcards

Τα topics είναι UTF-8 συμβολοσειρές όπου σημασιολογικά συνιστούν «κανάλια» επικοινωνίας στα οποία γίνονται publish και subscribe, φιλτράροντας έτσι τα μηνύματα που λαμβάνουν [22]. Μπορούν να έχουν ιεραρχική δομή χωρισμένα με *slash*, π.χ. «myhome/lights/livingroom/ bulb1». Κάθε client μπορεί να κάνει subscribe σε πολλά topics και κάθε topic να έχει πολλούς subscribers, επιτρέποντας έτσι ευελιξία και επεκτασιμότητα. Για τη δημιουργία topic δεν απαιτείται κάποια έξτρα ενέργεια παρά μόνο δημοσίευσης σε αυτό. Topics που ξεκινούν με το χαρακτήρα “$” δεν είναι εγγράψιμα καθώς είναι του συστήματος και αφορούν πληροφορίες και στατιστικά για το broker.

Χρησιμοποιώντας *wildcards* οι clients κάνουν subscribe (όχι για publish) σε πολλά topics που ταιριάζουν σε ένα μοτίβο, ακόμα και όταν είναι άγνωστο το ακριβής όνομά τους. Υπάρχουν δύο ειδών wildcard. Ο μονού επιπέδου wildcard “+” αντιστοιχεί σε όλα τα topics του ίδιου επιπέδου, π.χ. «myhome/lights/**+**/bulb1» θα αντιστοιχούσε στις πρώτες λάμπες όλων των δωματίων. Ο πολλαπλού επιπέδου wildcard “#” αντιστοιχεί σε όλα τα επιθέματα topics, π.χ. «myhome/lights/#» θα αντιστοιχούσε σε όλα τα φώτα του σπιτιού.

### 2.2.3 Δομή πακέτου

Καθώς το πρωτόκολλο MQTT είναι δυαδικό και όχι κειμένου, η επικοινωνία σε αυτό γίνεται με προκαθορισμένης μορφής πακέτα. Κάθε πακέτο ελέγχου αποτελείται από τα εξής μέρη [17][19]:

i. **Control Header** (1 byte). Τα πρώτα 4 bits είναι ο τύπος πακέτου (CONNECT, PUBLISH, κ.α., 16 συνολικά), ενώ τα επόμενα 4 είναι flags για το συγκεκριμένο τύπο πακέτου.
ii. **Remaining Length** (1-4 bytes). Δηλώνει πόσα bytes ακόμα περιλαμβάνει ολόκληρο το πακέτο μέχρι το τέλος του. Για τη κωδικοποίηση, στο κάθε byte το πρώτο bit δηλώνει συνέχεια και τα υπόλοιπα επτά το μέρος του αριθμού. Οπότε τα πακέτα καταλαμβάνουν από 2 έως θεωρητικά $2^{28}$ bytes. Τα πακέτα ACK έχουν μέγεθος 2 bytes.

iii. **Variable Header** (προαιρετικό). Ανάλογα με το τύπο πακέτου μπορεί να απουσιάζει τελείως, να ξεκινάει με *packet ID*, ή και να ακολουθείται από ένα μη ταξινομημένο σύνολο ιδιοτήτων όπου η καθεμία με δικό της αναγνωριστικό και τιμή.
iv. **Payload** (προαιρετικό). Ορισμένοι τύποι πακέτων περιλαμβάνουν περιεχόμενο για την εφαρμογή. Για παράδειγμα τα πακέτα PUBLISH περιέχουν το σώμα του μηνύματος. Η μορφή του payload καθορίζεται από την εφαρμογή και όχι από το πρωτόκολλο. Σε ορισμένα πακέτα (όπως το SUBACK κ.α.) στο payload περιλαμβάνεται ο 1-byte κωδικός αποτελέσματος (***Reason Code***) με τιμές μικρότερες του 0x80 να δηλώνουν επιτυχία ενώ μεγαλύτερες να δηλώνουν σφάλμα.

Ακολουθεί ενδεικτικό παράδειγμα πακέτου τύπου PUBLISH με συνολικό μήκος 15 bytes, για την έκδοση MQTT/3.1.1. Στη πρώτη στήλη είναι το μέρος του πακέτου, στη δεύτερη αναγράφονται τα bytes σε δεκαεξαδική μορφή χωρισμένα σημασιολογικά για λόγους ευκρίνειας, ενώ στη τρίτη είναι η περιγραφή του κάθε τμήματος. Οι συμβολοσειρές κωδικοποιούνται με UTF-8.

| | | |
|---|---|---|
| Fixed Header | 33 | Τύπος πακέτου PUBLISH (3) με flags (0011=3) DUP=0, QoS=1 και Retain=1 |
| | 0d | Υπολειπόμενο μήκος πακέτου (13) |
| Variable Header | 00 04 | Μήκος τιμής topic (4) |
| | 69 6e 66 6f | Topic=”info” |
| Payload | 00 12 | Αναγνωριστικό πακέτου (18) |
| | 68 65 6c 6c 6f | Message=”hello” |

*Παράδειγμα MQTT/3.1.1 πακέτου PUBLISH*

### 2.2.4 Quality of Service Levels

Το επίπεδο ποιότητας υπηρεσίας (*Quality of Service Level – QoS*) είναι μια συμφωνία μεταξύ του αποστολέα και του παραλήπτη για το επίπεδο εγγύησης μεταφοράς ενός μηνύματος [21]. Έχοντας επίπεδα QoS διευκολύνεται η επικοινωνία σε περιβάλλοντα ασταθούς σύνδεσης όπου απαιτείται αναμετάδοση μηνύματος. Καθώς η μεταφορά μηνύματος έχει δύο στάδια, από publisher σε broker και από broker σε subscriber, οι δύο client ορίζουν ξεχωριστά το QoS που επιθυμούν ώστε ο καθένας να μπορεί να διαλέξει μεταξύ αξιοπιστίας και απόδοσης. Το πρωτόκολλο ορίζει 3 επίπεδα QoS:

- ***QoS 0*** *– Το πολύ μία φορά*. Προσφέρει μηχανισμό *best-effort* όπου ο αποστολέας δεν περιμένει πίσω κανένα πακέτο αναγνώρισης ή εγγύησης αποστολής του μηνύματος.
- ***QoS 1*** *– Τουλάχιστον μία φορά*. Ο αποστολέας στέλνει το μήνυμα και το αποθηκεύει μέχρι να λάβει πακέτο αναγνώρισης *PUBACK* επιβεβαιώνοντας έτσι την αποστολή του. Σε διαφορετική περίπτωση συνεχίζει να το στέλνει κάθε ένα συγκεκριμένο χρονικό διάστημα και θέτει το *DUP flag*. Μόλις ο παραλήπτης στείλει PUBACK καθαρίζει το packet ID, όμως αν χαθεί το PUBACK τότε υπάρχει ασυμφωνία για το αν είναι ακόμα έγκυρο το packet ID, ο αποστολέας συνεχίζει να στέλνει PUBLISH

νομίζοντας πως δεν στάλθηκε, ενώ ο παραλήπτης βλέπει "καινούργιο" packet ID άρα και μήνυμα.

- ***QoS 2*** – *Ακριβώς μία φορά*. Το μήνυμα *επεξεργάζεται* (όχι απλά μεταδίδεται) στον παραλήπτη ακριβώς μία φορά και αυτό γίνεται με *χειραψία 4 βημάτων*. Αρχικά ο αποστολέας στέλνει πακέτο PUBLISH με QoS=2 μέχρι να λάβει πίσω *PUBREC*. Μετά στέλνει πακέτο *PUBREL* μέχρι να λάβει πίσω *PUBCOMP*. Έπειτα ξέρουν και οι δύο ότι το μήνυμα παραδόθηκε και επεξεργάστηκε μια φορά ως καινούργιο, αλλά και ότι ο αποστολέας έλαβε αναγνώριση. Έχοντας το ίδιο packet ID τα πακέτα αναφέρονται στο ίδιο PUBLISH και όχι σε άλλο.

Σε περίπτωση που το QoS μεταξύ των δύο clients διαφέρει τότε τίθεται σε ισχύ το χαμηλότερο το χαμηλότερο από αυτά. Τα packet ID καταλαμβάνουν μόνο 2 byte (έως 65536) διότι είναι διαφορετικά για κάθε ζευγάρι broker-client.

Όσο αφορά το σκοπό χρήσης του κάθε QoS, το QoS=0 είναι για όταν υπάρχει σταθερή σύνδεση και τα μηνύματα δεν είναι κρίσιμα όποτε αποδέχεται μερική απώλεια τους. Το πιο συνηθισμένο QoS=1 για όταν πρέπει να μεταδοθούν όλα τα μηνύματα και αν είναι πολλαπλά δεν παίζει ρόλο όπως αισθητήρες που συχνά λένε την ίδια τιμή, ενώ QoS=2 όταν είναι κρίσιμο να ληφθούν μόνο μία φορά.

### 2.2.5 Retained Messages

Το πρωτόκολλο για λόγους απόδοσης ορίζει πως τα μηνύματα που λαμβάνει και μεταδίδει ο broker να μην αποθηκεύονται, έτσι όταν ένας publisher δημοσιεύει σε ένα topic που δεν υπάρχει κανένας subscriber το μήνυμα αυτό χάνεται [18]. Όμως μαρκάροντας το μήνυμα ως συγκρατούμενο (*retained message*) ο publisher λέει στον broker να το συγκρατήσει. Έτσι όταν εμφανίζονται καινούργιοι subscribers λαμβάνουν αμέσως το τελευταίο συγκρατούμενο μήνυμα χωρίς να περιμένουν την επόμενη ενημέρωση (π.χ. από έναν αισθητήρα που ειδοποιεί μόνο όταν αλλάζει η κατάσταση του, ή για να στείλει οτιδήποτε στοιχεία που δεν αλλάζουν όπως id, IP, version κ.α.). Κάθε topic μπορεί να έχει μόνο ένα συγκρατούμενο μήνυμα, δηλαδή το καινούργιο αντικαταστεί το προηγούμενο, ενώ για να σβηστεί απλά τίθεται ως συγκρατούμενο το κενό μήνυμα.

Ένα άλλο είδος μηνύματος το *Last Will and Testament* (*LWT*) αποθηκεύεται όχι για κάθε topic αλλά για κάθε connection [20]. Κατά τη σύνδεση του ένας publisher μπορεί να ορίσει το LWT του, το οποίο στη περίπτωση που χάσει απρόσμενα τη σύνδεση του (και μόνο τότε) με τον broker, εκείνος το στέλνει στους subscribers του topic, ώστε εκείνοι να το καταλάβουν και να διατηρηθεί όσο γίνεται η ακεραιότητα του συστήματος.

### 2.2.6 Ασφάλεια

Για να ασφαλίσει κάποιος πλήρως ένα σύστημα με MQTT θα πρέπει να κοιτάξει σε τρία επίπεδα: δικτύου, μεταφοράς και εφαρμογής [23]. Για το επίπεδο δικτύου μπορεί να χρησιμοποιηθεί VPN μεταξύ του gateway στο τοπικό δίκτυο των clients και του απομακρυσμένου broker. Για το επίπεδο μεταφοράς στο TCP η επικοινωνία μπορεί να κρυπτογραφηθεί με TLS ώστε να μη διαβάζεται ή τροποποιείται από τρίτους στο διαδίκτυο.

Πέρα από αυτά, στο επίπεδο εφαρμογής το πρωτόκολλο προσφέρει δυνατότητες αυθεντικοποίησης χρήστη. Κατά τη σύνδεση με το broker, ο client με το πακέτο CONNECT στέλνει το *username* του κωδικοποιημένο με UTF-8 και το *password* σε δυαδική μορφή έως 64kb. Αφού ο broker επαληθεύσει τα στοιχεία επιστρέφει με το πακέτο CONNACK το

αποτέλεσμα αποδοχής ή άρνησης της σύνδεσης. Εναλλακτικά ορισμένες υλοποιήσεις broker δέχονται certificate από το client για την επαλήθευσή του.

Με τον όρο εξουσιοδότηση εννοούμε αν κάποιος έχει πρόσβαση ανάγνωσης και εγγραφής για κάποιο κοινόχρηστο πόρο ή δεδομένα. Χωρίς καμία πολιτική εξουσιοδότησης κάθε client μπορεί να γράφει και να διαβάζει οποιοδήποτε topic, πράγμα μη επιθυμητό σε ανοικτό δίκτυο με πιθανόν κακόβουλους χρήστες. Το πρωτόκολλο δεν έχει ενσωματωμένη την εξουσιοδότηση των clients, αλλά το αφήνει στην υλοποίηση του broker υποδεικνύοντας τον έλεγχο του hostname/IP, username, ClientID και topic [19].

### 2.2.7 Διαφορές εκδόσεων MQTT 3.1/3.1.1/5

Ο οργανισμός OASIS Standards παρέλαβε την έκδοση 3.1 από την IBM το 2013 και ένα χρόνο αργότερα έβγαλε την έκδοση 3.1.1 [16]. Καθώς έπρεπε να αυξηθεί η τιμή της έκδοσης στο πακέτο CONNECT από 3 σε 4 η έκδοση αυτή θεωρείται η 4, ενώ η επόμενη η 5 [24].

Η έκδοση 3.1.1 προσθέτει Session Present flag στο πακέτο CONNACK ώστε ο broker να δηλώσει στο client αν έχει διατηρήσει από τη προηγούμενη φορά τα subscriptions, μηνύματα κ.α. [25]. Προστίθενται επιπλέον κωδικοί σφάλματος ή επιτυχίας (*Reason Codes*) για ορισμένα πακέτα αναγνώρισης όπως SUBACK. Τα ClientID μπορούν να είναι κενά για να αναθέτονται αυτόματα από το broker, ενώ το όριο τους αυξάνεται από έως 23 bytes σε έως 64kb.

Η έκδοση 5 εισάγει τις κοινές συνδρομές (*shared subscriptions*) επιτρέποντας τους client να μοιράζονται μεταξύ τους τις ενημερώσεις από πολύ ενεργά topics για αποφυγή υπερφόρτωσης [16][24]. Προστίθενται Reason Codes για όλα τα ACK πακέτα για καλύτερο χειρισμό σφαλμάτων. Εισάγει το πακέτο AUTH για καλύτερες μεθόδους αυθεντικοποίησης τύπου *challenge/response* όπως OAuth. Εξατομικευμένες ιδιότητες (*user properties*) σε ορισμένα πακέτα όπως CONNECT. Τα μηνύματα μπορούν να έχουν καθορισμένη διάρκεια ζωής μέχρι τη παράδοση ή διαγραφή τους. Το Clean Session flag χωρίζεται σε *Clean Start* και *Session Expiry Interval*, δηλώνοντας την επιθυμία του client κατά τη σύνδεσή του για συνέχιση ή όχι της προηγούμενης. Τα topics μπορούν να έχουν ψευδώνυμα (*topic aliases*) *ως* αριθμοί για μείωση του μεγέθους των πακέτων.

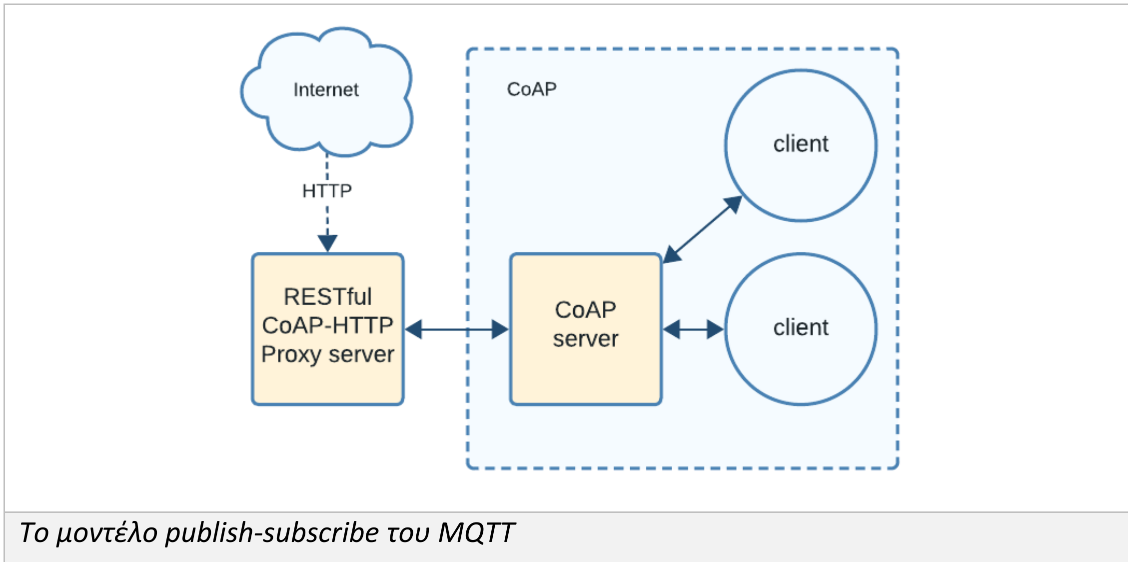
## *2.3* Το πρωτόκολλο CoAP

Το πρωτόκολλο *Constrained Application Protocol* (*CoAP*) είναι ένα εξειδικευμένο πρωτόκολλο επιπέδου εφαρμογής για το διαδίκτυο για χρήση σε περιορισμένους κόμβους ή δίκτυα χαμηλής κατανάλωσης και με απώλειες [26][27]. Τέτοια περιορισμένα δίκτυα και κόμβοι μπορεί να είναι τα ασύρματα δίκτυα αισθητήρων, microcontrollers χαμηλής απόδοσης, δίκτυα *6LoWPAN* υψηλής απώλειας πακέτων και με χαμηλό εύρος ζώνης κ.α. [27]. Πρόκειται για ένα ελαφρύ πρωτόκολλο τύπου client-server και request-response παρόμοιο με το HTTP, ακολουθώντας την αρχιτεκτονική REST όπου η πληροφορία αναπαρίσταται ως πόρους σε κάποιο URI [28].

Η αρχική έκδοση (RFC 7252) εκδόθηκε το 2014 από την ομάδα *CoRE* (*Constrained RESTful Environment*) της IETF (*Internet Engineering Task Force*). Αργότερα ακολούθησαν επεκτάσεις όπως CoAP πάνω σε TCP με TLS, σε WebSockets, και σε SMS [26]. Η επέκταση *Group Communication for CoAP* (*RFC 7390*) βασίζεται στη τεχνική multicast για τη μετάδοση μηνύματος σε group. Η επέκταση *Block-Wise Transfers in CoAP* (*RFC 7959*) αναιρεί τον

περιορισμό του ενός datagram και επιτρέπει την αποστολή μεγαλύτερων μηνυμάτων. Η επέκταση *RFC 7641* προσθέτει τη παρακολούθηση πόρων, ενώ η *RFC 8613* δυνατότητες αυθεντικοποίησης και εξουσιοδότησης σε επίπεδο εφαρμογής.

Καθώς λειτουργεί πάνω στο UDP, εκτός από unicast υποστηρίζει και *multicast* που είναι ιδιαίτερα χρήσιμο για IoT. Με τη τεχνική multicast ένα μήνυμα στέλνεται μόνο μία φορά από τη πηγή, κλωνοποιείται στους ενδιάμεσους κόμβους του δικτύου και φτάνει σε πολλούς προορισμούς μαζί. Άλλα χαρακτηριστικά του CoAP είναι: ασύγχρονη ανταλλαγή μηνύματος, χαμηλό κόστος επικεφαλίδων, μηχανισμός εύρεσης πόρων, χρήση URI (με πρόθεμα *coap* ή *coaps*) και Content-Format για τους πόρους, δυνατότητες Proxy και Caching, και αδιάκοπη δια-λειτουργικότητα με HTTP μέσω Proxy.




*Το μοντέλο publish-subscribe του MQTT*

### 2.3.1 Δομή πακέτου

Στο CoAP η επικοινωνία γίνεται με πακέτα μικρού μεγέθους απλής, προκαθορισμένης, και δυαδικής μορφής [26]. Βασική απαίτηση είναι να είναι αρκούντος μικρά ώστε να καταλαμβάνουν μόνο ένα πακέτο *datagram* στο UDP (ή frame για 6LoWPAN), αυξάνοντας έτσι τη πιθανότητα να μεταδοθεί ολόκληρο το μήνυμα. Το μικρότερο δυνατό πακέτο καταλαμβάνει 4 bytes έχοντας μόνο την επικεφαλίδα. Τα πακέτα CoAP έχουν την ακόλουθη μορφή [26][27]:

i. **Header** (4 bytes).
   - Bits 0-1. Έκδοση πρωτοκόλλου.
   - Bits 2-3. Τύπος πακέτου.
   - Bits 4-7. Μήκος του token.
   - Bits 8-15. Κωδικός αιτήματος ή απάντησης.
   - Bits 16-31. Αναγνωριστικό μηνύματος (message ID).

ii. **Token** (0 έως 8 bytes). Αριθμός που χρησιμοποιείται για την αντιστοίχιση request με response. Για κάθε διαφορετικό request ο client στέλνει και διαφορετικό token στο server και εκείνος το αντιγράφει στο response που επιστρέφει.

iii. **Options** (προαιρετικό). Κάθε μήνυμα μπορεί να έχει ένα πλήθος ρυθμίσεων όπου κωδικοποιούνται ως τριάδες: τύπου, μήκους τιμής, και τιμή. Εμφανίζονται σε

αύξουσα σειρά με βάση το τύπο. Για το τύπο και μήκος τιμής, χρησιμοποιείται μια μορφή delta κωδικοποίησης.

iv. **Payload** (προαιρετικό). Περιλαμβάνει το αληθινό περιεχόμενο του μηνύματος. Όταν υπάρχει ξεκινάει πάντα με τη τιμή 0xFF ενώ το μέγιστο μέγεθος του καθορίζεται από το μέγιστο μέγεθος ενός datagram.

### 2.3.2 Τύποι πακέτων

Υπάρχουν 4 τύποι πακέτων. Ένα request πακέτο μπορεί είναι μη επιβεβαιώσιμο *NON* (*non-confirmable*) ή επιβεβαιώσιμο *CON* (*confirmable*). Τα NON πακέτα στέλνονται από τον client το πολύ μία φορά χωρίς πακέτο αναγνώρισης (παρόμοιο με το QoS=0 του MQTT). Τα CON αναμεταδίδονται κάθε ένα συγκεκριμένο αλλά εκθετικά αυξανόμενο χρονικό διάστημα μέχρι να ληφθεί πακέτο αναγνώρισης ACK. Όταν ο παραλήπτης δε μπορεί να απαντήσει σε CON ούτε με κάποιο σφάλμα τότε απαντάει με τύπο πακέτου *RST* (*reset*) αντί ACK.

Με το message ID αντιστοιχούνται τα requests CON με το δικό τους ACK. Στη περίπτωση που η απάντηση χωράει και βρίσκεται στο ίδιο το πακέτο ACK (λεγόμενο *piggybacked response*) τότε το message ID και token έχουν ίδιο ρόλο [27]. Στην άλλη περίπτωση (*separate response*) ο server απαντάει με κενό ACK και όταν έχει έτοιμο το response στέλνει καινούργιο CON και αναμένει ACK από τον client. Το message ID χρησιμοποιείται επίσης για την εξάλειψη διπλότυπων πακέτων. Η διαφορά του με το token φαίνεται στο CON πακέτο του *separate* όπου μόνο το token πρέπει να παραμείνει ίδιο και όχι το message ID.

### 2.3.3 Κωδικοί αιτήματος και απάντησης

Σε κάθε μήνυμα είτε είναι request είτε response αναγράφεται ο κωδικός κατάστασης που έχει πολλά κοινά στη σημασιολογία με τους κωδικούς HTTP. Στο κανονικό CoAP υπάρχουν **4** κλάσεις κωδικών και καταλαμβάνουν τα 3 πρώτα bits από τα 8 συνολικά [27]. Η **πρώτη** κλάση είναι η μέθοδος αιτήματος και περιλαμβάνει: *EMPTY*, *GET*, *POST*, *PUT*, και *DELETE*. Στη **δεύτερη** είναι ο κωδικός επιτυχίας απάντησης (*Created*, *Deleted*, *Valid* κ.α.). Στη **τρίτη** ο κωδικός σφάλματος της μεριάς του client (*Bad Request, Unauthorized, Forbidden, Precondition Failed* κ.α.) ενώ στη **τέταρτη** της μεριάς του server (*Not implemented*, *Bad gateway,* κ.α.). Στην επέκταση για TCP υπάρχει και **πέμπτη** κλάση με κωδικούς σημάτων (*Ping*, *Pong*, *Release*, *Abort,* κ.α.).

### 2.3.4 CoAP Options

Η λειτουργία των μηνυμάτων μπορεί να προσαρμόζεται με προαιρετικές επιλογές-ρυθμίσεις (*options*) που είναι κοινές για request και response. Καθώς δεν είναι όλα τα option συμβατά με κάθε δυνατό κωδικό αιτήματος/απάντησης ή τύπο πακέτου, χωρίζονται σε critical ή elective ανάλογα με το αν πρέπει να απορριφθεί το μήνυμα ή να αγνοηθεί το option [27]. Εν συντομία, χωρίζονται επιπλέον σε Proxy Safe-to-Forward ή όχι, αν επηρεάζουν το caching, και αν είναι επαναλαμβανόμενες. Υπάρχουν 15 options: *Content-Format, ETag, Location-{Path,Query}, Max-Age, Proxy-{Uri,Scheme}, Uri-{Host,Path,Port,Query}, Accept, If-{Match,None-Match},* και *Size1.*

Το Content-Format option δηλώνει τη μορφή του payload στο μήνυμα. Υποστηρίζονται 6 μορφές (media types): κείμενο UTF-8, Link-Format, octet-stream, XML,

EXI, και JSON. Όταν απουσιάζει θεωρείται απροσδιόριστη μορφή και καθορίζεται από την εφαρμογή. Με το option Accept o client δηλώνει τη προτίμησή του στη μορφή του response.

Με τα επαναλαμβανόμενα υπό συνθήκη options If-Match και If-None-Match ο client ζητάει από τον server να εκτελέσει το request μόνο αν ικανοποιούνται οι συνθήκες αλλιώς εκείνος απαντάει με κωδικό σφάλματος "Precondition Failed". Χρήσιμο σε PUT request για να ελεγχθεί το ETag και να αποφευχθούν κατά λάθος αντικαταστάσεις πόρων.

### 2.3.5 Παρακολούθηση και εύρεση πόρων

Με το *OBSERVE* option του CoAP (επέκταση RFC 7641), ο client δηλώνει μέσω GET request ότι ενδιαφέρεται να παρακολουθεί κάποιο πόρο. Στη συνέχεια ο server στέλνει ασύγχρονα μηνύματα στο client κάθε φορά που ενημερώνεται ο πόρος. Για να ακυρώσει τη παρακολούθηση ο client στέλνει GET request με observe=1 (deregister) ή απαντάει στο επόμενο notification με Reset πακέτο. Έτσι αποφεύγονται το λεγόμενο *polling* που θα γινόταν στη περίπτωση του HTTP, δηλαδή δαπανηρά επαναλαμβανόμενα αιτήματα από το client στο server για να ελέγξει αν άλλαξε η κατάσταση του πόρου [28].

Για την εύρεση πόρων το πρωτόκολλο συνιστά στους CoAP servers να παρέχουν πληροφορίες πόρων στο URI "/.well-known/core", ώστε να το λαμβάνουν οι clients με ένα GET request [28]. Η μορφή αυτού του πόρου ακολουθεί το *CoRE Link Format* (RFC 6690).

### 2.3.6 Block-Wise Transfers

Η επέκταση πρωτοκόλλου *Block-Wise Transfers* (*RFC 7959*) επιτρέπει την αποστολή μεγαλύτερων μηνυμάτων από ένα datagram [29]. Προσθέτει δύο καινούργια block options block1 και block2 για κάθε κατεύθυνση του payload (request ή response αντίστοιχα). Η μεταφορά του μηνύματος γίνεται με ίδιου μεγέθους blocks (έως 1024 το καθένα) το ένα μετά το άλλο με εναλλασσόμενα CON/ACK πακέτα, όπου ο requester δηλώνει με το block2 (δηλ. το αντίθετο του) πιο block επιθυμεί. Στα POST/PUT πακέτα όπου το payload στέλνεται από τον requester οι ρόλοι των block# είναι ανάποδα.

Επειδή το συνολικό μήνυμα στέλνεται με ζευγάρια CON/ACK δεν επιτυγχάνεται μεγάλη χρήση του διαθέσιμου bandwidth αλλά ούτε υποστηρίζονται NON πακέτα. Αυτό αναλαμβάνει η επέκταση-διόρθωση *Block-Wise Transfer Options Supporting Robust Transmission* (*RFC 9177*) προσθέτοντας δύο καινούργια blocks (Q-Block1 και Q-Block2).

### 2.3.7 Ασφάλεια

Το πρωτόκολλο δεν παρέχει ενσωματωμένες μορφές αυθεντικοποίησης ή εξουσιοδότησης. Κατά τη μεταφορά-επικοινωνία μπορεί να χρησιμοποιηθεί *DTLS* (*Datagram Transport Layer Security*) το οποίο είναι αντίστοιχο του TLS για UDP, ώστε τα μηνύματα να μην υποκλαπούν ή αλλοιωθούν από τρίτους κατά τη μεταφορά τους. Η εφαρμογή του DTLS προσθέτει περίπου 13 bytes overhead στο μήνυμα [27]. Χωρίς DTLS χρησιμοποιείται το πρόθεμα *coap* με προκαθορισμένη θύρα τη 5683, ενώ αντίστοιχα με DTLS το πρόθεμα *coaps* και θύρα 5684. Ορίζονται 4 μορφές ασφάλειας [27]:

- **NoSec.** Όταν δε χρησιμοποιείται DTLS, μη ασφαλές.

- **PreSharedKey.** Με DTLS (pre-shared key mode). Καθορίζεται μια κοινή λίστα με κλειδιά (*pre-shared keys*) και κάθε κλειδί περιλαμβάνει μια λίστα διευθύνσεων κόμβων με τους οποίους μπορεί να επικοινωνήσει. Όταν ορισμένοι κόμβοι μοιράζονται το ίδιο κλειδί τότε αυθεντικοποιούνται ως μέλη ίδιας ομάδας αντί ως ξεχωριστοί.
- **RawPublicKey.** Με DTLS όπου η συσκευή έχει έτοιμο από πριν ένα ζευγάρι ασύμμετρου κλειδιού εγκατεστημένο από το κατασκευαστή, χωρίς όμως πιστοποιητικό.
- **Certificate.** Με DTLS όπου η συσκευή έχει και ασύμμετρο κλειδί αλλά και πιστοποιητικό υπογεγραμμένο από κοινή αρχή πιστοποίησης CA. Επιπλέον έχει και λίστα από CA.

# 3 Σύγκριση Πρωτοκόλλων

Σε αυτό το κεφάλαιο συγκρίνονται τα πρωτόκολλα του προηγούμενου κεφαλαίου με βάση τεχνικά κριτήρια όπως είναι η απόδοση τους και η ασφάλεια τους, αλλά και επιχειρησιακά κριτήρια όπως το πεδίο εφαρμογής και η ευκολία εγκατάστασης και χρήσης τους.

## *3.1* Απόδοση

Η απόδοση των πρωτόκολλων επικοινωνίας μπορεί να συγκριθεί ως προς 4 χαρακτηριστικά: χρήση εύρους ζώνης, καθυστέρησης αποστολής μηνύματος, αξιοπιστία αποστολής, και κατανάλωση ρεύματος η οποία είναι ανάλογη της χρήσης πόρων επεξεργαστή.

Οι συγκριτικές έρευνες πρωτοκόλλων IoT [30][31] αναφέρουν ότι το HTTP απαιτεί πολύ περισσότερους πόρους (και άρα κατανάλωση ρεύματος-μπαταρίας) από τα MQTT και CoAP, με το CoAP να απαιτεί τους λιγότερους. Κάτι το οποίο αναμενόταν διότι τα δύο τελευταία σχεδιάστηκαν με πρωταρχικό στόχο την απόδοση, αντίθετα με το HTTP. Επίσης το CoAP λειτουργεί σε UDP που είναι ελαφρύτερο από το TCP που χρησιμοποιεί το MQTT.

Όσο αφορά τη χρήση εύρους ζώνης, αυτό είναι δύσκολο να συγκριθεί καθώς κρίνεται περισσότερο από την εφαρμογή τους δηλαδή το μέγεθος και συχνότητα μηνυμάτων, αλλά και από τις υλοποιήσεις τους. Στην έρευνα τους οι [30] αναφέρουν ότι το CoAP χρησιμοποιεί το λιγότερο, ακολουθούμενο από το MQTT και μετά το HTTP. Αυτό ίσως οφείλεται στη διαφορά μεγέθους επικεφαλίδων αλλά και μεταξύ TCP με UDP.

Για τη καθυστέρηση δημιουργίας και αποστολής μηνύματος, οι [31] πειραματίζονται με μια συσκευή Arduino που τρέχει το client των πρωτοκόλλων επικοινωνώντας μέσω LAN με Laptop που τρέχει το server των πρωτοκόλλων. Μεταξύ MQTT και CoAP, τα αποτελέσματα τους δείχνουν ότι ο μέσος χρόνος δημιουργίας μηνύματος είναι 119 και 419 microseconds αντίστοιχα, και θεωρούν ότι η διαφορά πηγάζει από την υλοποίηση των βιβλιοθηκών τους. Ενώ ο χρόνος αποστολής μηνύματος, για το περιβάλλον δοκιμής τους, είναι 589 και 821 microseconds αντίστοιχα. Μία άλλη έρευνα που αναφέρουν εκεί λέει πως το MQTT έχει χαμηλότερη καθυστέρηση από το CoAP όταν το δίκτυο έχει χαμηλό ρυθμό απώλειας πακέτων, ενώ για υψηλό ρυθμό απώλειας το CoAP έχει μικρότερη καθυστέρηση.

Όσο αφορά την αξιοπιστία αποστολής μηνύματος, στο HTTP είναι περιορισμένη καθώς προσφέρει μόνο ότι προσφέρει το TCP. Αυτό δεν είναι αρκετό διότι ακόμα και όταν το TCP μεταφέρει επιτυχώς το μήνυμα, η συσκευή του παραλήπτη μπορεί να δυσλειτουργήσει ή σβήσει πριν προλάβει να το επεξεργαστεί (π.χ. αν λάβει υπερβολικά αιτήματα ταυτόχρονα ή τελειώσει η μπαταρία). Με τα 3 επίπεδα QoS του MQTT προσφέρεται σιγουριά ότι το μήνυμα όντως θα επεξεργαστεί στο παραλήπτη το πολύ μία φορά (QoS=0), τουλάχιστον μία (QoS=1), και ακριβώς μία (QoS=2). Το CoAP προσφέρει 2 επίπεδα αξιοπιστίας αποστολής ισοδύναμα με τα QoS=0 και QoS=1.

## 3.2 Ασφάλεια

Η σύγκριση της ασφάλειας των πρωτοκόλλων μπορεί να γίνει συγκρίνοντας αν αυτά παρέχουν: κρυπτογράφηση για την ακεραιότητα μηνύματος, αυθεντικοποίηση ή/και εξουσιοδότηση χρήστη.

Με τη κρυπτογράφηση παρέχεται ασφαλής επικοινωνία αποτρέποντας την υποκλοπή, αλλοίωση, ή και εισχώρηση μηνυμάτων από τρίτους κατά τη μεταφορά τους στο δίκτυο. Και τα 3 πρωτόκολλα μπορούν να χρησιμοποιηθούν με κρυπτογράφηση, εφαρμόζοντας TLS για HTTP και MQTT, και DTLS ή IPSec για το CoAP.

Για την αυθεντικοποίηση χρήστη τα HTTP και MQTT έχουν ενσωματωμένο μηχανισμό (ή αλλιώς διευκόλυνση) ενώ το CoAP όχι. Το HTTP διευκολύνει την αυθεντικοποίηση χρήστη με την επικεφαλίδα *Authorization*, είτε *Basic* με απλό plaintext username/password ή *Digest* όπου δηλαδή χρησιμοποιείται κάποιος επιπλέον αλγόριθμος επαλήθευσης ταυτότητας (π.χ. Bearer, OAuth κ.α.). Το MQTT προσφέρει μόνο username/password μέσα στο connect πακέτο.

Για την εξουσιοδότηση κάποιου χρήστη, δηλαδή για το αν του επιτρέπεται ή όχι να έχει πρόσβαση σε ένα πόρο ή λειτουργία, κανένα από τα τρία πρωτόκολλα δεν παρέχει ενσωματωμένες δυνατότητες για το σκοπό αυτό. Φυσικά αυτό δε σημαίνει ότι είναι αδύνατον, αλλά ότι θα πρέπει (αν χρειάζεται) να εφαρμοστεί στο επίπεδο της εφαρμογής.

## 3.3 Ευκολία μάθησης και ανάπτυξης-χρήσης

Η πολυπλοκότητα ενός πρωτοκόλλου επηρεάζει έμμεσα το αποτύπωμα σε μνήμη, χώρο και χρήση πόρων επεξεργαστή της υλοποίησής του, αλλά και το βαθμό δυσκολίας ανάπτυξης-χρήσης για το προγραμματιστή. Στη συγκριτική έρευνα [32] συγκρίνονται τα MQTT, CoAP (κ.α.) ως προς τη πολυπλοκότητά τους. Για να εκτιμηθεί με κάπως ποσοτικό τρόπο η πολυπλοκότητα ένας τρόπος είναι να συγκριθεί ο όγκος των προδιαγραφών τους και το πλήθος των καθορισμένων στοιχείων πρωτοκόλλου αλλά και το πλήθος των τύπων μηνυμάτων.

Πιο συγκεκριμένα για τον όγκο των προδιαγραφών το CoAP έχει 112 μονάδες έναντι 137 του MQTT, που δηλώνει ότι είναι πιο απλό ως προς τη μάθησή του [32]. Το CoAP έχει μόνο 7 τύπους μηνυμάτων ενώ το MQTT 14. Τα καθορισμένα στοιχεία πρωτοκόλλου είναι 24 έναντι 99, υποδεικνύοντας ότι το CoAP είναι πιο απλό και στην ανάπτυξη-χρήση. Επίσης αξίζει να σημειωθεί ότι αυτό εξαρτάται και από τις υλοποιήσεις, και επίσης διαφορετικές υλοποιήσεις μπορεί να υλοποιούν διαφορετικές επεκτάσεις και να έχουν διαφορετικό σύνολο λειτουργιών πρωτοκόλλου.

## 3.4 Πεδία εφαρμογής

Καθώς το πρωτόκολλο HTTP αρχικά σχεδιάστηκε για την επικοινωνία μεταξύ browser και server, τα πεδία εφαρμογής του είναι πάρα πολλά και δε περιορίζονται στο IoT. Το CoAP σχεδιάστηκε σαν παραλλαγή του HTTP για IoT, ενώ το MQTT σχεδιάστηκε εξαρχής για δίκτυα αισθητήρων. Οι συγγραφείς στο [32] διακρίνουν 4 τύπους συσκευών στο IoT. Οι *περιορισμένες συσκευές* που αποσκοπούν εξοικονόμηση ενέργειας πρέπει να ελαχιστοποιούν τον όγκο και τη συχνότητα μεταφοράς μηνυμάτων, και αν έχουν λίγη μνήμη και επεξεργαστική ισχύ τότε θέλουν επιπλέον απλά πρωτόκολλα. Για τις συσκευές όπου η σύνδεση στο δίκτυο κοστίζει (δίκτυα χαμηλής κατανάλωσης) τότε

θέλουν να ελαχιστοποιείται ο χρόνος σύνδεσης και ο όγκος δεδομένων. Οι άλλοι 2 τύποι συσκευών αφορούν αν μένουν συνέχεια ενεργές και συνδεδεμένες στο δίκτυο ή αν προσωρινά βγαίνουν εκτός. Τα MQTT και CoAP είναι εφαρμοστά και για τους 4 τύπους συσκευών ενώ το HTTP όχι.

Εφαρμογές των πρωτοκόλλων συμπίπτουν με τις εφαρμογές IoT που έχουν αναφερθεί στο κεφάλαιο 1. Λίγο ειδικότερα [30]: το HTTP χρησιμοποιείται κυρίως για ιστοσελίδες στο διαδίκτυο, το MQTT για αυτοματισμό σε σπίτια και εμπορικού επιπέδου εφαρμογές, ενώ το CoAP για έξυπνα σπίτια και πόλεις και κατασκευαστικές.

## *3.5* Συγκριτικός Πίνακας

Παρατίθεται περιληπτικός συγκριτικός πίνακας, προσαρμοσμένος από [29][31]:

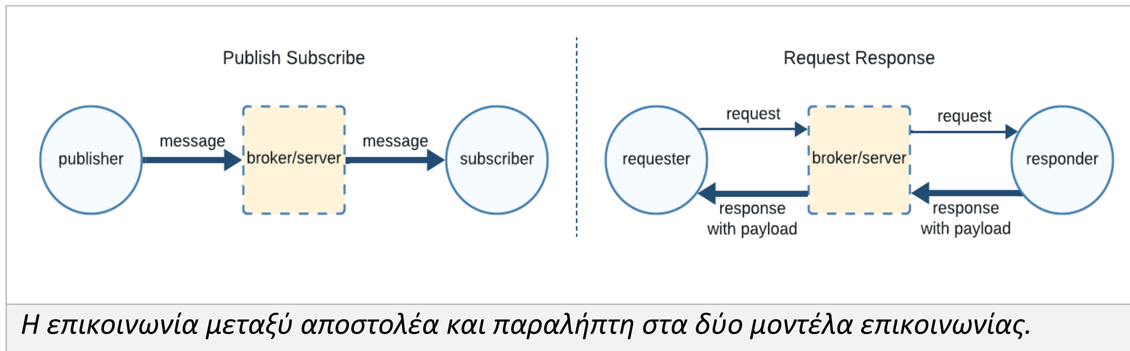
| | **HTTP** | **MQTT** | **CoAP** |
|---|---|---|---|
| **Αρχιτεκτονική** | Client-Server | Client-Broker | Client-Server, P2P |
| **Μοντέλο** | Request-Response (RESTful) | Publish-Subscribe (Topics) | Request-Response, Publish-Subscribe (RESTful και topics με επέκταση) |
| **Πρωτόκολλο μεταφοράς** | TCP | TCP | UDP |
| **Ασφάλεια** | TLS | TLS | DTLS, IPSec |
| **Συνδεσιμότητα** | Ένα προς ένα | Ένα προς ένα, ένα προς πολλά (με topics) | Ένα προς ένα, πολλά προς πολλά (με multicast) |
| **Μέγεθος επικεφαλίδας** | Μη καθορισμένο | 2 Byte | 4 Byte |
| **Μέγιστο μέγεθος μηνύματος** | Μη καθορισμένο | 256 MiB | 1 KiB (ένα datagram) ή έως 1 GiB (με επέκταση) |
| **Κατανάλωση ενέργειας** | Υψηλή | Χαμηλή | Χαμηλή |
| **Καθυστέρηση** | Υψηλή | Χαμηλότερη από HTTP | Χαμηλότερη από MQTT |
| **Χρήση εύρους ζώνης** | Υψηλή | Χαμηλή | Χαμηλότερη (λόγω UDP) |
| **Επίπεδο ποιότητας υπηρεσίας (QoS)** | Περιορισμένη (ότι παρέχεται από TCP) | QoS 0,1,2. Το πολύ μία, τουλάχιστον μία, ακριβώς μία. | Επιβεβαιώσιμο ή όχι (παρόμοιο με QoS 0 ή 1) |

*Πίνακας σύγκρισης πρωτοκόλλων HTTP, MQTT και CoAP*

# 4 Υλοποίηση συνδέσεων

## *4.1* Μεθοδολογία πειράματος

Στο πρακτικό κομμάτι της πτυχιακής δημιουργείται μια μικρή προσομοίωση με στόχο να μετρηθεί και συγκριθεί η απόδοση των πρωτοκόλλων MQTT-CoAP σε χρήση πόρων επεξεργαστή, η οποία κατ’ επέκταση είναι ανάλογη της κατανάλωσης ενέργειας. Θεωρώ ότι μια εφαρμογή IoT θα βασίζεται σε ένα από τα δύο μοντέλα επικοινωνίας, είτε Publish-Subscribe είτε Request-Response, και γι’ αυτό συγκρίνω την εφαρμογή των δύο πρωτοκόλλων στο κάθε ένα μοντέλο εφαρμογής ξεχωριστά.




*Η επικοινωνία μεταξύ αποστολέα και παραλήπτη στα δύο μοντέλα επικοινωνίας.*

Για να μετρηθεί η απόδοση σε κάθε μία από τις 4 περιπτώσεις μετριέται ο μέσος χρόνος αποστολής για διάφορα μεγέθη μηνυμάτων, από τη στιγμή που στέλνεται το μήνυμα μέχρι όταν είναι διαθέσιμο στο παραλήπτη στο επίπεδο εφαρμογής. Περιλαμβάνει τη κωδικοποίηση-αποκωδικοποίηση και το όποιο κόστος φέρει το πρωτόκολλο μεταφοράς. Καθώς ο αποστολέας και ο παραλήπτης βρίσκονται στην ίδια συσκευή, ο χρόνος δε περιλαμβάνει τη καθυστέρηση δικτύου (latency) αλλά ούτε υπάρχουν απώλειες πακέτων.

Το πείραμα εκτελείται σε υπολογιστή με λειτουργικό σύστημα *Windows 10 Pro (Version 22H2 Build 19045)*, επεξεργαστή *Intel Pentium G3450 @3.40GHz*, και *16 GB RAM*. Χρησιμοποιείται η *Python 3.11.6* με τις βιβλιοθήκες: *Paho MQTT 2.1.0* για τον MQTTv5 client, και *aiocoap 0.4.7* για τον CoAP client και server. Για τον MQTTv5 broker χρησιμοποιείται ο *Mosquitto Broker 2.0.18*.

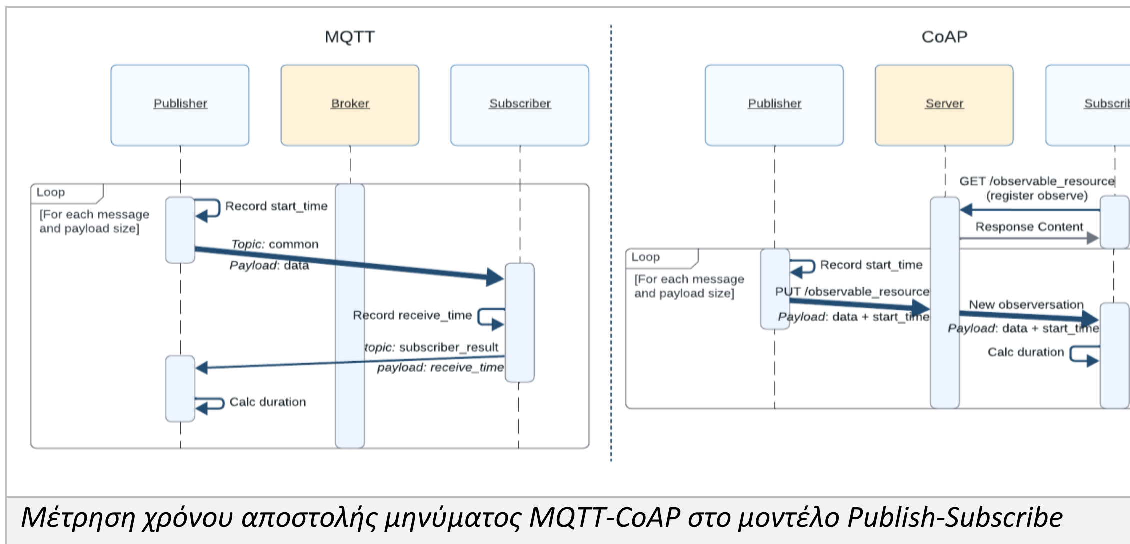
## *4.2* Publish-Subscribe

### 4.2.1 Περιγραφή

Για τη δοκιμή του MQTT υλοποιείται πρόγραμμα για τον publisher και για τον subscriber ξεχωριστά. Αρχικά ο publisher στέλνει κάθε μήνυμα με συγκεκριμένο μέγεθος payload μέσω του broker στον subscriber, χρησιμοποιώντας συγκεκριμένο topic. Έπειτα εκείνος μόλις το λάβει σημειώνει και στέλνει πίσω το χρόνο που το έλαβε, σε διαφορετικό topic. Έτσι ο publisher λαμβάνει το αποτέλεσμα και υπολογίζει το χρόνο που καταναλώθηκε. Αυτό γίνεται για κάθε μήνυμα και κάθε μέγεθος payload. Επίσης με αυτό το συντονισμό τους αποφεύγεται ο παραλληλισμός για τα μηνύματα. Όσο για το QoS, που όπως αναφέρθηκε στην ανάλυση πρωτοκόλλου μπορεί να είναι διαφορετικό μεταξύ των δύο clients, εδώ τίθεται το ίδιο για καλύτερη σύγκριση.

Για το CoAP επίσης υλοποιούνται δύο προγράμματα ξεχωριστά. Αρχικά ο subscriber κάνοντας GET request σε ένα observable resource μαζί το option observe=0 (register),

γνωστοποιεί στο server ότι επιθυμεί να λαμβάνει ειδοποιήσεις για τη κατάσταση του. Ο publisher κάνει PUT request στο ίδιο resource με συγκεκριμένο μέγεθος payload μαζί με το χρόνο που το έστειλε. Για να αποφευχθεί ο παραλληλισμός περιμένει λίγο χρόνο πριν το επόμενο μήνυμα ώστε να το έχει λάβει ο subscriber. Εκείνος μόλις λάβει το μήνυμα, δηλ. ενημέρωση του resource, υπολογίζει και καταγράφει τη διαφορά χρόνου.




*Μέτρηση χρόνου αποστολής μηνύματος MQTT-CoAP στο μοντέλο Publish-Subscribe*

### 4.2.2 Κώδικας προγράμματος

Ακολουθεί ο κώδικας και έπειτα περιγραφή του για το πείραμα σύγκρισης MQTT-CoAP στο μοντέλο Publish-Subscribe. Αποτελείται από 5 βασικά αρχεία που παρουσιάζονται με την ακόλουθη σειρά: *pubsub_mqtt_publisher.py*, *pubsub_mqtt_subscriber.py*, *pubsub_coap_publisher.py*, *pubsub_coap_subscriber.py*, και *coap_server.py*, αλλά και 1 βοηθητικό αρχείο *util.py* που περιέχει τις κοινές σταθερές. Για λόγους ευκρίνειας τα κοινά κομμάτια του κώδικα όπως imports και αρχικοποίηση σύνδεσης παρουσιάζονται μόνο μία φορά και παραλείπονται τις επόμενες, καθώς και εντολές print μικρής σημασίας.

```
 1 import paho.mqtt.client as mqtt; import time; import util
 2 curr_payload_size_idx = num_messages_done = start_time = avg_duration = 0
 3
 4 def on_connect(client: mqtt.Client, userdata, flags, reason_code, properties):
 5     client.subscribe(util.TOPIC_TEST_MSGS_RESULT, qos=2)
 6     send_next_message()
 7
 8 def send_next_message():
 9     global curr_payload_size_idx, start_time
10     payload_size = util.PAYLOAD_SIZES[curr_payload_size_idx]
11     payload = "DATA"*(payload_size//4)
12     start_time = time.time()
13     client.publish(topic=util.TOPIC_TEST_MSGS, payload=payload, qos=util.QOS)
```

```
14
15 def on_message(client: mqtt.Client, userdata, msg: mqtt.MQTTMessage):
16     global avg_duration, num_messages_done, curr_payload_size_idx
17     lastmsg_time = float(msg.payload.decode("ascii"))
18     duration = (lastmsg_time - start_time)*1000
19     avg_duration += duration
20     num_messages_done += 1
21     if num_messages_done < util.NUM_MESSAGES:
22         send_next_message()
23     else: # done with all messages with this payload size
24         num_messages_done = 0
25         avg_duration /= util.NUM_MESSAGES
26         print(f"Finished testing with payload size: \
27             {util.PAYLOAD_SIZES[curr_payload_size_idx]}. \
28             Avg Duration: {avg_duration}")
29         util.save_result_to_file(util.OUTPUT_FILE_PUBSUB, util.NUM_MESSAGES,\
30             util.QOS, util.PAYLOAD_SIZES[curr_payload_size_idx], avg_duration)
31         curr_payload_size_idx += 1
32         if curr_payload_size_idx < len(util.PAYLOAD_SIZES):
33             # first message of next payload size
34             send_next_message()
35         else:
36             # finished with all payload sizes
37             client.disconnect()
38
39 # common initialization code
40 client = mqtt.Client(client_id="", protocol=mqtt.MQTTv5, \
41 callback_api_version=mqtt.CallbackAPIVersion.VERSION2)
42 client.on_connect = on_connect
43 client.on_message = on_message
44 client.on_subscribe = on_subscribe
45 client.on_disconnect = on_disconnect
46 client.connect(util.MQTT_HOST, util.MQTT_PORT, keepalive=60)
47 try:
48     client.loop_forever()
49 except KeyboardInterrupt:
50     client.disconnect()
```

*Αρχείο pubsub_mqtt_publisher.py*

Η βιβλιοθήκη PahoMQTT ακολουθάει το μοντέλο προγραμματισμού οδηγούμενο από συμβάντα (event-driven). Στις γραμμές 39-50, οι οποίες είναι κοινές για όλα τα MQTT clients, αρχικοποιείται ο client με τα απαραίτητα event handlers και συνδέεται στο broker. Στη γραμμή 48 η εντολή *client.loop_forever()* μπλοκάρει την έξοδο του προγράμματος και διαχειρίζει στο παρασκήνιο το event queue. Μόλις συνδεθεί ο publisher κάνει subscribe για να λάβει αργότερα το αποτέλεσμα του subscriber. Η συνάρτηση *send_next_message()* δημιουργεί το payload με το ζητούμενο μέγεθος και το δημοσιεύει σε topic που ακούει ο subscriber, αφού πρώτα καταγράψει το χρόνο έναρξης αποστολής μηνύματος. Η συνάρτηση *on_message()* εκτελείται όταν ληφθεί μήνυμα από τον subscriber, υπολογίζει τη διαφορά χρόνου σε milliseconds, στέλνει το επόμενο μήνυμα, και όταν χρειαστεί ανανεώνει το μέγεθος του payload και αποθηκεύει το αποτέλεσμα σε αρχείο.

```
1 def on_connect(client: mqtt.Client, userdata, flags, reason_code, properties):
2     client.subscribe(util.TOPIC_TEST_MSGS, qos=util.QOS)
3
4 def on_message(client: mqtt.Client, userdata, msg: mqtt.MQTTMessage):
5     lastmsg_time = time.time()
6     client.publish(topic=util.TOPIC_TEST_MSGS_RESULT, qos=2, \
7                    payload=str(lastmsg_time))
```

*Αρχείο pubsub_mqtt_subscriber.py*

Μόλις συνδεθεί ο subscriber στο broker κάνει subscribe σε συγκεκριμένο topic για να δεχτεί τα μηνύματα από τον publisher. Το QoS του subscriber αλλά και του publisher για το μήνυμα δοκιμής είναι ίδιο (γρ. 2 εδώ και γρ. 13 πριν). Στη συνάρτηση *on_message()* μόλις ληφθεί το μήνυμα στέλνεται πίσω ο χρόνος παραλαβής του με καινούργιο publish αλλά σε διαφορετικό topic που ακούει ο publisher (γρ. 6 εδώ και γρ. 5 πριν). Ακολουθεί ο κώδικας για το CoAP.

```
 1 import asyncio; import aiocoap as aio; import struct
 2 from aiocoap.numbers.codes import Code
 3
 4 async def main():
 5     context = await aio.Context.create_client_context()
 6
 7     for payload_size in u.PAYLOAD_SIZES:
 8         padding = b'DATA'*(payload_size//4)
 9         for i in range(u.NUM_MESSAGES):
10             start_time = time.time()
11             # send current time + payload padding
12             payload = struct.pack(">d", start_time) + padding
13             request_msg = aio.Message(code=Code.PUT, payload=payload \
14                 uri=f"coap://{u.COAP_HOST}:{u.COAP_PORT}/observable_resource")
```

```
15              response = await context.request(request_msg).response
16              await asyncio.sleep(delay=0.050)
17
18 asyncio.run(main())
```

*Αρχείο pubsub_coap_publisher.py*

Η βιβλιοθήκη aiocoap ακολουθάει το μοντέλο του ασύγχρονου προγραμματισμού: στην εντολή 18 προστίθεται η *main()* στο event queue του asyncio και αυτό επιτρέπει τη χρήση του await. Στις γραμμές 10-12 δημιουργείται το payload με το σωστό μέγεθος περιέχοντας και το χρόνο αποστολής ως double (γρ. 12). Στις γραμμές 13-15 στέλνεται το μήνυμα ανανέωσης στο “*observable_resource*” resource με ρήμα PUT, και αναμένεται το response του (με το keyword await). Αυτό γίνεται για κάθε μήνυμα και κάθε μέγεθος payload (γρ. 7-9). Με το χρόνο αδράνειας ανάμεσα σε κάθε αποστολή μηνύματος (γρ. 16) αποφεύγεται εύκολα ο παραλληλισμός μηνυμάτων για καλύτερες εκτιμήσεις χρόνου.

```
 1 num_messages_done = avg_duration = 0
 2
 3 def observe_callback(message: aiocoap.Message):
 4     global avg_duration, num_messages_done
 5     [start_time] = struct.unpack(">d", message.payload[:8])
 6     duration = (time.time() - start_time)*1000
 7     avg_duration += duration
 8     num_messages_done += 1
 9     if num_messages_done == util.NUM_MESSAGES:
10         # done with all messages with this payload size
11         num_messages_done = 0
12         avg_duration /= util.NUM_MESSAGES
13         payload_size = len(message.payload) - 8
14         util.save_result_to_file(util.OUTPUT_FILE_PUBSUB, util.NUM_MESSAGES, \
15             "COAP", payload_size, avg_duration)
16         avg_duration = 0
17
18 async def main():
19     context = await aiocoap.Context.create_client_context()
20     request_msg = aiocoap.Message(code=aiocoap.GET, \
21         uri=f"coap://{util.COAP_HOST}:{util.COAP_PORT}/observable_resource")
22     request_msg.opt.observe = 0
23     pr = context.request(request_msg)
24     pr.observation.register_callback(observe_callback)
25     response = await pr.response
```

```
26      print(f"RespCode: {response.code} Payload[:10]: {response.payload[:10]}")
27      await asyncio.sleep(100000000) # let program run in event queue
```

*Αρχείο pubsub_coap_subscriber.py*

Η συνάρτηση *main()* του subscriber δημιουργεί για το "*observable_resource*" το αρχικό GET request με το option observe=0 και προσθέτει το event handler για τη παρακολούθηση του (γρ. 19-24). Στις γραμμές 25-26 στέλνεται το αρχικό request και εκτυπώνεται το response του με το προϋπάρχον περιεχόμενό. Η συνάρτηση *observe_callback()* που καλείται με κάθε ανανέωση του resource (δηλ. PUT από publisher), διαβάζει το χρόνο έναρξης και υπολογίζει τη διάρκεια (γρ. 5-7). Έπειτα μόλις τελειώσει με τα μηνύματα συγκεκριμένου μεγέθους (γρ. 9) αποθηκεύει τη μέση διάρκεια στο ίδιο αρχείο με του mqtt νωρίτερα (γρ. 14 και γρ. mqtt_publisher:29).

```
 1 import aiocoap.resource as resource
 2 
 3 class ObservResource(resource.ObservableResource):
 4     def __init__(self):
 5         super().__init__()
 6         self.content = b"Default content. Resource that can be PUT and observed."
 7     async def render_get(self, request):
 8         return aiocoap.Message(payload=self.content)
 9     async def render_put(self, request: aiocoap.Message):
10         self.content = request.payload # save content
11         self.updated_state() # notify the observers
12         return aiocoap.Message(code=Code.CHANGED, payload=b'')
13 
14 async def main():
15     root = resource.Site()
16     root.add_resource(['.well-known', 'core'],
17             resource.WKCResource(root.get_resources_as_linkheader))
18     root.add_resource(['observable_resource'], ObservResource())
19     await aiocoap.Context.create_server_context(site=root, \
20         bind=(util.COAP_HOST, util.COAP_PORT))
21     await asyncio.get_running_loop().create_future() # run forever
```

*Αρχείο coap_server.py*

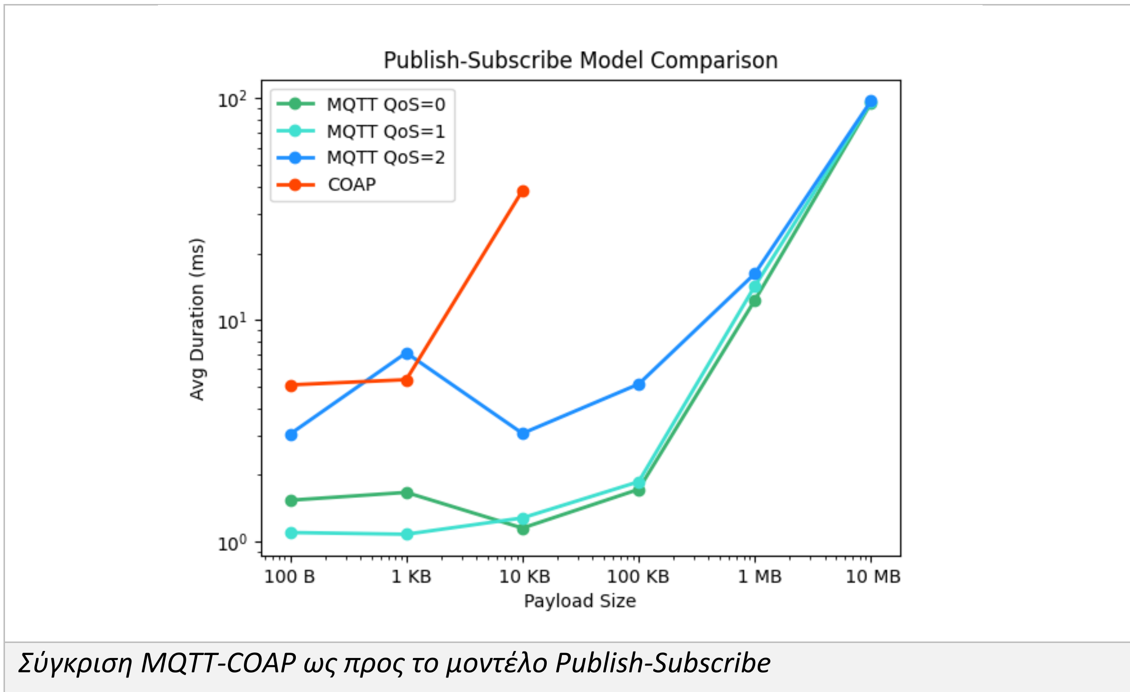
Ο CoAP server υλοποιείται επίσης με την ίδια βιβλιοθήκη aiocoap. Στις γραμμές 15-18 δημιουργούνται τα resources με τα URI τους, εδώ το "observable_resource". Το ".well-known/core" αν και δε χρησιμοποιείται εδώ όταν ζητείται επιστρέφει λίστα με τα resources στο server. Έπειτα ξεκινάει ο server στη τοπική IP και περιμένει αιτήματα (γρ. 19-21). Όταν ζητείται το "observable_resource" με GET απλά επιστρέφεται το περιεχόμενό του (γρ. 7-8), ενώ με PUT επιστρέφεται ο κωδικός CHANGED (γρ. 12) και ταυτόχρονα στέλνεται το καινούργιο περιεχόμενο σε όσους έχουν ενεργό observe (γρ. 11).

```
 1
 2 NUM_MESSAGES = 100 # num of messages for averaging time
 3 PAYLOAD_SIZES = [100, 1_000, 10_000, 100_000, 1_000_000, 10_000_000]
 4 QOS = 0 # same QOS for publisher and subscriber
 5 MQTT_HOST, MQTT_PORT = "...", 1883
 6 COAP_HOST, COAP_PORT = "...", 5683
 7 TOPIC_TEST_MSGS = "common_topic" # topic for sending test messages
 8 TOPIC_TEST_MSGS_RESULT = "subscriber_results" # topic for getting subscriber res
 9 TOPIC_REQ_CMD = "request_topic" # used for reqres model
10 TOPIC_RES_CMD = "response_topic" # used for reqres model
11 OUTPUT_FILE_PUBSUB = "pubsub_results.txt"
12 OUTPUT_FILE_REQRES = "reqres_results.txt"
13
14 import argparse
15 parser = argparse.ArgumentParser()
16 parser.add_argument('--qos', dest='qos', type=int, help='MQTT QoS')
17 args = parser.parse_args()
18 if args.qos:
19     QOS = args.qos
20
21 def save_result_to_file(filename, num_msgs, qosORcoap: int|str, \
22                         payload_size, avg_duration):
23     import os.path
24     existed = os.path.isfile(filename)
25     with open(filename, "a") as f:
26         if not existed:
27             f.write("num_messages,qosORcoap,payload_size,avg_duration\n")
28         f.write(f"{num_msgs},{qosORcoap},{payload_size},{avg_duration}\n")
29
```

*Αρχείο util.py*

Στο *util.py* ορίζονται οι κοινές σταθερές. Γραμμή 2 τα έξι μεγέθη payload που δοκιμάζονται. Για τις διευθύνσεις των servers (γρ. 4-5) τοποθετείται η τοπική IP του συστήματος (από την εντολή *ipconfig*). Οι γραμμές 13-18 επιτρέπουν τη ρύθμιση του QoS από το command line. Η συνάρτηση *save_result_to_file()* λαμβάνει το αποτέλεσμα μιας δοκιμής και το γράφει σε αρχείο τύπου CSV βάζοντας επικεφαλίδα αν χρειάζεται.

### 4.2.3 Αποτελέσματα




*Σύγκριση MQTT-COAP ως προς το μοντέλο Publish-Subscribe*

| Payload | 100 B | 1 KB | 10 KB | 100 KB | 1 MB | 10 MB |
|---|---|---|---|---|---|---|
| **MQTT QoS=0** | 1.53 | 1.67 | 1.15 | 1.72 | 12.29 | 95.25 |
| **MQTT QoS=1** | 1.10 | 1.08 | 1.28 | 1.86 | 14.23 | 96.42 |
| **MQTT QoS=2** | 3.05 | 7.11 | 3.08 | 5.14 | 16.20 | 97.37 |
| **COAP** | 5.10 | 5.38 | 38.36 | - | - | - |

*Τα παραπάνω δεδομένα σε μορφή πίνακα. Οι χρόνοι είναι σε milliseconds.*

Όπως βλέπουμε στο σχήμα οι χρόνοι που χρειάζεται το CoAP είναι μεγαλύτεροι από το MQTT για όλα τα QoS, κάτι το οποίο αναμενόταν αφού το πρωτόκολλο αρχικά σχεδιάστηκε για request-response μοντέλο. Για payload ≤ 1 KB και τα δύο πρωτόκολλα παίρνουν σχεδόν τον ίδιο χρόνο και πιθανότατα οφείλεται στο ότι δεν γίνεται fragmentation πακέτων στο πρωτόκολλο μεταφοράς. Ενώ για payload > 1 KB βλέπουμε ότι το CoAP αυξάνεται ταχύτερα, αφού το Block-Wise Transfer που χρησιμοποιεί στέλνει τα πακέτα με σειριακό τρόπο σε αντίθεση με το TCP του MQTT που χρησιμοποιεί TCP Windowing. Τα ελλιπής δεδομένα του CoAP οφείλονται σε άγνωστη ανικανότητα της βιβλιοθήκης aiocoap.

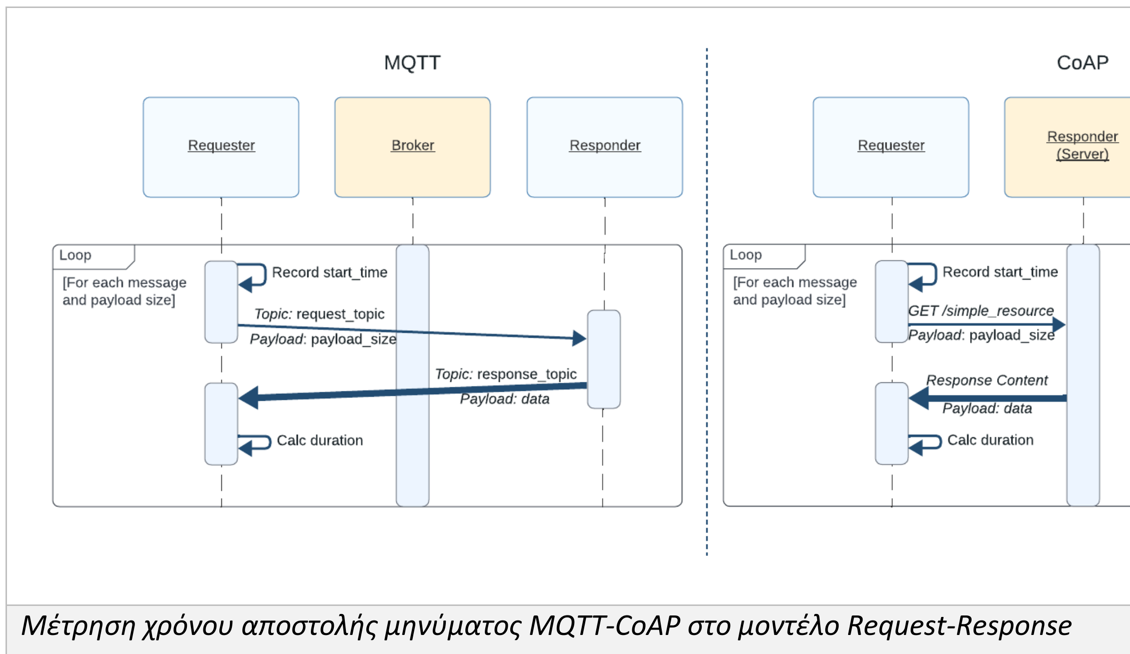
Για μεταξύ των QoS, βλέπουμε ότι τα QoS=0 και QoS=1 έχουν μικρή διαφορά μεταξύ τους και μεγάλη με το QoS=2. Αυτό ίσως γιατί μπορεί ο Broker να περιμένει να προχωρήσει περισσότερο η χειραψία 4 βημάτων του QoS=2 πριν το στείλει στον subscriber. Η απρόσμενη κορυφή στο QoS=2 για 1 KB μάλλον οφείλεται σε διακοπή το OS.

## 4.3 Request-Response

### 4.3.1 Περιγραφή

Για την εφαρμογή του MQTT στο Request-Response μοντέλο εφόσον δε προορίζεται για αυτό, απαιτούνται περισσότερα βήματα. Για το ανάλογο ενός GET request, αρχικά ο requester στέλνει μήνυμα μέσω publish σε topic που παρακολουθεί ο responder. Κανονικά το μήνυμα αυτό περιέχει δεδομένα αιτήματος και ένα response topic για λόγους συντονισμού, εδώ απλά περιέχει το payload size ως συμβολοσειρά και χωρίς το response topic διότι είναι σταθερό και δε χρειάζεται. Έχοντας το payload size ο responder απαντάει με το ζητούμενο μέγεθος payload στο response topic το οποίο ήδη από πριν έχει κάνει subscribe ο requester. Μόλις εκείνος το λάβει υπολογίζει και καταγράφει το συνολικό χρόνο αποστολής του προσομοιωμένου GET request.

Για το CoAP τα πράγματα είναι πιο απλά καθώς το πρωτόκολλο προορίζεται για αυτό το μοντέλο επικοινωνίας. Εδώ ο requester παραμένει CoAP client αλλά ο responder είναι ουσιαστικά ο CoAP server, καθώς αυτό ισχύει σε μία κανονική εφαρμογή του CoAP. Όπως και νωρίτερα ο requester στέλνει μαζί με το GET request το επιθυμητό για τη δοκιμή payload size στον responder, και εκείνος απαντάει με το αντίστοιχο payload, και τελικά καταγράφεται ο χρόνος.




*Μέτρηση χρόνου αποστολής μηνύματος MQTT-CoAP στο μοντέλο Request-Response*

### 4.3.2 Κώδικας προγράμματος

Ακολουθεί ο κώδικας και η περιγραφή του για τη σύγκριση MQTT-CoAP στο μοντέλο Request-Response. Αποτελείται από τα 4 αρχεία: *reqres_mqtt_requester.py, reqres_mqtt_responder.py, reqres_coap_requester.py* αλλά και τις προσθήκες στο *coap_server.py*. Παραλείπεται κοινός κώδικας που ήδη αναφέρθηκε στη προηγούμενη ενότητα όπως imports, κώδικας σύνδεσης mqtt, αλλά και *util.py*.

```
1 curr_payload_size_idx = num_messages_done = start_time = avg_duration = 0
```

```
 2
 3 def on_connect(client: mqtt.Client, userdata, flags, reason_code, properties):
 4     client.subscribe(util.TOPIC_RES_CMD, qos=util.QOS)
 5     send_next_message()
 6
 7 def send_next_message():
 8     global curr_payload_size_idx, start_time
 9     # tell responder the payload size to use for that response
10     payload = str(util.PAYLOAD_SIZES[curr_payload_size_idx])
11     start_time = time.time()
12     client.publish(topic=util.TOPIC_REQ_CMD, payload=payload, qos=util.QOS)
13
14 def on_message(client: mqtt.Client, userdata, msg: mqtt.MQTTMessage):
15     global avg_duration, num_messages_done, curr_payload_size_idx
16     duration = (time.time() - start_time)*1000
17     avg_duration += duration
18     num_messages_done += 1
19     if num_messages_done < util.NUM_MESSAGES:
20         send_next_message()
21     else: # done with all messages with this payload size
22         num_messages_done = 0
23         avg_duration /= util.NUM_MESSAGES
24         print(f"Finished testing with payload size: \
25             {util.PAYLOAD_SIZES[curr_payload_size_idx]}. \
26             Avg Duration: {avg_duration}")
27         util.save_result_to_file(util.OUTPUT_FILE_REQRES, util.NUM_MESSAGES,\
28             util.QOS, util.PAYLOAD_SIZES[curr_payload_size_idx], avg_duration)
29         curr_payload_size_idx += 1
30         if curr_payload_size_idx < len(util.PAYLOAD_SIZES):
31             # first message of next payload size
32             send_next_message()
33         else:
34             # finished with all payload sizes
35             client.disconnect()
```

*Αρχείο reqres_mqtt_requester.py*

Μόλις συνδεθεί ο requester στο broker κάνει subscribe στο “response_topic” για να λάβει αργότερα την απάντηση από τον responder (γρ. 4) και στέλνει το πρώτο μήνυμα. Κάθε μήνυμα που στέλνεται στην *send_next_message()* περιέχει το τωρινό μέγεθος payload για το response. Όταν λάβει την απάντηση στη συνάρτηση *on_message()* υπολογίζει τη

συνολική διάρκεια (γρ. 16) και στέλνει το επόμενο μήνυμα (γρ. 19-20). Μόλις τελειώσουν τα μηνύματα για το συγκεκριμένο μέγεθος payload υπολογίζει τη μέση διάρκεια τους και τη καταγράφει σε αρχείο (γρ. 21-35).

```
1 def on_connect(client: mqtt.Client, userdata, flags, reason_code, properties):
2     client.subscribe(util.TOPIC_REQ_CMD, qos=util.QOS)
3
4 def on_message(client: mqtt.Client, userdata, msg: mqtt.MQTTMessage):
5     payload_size = int(msg.payload.decode("ascii"))
6     payload = "DATA"*(payload_size//4)
7     client.publish(topic=util.TOPIC_RES_CMD, qos=util.QOS, payload=payload)
```

*Αρχείο reqres_mqtt_responder.py*

Για τον MQTT responder τα πράγματα είναι πιο απλά. Επίσης μόλις συνδεθεί κάνει subscribe αλλά στο “request_topic” (γρ. 1-2). Κάθε φορά που λαμβάνει μήνυμα από τον requester (γρ. 4) διαβάζει το ζητούμενο μέγεθος payload, το δημιουργεί, και το στέλνει πίσω δημοσιεύοντας το στο “response_topic” (γρ. 5-7). Ακολουθεί ο σχετικός κώδικας για το CoAP.

```
 1 async def main():
 2     context = await aio.Context.create_client_context()
 3
 4     for payload_size in util.PAYLOAD_SIZES:
 5         avg_duration = 0
 6         for i in range(util.NUM_MESSAGES):
 7             start_time = time.time()
 8             # tell responder (server) the payload size to use for that response
 9             payload = struct.pack(">i", payload_size)
10             request_msg = aio.Message(code=Code.GET, payload=payload, \
11                 uri=f"coap://{util.COAP_HOST}:{util.COAP_PORT}/simple_resource")
12             response = await context.request(request_msg).response
13             duration = (time.time() - start_time)*1000
14             avg_duration += duration
15             await asyncio.sleep(delay=0.050)
16
17         avg_duration /= util.NUM_MESSAGES
18         util.save_result_to_file(util.OUTPUT_FILE_REQRES, util.NUM_MESSAGES,\
19             "COAP", payload_size, avg_duration)
```

*Αρχείο reqres_coap_requester.py*

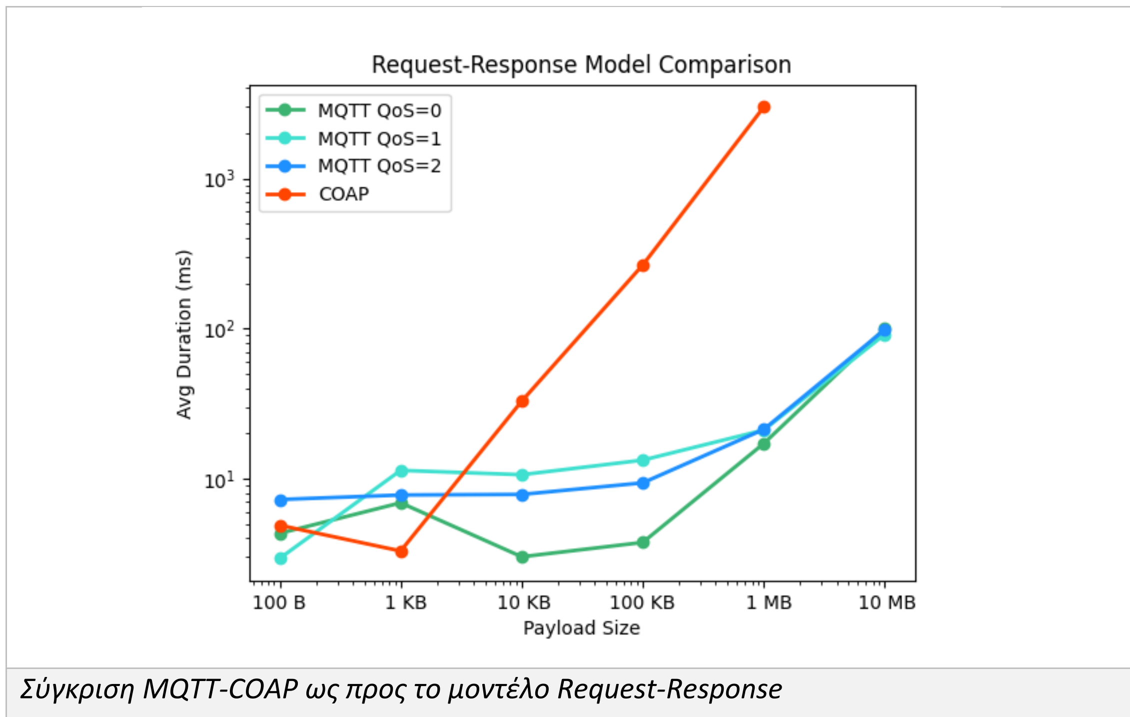
Ο CoAP requester σε κάθε GET request για το “simple_resource” στέλνει το μέγεθος payload για την απάντηση (γρ. 9-12). Στην εντολή *struct.pack()* η παράμετρος “>i” σημαίνει big-endian κωδικοποίηση ακεραίου 4 bytes. Μόλις λάβει την απάντηση υπολογίζει και αποθηκεύει τη μέση διάρκεια του GET request μαζί με το response (γρ. 12-19). Η εντολή *asyncio.sleep()* στη γραμμή 15, όπως και προηγουμένως προσφέρει σιγουριά ότι θα έχει τελειώσει το τωρινό GET πριν το επόμενο.

```
 1 # .....
 2 class SimpleResource(resource.Resource):
 3     async def render_get(self, request: aiocoap.Message):
 4         [response_size] = struct.unpack(">i", request.payload)
 5         payload = b'DATA'*(response_size//4)
 6         return aiocoap.Message(payload=payload, code=Code.CONTENT)
 7
 8 async def main():
 9     # .....
10     root = resource.Site()
11     root.add_resource(['simple_resource'], SimpleResource())
12     # .....
```

*Αρχείο coap_server.py (προσθήκες)*

Παραπάνω φαίνονται οι προσθήκες για τον CoAP server (βλ. 4.2.2 για το αρχικό). Στη γραμμή 11 προστίθεται το URI για το “simple_resource” που αντιστοιχεί στη κλάση *SimpleResource.* Όταν ληφθεί GET request ο server (responder) στις γραμμές 3-6 διαβάζει το ζητούμενο μέγεθος payload, το δημιουργεί και απαντάει με αυτό με κωδικό απάντησης Content.

### 4.3.3 Αποτελέσματα




*Σύγκριση MQTT-COAP ως προς το μοντέλο Request-Response*

| Payload | 100 B | 1 KB | 10 KB | 100 KB | 1 MB | 10 MB |
|---|---|---|---|---|---|---|
| **MQTT QoS=0** | 4.31 | 6.90 | 3.01 | 3.76 | 17.15 | 100.08 |
| **MQTT QoS=1** | 2.93 | 11.35 | 10.62 | 13.29 | 21.08 | 91.57 |
| **MQTT QoS=2** | 7.25 | 7.78 | 7.85 | 9.37 | 21.30 | 98.59 |
| **COAP** | 4.89 | 3.29 | 32.83 | 265.24 | 2994.07 | - |

*Τα παραπάνω δεδομένα σε μορφή πίνακα. Οι χρόνοι είναι σε milliseconds.*

Για payload ≤ 1 KB βλέπουμε ότι το CoAP έχει καλύτερους χρόνους και αυτό οφείλεται στο γεγονός ότι για μικρά μηνύματα η απάντηση βρίσκεται στο ίδιο το ACK πακέτο (δηλ. piggybacked response). Για τα υπόλοιπα αυξάνεται γρηγορότερα από τι το MQTT για τους ίδιους λόγους και με το άλλο μοντέλο. Ο χρόνος του CoAP για 10 MB payload λείπει γιατί ήταν υπερβολικά μεγάλος πάνω από 1 λεπτό.

Για μεταξύ των QoS το μόνο που μπορούμε να διακρίνουμε είναι ότι υπάρχει μια σχεδόν σταθερή διαφορά μεταξύ QoS=0 με QoS=1,2 που μάλλον οφείλεται στον ίδιο λόγο με το άλλο μοντέλο δηλ. στη στιγμή της χειραψίας που προωθεί το μήνυμα ο broker.

# 5 Συμπεράσματα

Σε αυτή τη πτυχιακή ασχολήθηκε με τρία από τα πιο διαδεδομένα πρωτόκολλα επικοινωνίας τα HTTP, MQTT, και CoAP, με τα δύο τελευταία να είναι τα πιο διαδεδομένα στο IoT ενώ το πρώτο λειτούργησε ως βάση σύγκρισης. Αναλύθηκαν οι μηχανισμοί και οι ιδιαιτερότητες του κάθε πρωτοκόλλου από μεριά σχεδιασμού αντί υλοποίησης, και με περιληπτικό αλλά περιεκτικό τρόπο. Έπειτα έγινε αναφορά βιβλιογραφίας για τη σύγκριση και των τριών ως προς διάφορες πτυχές τους. Η σύγκριση αυτή επεκτάθηκε για τα MQTT-CoAP υλοποιώντας πρόγραμμα προσομοίωσης για μέτρηση χρήσης πόρων επεξεργαστή, ως προς τα δύο πιο συχνά μοντέλα επικοινωνίας σε εφαρμογή IoT δηλαδή το Publish-Subscribe και το Request-Response.

Τα αποτελέσματα αυτού του πειράματος έδειξαν ότι και τα δύο πρωτόκολλα μπορούν να χρησιμοποιηθούν και για τα δύο μοντέλα επικοινωνίας. Με το CoAP να αποδίδει καλύτερα από το MQTT μόνο στο Request-Response και για σχετικά μικρού μεγέθους μηνύματος κάτω από 1 MB. Είναι σημαντικό να σημειωθεί ότι τα αποτελέσματα αυτά σχετίζονται μόνο με τις δύο πιο δημοφιλής υλοποιήσεις των πρωτοκόλλων σε Python (τις pahomqtt και aiocoap).

Η μελέτη αυτή θα μπορούσε να επεκταθεί εξετάζοντας περισσότερες υλοποιήσεις των πρωτοκόλλων σε αυτά τα δύο μοντέλα επικοινωνίας. Επιπλέον θα μπορούσε να εξεταστεί με παρόμοιο τρόπο η χρήση πόρων για περισσότερα λιγότερο δημοφιλής πρωτόκολλα IoT.

# 6 Βιβλιογραφικές Πηγές

1. **OSI model.** Wikipedia contributors. (2023, December 6). In *Wikipedia, The Free Encyclopedia*. Retrieved 11:56, December 7, 2023, from https://en.wikipedia.org/w/index.php?title=OSI_model&oldid=1188619108
2. **What is the internet of things?** https://www.ibm.com/topics/internet-of-things
3. Bassi, A., Bauer, M., Fiedler, M., Kramp, T., Van Kranenburg, R., Lange, S., & Meissner, S. (2013). ***Enabling Things to Talk: Designing IoT solutions with theIoT Architectural Reference Model*** (p. 379). Springer Nature.
4. **internet of things (IoT)** https://www.techtarget.com/iotagenda/definition/Internet-of-Things-IoT
5. **MDN Web Docs.** *HTTP* (2023, December 13) https://developer.mozilla.org/en-US/docs/Web/HTTP
6. **HTTP.** Wikipedia contributors. (2023, December 11). In *Wikipedia, The Free Encyclopedia*. Retrieved 14:55, December 13, 2023, from https://en.wikipedia.org/w/index.php?title=HTTP&oldid=1189333921
7. **Uniform Resource Identifier.** Wikipedia contributors. (2023, December 14). In *Wikipedia, The Free Encyclopedia*. Retrieved 11:05, December 16, 2023, from https://en.wikipedia.org/w/index.php?title=Uniform_Resource_Identifier&oldid=1189824644
8. **MDN Web Docs.** *HTTP request methods* (2023, December 16) https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods
9. **IETF HTTP Working Group.** *HTTP Semantics.* (2023, December 17) https://httpwg.org/specs/rfc9110.html
10. **MDN Web Docs.** *HTTP headers* (2023, December 17) https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers
11. **MDN Web Docs.** *Using HTTP cookies* (2023, December 17) https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies
12. **HTTPS.** Wikipedia contributors. (2023, December 14). In *Wikipedia, The Free Encyclopedia*. Retrieved 11:52, December 18, 2023, from https://en.wikipedia.org/w/index.php?title=HTTPS&oldid=1189857133
13. **A Brief Overview on HTTP.** Wendroth, J., & Jaeger, B. (2022). *Network*, *59*.
14. **Introduction to HTTP/2.** Ilya Grigorik, Surma. (2016, September 29) https://web.dev/articles/performance-http2
15. **HTTP/3 From A To Z: Core Concepts.** Robin Marx. (2021, August 9) *Smashing Magazine.* https://www.smashingmagazine.com/2021/08/http3-core-concepts-part1/

16. **MQTT.** Wikipedia contributors. (2023, December 22). In *Wikipedia, The Free Encyclopedia*. Retrieved 15:07, January 3, 2024, from https://en.wikipedia.org/w/index.php?title=MQTT&oldid=1191236512
17. **Understanding the MQTT Protocol Packet Structure.** Stephen Cope. (2023, April 11) *Steve's Internet Guide.* http://www.steves-internet-guide.com/mqtt-protocol-messages-overview/
18. **MQTT Retained Messages Explained.** Stephen Cope. (2023, February 11) *Steve's Internet Guide.* http://www.steves-internet-guide.com/mqtt-retained-messages-example/
19. **MQTT Version 5.0**. Andrew Banks, Ed Briggs, Ken Borgendale, Rahul Gupta. (2019, March 7) *OASIS Standard.* https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html
20. **What is MQTT Last Will and Testament (LWT)?** HiveMQ Team. (2023, June 27) *MQTT Essentials*. https://www.hivemq.com/blog/mqtt-essentials-part-9-last-will-and-testament/
21. **What is MQTT Quality of Service (QoS) 0,1, & 2?** HiveMQ Team. (2023, June 20) *MQTT Essentials*. https://www.hivemq.com/blog/mqtt-essentials-part-6-mqtt-quality-of-service-levels/
22. **MQTT Topics, Wildcards, & Best Practices** HiveMQ Team. (2023, June 20) *MQTT Essentials*. https://www.hivemq.com/blog/mqtt-essentials-part-5-mqtt-topics-best-practices/
23. **MQTT Security Fundamentals – Multipart Series.** HiveMQ Team. (2015, June 8) *MQTT Security Fundamentals*. https://www.hivemq.com/mqtt/mqtt-security-fundamentals/
24. **Introduction to MQTT 5 Protocol - MQTT 5 Essentials Part 1 & 2.** Mary Brickenstein-Hofschen. (2019, October 1) *MQTT 5 Essentials*. https://www.hivemq.com/blog/mqtt5-essentials-part1-introduction-to-mqtt-5/
25. **6 facts why it's worth upgrading to the brand new MQTT 3.1.1 version.** HiveMQ Team. (2014, October 30) *HiveMQ Articles*. https://www.hivemq.com/article/6-facts-why-its-worth-upgrading-to-mqtt-3-1-1/
26. **Constrained Application Protocol.** Wikipedia contributors. (2023, October 18). In *Wikipedia, The Free Encyclopedia*. Retrieved 15:07, January 3, 2024, from https://en.wikipedia.org/w/index.php?title=Constrained_Application_Protocol&oldid=1180725414
27. **The constrained application protocol (CoAP).** Shelby Zach, Klaus Hartke, and Carsten Bormann. (2014, June) *IETF RFC 7252*. https://datatracker.ietf.org/doc/rfc7252/
28. **CoAP: An Application Protocol for Billions of Tiny Internet Nodes.** Bormann Carsten, Angelo P. Castellani, and Zach Shelby. *IEEE Internet Computing* 16.2 (2012): 62-67.

29. **Block-Wise Transfers in the Constrained Application Protocol (CoAP).** Z. Shelby, and C. Bormann. (2016, August) *IETF RFC 7252*. https://datatracker.ietf.org/doc/html/rfc7959
30. **Analysis and evaluation of communication protocols for IoT applications.** Sidna, Jeddou, et al. (2020, September). *Proceedings of the 13th international conference on intelligent systems: theories and applications*.
31. **Comparative analysis of IoT communication protocols.** Çorak, Burak H., et al. (2018, June). *IEEE International symposium on networks, computers and communications*.
32. **Messaging protocols for IoT systems—A pragmatic comparison.** Wytrębowicz, Jacek, Krzysztof Cabaj, and Jerzy Krawiec. *Sensors* 21.20 (2021): 6904.