**ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ**

**Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών**

Τομέας Συστημάτων Επικοινωνιών, Ηλεκτρονικής
& Συστημάτων Πληροφορικής

# Ανάπτυξη Υποδομής MQTT για IoT Εφαρμογές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

**ΑΘΗΝΑΣ ΠΛΑΣΚΑΣΟΒΙΤΗ**

**Επιβλέπων**: Συκάς Ευστάθιος
Καθηγητής Ε.Μ.Π.

ΕΡΓΑΣΤΗΡΙΟ ΔΙΚΤΥΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

Αθήνα, Ιούνιος 2020

**ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ**

**Σχολή Ηλεκτρολόγων Μηχανικών και Μηχανικών Υπολογιστών**

Τομέας Συστημάτων Επικοινωνιών, Ηλεκτρονικής
& Συστημάτων Πληροφορικής

# Ανάπτυξη Υποδομής MQTT για IoT Εφαρμογές

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

**ΑΘΗΝΑΣ ΠΛΑΣΚΑΣΟΒΙΤΗ**

**Επιβλέπων**: Συκάς Ευστάθιος
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 22η Ιουνίου 2020:

| ..... | ..... | ..... |
|---|---|---|
| Ευστάθιος Συκάς | Συμεών Παπαβασιλείου | Ιωάννα Ρουσσάκη |
| Καθηγητής Ε.Μ.Π. | Καθηγητής Ε.Μ.Π. | Επίκουρη Καθηγήτρια |

ΕΡΓΑΣΤΗΡΙΟ ΔΙΚΤΥΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
Αθήνα, Ιούνιος 2020

*(Υπογραφή)*

.....

**Αθηνά Α. Πλασκασοβίτη**

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών

# Περίληψη

Τα τελευταία χρόνια, παράλληλα με τις αλματώδεις προόδους σε όλους τους τομείς της επιστήμης και της τεχνολογίας, αναπτύσσεται ραγδαία και ο τομέας του Διαδικτύου των Πραγμάτων, τόσο σε πειραματικό/εργαστηριακό περιβάλλον όσο και σε περιβάλλον παραγωγης. Ο όρος Διαδίκτυο των Πραγμάτων (Internet of Things – IoT) αναφέρεται σε ένα ευρύ δίκτυο συσκευών και ανθρώπων συνδεδεμένων στο ίντερνετ με τέτοιο τρόπο ώστε να μπορούν να ανταλλάσουν μεταξύ τους άμεσα πλήθος πληροφοριών που αφορούν τη λειτουργία και το περιβάλλον τους.

Πιο συγκεκριμένα, όσον αφορά συστήματα με μεγάλο πλήθος συνδεδεμένων συσκευών που στέλνουν αυτοματοποιημένα μηνύματα για το συντονισμό της λειτουργίας του δικτύου που εξυπηρετούν, τα δεδομένα που ανταλλάσονται κάθε στιγμή είναι πάρα πολλά. Σε μια τέτοια περίπτωση, είναι μεγάλη η ανάγκη δόμησης ενός συστήματος που θα είναι αξιόπιστο και θα ανταποκρίνεται ακόμα και σε σενάρια εξαιρετικά μεγάλου φόρτου πληροφορίας – που θα μπορεί να διαχειριστεί μεγάλο πλήθος ταυτόχρονων συνδέσεων, ανταλλαγή πολλών χιλιάδων μηνυμάτων, με όσο το δυνατόν λιγότερες επιπτώσεις στη λειτουργία του δικτύου.

Στόχος αυτής της διπλωματικής εργασίας είναι η διερεύνηση των ανοιχτού λογισμικού υλοποιήσεων server ως προς τον τρόπο λειτουργίας τους και της εκάστοτε διαχείρησης δικτυακής κίνησης σε πειραματικό περιβάλλον. Η μελέτη αυτή θα γίνει πραγματοποιώντας τα υποθετικά σενάρια που καταναλώνουν περισσότερα resources, προκειμένου να διαπιστωθεί κατα πόσο η επιλεγμένη κάθε φορά υλοποίηση (standalone server ή clustered) μπορεί να ανταποκριθεί βέλτιστα και αξιόπιστα ακόμα και σε περιβάλλον παραγωγής.

Η διπλωματική αυτή εργασία πραγματοποιήθηκε σε συνεργασία με την COSMOTE.

**Λέξεις κλειδιά:**

Αρχιτεκτοντική συστήματος IoT, πρωτόκολλο MQTT, Mosquitto server, EMQX server, clustering, HAPROXY load balancer, Python, C, BASH, JSON

# Abstract

In the recent years, along with the leaping progress in all sectors of science and technology, there has been a great development regarding the concept of the Internet of Things, both in a theoretic-experimental level as we as in production. The term "Internet of Things – IoT" is referring to a vast network of devices and people, all connected to the internet, exchanging constantly a large number of information related to their function and their surroundings.

Specifically, when the systems in inquiry have a great number of connected devices, that send automated messages and signals to coordinate the network at service, the load of data being processed every moment is extremely big. In a case like this, there is great need to implement a system both trustworthy and responsive even in the worst case scenarios – a system capable to manage a large number of concurrent connections and the exchange of millions of messages with the minimum loss of information.

The aim of this diploma thesis is to experiment with open source server software, understanding the different ways to manage network traffic in a testing environment. This study will make use of the traffic scenarios that make heavy use of the available resources, in order to realize how the selected implementation (standalone server or clustering) is able to respond optimally, even in a production environment.

This diploma thesis is conducted in cooperation with COSMOTE.

Keywords:

IoT system architecture, MQTT protocol, Mosquitto server, EMQX server, clustering, HAPROXY load balancer, Python, C, BASH, JSON

# Περιεχόμενα

# Διαγράμματα - Πίνακες

# Ευχαριστίες

Αρχικά θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή κ. Ευστάθιο Συκά, ο οποίος δέχτηκε να συνεργαστεί μαζί μου και μου έδωσε την ευκαιρία να ασχοληθώ με ένα θέμα που συνδύασε εξαιρετικά πολλές διαφορετικές γνώσεις που ήθελα να εξασκήσω.

Θα ήθελα επίσης να ευχαριστήσω τον κ. Γιώργο Λυμπερόπουλο, επικεφαλή του τμήματος έρευνας και ανάπτυξης της Cosmote, για την εμπιστοσύνη που μου έδειξε αναθέτοντάς μου αυτό το θέμα, καθώς και για τον χώρο και χρόνο που μου διέθεσε για να εργαστώ στο Leonardo Lab στο OTE Academy.

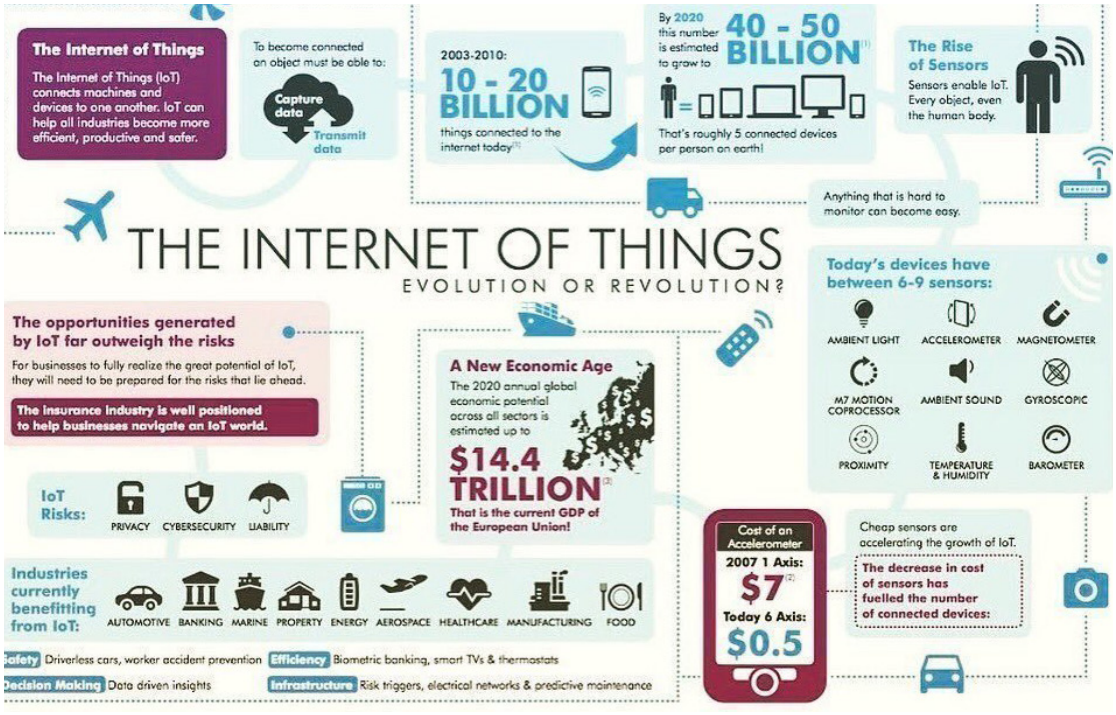
Σε συνέχεια αυτού, ευχαριστώ θερμά τον Θωμά Πάζιο για την καθοδήγηση και υποστήριξη που μου πρόσφερε κατά τη διεκπόνηση της εργασίας.

Τέλος, θέλω να ευχαριστήσω τους γονείς μου, καθώς και την υπόλοιπη οικογένειά μου, φιλική και συντροφική, οι οποίοι ήταν δίπλα μου σε όλα τα χρόνια της ζωής μου και χάριν στους οποίους βρίσκομαι εδώ σήμερα.

# Κεφάλαιο 1
# Εισαγωγή

## *1.1 Περιγραφή προβλήματος*

Καθώς το Διαδίκτυο των Πραγμάτων έχει αρχίσει να βρίσκει εφαρμογή σε μεγάλο εύρος συνθηκών τα τελευταία χρόνια, ανταπτύσσονται και εξετάζονται συνεχώς νέες δομές οργάνωσης εξυπηρετητών (servers) που αναλαμβάνουν τον συντονισμό δικτύων έξυπνων συσκευών, αισθητήρων, καμερών, ακόμα και ανθρώπων. Τα δίκτυα αυτά συναντώνται τόσο σε οικειακό περιβάλλον, όσο και σε εργαστηριακό και περιβάλλον παραγωγής, επομένως το πλήθος συνδεδεμένων συσκευών ποικίλει. Οι συσκευές ανταλλάσουν μηνύματα μεταξύ τους μέσω του εξυπηρετητή, αλλά και με τον ίδιο, για την εύρυθμη λειτουργία του δικτύου.




*Εικονα 1.1: Εξέλιξη της τεχνολογίας του Διαδικτύου των Πραγμάτων*

Προφανώς ανάλογα με τη συνθήκη και το περιβάλλον χρήσης του δικτύου αλλάζουν και οι απαιτήσεις του σε ταχύτητα, ανταποκρισιμότητα, αξιοπιστία. Σε ένα οικειακό περιβάλλον για παράδειγμα, οι ανάγκες ενός δικτύου είναι αρκετά μικρές, αφού το πλήθος των IoT συσκευών θα είναι σχετικά μικρό και η συχνότητα επικοινωνίας μεταξύ τους περιορισμένη. Αντίστοιχα σε ένα εργαστηριακό περιβάλλον, το δίκτυο θα πρέπει να είναι ικανό να χειριστεί πολύ μεγαλύτερο πλήθος συσκευών, και θα πρέπει ανταποκρίνεται κάθε στιγμή άμεσα και να διακινεί συνεχώς μεγάλο αριθμό μηνυμάτων, χωρίς απώλειες, ή αν υπάρχουν, να ακολουθείται πολιτική διαχείρισης μεγάλης δικτυακής κίνησης για να διατηρηθεί η αξιοπιστία του. Οι ανάγκες ενός δικτύου σε περιβάλλον παραγωγής είναι προφανώς ακόμα πιο απαιτητικές.

Η ανταλλαγή μηνυμάτων συσκευών IoT γίνεται με χρήση του πρωτοκόλλου MQTT, το οποίο είναι ιδανικό για μεταφορά μηνυμάτων μικρού μεγέθους μεταξύ απομακρυσμένων συσκευών. Οι εξυπηρετητές τέτοιων δικτύων έχουν λοιπόν να διαχειριστούν μηνύματα από πολλές συσκευές, συγκεκριμένης μορφής, και να ανταποκρίνονται σε μεγάλο πληροφοριακό φόρτο και υψηλή συχνότητα, εάν αυτό είναι απαραίτητο.

Οι επιλογές ανοιχτού λογισμικού για την υλοποίηση εξυπηρετητών σε δίκτυα IoT συσκευών είναι πολλές, και καθεμιά ανταποκρίνεται σε διαφορετικές ανάγκες και συνθήκες λειτουργίας του υποψήφιου δικτύου.

## *1.2 Σκοπός διπλωματικής εργασίας*

Η παρούσα διπλωματική εργασία επικεντρώνεται γύρω από τους παρακάτω βασικούς άξονες:

- Την διερεύνηση των πιθανών σεναρίων δικτυακού φόρτου σε ένα εργαστηριακό περιβάλλον, με σκοπό την εύρεση του δυσχερέστερου και την περαιτέρω εφαρμογή του για παρατήρηση της συμπεριφοράς του.
- Την ανάπτυξη δυνατότητας εφαρμογής των σεναρίων αυτών σε εικονικό περιβάλλον, δηλαδή την χρήση εικονικών μηχανημάτων και τον σχεδιασμό των προγραμμάτων που θα προσομοιώσουν επιτυχώς και με μεγάλο βαθμό ακριβείας ένα δίκτυο IoT με μεγάλο πλήθος

συνδεδεμένων συσκευών.

- Τον σχεδιασμό και την υλοποίηση των μερών ενός δικτύου IoT, όπως αυτό θα περιγραφεί στη συνέχεια.
- Την δοκιμή των πιο απαιτητικών σεναρίων σε διαφορετικές δομές συστημάτων εξυπηρετητών ανοιχτού λογισμικου, δηλαδή σε αρχιτεκτονικές που περιλαμβάνουν μοναδικό εξυπηρετητή (standalone server or server) αλλά και σε αντίστοιχες με συστοιχία εξυπηρετητών (clustered server), και την καταγραφή μετρήσεων απόδοσης εξυπηρετητή και εύρυθμης λειτουργίας δικτύου.
- Την σύγκριση των αποτελεσμάτων μεταξύ των διαφορετικών συστημάτων για την εύρεση της πιο αξιόπιστης και αποτελεσματικής υλοποίησης.

## *1.3 Περιγραφή λύσης*

Για να βρεθεί η αρχιτεκτονική που εξυπηρετεί βέλτιστα ένα σενάριο μεγάλου φόρτου σε δίκτυο συσκευών IoT, πρέπει να γίνει μελέτη τόσο του πρωτοκόλλου επικοινωνίας και του τρόπου επικοινωνίας μεταξύ συσκευών και εξυπηρετητή, όσο και της κατανάλωσης πόρων από το σύστημα του εξυπηρετητή σε αντιστοιχία με την αύξηση των διακινούμενων δεδομένων. Επομένως θα ακολουθηθεί η εξής διερευνητική πορεία:

***Πρωτόκολλο επικοινωνίας των συσκευών IoT***: Το πιο διαδεδομένο πρωτόκολλο επικοινωνίας που χρησιμοπουείται για την επικοινωνία σε δίκτυα IoT είναι το MQTT (MQ Telemetry Transport). Θα γίνει, επομένως, μελέτη του πρωτοκόλλου MQTT, του δυνατού μεγέθους των μηνυμάτων και των δυνατοτήτων μετάδοσης πληροφορίας μέσω αυτού.

***Λογισμικό εξυπηρετητή***: Υπάρχουν πολλά λογισμικά για την υλοποίηση του συστήματος του εξυπηρετητή σε IoT δίκτυα. Θα γίνει διερεύνηση του λογισμικού του Mosquitto Server, καθώς και του EMQX server αντίστοιχα, τα οποία αποτελούν δύο από τις πιο διαδεδομένες επιλογές ανοιχτού λογισμικού για εξυπηρετητές σε δίκτυα IoT.

***Πιθανά σενάρια μεγάλης δικτυακής κίνησης:*** Υπάρχουν αρκετά σενάρια σε ένα δίκτυο τα οποία μπορούν να δημιουργήσουν προβλήματα στην επικοινωνία μεταξύ των μελών του, απώλεια μηνυμάτων, καθυστέρηση στην ανταπόκριση του εξυπηρετητή ή σε κάποιες περιπτώσεις ακόμα και

αδυναμία ανταπόκρισής του. Προφανώς σημαντικό ρόλο σε αυτά, πέρα από τις δικτυακές παραμέτρους, παίζουν και οι διαθέσιμοι πόροι του εξυπηρετητή. Η μελέτη των σεναρίων αυτών στόχο έχει να βρεθούν τα πιο απαιτητικά από αυτά έτσι ώστε η δοκιμή σε εργαστηριακό περιβάλλον και τα αποτελέσματά της να είναι όσο το δυνατόν πιο κοντά σε ρεαλιστικές συνθήκες.

***Εύρεση της αποδοτικότερης αρχιτεκτονικής του συστήματος εξυπηρετητή:*** Ανάλογα με τις συνθήκες λειτουργίας και το πλήθος των επιθυμητών συνδέσεων κάθε δικτύου, επιλέγεται συγκεκριμένη αρχιτεκτονική, με μοναδικό εξυπηρετητή ή σύμπλεγμα εξυπηρετητών (clustering). Θα διερευνηθούν και τα δύο αυτά μοντέλα, με σκοπό τη σύγκριση των δυνατοτήτων τους σε δυσχερείς δικτυακές συνθήκες.

## *1.4 Περιγραφή απαιτήσεων*

Για την αναπαραγωγή της περιγραφείσας έρευνας και την εκτέλεση πειραμάτων για την αναγνώριση των δυνατοτήτων των συστημάτων εξυπηρετητών σε δίκτυα IoT, είναι απαραίτητη η δημιουργία ενός εικονικού περιβάλλοντος που θα προσομοιώνει ρεαλιστικά ένα δίκτυο κάποιων χιλιάδων χρηστών που θα επικοινωνούν μεταξύ τους με μεταβλητή συχνότητα μέσω ενός εξυπηρετητή. Επομένως είναι αναγκαίο να δημιουργηθούν οι παρακάτω δομές:

- εξυπηρετητής, ή/και σύμπλεγμα εξυπηρετητών, δηλαδή η φυσική δομή και τα λογισμικά που αναφέρθηκαν παραπάνω, τα οποία θα πρέπει να ρυθμιστούν με τρόπο τέτοιο ώστε να εξυπηρετούν τις ανάγκες του δικτύου.
- εικονικές συσκευές, τόσο για τον εξυπηρετητή όσο και για την προσομοίωση των συσκευών IoT του δικτύου, καθότι είναι αρκετά δύσκολο να δημιουργηθεί δίκτυο IoT αρκετών χιλιάδων συσκευών για πειραματικούς σκοπούς, αυτό επιτυγχάνεται μέσω εικονικών μηχανημάτων στα πλαίσια προσομοίωσης.
- ανάπτυξη προγραμμάτων παρακολούθησης των επιδόσεων του εξυπηρετητή και του δικτύου (logging), για να γίνει η συλλογή των απαραίτητων στοιχείων και μετέπειτα η ανάλυση τους για διεξαγωγή

συμπερασμάτων σε σχέση με τις επιδόσεις του δικτύου.

- επιλογή της βάσης δεδομένων που θα αποστέλλονται οι μετρήσεις από τον εξυπηρετητή για αποθήκευση, αλλά και του λογισμικού που θα κάνει την γραφική απεικόνιση της βάσης δεδομένων για διευκόλυνση της αναλυτικής διαδικασίας.

Αυτοί είναι και οι βασικοί άξονες έρευνας για αυτή τη διπλωματική, που θα αναλυθούν στα επόμενα κεφάλαια, τόσο θεωρητικά, όσο και πρακτικά στην τοποθέτησή τους μέσα στο δίκτυο των πειραμάτων.

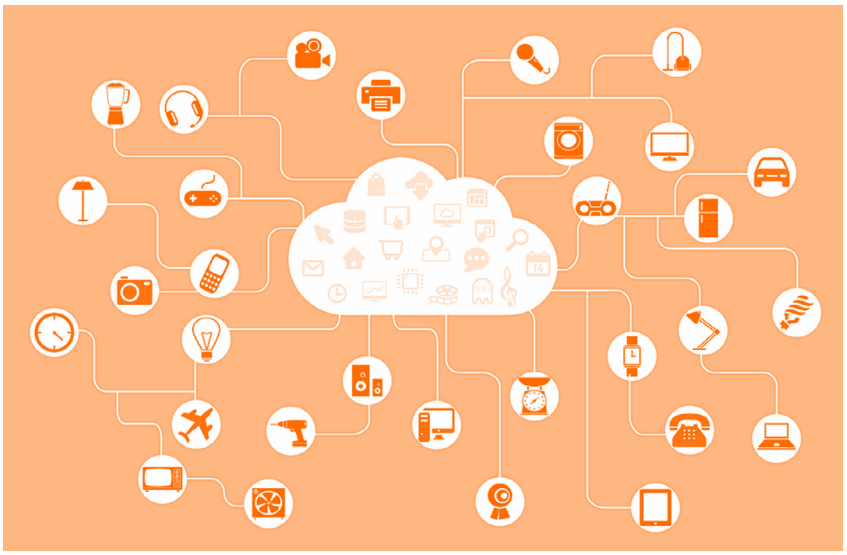
## *1.5 Οργάνωση Τόμου*

Η διπλωματική εργασία που παρουσιάζεται οργανώνεται σε 6 κεφάλαια. Το πρώτο κεφάλαιο που διανύουμε αποτελεί μια συνοπτική περιγραφή του θέματος, των ζητημάτων που προκύπτουν και του σχεδιασμού του προτεινόμενου πλάνου για την εύρεση της λύσης. Στο δεύτερο κεφάλαιο γίνεται αναφορά στην τεχνολογία του Διαδικτύου των Πραγμάτων, και σε βασικές αρχές λειτουργίας ενός δικτύου, στοιχεία που είναι απαραίτητα να ληφθούν υπόψιν για την διεξαγωγή των συμπερασμάτων της διπλωματικής εργασίας. Στο τρίτο κεφαλαιο θα γίνει λεπτομερής περιγραφή της αρχιτεκτονικής των συστημάτων εξυπηρετητών και του εικονικού περιβάλλοντος που χρησιμοποιήθηκε για την προσομοίωση του δικτύου IoT, όπως επίσης και σύντομη επεξήγηση των προγραμμάτων που χρησιμοποιήθηκαν και αναφορά στην παραμετροποίηση των μερών του δικτύου, για να υπάρχει ακριβής εικόνα για τον τρόπο λειτουργίας του. Στο τέταρτο κεφάλαιο θα γίνει αναλυτική παρουσίαση των πειραμάτων και των μετρήσεων που λήφθηκαν. Στο πέμπτο θα αναλυθούν τα συμπεράσματα από την πειραματική διαδικασία. Στο έκτο κεφάλαιο θα γίνει αναφορά σε προβλήματα που προέκυψαν και πιθανές προεκτάσεις και προτάσεις πάνω σε αυτά. Στο έβδομο κεφάλαιο θα παρατεθούν οι κώδικες που αναπτύχθηκαν, όπως επίσης και τεχνικά στοιχεία της υλοποίησης. Τέλος παρατίθεται και βιβλιογραφία.

# Κεφάλαιο 2
# Διαδίκτυο των Πραγμάτων

## *2.1 Ορισμός*

Ο όρος Διαδίκτυο των Πραγμάτων (Internet of Things – IoT) αναφέρεται σε ένα ευρύ δίκτυο επικοινωνίας μεγάλου πλήθους συσκευών που ενσωματώνουν ηλεκτρονικά μέσα, λογισμικό, αισθητήρες και συνδεσιμότητα στο διαδίκτυο ώστε να επιτρέπεται η σύνδεση και η ανταλλαγή δεδομένων μεταξύ τους. Η ονομασία άρχισε να χρησιμοποιείται στα τέλη της δεκαετίας του 1990, όταν μία ομάδα έρευνας στο MIT ανακάλυψε τον τρόπο να συνδέονται αντικείμενα στο διαδίκτυο μέσω μίας πλακέτας RFID.



*Εικόνα 2.1: Σχηματική απεικόνιση του Διαδικτύου των Πραγμάτων*

Χρησιμοποιείται η λέξη “πράγματα” μιας και όντως οι συσκευές που αναφέρονται στον ορισμό δεν περιορίζονται σε ηλεκτρονικές: μπορούν να είναι υπολογιστές, μηχανήματα, ψηφιακές συσκευές, ζώα και άνθρωποι.

Το καθένα φέρει ένα μοναδικό αναγνωριστικό και τη δυνατότητα να ανταλλάσσει δεδομένα στο διαδίκτυο, με τον εξυπηρετητή του δικτύου ή και με τις υπόλοιπες συσκευές, εάν αυτό είναι απαραίτητο, χωρίς να απαιτείται διαμεσολάβηση ανθρώπινου παράγοντα (machine-to-machine ή M2M διασύνδεση).

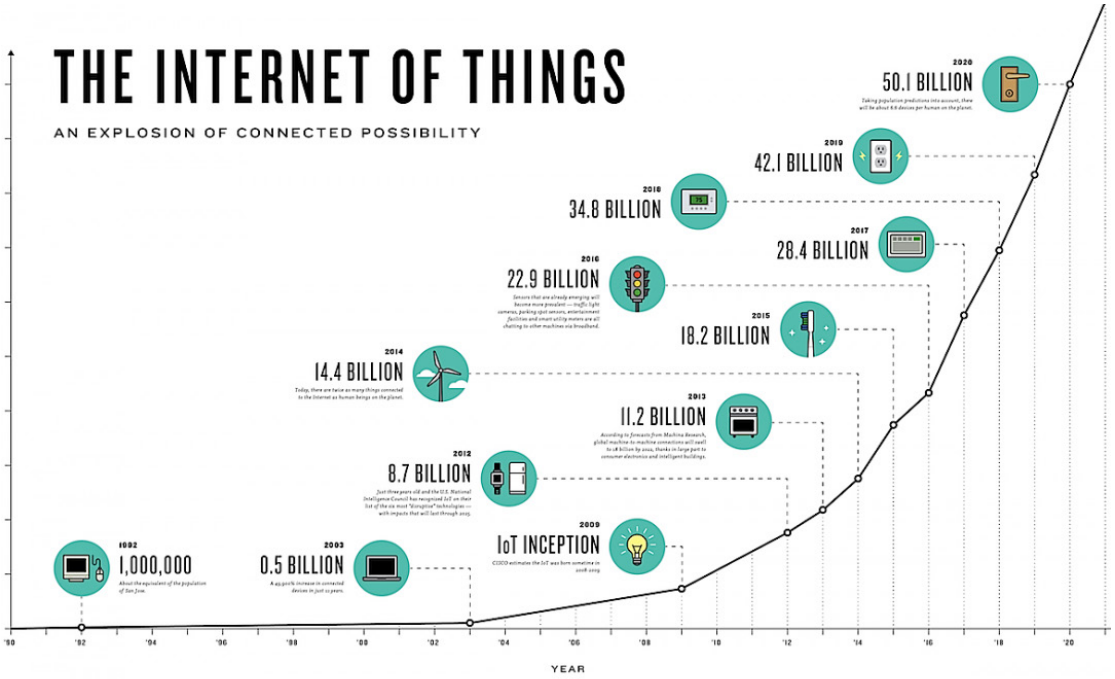
Στο διαδίκτυο συναντώνται διάφοροι ορισμοί του Διαδικτύου των πραγμάτων, όλοι όμως έχουν ως κοινό παρονομαστή το χαρακτηριστικό της διασύνδεσης καθημερινών συσκευών, με την προσθήκη υλικού κατάλληλα διαμορφωμένου, και την ανταλλαγή πληροφοριών μεταξύ τους. Η τεχνολογία αυτή, από τη στιγμή της ανακάλυψής της έχει συναντήσει ραγδαία ανάπτυξη και εξαπλώνεται σε διάφορους τομείς της καθημερινότητας και της επιστήμης.

## *2.2 Εξέλιξη του Διαδικτύου των Πραγμάτων*

Μπορεί η ορολογία διαδίκτυο των πραγμάτων να ξεκίνησε να χρησιμοποιείται στα τέλη του 1990, όμως η έννοια των συνδεδεμένων συσκευών υπήρχε αρκετό καιρό πριν.

Η ιδέα για τη δημιουργία ενός δικτύου έξυπνων συσκευών άρχισε να συζητείται από το 1982 στο Carnegie Mellon University, όπου δημιουργήθηκε η πρώτη συνδεδεμένη με το διαδίκτυο συσκευή, ένας αυτόματος πωλητής αναψυκτικών, που ανέφερε αν τα αναψυκτικά που είχαν προστεθεί ήταν κρύα. Γύρω στο 1991 (MarkWeiser) έγινε μια σειρά από δημοσιεύσεις σχετικά με την εξέλιξη των υπολογιστών τον 21ο αιώνα, η οποία έθεσε τη βάση για την ανάπτυξη του IoT. Το 1994 (Reza Raji) έγινε η πρώτη περιγραφή της ιδέας του IoT η οποία ήταν η εξής: «η μεταφορά μικρών πακέτων δεδομένων σε ένα μεγάλο σύνολο κόμβων, προκειμένου να ενοποιηθούν και να αυτοματοποιηθούν τα πάντα, από μικρές οικιακές συσκευές μέχρι ολόκληρα εργοστάσια». Μεταξύ του 1993 και του 1996 αρκετές εταιρείες πρότειναν κάποιες λύσεις , όμως τελικά το 1999 άρχισε η πραγματική εξέλιξη του κλάδου, όταν προτάθηκε η επικοινωνία συσκευής με συσκευή στο Παγκόσμιο Οικονομικό Φόρουμ. Η ιδέα του Internet of Things έγινε πρώτη φορά γνωστή το 1999 μέσω του Auto-ID Center του Πανεπιστημίου της Μασαχουσέτης (MIT), μέσα από δημοσιεύσεις σχετικές με την ανάλυση της αγοράς. Η ταυτοποίηση μέσω ραδιοσυχνοτήτων (Radio Frequency Identification, RFID) θεωρήθηκε από τους ιδρυτές του Auto-ID

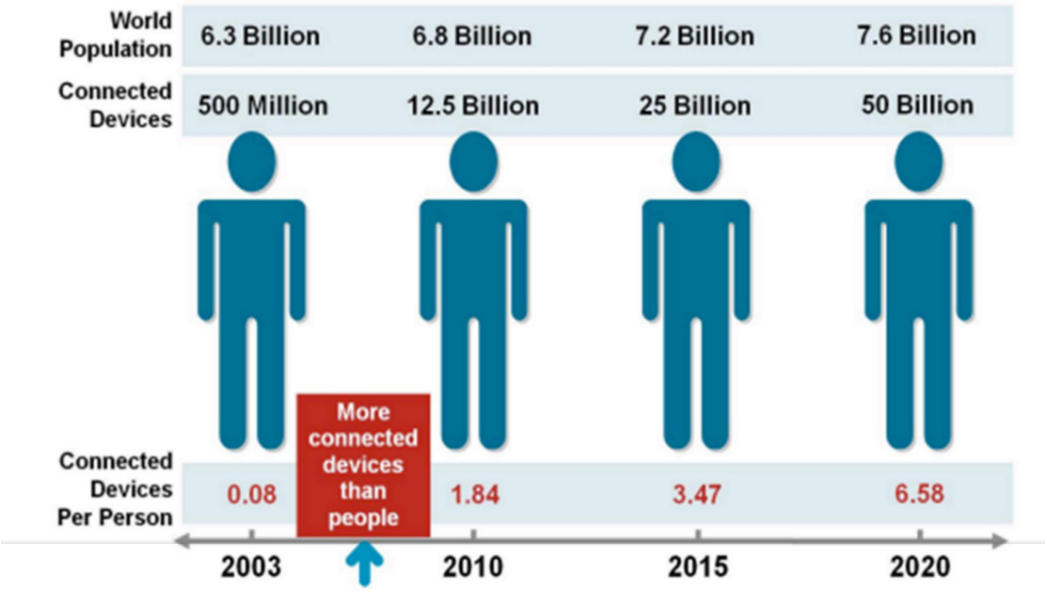
ως προαπαιτούμενο για την ανάπτυξη του IoT. Η λογική ήταν ότι εάν όλα τα αντικείμενα είχαν κάποια διακριτικά (identifiers), τότε οι υπολογιστές θα μπορούσαν να τα διαχειριστούν και να τα καταγράψουν ευκολότερα. Εκτός από τη χρήση του RFID για τον χαρακτηρισμό των αντικειμένων θα μπορούσαν να χρησιμοποιηθούν και άλλες τεχνολογίες, όπως η επικοινωνία




*Εικόνα 2.2: Ιραφική παράσταση του όλο και αυξανόμενου πλήθους IoT συσκευών στο χρόνο (1980-2020)*

κοντινού πεδίου (Near Field Communication, NFC), τα barcodes, τα QR codes καθώς και το ψηφιακό υδατογράφημα. Ως αρχική σκέψη, στόχος ήταν η υλοποίηση του IoT μέσω του εξοπλισμού όλων των αντικειμένων παγκοσμίως με μικρές συσκευές αναγνώρισης ή διακριτικά που μπορούν να αναγνωστούν από μηχανές, κάτι που θα είχε ως αποτέλεσμα μια πλήρη αλλαγή στην καθημερινότητα. Ο όρος Internet of Things εισήχθη από τον Kevin Ashton έναν εκ των ιδρυτών του Auto-ID Center στο οποίο έγινε αναφορά παραπάνω. Τον χρησιμοποιήσε για μία παρουσίαση το 1999 και έκτωτε καθιερώθηκε σαν όρος. Την τελευταία δεκαετία ο ορισμός αυτός κατέληξε να προσδιορίζει ένα ολοένα και μεγαλύτερο φάσμα εφαρμογών στους τομείς της υγείας, των δημοσίων υπηρεσιών, των μεταφορών κλπ. Παρόλο που η έννοια του όρου 'Thing' έχει μεταβληθεί με την εξέλιξη της τεχνολογίας, ο στόχος της χρήσης πληροφοριών από υπολογιστικά

συστήματα χωρίς την ανθρώπινη παρέμβαση παραμένει ο ίδιος. Σύμφωνα με την Cisco IBSG το Internet of Things «γεννήθηκε» πραγματικά τα έτη 2008-2009 όταν και έιχαμε εκρηκτική αύξηση στον αριθμό των συσκευών με δυνατότητα σύνδεσης στο διαδίκτυο. Πιο συγκεκριμένα, το έτος 2003 με πληθυσμό 6.3 δισεκατομμύρια είχαμε 500 εκατομμύρια συνδεδεμένες συσκευές. Το έτος 2010 ο πληθυσμός της γης αυξήθηκε σε 6.8 δισεκατομμύρια και οι συνδεδεμένες συσκευές σε 12.5 δισεκατομμύρια που σημαίνει ότι αναλογούσαν 1.84 συσκεύες στον κάθε άνθρωπο. Σύμφωνα με εκτιμήσεις της ίδιας εταιρείας το 2020 θα έχουμε 50 δισεκατομμύρια συνδεδεμένες στο διαδίκτυο συσκευές. Αυτή η εκτίμηση φαίνεται να είναι πολύ κοντά στα πραγματικά δεδομένα.




*Εικόνα 2.3: Απεικόνιση των IoT συσκευών σε συνάρτηση με τον πληθυσμό στην περίοδο 2000-2020*

Σημαντικό ρόλο στην ανάπτυξη του Διαδικτύου των Πραγμάτων έπαιξαν φυσικά οι ηλεκτρονικοί υπολογιστές – η εξέλιξή τους δηλαδή από τους κεντρικούς υπολογιστές στους ηλεκτρονικούς υπολογιστές (PC), από τα PC στα notebooks και από τα notebooks στις φορητές συσκευές (όπως τα tablets, smartphones και τα wearables). Κατά τη διάρκεια αυτής της εξέλιξης, οι συσκευές έγιναν μικρότερες – πια υπάρχουν ακόμα και συσκευές που φοριούνται - καθώς όλα τα εξαρτήματα περιορίστηκαν σε

μέγεθος, ενώ οι επιδόσεις τους βελτιώθηκαν - τόσο που αρκετές συσκευές σήμερα ξεπερνούν σε επιδόσεις τους ογκώδεις κεντρικούς υπολογιστές του παρελθόντος.

Παράλληλα και οι τιμές των προϊόντων και των εξαρτημάτων περιορίστηκαν και πλέον είναι οικονομικά προσιτά, κάτι που διευκολύνει την ευρεία υιοθέτηση τους από τους καταναλωτές. Σε τελική ανάλυση, όλες αυτές οι εξελίξεις είχαν ως αποτέλεσμα την κυκλοφορία προϊόντων που συνδέονται μεταξύ τους – συσκευών δηλαδή που αποτελούν το βασικό στοιχείο και την πόρτα που ανοίγει το δρόμο του IoT.

Εκτός από την εξέλιξη του υλικού (hardware) στους υπολογιστές, η πρόοδος σε θέματα συνδεσιμότητας είναι εξίσου σημαντική. Τεχνολογίες όπως η 3G και 4G LTE έχουν προσφέρει στους καταναλωτές τη δυνατότητα να είναι συνδεδεμένοι όπου και αν βρίσκονται. Παράλληλα, οι λειτουργίες σύνδεσης των συσκευών προσφέρουν τη δυνατότητα για άμεση σύνδεση δύο ή και περισσότερων συσκευών μεταξύ τους ή και έμμεση σύνδεση τους μέσω Wi-Fi ώστε οι συσκευές αυτές να επικοινωνούν μεταξύ τους.

## *2.3 Τομείς εφαρμογής*

Τα πεδία στα οποία έχει εισχωρήσει και βρίσκει εφαρμογές το Διαδίκτυο των Πραγμάτων είναι πλέον πολλά. Το πιο άμεσο στον καταναλωτή είναι οι εφαρμογές οικιακών αυτοματισμών, που μπορούν να ελέγχουν σε πρώτο επίπεδο τον φωτισμό, την θέρμανση/ψύξη, τα πολυμέσα και τα συστήματα ασφαλείας ενός σπιτιού, και πιο μακροπρόθεσμα την ενεργειακή απόδοση του σπιτιού προσαρμόζοντας όλα τα παραπάνω βάσει πληροφοριών που αφορούν στην καθημερινότητα των κατοίκων. Αντίστοιχα είναι τα συστήματα οικιακού αυτοματισμού που αποσκοπούν στην διευκόλυνση της ζωής των γηραιότερων.

Άλλη μια μείζονος σημασίας εφαρμογή του IoT είναι στον τομέα της υγείας. Εκεί γίνεται χρήση της διασύνδεσης πολλών συσκευών με στόχο την άμεση πρόσβαση σε πληροφορίες για τους ασθενείς, το ιστορικό τους αλλά και στοιχεία που λαμβάνονται κατά τη νοσηλεία τους. Σημαντική είναι η συμβολή των IoT συσκευών και σε περιπτώσεις ανάγκης απομακρυσμένου ελέγχου ιατρικών συσκευών (βηματοδότες, ειδικά μοσχεύματα κ. α.) και ενεργοποίηση συστημάτων ασφαλείας και

ειδοποιήσεων. Προσφέρουν εν ολίγοις στο ιατρικό προσωπικό μεγαλύτερη ευελιξία στις μεθόδους θεραπείας των ασθενών αλλά και άμεση πρόσβαση σε ζωτικές πληροφορίες, ενώ διευκολύνουν και την παρακολούθηση των ασθενών χωρίς να είναι απαραίτητη η συνεχής ανθρώπινη παρουσία.

Η τεχνολογία του Διαδικτύου των Πραγμάτων συναντάται και στον τομέα των μεταφορών και της αυτοκίνησης. Τα σύγχρονα αυτοκίνητα διαθέτουν αισθητήρες GPS, υγρασίας, θερμοκρασίας, ταχύτητας οι οποίοι στέλνοντας πληροφορίες συνεχώς ενημερώνουν τους οδηγούς για τις συνθήκες καιρού, για τα σημεία κυκλοφοριακής συμφόρησης σε μια προκαθορισμένη διαδρομή, ακόμα και για την κατάσταση του οχήματός τους λεπτομερώς, ανά πάσα στιγμή. Ακόμα στις μαζικές συγκοινωνίες υλοποιούνται πλέον συστήματα έξυπνων πόλεων, που δίνουν στους χρήστες τους ακριβή δρομολόγια και τοποθεσία των Μέσων Μαζικής Μεταφοράς.

Από τις παραπάνω αναφορές σε κάποιες εφαρμογές γίνονται ήδη αντιληπτές οι προεκτάσεις που έχει το Διαδίκτυο των Πραγμάτων στην καθημερινότητα των ανθρώπων. Ακόμα μεγαλύτερης σημασίας είναι η καινοτομία που έχει εισαχθεί στον τεχνολογικό/επιστημονικό τομέα και στον τομέα της μαζικής παραγωγής. Η διαρκής συλλογή πληροφοριών από πολλαπλούς αισθητήρες, η επεξεργασία και διανομή αυτής της πληροφορίας στο διαδίκτυο, καθώς και η δυνατότητα απομακρυσμένου ελέγχου αυξάνουν το εύρος των επιστημονικών δυνατοτήτων συνεχώς: έλεγχος και διατήρηση πάρκων εναλλακτικής ενέργειας (αιολικών, ηλιακών), επίβλεψη αγροτικών καλλιεργειών, βελτιστοποίηση ενεργειακής απόδοσης εργοστασίων και επιχειρήσεων, αλλαγή στη διαδικασία της βιομηχανικής παραγωγής, αυτοματοποιημένη παρακολούθηση εργαστηριακών συνθηκών, είναι λίγα μόνο από τα παραδείγματα δραστικών αλλαγών που έχει επιφέρει το Διαδίκτυο των Πραγμάτων.

Αναγκαία αναφορά αποτελεί και η αυτοματοποίηση των υπηρεσιών και η διαχείριση των δικτύων πολλών συνδέσεων, το οποίο είναι και το κύριο θέμα αυτής της διπλωματικής εργασίας. Η ανάπτυξη λογισμικού εξυπηρετητών συγκεκριμένου για την διαχείριση δικτυακής κίνησης σε ένα δίκτυο IoT βρίσκει μεγάλη χρήση σε εργαστηριακό και βιομηχανικό περιβάλλον, για την καλύτερη και κατά το δυνατόν άμεση εξυπηρέτηση των πελατών ακόμα και σε συνθήκες υψηλής κίνησης στο δίκτυο. Η συγκεκριμένη εφαρμογή, η υλοποίησή της και οι ελλείψεις της θα

αναλυθούν και παρακάτω.

## *2.4 Προκλήσεις και Προβλήματα*

Η ραγδαία ανάπτυξη του Διαδικτύου των Πραγμάτων και η όλο και αυξανόμενη ενσωμάτωσή του δεν θα μπορούσαν παρά να δημιουργούν όλο και περισσότερες προκλήσεις που η τεχνολογική κοινότητα και οι χρήστες καλούνται να λάβουν υπόψιν. Παρά την μεγάλη προσαρμοστικότητα των IoT δικτύων και των συσκευών τους, η επίλυση των ζητημάτων αυτών αποτελεί μεγάλη προτεραιότητα για την περαιτέρω ανάπτυξη του τομέα του Διαδικτύου των Πραγμάτων και την ενίσχυση της αξιοπιστίας των δικτύων αυτών. Τα κύρια ζητήματα αναφέρονται παρακάτω.

*Μεγάλος όγκος δεδομένων:* Οι αισθητήρες σε ένα δίκτυο IoT συλλέγουν πληροφορίες ανά τακτά χρονικά διαστήματα, κάτι που σε συνεχή λειτουργία του συστήματος οδηγεί σε συσσώρευση τεράστιου όγκου πληροφορίας προς επεξεργασία. Είναι λοιπόν πρόκληση για τους προγραμματιστές να βελτιστοποιήσουν την επεξεργασία, την εκκαθάριση και την διαλογή, διανομή, και αποθήκευση της χρήσιμης πληροφορίας.

*Ζητήματα συμβατότητας:* Οι διάφορες συσκευές IoT που κυκλοφορούν στην αγορά λειτουργούν βασισμένες σε μη ενιαία πολιτική υλικού και λογισμικού και χρησιμοποιούν πολλές φορές διαφορετικά μέσα για τη σύνδεσή τους στο διαδίκτυο. Αυτή η έλλειψη κυρίαρχης πολιτικής στην ανάπτυξη των συσκευών οδηγεί πολλές φορές σε προβλήματα συμβατότητας με τις υπόλοιπες συσκευές σε ένα δίκτυο, ζητήματα συνδεσιμότητας καθώς και αδυναμία ταυτόχρονης αναβάθμισης λογισμικού, ιδιαίτερα στις παλαιότερες ή φθηνότερες συσκευές.

*Πρωτόκολλο επικοινωνίας:* Το ήδη μεγάλο πλήθος των IoT συσκευών και οι προβλέψεις για το μέλλον του Διαδικτύου των Πραγμάτων, σε συνδυασμό με την εξάντληση των διαθέσιμων διευθύνσεων IPv4 δείχνουν πως για την ευρύτερη υιοθέτηση τους θα είναι αναγκαία η ολοκληρωτική μετάβαση στο πρωτόκολλο IPv6, προκειμένου να εξυπηρετηθούν μελλοντικά τα IoT δίκτυα.

*Ενέργεια:* Ο απομακρυσμένος έλεγχος που προσφέρει το Διαδίκτυο των Πραγμάτων προϋποθέτει ότι πολλές συσκευές IoT βρίσκονται σε σημεία αρκετά μακριά από κάποια πηγή ενέργειας. Είναι απαραίτητη λοιπόν η ανάπτυξη διάφορων επιλογών παροχής ενέργειας που θα εξυπηρετούν τα δίκτυα IoT χωρίς όμως να αποτελούν ταυτόχρονα κοστοβόρες και πιθανόν μη υλοποιήσιμες προτάσεις.

*Ασφάλεια και Ιδιωτικότητα:* Ίσως το μεγαλύτερο έλλειμμα όσον αφορά το Διαδίκτυο των Πραγμάτων. Οι συσκευές IoT, καθώς στην πλειοψηφία τους έχουν εγκατεστημένους μικρούς αισθητήρες μόνο, έχουν ευάλωτο λογισμικό ασφαλείας, ή και καθόλου, γεγονός που επιτρέπει πολύ εύκολα σε κακόβουλους χρήστες να διαμεσολαβήσουν των συσκευών και του κεντρικού συστήματος και να υποκλέψουν τα δεδομένα που στέλνονται μεταξύ αυτών. Όσον αφορά την ιδιωτικότητα, άλλο ένα ζήτημα είναι η χρήση των δεδομένων που συλλέγονται και κατά πόσο αυτές χρησιμοποιούνται από τρίτους (διαφημιστικές εταιρείες, έρευνες κ.α. για εύρεση μοτίβων συμπεριφοράς καταναλωτών).

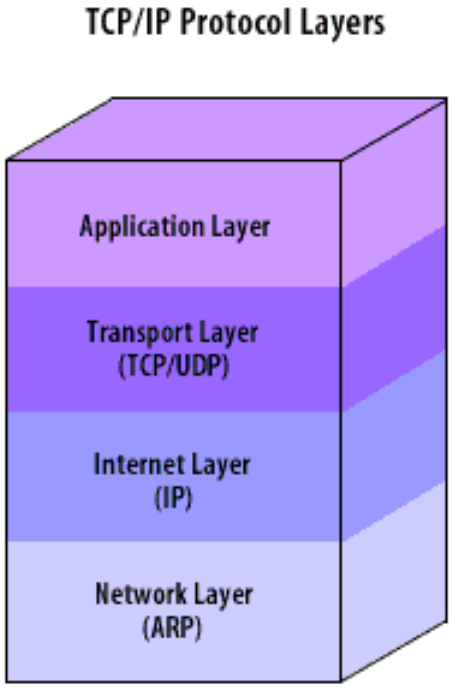
*Εξυπηρέτηση:* Με το Διαδίκτυο των Πραγμάτων να εξελίσσεται συνεχώς, τα δίκτυα IoT θα πρέπει να εξυπηρετούν από μερικές χιλιάδες έως και εκατομμύρια συσκευές ταυτόχρονα χωρίς απώλεια δεδομένων. Ο δικτυακός φόρτος σε τέτοιες περιπτώσεις είναι εξαιρετικά μεγάλος, αλλά με την κατάλληλη δόμηση του συστήματος μπορεί να αντιμετωπιστεί με βέλτιστη απόδοση. Αντικείμενο αυτής της διπλωματικής εργασίας είναι η διερεύνηση των δυνατοτήτων και των ορίων διάφορων δομών εξυπηρετητών σε δίκτυα IoT, υπό συνθήκες μεγάλης δικτυακής κίνησης.

## *2.5 Μοντέλα IoT*

### *2.5.1 Επίπεδα δικτύων και επίπεδα μοντέλου IoT*

Φυσική διασύνδεση, επικοινωνία δικτύου, μεταφορά δεδομένων, εφαρμογή, είναι τα τέσσερα βασικά επίπεδα στο μοντέλο της δικτυακής επικοινωνίας. Με την εξέλιξη στον τομέα του διαδικτύου έχουν δημιουργηθεί πολλά πρωτόκολλα επικοινωνίας , καθένα από τα οποία

έχουν συγκεκριμένα χαρακτηριστικά που τα κατατάσσουν στα παραπάνω επίπεδα και πλεονεκτήματα έναντι άλλων σε σχέση με την περίπτωση και το επίπεδο της εφαρμογής που χρησιμοποιούνται.




*Εικόνα 2.4: Επίπεδα του πρωτοκόλλου TCP/IP*

Στην αρχιτεκτονική του Διαδικτύου των Πραγμάτων συναντώνται τέσσερα επίπεδα οργάνωσης, για να γίνει εφικτή η σύνδεση φυσικού και εικονικού κόσμου. Ο σχεδιασμός της αρχιτεκτονικής ορίζει τα επίπεδα των αισθητήρων (Sensor Layer), της διαχείρισης υπηρεσιών (Management Service Layer), του δικτύου (Gateway & Network Layer) και το επίπεδο εφαρμογής (Application Layer).

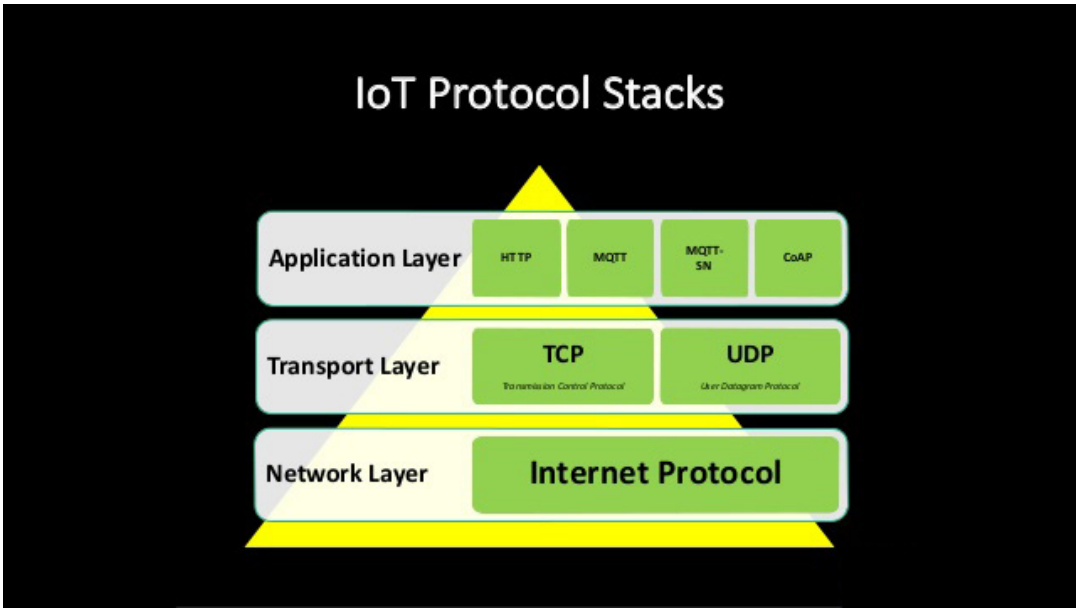
**Sensor Layer**

Το επίπεδο των αισθητήρων αποτελείται από όλες τις συσκευές και τα άτομα που διαθέτουν ενσωματωμένους αισθητήρες για να συλλέγουν πληροφορίες και να αλληλεπιδρούν με το περιβάλλον τους. Κάθε τέτοιο αντικείμενο φέρει ένα χαρακτηριστικό αναγνωριστικό, το UUID (Universal Unique Identifier), ου του προσδίδει μοναδική ψηφιακή ταυτότητα μέσα στο δίκτυο. Το UUID μπορεί να είναι όνομα ή διεύθυνση. Για να θεωρείται

ένα αντικείμενο μέρος ενός δικτύου που υλοποιείται στα πλαίσια του Διαδικτύου των Πραγμάτων θα πρέπει να έχει τη δυνατότητα συλλογής δεδομένων και ίσως ελαφριάς επεξεργασίας τους, καθώς και δυνατότητα να συνδέεται στο διαδίκτυο για να στέλνει αυτά τα δεδομένα στα υπόλοιπα μέρη του δικτύου.

### Gateway & Network Layer

Στο επίπεδο αυτό γίνεται η ευρύτερη σύνδεση όλων των αντικειμένων ώστε να δημιουργήσουν ένα ευφυές δίκτυο. Οι πύλες (Gateways) είναι συσκευές ή λογισμικό που συνήθως λειτουργούν κοντά στους αισθητήρες και διαμεσολαβούν μεταξύ αυτών και της δομής του νέφους (Cloud) και των εξυπηρετητών που τους διαχειρίζονται. Σε μεγάλα δίκτυα οι πύλες μπορεί να οργανώνονται σε πολλά επίπεδα, δημιουργώντας έτσι περίπλοκες τοπογραφίες για να εξυπηρετήσουν τις ανάγκες σύνδεσης και μεταφοράς πληροφορίας μεταξύ των συμμετεχόντων.




*Εικόνα 2.5: Επίπεδα δικτύων και πρωτόκολλα στο IoT*

### Management Service Layer

Η αποθήκευση των πληροφοριών του δικτύου και η επεξεργασία και πιθανή επαναπροώθησή τους από τους εξυπηρετητές είναι λειτουργίες που

γίνονται στο επίπεδο της διαχείρισης υπηρεσιών. Οι πληροφορίες συνήθως αποθηκεύονται στο νέφος (Cloud) για εύκολη διαχείριση και εύρεσή τους από τους εξυπηρετητές.

**Application Layer**

Στο τελικό επίπεδο οργάνωσης της αρχιτεκτονικής ενός δικτύου IoT γίνεται η παρουσίαση των αποτελεσμάτων των μετρήσεων που έχουν ληφθεί από τους αισθητήρες και η οπτικοποίησή τους. Γίνεται επομένως το επίπεδο όπου γίνεται χρήση αναλυτικών εργαλείων για καλύτερη ανάλυση των ακατέργαστων δεδομένων έτσι ώστε να είναι εύκολα αναγνώσιμα από τον τελικό χρήστη του δικτύου IoT.

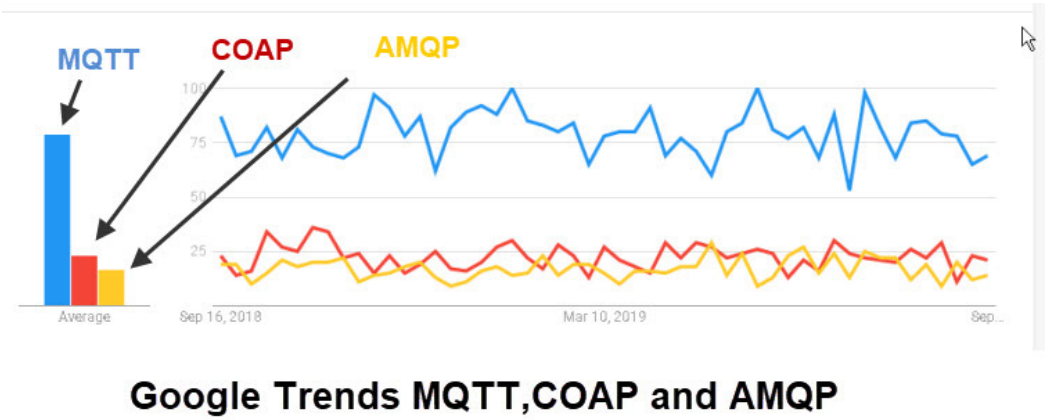
### *2.5.2 Πρωτόκολλα του Διαδικτύου των Πραγμάτων*

Όσον αφορά το Διαδίκτυο των Πραγμάτων, υπάρχουν διάφορα πρωτόκολλα, με πιο γνωστά τα MQTT, CoAP, AMQP, XMPP, DDS. Ενδεικτικά, το πρωτόκολλο AMQP είναι βασισμένο στο TCP/IP, με φιλοσοφία δημοσίευσης/εγγραφής, και είναι σχεδιασμένο για την επικοινωνία μεταξύ των εξυπηρετητών με πρωταρχικό στόχο την ελάχιστη απώλεια μηνυμάτων. Είναι ιδιαίτερα αξιόπιστο στην επεξεργασία χιλιάδων συναλλαγών βάσει σειράς προτεραιότητας, μιας και αναπτύχθηκε στα πλαίσια τραπεζικών συναλλαγών, όπου και συνεχίζει να έχει βασική εφαρμογή.

Το πρωτόκολλο CoAP (Constrained Application Protocol) θεωρείται μια πιο ελαφριά από πλευράς πληροφορίας εκδοχή του HTTP πρωτοκόλλου, και δουλεύει βασισμένο στο πρωτόκολλο UDP, με κύρια χρήση σε συσκευές περιορισμένων πόρων που συνδέονται στο ίδιο δίκτυο ή συσκευές διαφορετικών δικτύων που ωστόσο επικοινωνούν μεταξύ τους μέσω κοινού δικτύου.

Το XMPP (Extensible Messaging and Presence Protocol, παλιότερα γνωστό και ως Jabber) έχει σχεδιαστεί για εφαρμογές ανταλλαγής άμεσων γραπτών μηνυμάτων (Instant Messaging), τρέχει πάνω σε HTTP (και άρα πάνω σε TCP) και διευκολύνει την απεύθυνση σε συγκεκριμένη

συσκευή, αφού λειτουργεί με διευθυνσιοδότηση τύπου name@domain.com. Το χαρακτηριστικό αυτό είναι ιδιαίτερα χρήσιμο όταν τα δεδομένα ανταλλάσσονται μεταξύ απομακρυσμένων και όχι άμεσα συσχετιζόμενων συσκευών, όπως για παράδειγμα η επικοινωνία μεταξύ ατόμων.




*Εικόνα 2.5: Χρήση των πρωτοκόλλων MQTT, CoAP, AMQP σε εφαρμογές τα τελευταία 2 χρόνια (πηγή: Google Trends)*

Το πρωτόκολλο DDS (Data-Distribution Service) στοχεύει στις συσκευές που χρησιμοποιούν άμεσα δεδομένα, και τα διαμοιράζει σε άλλες συσκευές, χωρίς να είναι απαραίτητη η διαμεσολάβηση δομής εξυπηρέτησης – συνδέει δηλαδή συσκευές απευθείας μεταξύ τους. Μπορεί να παραδώσει εκατομμύρια μηνύματα ανά δευτερόλεπτο σε πολλές συσκευές ταυτόχρονα. Προσφέρει τη δυνατότητα διαλογής της πληροφορίας, κάνει χρήση του μοντέλου δημοσίευσης/εγγραφής και επιπέδων ποιότητας υπηρεσίας (Quality of Service – QoS).

Το πρωτόκολλο MQTT, που αποτελεί και το βασικό αντικείμενο της διπλωματικής αυτής εργασίας, αναλύεται παρακάτω.

### *2.5.3 Χαρακτηριστικά πρωτοκόλλου MQTT*

Ένα από τα πλέον διαδεδομένα πρωτόκολλα, και αυτό στο οποίο βασίζονται οι τεχνολογίες που θα χρησιμοποιηθούν στην διπλωματική αυτή εργασία, είναι το πρωτόκολλο MQTT (Message Queue Telemetry Transport). Είναι ένα πρωτόκολλο του επιπέδου Εφαρμογής, που είναι βασισμένο πάνω στο TCP/IP των επιπέδων Μεταφοράς και Δικτύου. Έχει

σχεδιαστεί πάνω στη φιλοσοφία δημοσίευσης/εγγραφής (publish/subscribe), είναι αρκετά ελαφρύ όσον αφορά το μέγεθος των μηνυμάτων και έχει τη δυνατότητα να μεταφέρει οποιονδήποτε τύπο πληροφορίας (industry-agnostic). Επιτρέπει λοιπόν στις συσκευές να επικοινωνούν ασύγχρονα και ιδιαίτερα αποδοτικά σε απομακρυσμένα δίκτυα μικρής αξιοπιστίας και με μεγάλο κόστος ανά byte.

Το MQTT χρησιμοποιεί μοντέλο δημοσίευσης/εγγραφής, ή αλλιώς publish/subscribe, δηλαδή οι αποστολείς των μηνυμάτων, οι publishers, δεν στέλνουν μηνύματα σε συγκεκριμένους παραλήπτες (subscribers), αλλά κατηγοριοποιούν τα μηνύματα σε κλάσεις (topics), ανάλογα με την πληροφορία που αυτά περιέχουν, και δεν γνωρίζουν ποιοι παραλήπτες – αν υπάρχουν – θα λάβουν τα μηνύματα. Αντίστοιχα, οι παραλήπτες δηλώνουν το ενδιαφέρον τους σε κάποια κλάση μηνυμάτων και εγγράφονται σε αυτήν, λαμβάνοντας έτσι μηνύματα πάνω σε ένα θέμα χωρίς να γνωρίζουν τον αποστολέα. Αυτό το μοντέλο οργάνωσης των μηνυμάτων και των πελατών προσφέρει το πλεονέκτημα της μεγάλης επεκτασιμότητας του δικτύου, αφού οι αποστολείς και οι παραλήπτες δεν είναι ρητά συνδεδεμένοι μεταξύ τους, ούτε και είναι απαραίτητο να γνωρίζουν την τοπολογία του δικτύου.

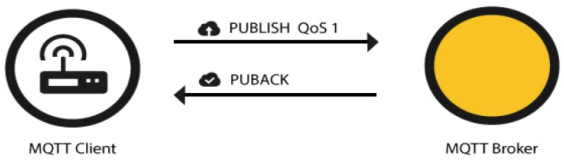
Το ελάχιστο μέγεθος ενός μηνύματος MQTT είναι 2 bytes δεδομένων, ενώ μπορεί να φτάσει μέχρι και τα 256 Mbs. Η πληροφορία στέλνεται ως απλό κείμενο (plaintext), επομένως η προστασία της από τυχόν υποκλοπές μπορεί να διασφαλιστεί από το TCP πρωτόκολλο που χρησιμοποιείται στα κατώτερα επίπεδα της επικοινωνίας.

Το MQTT επίσης υποστηρίζει βαθμονόμηση ποιότητας υπηρεσίας (Quality of Service – QoS), σε τρία επίπεδα:

- **QoS 0:** το πολύ μία φορά, δηλαδή το μήνυμα στέλνεται μία φορά και πελάτης/εξυπηρετητής δεν κάνουν κάποια άλλη κίνηση να επιβεβαιώσουν την παράδοσή του

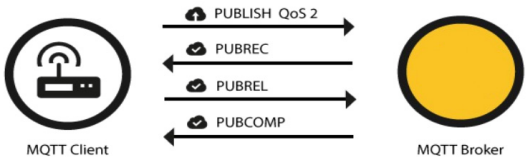

*Εικόνα 2.6: Σχηματική απεικόνιση επιπέδου QoS 0*

- **QoS 1:** τουλάχιστον μία φορά, δηλαδή το μήνυμα στέλνεται ξανά από τον αποστολέα μέχρι να γίνει επιβεβαίωση της παράδοσής του




*Εικόνα 2.7: Σχηματική απεικόνιση επιπέδου QoS 1*

- **QoS 2:** ακριβώς μία φορά, αποστολέας και παραλήπτης φέρουν μια χειραψία δύο επιπέδων για να διαβεβαιώσουν ότι το μήνυμα στάλθηκε ακριβώς μία φορά.

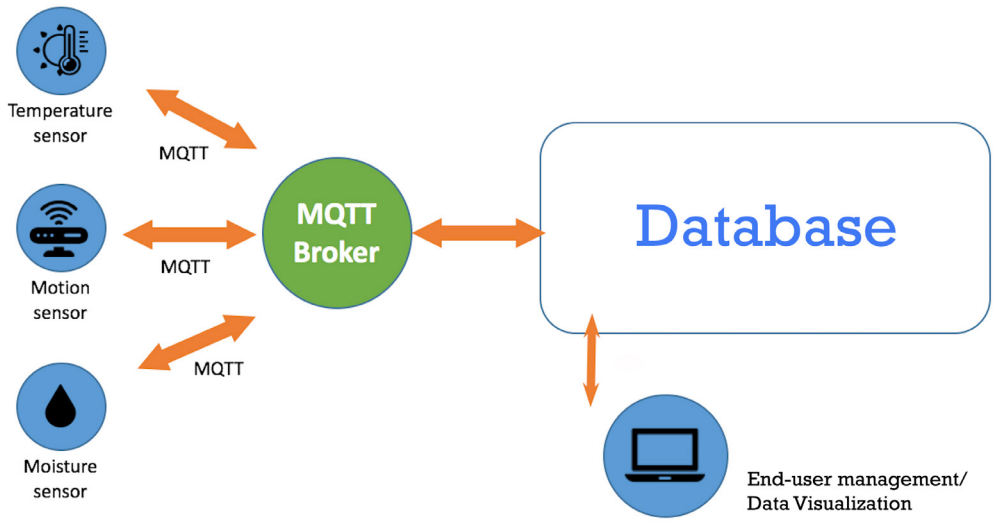



*Εικόνα 2.8: Σχηματική απεικόνιση επιπέδου QoS 2*

Οι συσκευές (clients) ενός δικτύου IoT που λειτουργεί βασισμένο στο πρωτόκολλο MQTT, συνδέονται απαραίτητα με έναν εξυπηρετητή (MQTT Broker/broker), όπου εγγράφονται σε συγκεκριμένα θέματα (topics) και στέλνουν εκεί τις μετρήσεις και τις πληροφορίες που λαμβάνουν. Η λειτουργία του εξυπηρετητή και του τρόπου ανταλλαγής μηνυμάτων μεταξύ συσκευών και εξυπηρετητή θα αναλυθεί περισσότερο στο Κεφάλαιο 3.

# Κεφάλαιο 3
# Αρχιτεκτονική

Σε αυτό το κεφάλαιο θα γίνει αναλυτική περιγραφή της αρχιτεκτονικής του δικτύου που σχεδιάστηκε και χρησιμοποιήθηκε για την τέλεση των πειραμάτων. Σε κάθε βαθμίδα οργάνωσης θα αναλυθεί τόσο ο τρόπος λειτουργίας της, όσο και το λογισμικό που χρησιμοποιήθηκε.




*Εικόνα 3.1: Σχεδιάγραμμα της αρχιτεκτονικής του δικτύου*

Ένα απλό δίκτυο IoT, όπως αυτό που χρησιμοποιείται στην παρούσα διπλωματική εργασία, αποτελείται από 4 βασικά επίπεδα, τα οποία θεωρητικά αναλύθηκαν στο κεφάλαιο 2, και παρουσιάζονται ξανά εδώ υλοποιημένα ως εξής:

- Τους αισθητήρες, που κάνουν τις απαραίτητες μετρήσεις για το δεδομένο δίκτυο (Sensor Layer)

- Τον εξυπηρετητή, ο οποίος διατηρεί συνδέσεις με τους αισθητήρες, δέχεται και συγκεντρώνει τις πληροφορίες από αυτούς μέσω πακέτων που του αποστέλλουν, ελέγχει την σωστή λειτουργία τους (Management Service Layer)
- Τη βάση δεδομένων, όπου αποθηκεύονται οι πληροφορίες που επεξεργάζεται ο εξυπηρετητής (Management Service Layer)
- Τη μονάδα οπτικοποίησης και ηλεκτρονικής παρακολούθησης της βάσης δεδομένων (Application Layer).

## *3.1 Αισθητήρες*

Οι αισθητήρες είναι οι συσκευές που συλλέγουν την σχετική με τη λειτουργία τους πληροφορία. Όπως αναφέρθηκε ήδη, οι συσκευές μπορεί να είναι

- ηλεκτρονικές ή αναλογικές, όπως: καθημερινές συσκευές (κάμερες, κλιματιστικά, φώτα, πρίζες, smartwatches, και πολλές ακόμα) που συναντώνται σε ένα σπίτι, αισθητήρες (κλίσης, θερμοκρασίας, υγρασίας, τοποθεσίας κ.α.), διακόπτες λειτουργίας, αλλά και ολοκληρωμένα συστήματα που χρησιμοποιούνται στον τομέα της υγείας, σε εργοστάσια, εργαστήρια, ΜΜΜ.
- αντικείμενα, ζώα και άνθρωποι, με τη δυνατότητα να αποστέλλουν πληροφορίες στο δίκτυο.

Κοινό χαρακτηριστικό όλων των συσκευών σε ένα IoT δίκτυο είναι το προσαρμοσμένο λογισμικό που τους δίνει τη δυνατότητα να συνδέονται στο διαδίκτυο και να μπορούν να στέλνουν πληροφορίες για την λειτουργία τους. Κάθε συσκευή έχει ένα μοναδικό αναγνωριστικό, με το οποίο μπορεί να βρεθεί εύκολα από τον διαχειριστή του συστήματος το είδος της συσκευής, ο τόπος λειτουργίας της και άλλες χρήσιμες πληροφορίες.

Ανάγκη λοιπόν για την εκτέλεση πειραμάτων στην παρούσα εργασία είναι η δημιουργία συσκευών (clients) που θα στέλνουν πληροφορία στον εξυπηρετητή (broker).

Ως MQTT client θα ορίζεται κάθε συσκευή η οποία έχει ενσωματωμένη

την βιβλιοθήκη MQTT και συνδέεται σε έναν MQTT broker μέσα σε ένα δίκτυο.

Το είδος των συσκευών δεν παίζει ρόλο στην συγκεκριμένη πειραματική διαδικασία, παρά μόνο το πλήθος τους, μιας και γίνεται έλεγχος των δυνατοτήτων του δικτύου σε δυσμενείς περιπτώσεις, και το μέγεθος των μηνυμάτων. Δεδομένου αυτού έγινε προσομοίωση συσκευών-αισθητήρων σε εικονικό περιβάλλον, με τη χρήση προγραμμάτων (ο κώδικας των οποίων παρατίθεται στο παράρτημα) τα οποία δημιουργούν το επιθυμητό πλήθος συσκευών, με συγκεκριμένο ρυθμό δημιουργίας νέων συνδέσεων στον server. Η κάθε συσκευή συνδέεται με τον broker (ο τρόπος λειτουργίας του θα αναλυθεί παρακάτω) και αποστέλλει μηνύματα με επιλεγμένη συχνότητα.

Το περιβάλλον δημιουργίας client υλοποιήθηκε με την χρήση εικονικών μηχανημάτων (Virtual Machines – VMs), πάνω στην υποδομή Cloud του Leonardo Lab της COSMOTE, που είναι χτισμένο πάνω στο ανοιχτό λογισμικό του Openstack. Η τοπολογία του δικτύου όσον αφορά αυτό το επίπεδο οργάνωσης δεν αποτέλεσε σημείο μελέτης της διπλωματικής, παρά μέσο για την υλοποίησή της, και για αυτόν τον λόγο δεν αναλύεται περαιτέρω.

Για τα προγράμματα της προσομοίωσης, αρχικά επιλέχθηκε η προγραμματιστική γλώσσα Python. Συγκεκριμένα, γράφτηκε ένα πρόγραμμα που δημιουργεί μία ενεργή σύνδεση μιας συσκευής με τον server, και στέλνει μηνύματα σε ένα συγκεκριμένο θέμα (topic) αυτού. Το πρόγραμμα αυτό χρησιμοποιεί την βιβλιοθήκη Paho της Python, για την δημιουργία ενός αντικειμένου πελάτη (client object) που ανοίγει και διατηρεί μια σύνδεση με τον εξυπηρετητή, και στέλνει μηνύματα ή/και συνδέεται σε κάποιο topic. Στη συνέχεια, με ένα πρόγραμμα γραμμένο σε BASH, γίνεται παράλληλη εκτέλεση του παραπάνω προγράμματος, ανά συγκεκριμένο χρονικό διάστημα (συχνότητα δημιουργίας συνδέσεων – spawning frequency).

Αξίζει να σημειωθεί σε αυτό το σημείο, ότι κατά την σύνταξη των προγραμμάτων παρατηρήθηκε ότι το πρόγραμμα της μίας συσκευής, γραμμένο σε Python, λόγω της συνεχώς αυξανόμενης παράλληλης εκτέλεσής του, κατανάλωνε με εξαιρετικά γρήγορους ρυθμούς τους πόρους των VM, και συγκεκριμένα την μνήμη RAM αυτών, με αποτέλεσμα να είναι δύσκολη η αναπαραγωγή σεναρίου δικτύου με πολλούς συνδεδεμένους

clients, το οποίο ήταν και το ζητούμενο. Δεδομένου ότι η πρόταση του να παραχωρηθούν περισσότερα VM δεν ήταν βέλτιστη, το πρόβλημα αυτό αντιμετωπίστηκε με την μετατροπή της εκτέλεσης του προγράμματος από Python σε C, λύση που αποδείχτηκε εξαιρετικά αποδοτική, αφού η παράλληλη εκτέλεση των προγραμμάτων γραμμένων σε C μείωσε δραστικά την κατανάλωση της RAM ανά δημιουργούμενη συσκευή, με αποτέλεσμα να επιτευχθεί η προσομοίωση δικτύου με πολλές ενεργές συνδέσεις χρησιμοποιώντας μικρό πλήθος (2-4) εικονικών μηχανημάτων (συγκριτικά: μέχρι 400 ενεργοί clients ανά VM με Python, 1000+ ενεργοί clients ανά VM με C).

```c
CLIENTID = r;

MQTTClient client;
MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
MQTTClient_message pubmsg = MQTTClient_message_initializer;
MQTTClient_deliveryToken token;
int rc;

MQTTClient_create(&client, ADDRESS, CLIENTID,
    MQTTCLIENT_PERSISTENCE_NONE, NULL);
conn_opts.keepAliveInterval = 20;
conn_opts.cleansession = 1;

MQTTClient_setCallbacks(client, NULL, connlost, msgarrvd, delivered);

if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
{
    printf("Failed to connect, return code %d\n", rc);
    exit(EXIT_FAILURE);
}

MQTTClient_subscribe(client, TOPIC, QOS);

while(1){
        delay(3000);
        pubmsg.payload = PAYLOAD;
        pubmsg.payloadlen = strlen(PAYLOAD);
        pubmsg.qos = QOS;
        pubmsg.retained = 0;
        MQTTClient_publishMessage(client, TOPIC, &pubmsg, &token);
}


rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
MQTTClient_disconnect(client, 10000);
MQTTClient_destroy(&client);
```

*Εικόνα 3.2: Κώδικας ενός Client object σε γλώσσα C.*

## *3.2 Εξυπηρετητής*

### *3.2.1 Θεωρητικό υπόβαθρο*

Ο εξυπηρετητής είναι είναι υλικό ή / και λογισμικό που αναλαμβάνει την παροχή διάφορων υπηρεσιών, «εξυπηρετώντας» αιτήσεις άλλων προγραμμάτων, γνωστών ως πελάτες (**clients**) που μπορούν να τρέχουν στον ίδιο υπολογιστή ή σε σύνδεση μέσω δικτύου. Είναι η δομή που είναι υπεύθυνη για την επαλήθευση της εγκυρότητας των μηνυμάτων, την επεξεργασία τους και την δρομολόγησή τους μέσα στο δίκτυο, και με αυτόν τον τρόπο ελαχιστοποιεί την εξάρτηση των υπόλοιπων συσκευών μεταξύ τους.

Τα βασικά χαρακτηριστικά ενός εξυπηρετητή συνοψίζονται παρακάτω:

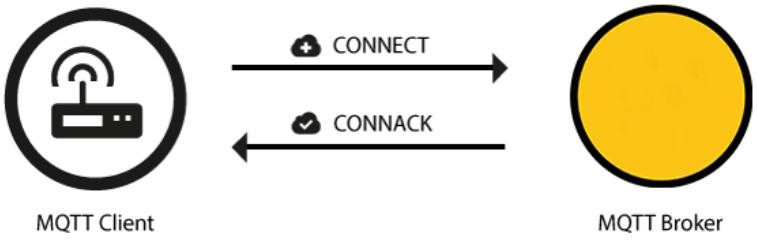
- Δρομολόγηση μηνυμάτων προς μία ή πολλές κατευθύνσεις
- Μετατροπή των μηνυμάτων σε διαφορετική μορφή
- Θρυμματισμό μηνυμάτων, σε πολλά μικρότερα για αποστολή στον προορισμό τους, και ανασύνθεση των απαντήσεων για επιστροφή στον αρχικό αποστολέα
- Επικοινωνία με εξωτερικές δομές για αποθήκευση μηνυμάτων (Cloud)
- Αίτημα σε διαδικτυακές υπηρεσίες για επανάκτηση μηνυμάτων
- Αναφορά συμβάντων ή λαθών
- Δρομολόγηση μηνυμάτων βάσει περιεχομένου η θέματος, σύμφωνα με την λογική του μοντέλου δημοσίευσης/εγγραφής.

#### *a. MQTT Broker*

Εφόσον γίνεται αναφορά σε δίκτυο IoT, ο εξυπηρετητής συνήθως αναφέρεται ως διακομιστής MQTT ή MQTT Broker και όπως αναφέρθηκε παραπάνω ακολουθεί μοντέλο publish/subscribe. Σε αυτό το μοντέλο, δεν υφίσταται η συνήθης σχέση client-server, όπου υπάρχει επικοινωνία του client με ένα τελικό σημείο αναφοράς (endpoint), αλλά ο κάθε client δεν γνωρίζει καν την ύπαρξη των υπόλοιπων clients του δικτύου και την μεταξύ τους επικοινωνία αναλαμβάνει ο broker, που κάνει τη διαλογή των μηνυμάτων

και την επαναπροώθησή τους στους εκάστοτε συνδρομητές-subscribers. Το σύστημα αυτό προσφέρει μεγάλη επεκτασιμότητα αφού publisher και subscriber δεν χρειάζεται να τρέχουν ταυτόχρονα και η λειτουργία τους δεν διακόπτεται από την αποστολή ή λήψη του μηνύματος. Αξίζει να σημειωθεί ότι η δυνατότητα του publish/subscribe μπορεί να γίνει από το ίδιο αντικείμενο MQTT client (MQTT client object).

Η φιλοσοφία της υλοποίησης που μελετάται βασίζεται στην ταξινόμηση ανά θέμα (topic based): κάθε client εγγράφεται σε ένα ή περισσότερα topic του broker, ανάλογα με τη λειτουργία του στο δίκτυο. Η εγγραφή σε ένα topic αποτελεί μια ενεργή σύνδεση με τον broker: πιο αναλυτικά, η σύνδεση ενός client με τον broker γίνεται με την τέλεση μιας χειραψίας μεταξύ τους. Ο client στέλνει ένα μήνυμα για έναρξη της σύνδεσης, το CONNECT message, και ο broker απαντάει με το αντίστοιχο CONNACK (connection aknowlegment) και την κατάστασή του. Η σύνδεση διατηρείται μέχρι ο client να στείλει ένα μήνυμα αποσύνδεσης ή να διακοπεί η σύνδεση από άλλους λόγους.




*Εικόνα 3.3: Σχεδιάγραμμα της χειραψίας MQTT Client-MQTT Broker κατά την δημιουργία νέας σύνδεσης*

Τα μηνύματα αποστέλλονται από τους publishers, οι οποίοι μπορεί είτε να είναι ήδη εγγεγραμμένοι σε κάποιο topic ή να συνδέονται με τον broker για να στείλουν ένα μήνυμα σε κάποιο topic. Ο broker είναι υπεύθυνος για την προώθηση των μηνυμάτων που θα στείλει κάποιος publisher σε όλους τους subscribers αυτού του topic. Για τις ενεργές συνδέσεις στον broker, το πρωτόκολλο MQTT παρέχει τις παρακάτω δυνατότητες:

- όνομα χρήστη, **ClientID**, το οποίο πρέπει να είναι μοναδικό για κάθε client.
- προτεραιότητα στην αποστολή των μηνυμάτων, όπως αναφέρθηκε ήδη,

που γίνεται με τα επίπεδα ποιότητας εξυπηρέτησης, το **Quality of Service.**

- δυνατότητα να διατηρηθεί κάποιο μήνυμα, τα λεγόμενα **retained messages**, δηλαδή να αποθηκευτεί από τον broker και να αποστέλλεται σε κάθε client που εγγράφεται σε κάποιο topic για πρώτη φορά.
- δυνατότητα διατήρησης της σύνδεσης, αναφερόμενη ως **clean session flag** (false for persistent connections) έτσι ώστε ακόμα και αν αποσυνδεθεί ο client, με την επανασύνδεσή του να του αποσταλούν όσα μηνύματα δεν έλαβε όσο βρισκόταν εκτός δικτύου.
- την επιλογή **SessionPresent**, που ενημερώνει τον client αν ο broker είχε ήδη μία διατηρημένη σύνδεση με τον ίδιο στο παρελθόν.
- επιλογή του μέγιστου χρονικού διαστήματος που μπορεί να διατηρηθεί η σύνδεση client και broker χωρίς να γίνεται αποστολή μηνυμάτων, η επιλογή **KeepAlive**. Μετράται σε δευτερόλεπτα και περιέχεται στο αρχικό μήνυμα σύνδεσης του client, και ουσιαστικά αποτελεί μια συμφωνία ότι ο client θα αποστέλλει Ping Requests στον broker για επιβεβαίωση της σύνδεσης.
- την δυνατότητα ενημέρωσης των clients που είναι συνδεδεμένοι για το αν κάποιος client σε αυτό το topic αποσυνδέθηκε βίαια, που ονομάζεται **Last Will and Testament** (LWT).
- σύνδεση με **username/password**, μια επιλογή που αν δεν είναι κρυπτογραφημένη με κάποιο τρόπο αποτελεί σοβαρό κενό ασφαλείας, μιας και οι πληροφορίες αυτές αποστέλλονται ως απλό κείμενο σε ένα μήνυμα MQTT.

Ο ίδιος ο broker έχει συγκεκριμένα topics, τα systopics, στα οποία στέλνει μηνύματα που αφορούν την κατάστασή του και αυτή των ενεργών και παρελθόντων συνδέσεων του δικτύου. Η πρόσβαση στα systopics γίνεται με την άδεια από πλευράς broker, και μέσω αυτών μπορεί να γίνει η παρακολούθηση του δικτύου, το logging. Για την καταγραφή διάφορων στοιχείων των πειραμάτων, γράφτηκαν συγκεκριμένα προγράμματα logger, τα οποία μέσω των systopics έκαναν συλλογή μετρήσεων σε σχέση με το δίκτυο και την κατανάλωση πόρων στο μηχάνημα του broker, κατά την εκτέλεση των προσομοιώσεων. Περισσότερες λεπτομέρειες για τους logger θα δοθούν στις Ενότητες 3.2.α και 3.2.β.

### *β. Διαθέσιμα λογισμικά*

Υπάρχουν πολλά διαθέσιμα λογισμικά για την υλοποίηση ενός MQTT broker, ελεύθερα (**open source**) και εμπορικά, τα οποία διαφοροποιούνται βάσει των δυνατοτήτων που προσφέρουν και των χαρακτηριστικών που πρέπει να ικανοποιηθούν στο εκάστοτε σύστημα. Ενδεικτικά αναφέρονται τα πιο γνωστά από αυτά: Apache ActiveMQ/Kafka/Qpid, Eclipse Mosquitto MQTT broker, HiveMQ, HornetQ, IBM MQ, Joram, RabbitMQ, EMQX Broker.

Για την εκτέλεση των πειραμάτων της εργασίας αυτής επιλέχθηκαν δύο από τα παραπάνω λογισμικά MQTT Broker, το Moqsuitto MQTT Broker της Eclipse και το EMQX Broker. Ο λόγος που επιλέχθηκε ο Mosquitto MQTT Broker είναι επειδή αποτελεί μια πολύ ελαφριά υλοποίηση broker και μάλιστα ανοιχτού λογισμικού, που χρησιμοποιεί το πρωτόκολλο MQTT και μπορεί να εγκατασταθεί σε μεγάλο εύρος συσκευών και πάνω σε πολλά λειτουργικά συστήματα, από προσωπικούς υπολογιστές μέχρι server υψηλών πόρων. Ο Mosquitto Broker επίσης διαθέτει πολλά χαρακτηριστικά που προσφέρουν και άλλες υλοποιήσεις κλειστού λογισμικού και είναι σε μεγάλο βαθμό ανοιχτός στην παραμετροποίηση, ενώ επιπλέον υποστηρίζει πολλές εκδόσεις του πρωτοκόλλου MQTT, γεγονός που διευκολύνει την σύνδεση πολλών συσκευών που ίσως δεν είναι όλες ενημερωμένες στον ίδιο βαθμό.

Δεδομένου ότι η υλοποίηση clustered server δεν υποστηρίζεται από τον Mosquitto Broker, για το συγκεκριμένο σκοπό επιλέχθηκε ο EMQX Broker, επίσης ανοιχτού λογισμικού, ο οποίος αποτελεί ένα εξαιρετικά παραμετροποιήσιμο λογισμικό με μεγάλη συμβατότητα, καθώς υποστηρίζει τα περισσότερα από τα πρωτόκολλα του IoT, και μεγάλη επεκτασιμότητα (scalability). Αποτελεί επίσης ένα αρκετά ελαφρύ λογισμικό, παρόλα αυτά με πολλές δυνατότητες.

Τα χαρακτηριστικά των δύο αυτών πακέτων λογισμικών broker θα αναλυθούν στα δύο επόμενα υποκεφάλαια. Σημαντικό είναι να αναφερθεί ότι η επιλογή ανοιχτού λογισμικού έγινε με κύριο γνώμονα την ιδιαίτερα σημαντική δυνατότητα που αυτό προσφέρει, την δόμηση ενός συστήματος εξαρχής με παραμέτρους που μπορούν να μεταβάλλονται ελεύθερα και κατά τις ανάγκες που παρουσιάζονται. Σημαντικός λόγος είναι επίσης η συμβολή στην κοινότητα του ανοιχτού λογισμικού με την διερεύνηση

των δυνατοτήτων και των ορίων του λογισμικού broker στην χρήση του σε εκτεταμένες εφαρμογές, όπως ένα εργαστηριακό περιβάλλον ή ένα περιβάλλον παραγωγής, καθώς και την περαιτέρω βελτίωση των λογισμικών αυτών.

### *γ. Clustering και Load Balancing*

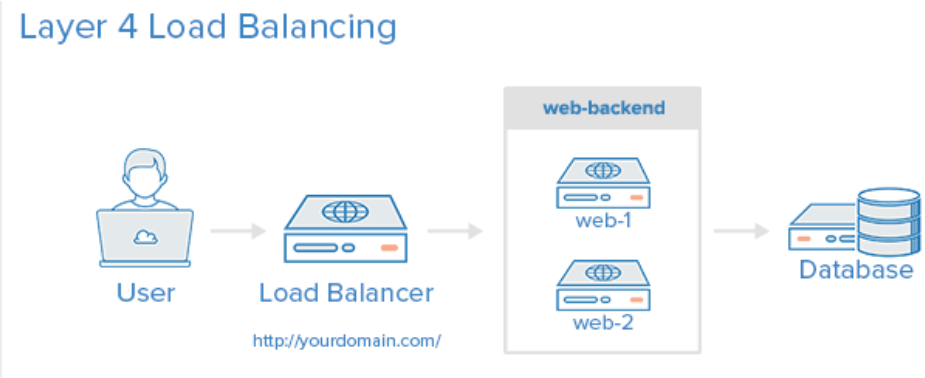
Ανάλογα με τις ανάγκες της υλοποίησης, μπορεί ο broker να είναι ένα μηχάνημα αλλά μπορεί να είναι και πολλά μηχανήματα, σε διαφορετική φυσική τοποθεσία, συνδεδεμένα μεταξύ τους και σε άμεση επικοινωνία για την εξυπηρέτηση των clients. Γίνεται λοιπόν εισαγωγή στην έννοια του **clustering**, την σύνδεση δηλαδή δύο ή περισσότερων μηχανημάτων και την κοινή λειτουργία τους έτσι ώστε να ανταποκρίνονται σαν ένα ενιαίο σύστημα. Κάθε μηχάνημα σε ένα cluster (**cluster node**) τρέχει – συνήθως – το ίδιο λογισμικό με τα υπόλοιπα, και συνδέεται με αυτά δικτυακά, λειτουργώντας με ένα κοινό σύνολο κανόνων που αφορά στην εξυπηρέτηση των clients. Μέσω αυτής της μεθόδου επιτυγχάνεται η απόδοση του συστήματος και μειώνεται δραστικά η πιθανότητα αποτυχίας εξυπηρέτησης, αφού ο φόρτος μοιράζεται ανάμεσα στα μηχανήματα και σε περίπτωση που κάποιο από αυτά χάσει τη δυνατότητα απόκρισης, αναλαμβάνουν τα υπόλοιπα (no single point of failure). Η μέθοδος αυτή εξερευνάται στα πλαίσια αυτής της εργασίας ως μέσο αντιμετώπισης προβλημάτων στην ανταπόκριση ενός broker υπό έντονες δικτυακές συνθήκες.

Υπάρχουν διάφοροι αλγόριθμοι για τον διαμοιρασμό των εργασιών ανάμεσα στα nodes ενός cluster, οι οποίοι υλοποιούνται από μία ξεχωριστή δομή εξισορρόπησης που λέγεται **load balancer**. Η λογική πίσω από το load balancing βασίζεται στην αναγκαιότητα συνεχούς ανταπόκρισης του συστήματος του broker, και την αποφυγή υπερφόρτωσης οποιουδήποτε μέρους αυτού.

Δύο είναι οι συνήθεις τύποι load balancing, load balancing επιπέδου 4 (επίπεδο μεταφοράς) και επιπέδου 7 (επίπεδο εφαρμογής).

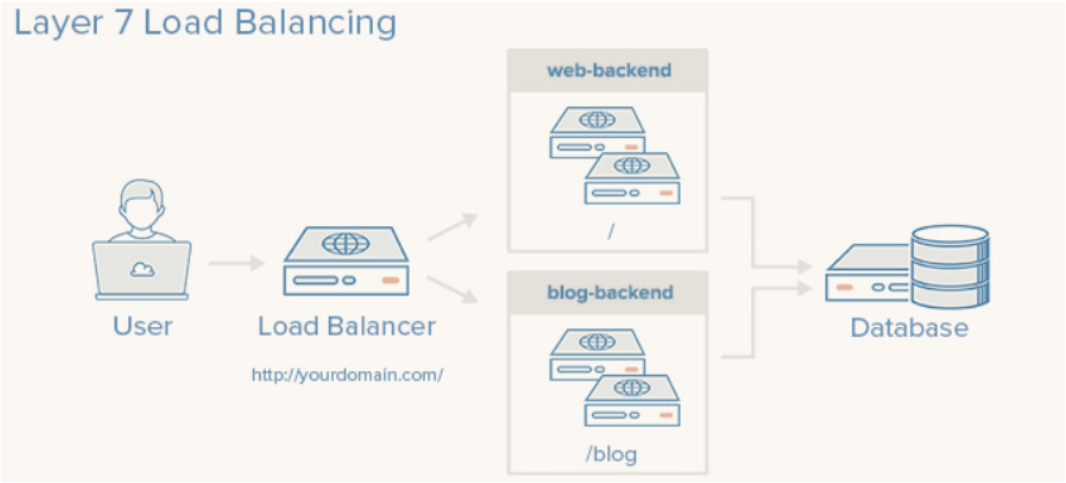
Στο επίπεδο μεταφοράς, η κίνηση προωθείται σε διεύθυνση και θύρα που έχει προκαθοριστεί, έτσι ώστε ο client να στέλνει πληροφορίες στον load balancer (frontent) και ο load balancer προωθεί το αίτημα του χρήστη

στο διαδικτυακό υπόβαθρο (web backend) που έχει οριστεί. Όλα τα nodes του web-backend συνδέονται στην ίδια βάση δεδομένων.




*Εικόνα 3.4: Load Balancing επιπέδου 4*

Στο επίπεδο εφαρμογής, ο load balancer προωθεί το αίτημα του client σε συγκεκριμένο, ανάλογα με το περιεχόμενο του αιτήματος, web-backend. Ο τρόπος αυτός είναι πιο περίπλοκος αλλά δίνει τη δυνατότητα διαφορετικών εφαρμογών server πίσω από την ίδια διεύθυνση και θύρα.




*Εικόνα 3.5: Load Balancing επιπέδου 7*

Οι πιο συνήθεις αλγόριθμοι load balancing σε δομές που χρησιμοποιούν clustering είναι:

- round-robin, όπου ο κάθε node του cluster χρησιμοποιείται εναλλάξ και ανάλογα με το βάρος που του έχει ανατεθεί (βαθμός προτεραιότητας).

Είναι ο πιο συχνά χρησιμοποιούμενος αλγόριθμος, καθώς προσφέρει μια ομαλή και δίκαιη κατανομή φόρτου στο δίκτυο.

- leastconn, δηλαδή προώθηση των νέων συνδέσεων στο node με τις λιγότερες ενεργές συνδέσεις.
- static-rr, παρόμοιας λογικής με τον round-robin, με μοναδική διαφορά ότι είναι στατικός αλγόριθμος, δηλαδή τα βάρη του δεν μπορούν να αλλάξουν οποιαδήποτε στιγμή κατά τη λειτουργία του cluster.
- source, όπου η IP διεύθυνση του client διαιρείται με το πλήθος των ενεργών nodes για να αποφασιστεί ποιο από αυτά θα αναλάβει τη σύνδεση. Με αυτόν τον αλγόριθμο διασφαλίζεται ότι κάθε client θα συνδέεται πάντα με τον node που συνδέθηκε για πρώτη φορά, εκτός και αν κάποιο node βγει εκτός δικτύου, οπότε και θα αλλάξει η δρομολόγηση των clients.

Στην υλοποίηση του clustered server που πραγματοποιήθηκε για την εκτέλεση του δεύτερου μέρους των πειραμάτων της διπλωματικής εργασίας, ως load balancer επιλέχθηκε ο HAProxy Load Balancer, ανοιχτού λογισμικού, ο οποίος προτείνεται σε συνδυασμό με το λογισμικό του EMQX Broker, και προσφέρει μεγάλη διαθεσιμότητα και απόδοση σε TCP/HTTP φορτία.

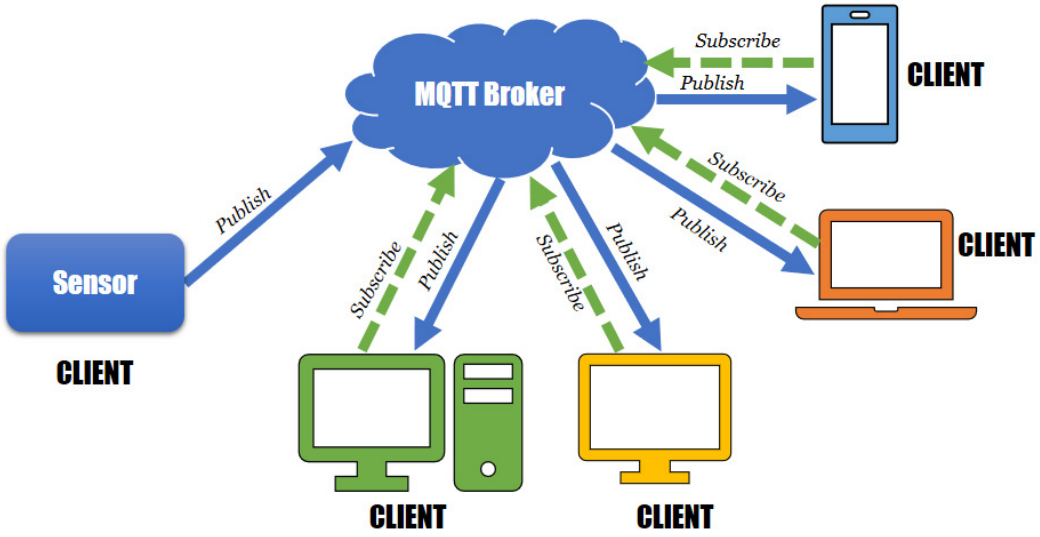
### *3.2.2 Υλοποίηση εξυπηρετητή*

Η πειραματική διαδικασία για την διερεύνηση της ανταποκρισιμότητας ενός δικτύου IoT σε συνθήκες μεγάλου δικτυακού φόρτου έγινε σε δύο επίπεδα: πρώτα δοκιμάστηκε μια απλή αρχιτεκτονική MQTT Broker, με χρήση του λογισμικού Mosquitto της Eclipse, και στη συνέχεια τα ίδια πειράματα εκτελέστηκαν στο ίδιο δίκτυο, αλλάζοντας τον απλό broker με μια δομή clustering, με δύο nodes, χρησιμοποιώντας το λογισμικό EMQX. Παρακάτω θα αναλυθεί η κάθε αρχιτεκτονική εξυπηρετητή ξεχωριστά. Τα εικονικά μηχανήματα που χρησιμοποιήθηκαν για να στηθεί το δίκτυο ήταν εξοπλισμένα με λειτουργικό σύστημα Ubuntu 18.04 LTS.

### *α. Non clustered implementation*

Για την υλοποίηση του non clustered σεναρίου χρησιμοποιήθηκε το λογισμικό Mosquitto. Ο Mosquitto MQTT Broker είναι μια υλοποίηση MQTT broker ανοιχτού λογισμικού του ιδρύματος Eclipse. Υλοποιεί τα πρωτόκολλα MQTT 5.0, 3.1.1, και 3.1, είναι αρκετά ελαφρύς από πλευράς κατανάλωσης πόρων και είναι ιδανικός για εφαρμογή σε μεγάλο εύρος συσκευών. Ακολουθεί το μοντέλο publish/subscribe και διαθέτει βιβλιοθήκες σε διάφορες προγραμματιστικές γλώσσες για την υλοποίηση MQTT clients, στα πλαίσια του Eclipse Paho Project.

Για την εργασία αυτή, το δίκτυο που υλοποιήθηκε αποτελείται από τους clients οι οποίοι συνδέονται στον broker και στέλνουν/λαμβάνουν δεδομένα. Μια απλή απεικόνιση αυτής της αρχιτεκτονικής φαίνεται παρακάτω.




*Εικόνα 3.6: Αναπαράσταση ενεργών συνδέσεων clients με τον Mosquitto Broker*

Η εγκατάσταση του broker έγινε μέσω του terminal των Ubuntu και σύμφωνα με τις οδηγίες του ιστότοπου της Eclipse. Δεν χρειάστηκε κάποια ιδιαίτερη παραμετροποίηση στο αρχείο διαμόρφωσης του Mosquitto (το οποίο βρίσκεται στην τοποθεσία /etc/mosquitto/mosquitto.conf). Οι συνδέσεις των clients γίνονται με εκείνους να απευθύνονται στην διεύθυνση IP του broker, και συγκεκριμένα στην θύρα 1883 που έχει προκαθοριστεί να είναι ανοιχτή και να χρησιμοποιείται από τον broker. Το

VM του broker διαθέτει quadcore επεξεργαστή, ισχύος 2.095 GHz με 8Gb RAM.

Η βασική υλοποίηση των πειραμάτων έγινε χωρίς κάποια ρύθμιση ασφαλείας για τις συνδέσεις με τον broker, για απλούστευση της διαδικασίας σύνδεσης μιας και εκτιμάται ότι η προσθήκη δημόσιου πιστοποιητικού θα αύξανε αρκετά τον χρόνο δημιουργίας και εγκαθίδρυσης νέων συνδέσεων.

Η μέτρηση των αποδόσεων του Mosquitto έγινε με την χρήση προγράμματος logger, που φτιάχτηκε με τέτοιο τρόπο ώστε να παρακολουθεί τα systopics του broker και να λαμβάνει πληροφορίες για τις ενεργές συνδέσεις, τις συνδέσεις που διακόπηκαν, πλήθος απεσταλμένων μηνυμάτων και χαμένων μηνυμάτων στη μονάδα του χρόνου και σε σχέση με το αυξανόμενο πλήθος συνδέσεων. Επίσης, ο logger παρακολουθούσε μέσω του προγράμματος ps των Linux την κατανάλωση RAM και CPU του Mosquitto στο μηχάνημα. Συγκεκριμένα, ο Mosquitto με την πρώτη ενεργοποίησή του ξεκινάει να τρέχει στο προσκήνιο του VM μόνιμα (μέχρι την χειροκίνητη παρέμβαση του διαχειριστή) ως μοναδική διεργασία (unique process) με δικό της αναγνωριστικό PID (Process Identification), γεγονός που κάνει την παρακολούθησή του μέσω του ps όχι μόνο δυνατή, αλλά και αρκετά ακριβή, με την εντολή:

**ps -aux**

και στη συνέχεια την απομόνωση των δεδομένων περί CPU usage και Memory usage από την απάντηση της εντολής για καταγραφή τους.

### *β. Clustered implementation*

Το δεύτερο μέρος της πειραματικής διαδικασίας αποτελεί την εκτέλεση των ίδιων πειραμάτων σε σύστημα clustered broker, για την παρατήρηση διαφορών στην απόδοση και την ανταποκρισιμότητα του δικτύου.

Η υλοποίηση του clustered broker έγινε με χρήση του λογισμικού EMQX Broker, ενός από τα πιο δημοφιλή ανοιχτά λογισμικά broker που βρίσκονται διαθέσιμα στο διαδίκτυο. Η εγκατάσταση έγινε σύμφωνα με τις επίσημες οδηγίες του broker, σε δύο εικονικά μηχανήματα που βρίσκονταν στο ίδιο δίκτυο. Τα μηχανήματα διέθεταν το ίδιο λειτουργικό Ubuntu 18.04

LTS και η μόνη διαφορά τους ήταν οι διαθέσιμοι πόροι του καθενός, με το ένα από τα δύο να έχει διπλάσια RAM και πλήθος πυρήνων από το άλλο:

| **Node 172.16.8.203** | **Node 172.16.8.215** |
|---|---|
| Quadcore, 2.095 GHz<br>RAM 8Gb<br>no swap | Dualcore, 2.095 GHz<br>RAM 8Gb<br>no swap |

*Εικόνα 3.7: Στοιχεία αρχιτεκτονικής των nodes του EMQX Cluster που χρησιμοποιήθηκε*

Για την δημιουργία cluster είναι απαραίτητη η παραμετροποίηση των μηχανημάτων μέσω των αρχείων διαμόρφωσής τους (στην τοποθεσία **/etc /emqx/emqx.conf** κάθε μηχανήματος). Συγκεκριμένα, είναι απαραίτητη η αλλαγή του ονόματος κάθε μηχανήματος, που είναι προκαθορισμένο ως emqx@127.0.0.1 (δηλαδή την loopback διεύθυνση IP, που αναφέρεται στο ίδιο το μηχάνημα), σε μια χαρακτηριστική διεύθυνση ορατή μέσα στο δίκτυο. Έτσι λοιπόν κάθε node ρυθμίστηκε ως εξής:

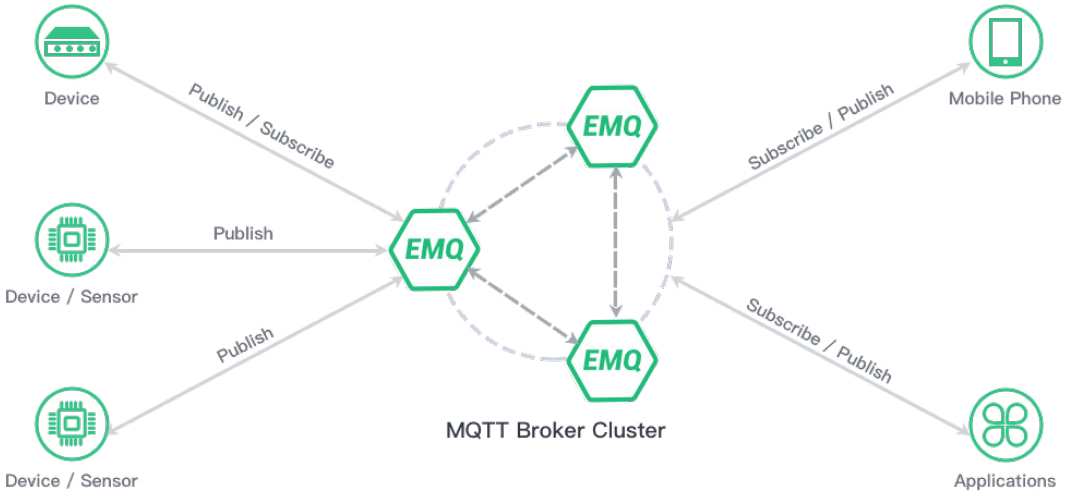
```
## Value: <name>@<host>
##
## Default: emqx@127.0.0.1
node.name = emqx1@172.16.8.203
```

*Εικόνα 3.8: Ονοματοδοσία κόμβου (node) του cluster στο αρχείο ρυθμίσεων /etc/emqx/emqx.conf για το node 172.16.8.203*

Αντίστοιχα ρυθμίστηκε και το node 172.16.8.215. Στη συνέχεια, για τη σύνδεση μεταξύ των nodes και τη δημιουργία του cluster, ακολουθήθηκε η εξής διαδικασία στο node 172.16.8.203:

**$ ./bin/emqx_ctl cluster join emqx2@172.16.8.215**

Με την επιτυχή εκτέλεση αυτής της εντολής, τα δύο nodes συνδέονται και η τοπολογία του δικτύου πλέον είναι αυτή που παρουσιάζεται στην Εικόνα 3.9 παρακάτω.



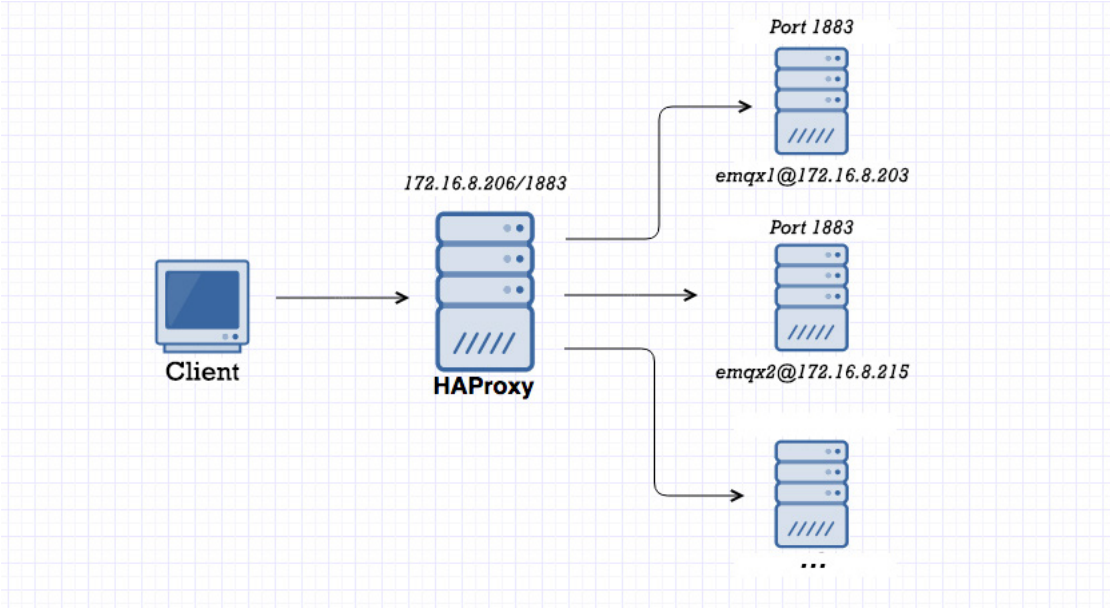

*Εικόνα 3.9: Τοπολογία δικτύου με clustered δομή εξυπηρετητή*

Επιπλέον στήθηκε σε τρίτο VM (*Quadcore, 2.095GHZ, RAM 8Gb, no swap, Ubuntu LTS 18.04*) μονάδα διαχείρισης του cluster, με χρήση του λογισμικού HAProxy Load Balancer. Ο load balancer ρυθμίστηκε σύμφωνα με τον αλγόριθμο roundrobin να διανέμει τους εισερχόμενους clients στα δύο nodes δίνοντας προτεραιότητα στο emqx1@172.16.8.203, μιας και πρόκειται για το πιο δυνατό από τα δύο. Στο αρχείο διαμόρφωσής του, το οποίο παρατίθεται και στο Κεφάλαιο 7, ορίστηκε η εξωτερική όψη του δικτύου (frontend) και η εσωτερική (backend), με τον load balancer να αποτελεί την πύλη σύνδεσης των clients και την εν συνεχεία δρομολόγησή τους στα δύο nodes.

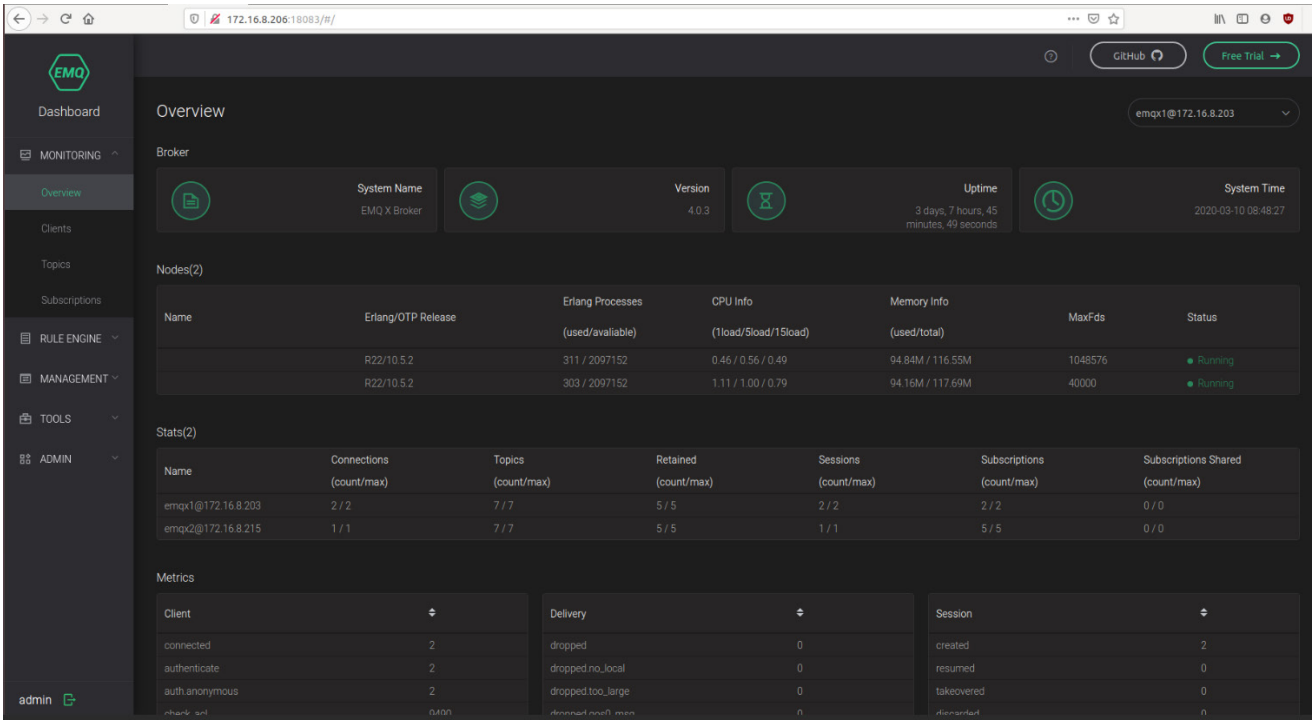
Εν τέλει η τοπολογία του δικτύου μέχρι και τον εξυπηρετητή φαίνεται στην εικόνα 3.10.

Ο EMQX Broker διαθέτει διαδικτυακή πλατφόρμα (web dashboard) παρουσίασης των δεδομένων λειτουργίας του, στην οποία αποκτάται πρόσβαση από τον φυλλομετρητή στην διεύθυνση IP του κάθε node και στη θύρα 18083, με όνομα χρήστη 'admin' και κωδικό 'public'. Μετά και την εγκατάσταση του load balancer, το dashboard στήθηκε στην IP του load balancer, δηλαδή στην διεύθυνση:

**http://172.16.8.206:18083/#/**




*Εικόνα 3.10: Τοπολογία δικτύου IoT, με Cluster και Load Balancer*



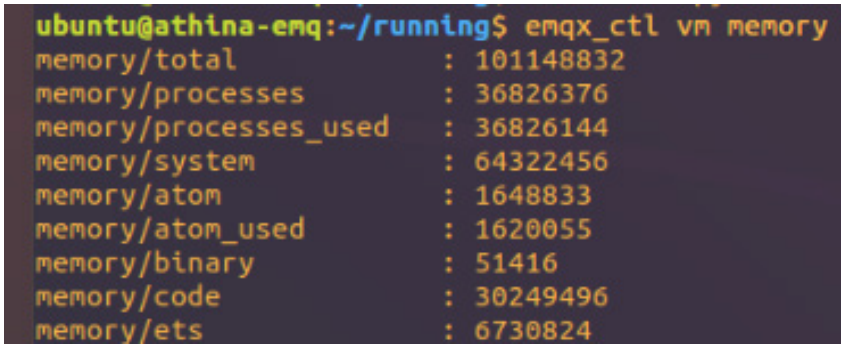

*Εικόνα 3.11: Web dashboard του EMQX Broker, στη διεύθυνση http://172.16.8.206:18083/#/*

Για την παραπάνω αρχιτεκτονική δημιουργήθηκαν και πάλι προγράμματα logger για την λήψη των απαραίτητων δεδομένων λειτουργίας του συστήματος. Και αυτή τη φορά χρησιμοποιήθηκαν τα systopics του EMQX, για τα οποία χρειάστηκε να δημιουργηθεί άδεια από το dashboard του broker για ελεύθερη πρόσβαση και αποστολή αιτημάτων μέσω curl requests (cURL: Client URL, βιβλιοθήκη libcurl), δηλαδή μέσω ενός λογισμικού που επιτρέπει τη μεταφορά δεδομένων με τη χρήση πολλών ιντερνετικών πρωτοκόλλων. Τα curl requests, αφού η πρόσβασή τους γίνεται αποδεκτή από τον broker (μέσω δημιουργίας ειδικής άδειας από το dashboard του EMQX, περισσότερα στην Ενότητα 7.2.3), στέλνουν πίσω απάντηση της μορφής:

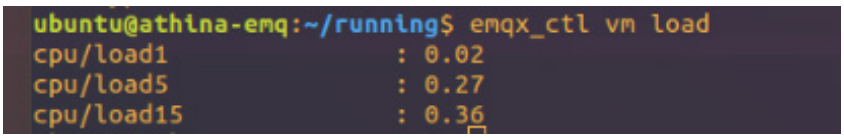

{"code":0,"data":[{"node":"emqx1@172.16.8.203","stats":{"subscriptions.shared.max":0,"subscriptions.max":10,"subscribers.max":10,"resources.max":0,"topics.count":8,"channels.count":6,"subscriptions.count":10,"suboptions.max":10,"topics.max":8,"connections.max":6,"actions.count":5,"retained.count":3,"rules.count":0,"routes.count":8,"subscriptions.shared.count":0,"suboptions.count":10,"sessions.count":6,"channels.max":6,"actions.max":5,"retained.max":3,"sessions.max":6,"rules.max":0,"routes.max":8,"resources.count":0,"subscribers.count":10,"connections.count":6}}]}
{"code":0,"data":[{"node":"emqx1@172.16.8.203","metrics":{"client.connack":9,"delivery.dropped.too_large":0,"packets.puback.inuse":0,"rules.matched":0,"messages.dropped.expired":0,"messages.sent":8248,"packets.disconnect.sent":0,"bytes.sent":1447322,"packets.disconnect.received":1,"packets.pingresp.sent":54,"packets.pingreq.received":0,"packets.unsubscribe.received":0,"packets.pubcomp.missed":0,"packets.puback.missed":0,"packets.pubcomp.sent":0,"packets.pubcomp.received":1,"session.created":9,"packets.pubrec.missed":0,"client.auth.anonymous":9,"packets.connack.auth_error":0,"delivery.dropped.no_local":0,"messages.publish":4126,"packets.pubcomp.inuse":0,"actions.failure":0,"client.unsubscribe":0,"packets.pubrec.inuse":0,"client.disconnected":1,"packets.suback.sent":9,"packets.puback.sent":1,"session.takeovered":0,"delivery.dropped.expired":0,"messages.retained":3,"messages.received":4126,"packets.connect.received":9,"messages.delivered":8248,"messages.forward":0,"packets.pubrel.missed":0,"packets.publish.received":4126,"packets.connack.sent":9,"session.terminated":1,"session.resumed":0,"delivery.dropped.qos0_msg":0,"client.authenticate":9,"packets.subscribe.received":9,"packets.pubrel.received":0,"packets.pubrec.received":1,"packets.puback.received":2,"packets.sent":8323,"packets.received":4149,"bytes.received":724782,"messages.delayed":0,"messages.dropped":0,"packets.publish.dropped":0,"session.discarded":2,"packets.pubrel.sent":1,"packets.pubrec.sent":1,"packets.publish.sent":8248,"messages.dropped.no_subscribers":0,"actions.success":0,"packets.publish.error":0,"packets.unsubscribe.error":0,"client.check_acl":73,"delivery.dropped.queue_full":0,"client.subscribe":9,"delivery.dropped":0,"messages.qos2.received":1,"messages.qos1.received":1,"messages.qos0.received":4124,"packets.auth.sent":0,"client.connect":9,"messages.acked":3,"messages.qos2.sent":2,"messages.qos1.sent":2,"messages.qos0.sent":8244,"client.connected":9,"packets.auth.received":0,"packets.unsuback.sent":0,"packets.connack.error":0,"packets.publish.auth_error":0,"packets.subscribe.error":0,"packets.subscribe.auth_error":0}}]}

*Εικόνα 3.12: Μορφή απάντησης σε ερώτημα CURL από τον broker. Η απάντηση αυτή είναι σε μορφή JSON, και επεξεργάζεται ανάλογα από τον logger για απομόνωση των στοιχείων που είναι αναγκαία να καταγραφούν*

η οποία είναι σε μορφή JSON, την επεξεργάζεται ο logger και κρατούνται τα πεδία ενδιαφέροντος σχετικά με το πείραμα. Όσον αφορά την κατανάλωση πόρων CPU και RAM, επειδή ο broker δεν αποτελεί μεμονωμένο μηχάνημα, και επειδή επιπλέον το λογισμικό του EMQX δεν υφίσταται ως μεμονωμένη διεργασία σε κάθε node, δεν ήταν δυνατή η λήψη των μετρήσεων με τον ίδιο τρόπο όπως στον Mosquitto Broker. Ωστόσο ο EMQX προσφέρει τη δυνατότητα μέσω εντολών από το terminal των node να ληφθεί η κατανάλωση πόρων κάθε 1/5/15 δευτερόλεπτα.




*Εικόνα 3.13: Ερώτηση για την μνήμη μεμονομένου node στην κονσόλα και μορφή της απάντησης που επεξεργάζεται ο logger.*




*Εικόνα 3.14: Ερώτηση για την κατανάλωση πόρων του επεξεργαστή μεμονομένου node στην κονσόλα και μορφή της απάντησης που επεξεργάζεται ο logger.*

### *3.3 Βάση δεδομένων*

Με τον όρο βάση δεδομένων γίνεται αναφορά σε οργανωμένες, διακριτές συλλογές σχετιζόμενων δεδομένων ηλεκτρονικά και ψηφιακά αποθηκευμένων, στο λογισμικό που χειρίζεται τέτοιες συλλογές και στο γνωστικό πεδίο που το μελετά. Πέρα από την εγγενή της ικανότητα να αποθηκεύει δεδομένα, η βάση δεδομένων παρέχει μέσω του σχεδιασμού και του τρόπου ιεράρχησης των δεδομένων, τα αποκαλούμενα συστήματα διαχείρισης περιεχομένου, δηλαδή τη δυνατότητα γρήγορης άντλησης και ανανέωσης των δεδομένων.

Η αναγκαιότητα της βάσης δεδομένων σε ένα δίκτυο IoT είναι αδιαμφισβήτητη, μιας και είναι το μέρος που συγκεντρώνονται όλες οι μετρήσεις και τα δεδομένα που αποστέλλουν οι συσκευές IoT και που συλλέγονται και επεξεργάζονται από τον broker. Ο broker ρυθμίζεται

έτσι ώστε να έχει ασφαλή πρόσβαση στη βάση δεδομένων και να στέλνει τις πληροφορίες που είναι αναγκαίο να καταγραφούν ανά τακτά χρονικά διαστήματα.

Η βάση δεδομένων που χρησιμοποιήθηκε στην παρούσα εργασία είναι η InfluxDB, μία ανοιχτού λογισμικού βάση δεδομένων χρονικής σειράς, που έχει αναπτυχθεί από την Influx Data. Η InfluxDB είναι γραμμένη σε γλώσσα Go και είναι μία βέλτιστη λύση για γρήγορη και διαρκώς διαθέσιμη αποθήκευση και ανάκτηση δεδομένων χρονικής σειράς. Βρίσκει εφαρμογή σε πεδία όπως παρακολούθηση και μέτρηση απόδοσης εφαρμογών, συλλογή δεδομένων αισθητήρων IoT, και ανάλυση δεδομένων πραγματικού χρόνου. Είναι εξαιρετικά αξιόπιστη, καθώς έχει τη δυνατότητα σύγχρονης αποθήκευσης χιλιάδων χρονοσειρών και άρα μπορεί να υποστηρίξει δίκτυα IoT με μεγάλο πλήθος συσκευών. Επίσης, δέχεται δεδομένα με χρήση των πρωτοκόλλων HTTP, TCP και UDP, κάνοντάς την έτσι εύκολα προσαρμόσιμη στις ανάγκες του χρήστη.

Η InfluxDB λειτουργεί χωρίς την εξάρτηση από άλλα λογισμικά και προσφέρει μια γλώσσα παρόμοιας λογικής με την SQL για δημιουργία ερωτήσεων από τον χρήστη. Κάθε σημείο απαρτίζεται από ζεύγος κλειδιών-τιμών το οποίο ονομαζέται fieldset και όταν συγκεντρώσουν ένα σύνολο τιμών τότε δημιουργείται μία σείρα η οποία ονομάζεται tagset. Στη συνέχεια οι σειρές ομαδοποιούνται από μία αναγνωριστική συμβολοσειρά η οποία αποτελεί και την μέτρηση. Οι τιμές μπορούν να αποτελούν 64-bit ακέραιους, 64-bit δείκτες κινητής αποστολής, μία συμβολοσειρά ή ένα λογικο τελεστή.

Η συλλογή των δεδομένων γίνεται με τη χρήση ενός διαμεσολαβητή μεταξύ broker και βάσης δεδομένων ο οποίος εκτελεί την ανάγνωση των δεδομένων από τον broker και την προώθησή τους στη βάση δεδομένων στην επιθυμητή προς καταγραφή μορφή. Ο διαμεσολαβητής αυτός ονομάζεται Telegraf και αποτελεί μέρος της σουίτας λογισμικού TICK Stack, μαζί με την InfluxDB και τον Chronograf, του οποίου η χρήση θα αναλυθεί παρακάτω. Το TICK Stack αποτελεί μια ολοκληρωμένη λύση ανοιχτού λογισμικού για την συλλογή, επεξεργασία, καταγραφή και οπτικοποίηση δεδομένων σειράς και συμβάντων που διαχειρίζεται ο broker σε ένα δίκτυο IoT.

Αναφέρεται, επιπλέον, πως η InfluxDB εγκαταστάθηκε σε ανεξάρτητο από τον broker VM, το οποίο είχε επεξεργαστή Dualcore, 2.095GHz, και

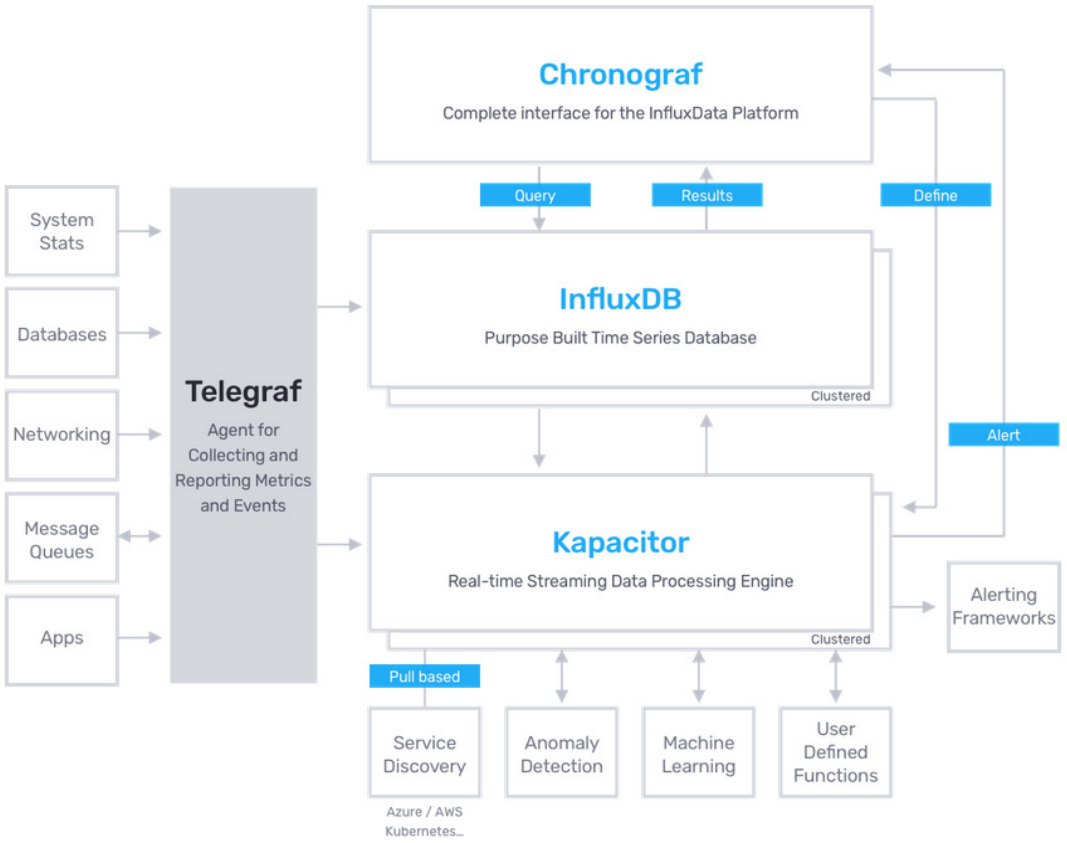
μνήμη RAM 8Gb, με 16Gb επιπλέον swap. Το VM είχε λειτουργικό Ubuntu LTS 18.04.

### *3.4 Οπτικοποίηση βάσης δεδομένων*

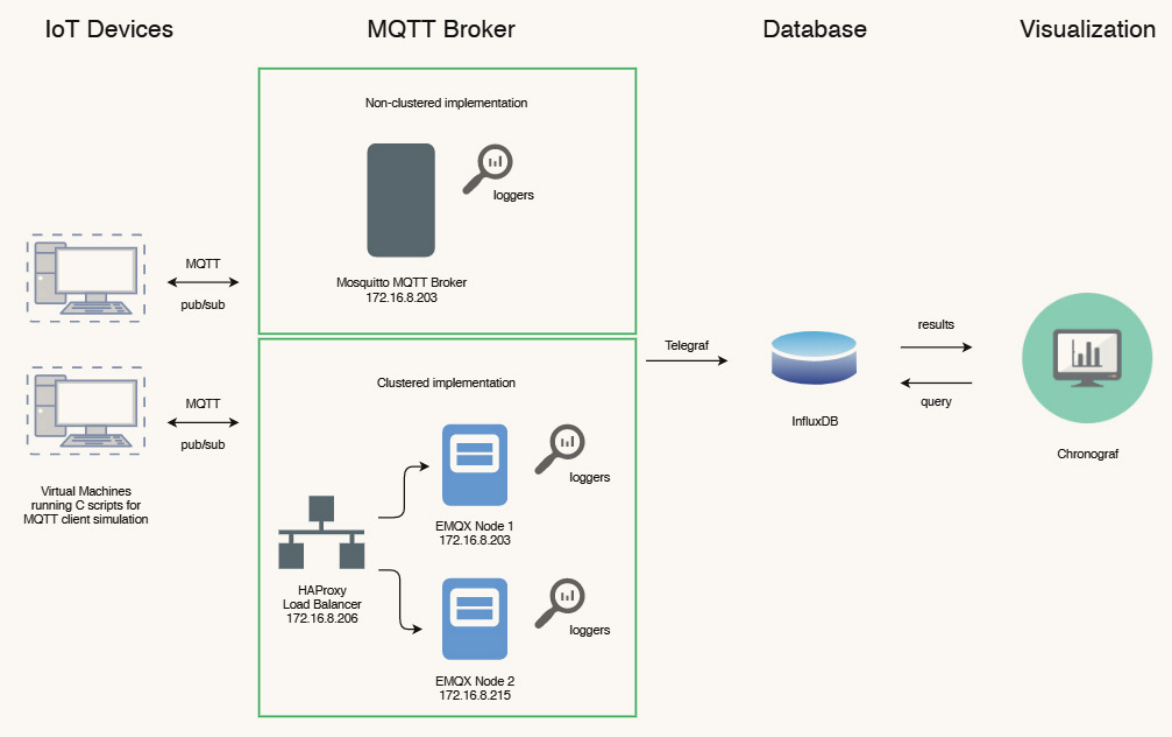
Στο τέλος της αλυσίδας ενός δικτύου IoT τοποθετείται η μονάδα οπτικοποίησης και παρακολούθησης των δεδομένων που έχουν παραληφθεί από τον broker και έχουν καταγραφεί στη βάση δεδομένων. Η βαθμίδα αυτή είναι απαραίτητη για την ευκολότερη ανάγνωση των συμβάντων της λειτουργίας του δικτύου από τον διαχειριστή του, αλλά και για την επεξεργασία και εξαγωγή αποτελεσμάτων και συμπερασμάτων σχετικά με την κατασκευή και την εύρυθμη λειτουργία των μελών της.

Η διαδικασία αυτή μπορεί να γίνει από ένα εργαλείο του TICK Stack, όπως αναφέρθηκε παραπάνω, το Chronograf. Ο Chronograf αποτελεί την διαδικτυακή πλατφόρμα της βάσης δεδομένων InfluxDB, μέσω της οποίας πέρα από την παρουσίαση των περιεχομένων που συλλέγονται στη βάση δεδομένων μπορεί να γίνει και η διαχείριση αυτής, η οπτικοποίηση του δικτύου, των μελών του και των αδειών που έχουν παραχωρηθεί σε αυτά για πρόσβαση στα δεδομένα. Διαθέτει πλήθος ερωτημάτων (queries) για την γραφική απεικόνιση των δειγμάτων των μετρήσεων, και ευρύ περιθώριο στην παραμετροποίηση αυτών, έτσι ώστε ο χρήστης να έχει πολλές επιλογές για την βέλτιστη απόδοση συμπερασμάτων πάνω στα δεδομένα κίνησης του δικτύου.

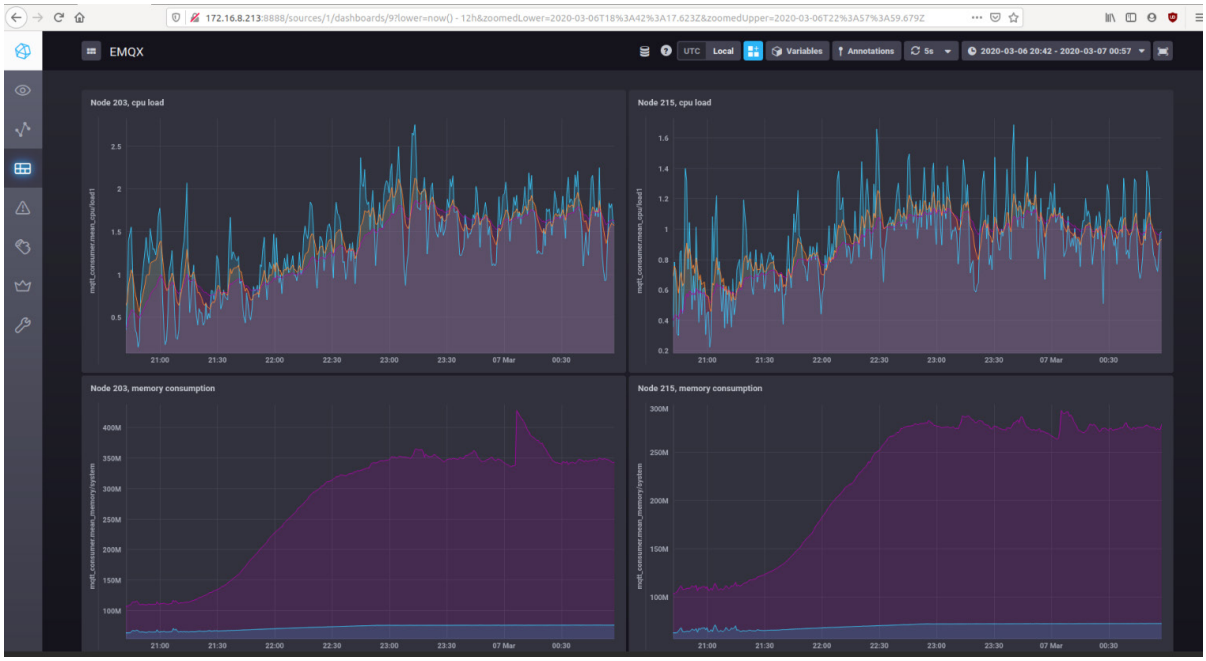
Παρακάτω απεικονίζεται η σύνδεση μεταξύ του Broker, του Telegraf, της InfluxDB του Chronograf, και κατ' επέκταση η ροή της πληροφορίας από την συλλογή της μέχρι την παρουσίασή της.




*Εικόνα 3.14: Συνολική αρχιτεκτονική του TICK Stack (στην υλοποίηση του δικτύου που μελετάται δεν χρησιμοποιείται ο Kapacitor)*




*Εικόνα 3.15: Συγκεντρωτικό σχεδιάγραμμα της αρχιτεκτονικής του δικτύου*




*Εικόνα 3.16: Στιγμιότυπο από την λειτουργία του Chronograf, με αναπαράσταση δεδομένων από την InfluxDB*

# Κεφάλαιο 4
# Πειραματική Διαδικασία

Στο κεφάλαιο αυτό θα εξηγηθεί η πειραματική διαδικασία που ακολουθήθηκε και θα παρουσιαστούν οι μετρήσεις που λήφθηκαν με σκοπό την εξαγωγή των αποτελεσμάτων της έρευνας πάνω στο δίκτυο IoT που περιγράφηκε στα προηγούμενα κεφάλαια.

## *4.1 Σχεδιασμός πειραμάτων*

Για την διερεύνηση των ορίων και της απόδοσης ενός δικτύου IoT υπό συνθήκες μεγάλου δικτυακού φόρτου, πρώτη προτεραιότητα είναι η προσομοίωση των clients του δικτύου, η οποία, όπως αναλυθηκε στο Κεφάλαιο 3, έγινε με τη χρήση εικονικών μηχανημάτων του Cloud του Leonardo Lab της COSMOTE. Τα χρησιμοποιούμενα VMs είχαν τη δυνατότητα αναπαράστασης περίπου 4000 χιλιάδων clients.

Στη συνέχεια είναι αναγκαία η εύρεση της συνθήκης στην οποία λειτουργούν οι clients, έτσι ώστε να παράγουν μεγάλο φόρτο στον broker του δικτύου, με άλλα λόγια η εύρεση του κατάλληλου σεναρίου προς εφαρμογή. Η λειτουργία του μοντέλου publish/subscribe ορίζει ότι όταν ένας client στέλνει ένα μήνυμα σε ένα topic, τότε ο broker προωθεί αυτό το μήνυμα σε όλους τους subscribers του topic αυτού, κάτι που συμβαίνει προφανώς σε κάθε μεμονωμένο publish message που στέλνεται σε αυτόν. Είναι εύκολο να αντιληφθούμε ότι σε ένα σενάριο που υπάρχουν αρκετά published messages σε ένα topic με μεγάλο αριθμό subscribers, θα πρέπει το καθένα από αυτά να προωθείται σε όλους τους subscribers του topic.

Έτσι λοιπόν τα δύο σενάρια που σύμφωνα με την παραπάνω λογική δημιουργούν εκθετικά αυξανόμενη ανταλλαγή μηνυμάτων ανάμεσα σε χιλιάδες συσκευές του δικτύου είναι τα παρακάτω σενάρια:

- Μοναδικός publisher/πλήθος subscribers (unique publisher/multiple subscribers)

- Πλήθος publishers/πλήθος subscribers (multiple publishers/multiple subscribers)

Μεταξύ των δύο σεναρίων είναι προφανές ότι στο ίδιο χρονικό διάστημα, το σενάριο mulitple publishers/multiple subscribers είναι αυτό που επιφέρει τον μεγαλύτερο φόρτο στο δίκτυο. Αποφασίστηκε λοιπόν να είναι το σενάριο το οποίο θα χρησιμοποιηθεί για τις προσομοιώσεις, δεδομένου ότι εξετάζοντας τις μεταβολές στις παραμέτρους του δικτύου ενώ λαμβάνει χώρα το "worst case scenario" τα συμπεράσματα που θα προκύψουν αφορούν συνολικά τις δυνατότητες διαχείρισης του broker.

Ο κώδικας υλοποίησης του σεναρίου multiple publishers/multiple subscribers, ο οποίος, όπως αναφέρθηκε στο κεφάλαιο 3, είναι γραμμένος σε BASH, Python και C, παρατίθεται ολόκληρος στο Κεφάλαιο 7.

Σύμφωνα με τη δομή της υλοποίησης των προγραμμάτων, το C script είναι αυτό που διαχειρίζεται την δημιουργία ενός client object, τη σύνδεσή του με τον broker, τη συνδρομή του σε ένα topic και την αποστολή μηνυμάτων σε αυτό. Το BASH script είναι υπεύθυνο για την παράλληλη εκτέλεση των C scripts στο επιθυμητό πλήθος φορών, και το Python script στέλνει μηνύματα στον broker για το πλήθος των προσπαθειών ενεργών συνδέσεων. Οι βασικές παράμετροι αυτού του σεναρίου αυτού, όπως φαινεται από τον κώδικα, είναι δύο:

- το **spawning delay**, που είναι ο χρόνος ανάμεσα στη δημιουργία καινούριων clients, και
- το **publish delay**, δηλαδή ο χρόνος μεταξύ της αποστολής δύο διαδοχικών μηνυμάτων/publish σε topic από έναν client.

Οι δύο αυτές παράμετροι μεταβάλλονται σε κάθε πείραμα για να παρατηρηθούν οι αντίστοιχες διαφορές που επιφέρουν στο δίκτυο.

Οι υπόλοιπες παράμετροι του σεναρίου συνοψίζονται στο μέγεθος του μηνύματος publish που είναι αμελητέο (payload 8 χαρακτήρων) και επιπέδου QoS 1. Το μέγεθος του μηνύματος δεν κρίθηκε απαραίτητο να μεταβάλλεται στις προσομοιώσεις, μιας και η σύνδεση μεγέθους μηνύματος και κατανάλωσης bandwidth σε κάθε σύνδεση client με τον broker είναι προφανής. Προτιμήθηκε να κριθεί η ανταπόκριση του broker ελέγχοντας την ανοχή σε βομβαρδισμό μηνυμάτων προς διαχείριση και επαναπροώθηση παρά την δυνατότητα αυτή με μεγάλο μέγεθος μηνύματος.

## *4.2 Εκτέλεση πειραμάτων*

### *4.2.1 Non clustered implementation*

Στην non clustered δομή του δικτύου με τον Mosquitto broker εκτελέστηκαν 15 συνολικά προσομοιώσεις, τα χαρακτηριστικά των οποίων παρουσιάζονται συνοπτικά στον παρακάτω πίνακα:

| **Mosquitto Broker – Εκτέλεση Προσομοιώσεων** | | | |
|---|---|---|---|
| **Προσομοίωση/ Διάγραμμα** | **Publish Delay (in ms)** | **Spawning Delay (in s)** | **Number of VMs** |
| 1 / M1 | 50 | 1 | 1 |
| 2 / M2 | 100 | 1 | 1 |
| 3 / M3 | 500 | 1 | 1 |
| 4 / M4 | 3000 | 1 | 1 |
| 5 / M5 | 50 | 0.5 | 1 |
| 6 / M6 | 100 | 0.5 | 1 |
| 7 / M7 | 500 | 0.5 | 1 |
| 8 / M8 | 3000 | 0.5 | 1 |
| 9 / M9 | 50 | 3 | 1 |
| 10 / M10 | 100 | 3 | 1 |
| 11 / M11 | 500 | 3 | 1 |
| 12 / M12 | 3000 | 3 | 1 |
| 13 / M13 | 100 | 1 | 2 |
| 14 / M14 | 50 | 3 | 2 |
| 15 / M15 | 3000 | 0.5 | 2 |

*Πίνακας 1: Πίνακας παραμέτρων προσομοιώσεων για τον Mosquitto Broker*

Για να γίνει πιο εύκολη η ανάλυση των πειραματικών αποτελεσμάτων, θα γίνει παρουσίαση αυτών ομαδοποιημένων βάσει της μεταβλητής Spawning Delay, η οποία παίρνει τις τιμές 0.5, 1, 3 και μετράται σε δευτερόλεπτα.

**Σε κάθε ζεύγος γραφικών παραστάσεων γίνεται απεικόνιση**:

- (πάνω): Γραφική παράσταση των καταναλωθέντων πόρων CPU (μωβ χρώμα) και MEM (κόκκινο χρώμα) συναρτήσει του χρόνου.
- (κάτω): Γραφική παράσταση των clients που είναι ενεργά συνδεδεμένοι με τον Mosquitto Broker.

έτσι ώστε να μπορούν να γίνουν παρατηρήσεις στην απόδοση του Mosquitto

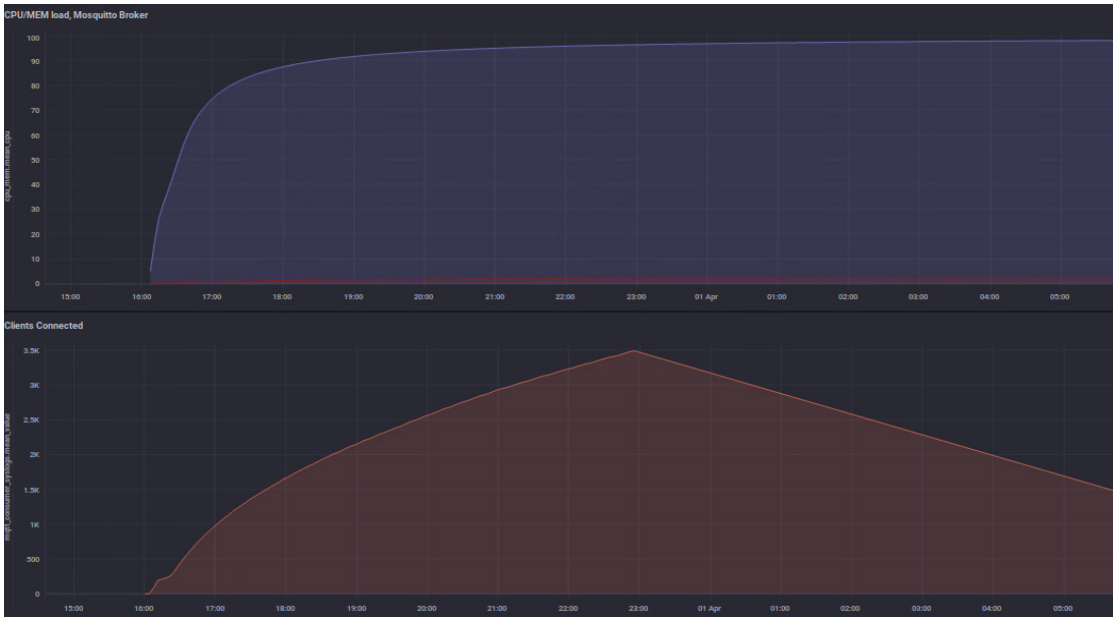
Broker συναρτήσει και του χρόνου λειτουργίας, αλλά κυρίως των ενεργών και ολοένα αυξανόμενων συνδέσεων clients.

Στα παρακάτω διαγράμματα που έχουν ληφθεί από την InfluxDB, παρατηρείται στις γραφικές παραστάσεις των συνδεδεμένων clients αρκετές φορές ένα σημείο καμπής και στη συνέχεια γραμμική μείωση των clients. Αυτό ωστόσο δεν αποτελεί πραγματική καταγραφή της εξέλιξης των συνδεδεμένων clients, αλλά προκύπτει λόγω του γεγονότος πως το σημείο καμπής είναι το σημείο που ο broker παύει να ανταποκρίνεται στην δημιουργία νέων συνδέσεων, φαινόμενο που θα εξηγηθεί παρακάτω. Έτσι διακόπτεται η καταγραφή των ενεργών συνδέσεων, που είναι στοιχείο που λαμβάνεται από τα systopics του broker, και η InfluxDB ενώνει το σημείο αυτό με το αμέσως επόμενο που θα αντιστοιχεί σε επόμενη μέτρηση (το οποίο προκύπτει μετά από επανεκκίνηση του broker και δηλαδή τερματισμό του πειράματος. Ο λόγος που η γραφική απεικόνιση των CPU/MEM δεν είναι αντίστοιχη είναι επειδή οι μετρήσεις αυτές λαμβάνονται από το VM του broker, το οποίο παρότι εξαντλούνται οι πόροι του συνεχίζει να στέλνει τα δεδομένα αυτών απευθείας στην InfluxDB.

### *α. Spawning Delay = 1s*

Οι παρακάτω γραφικές παραστάσεις απεικονίζουν τη κατανάλωση πόρων CPU/MEM στο μηχάνημα του Mosquitto Broker, για τις τιμές Publish Delay = 50/100/500/3000ms.

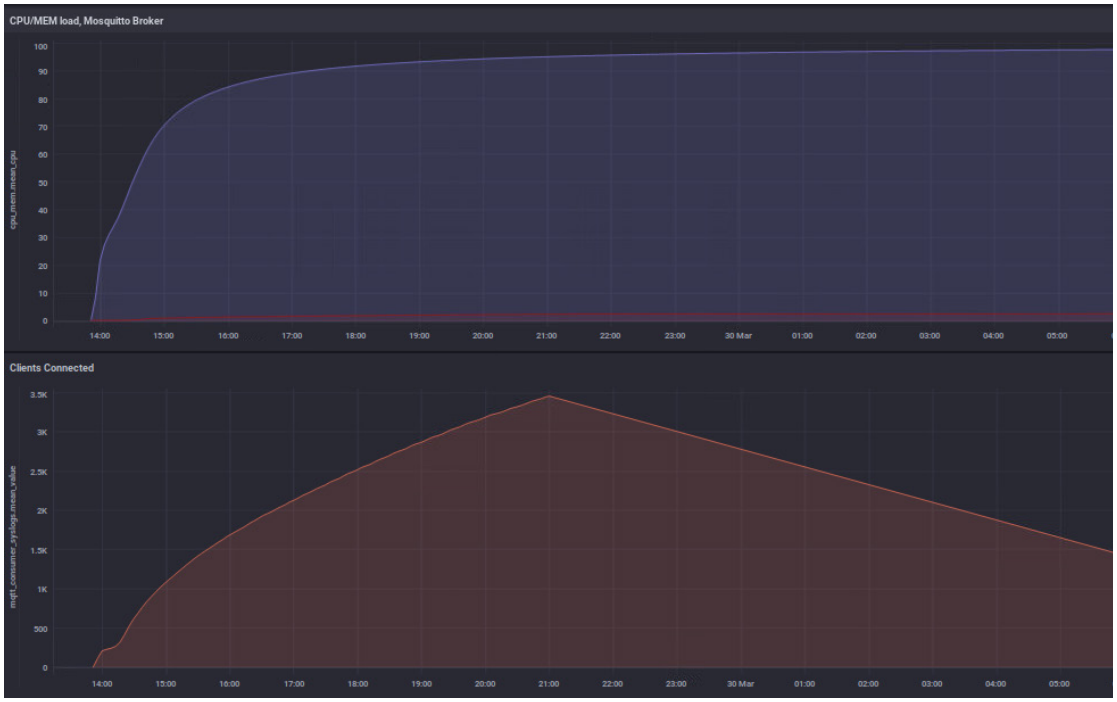
**Διάγραμμα Μ1:**
**Spawning Delay = 1s, Publish Delay = 50ms**




Παρατηρείται μέγιστος αριθμός συνδέσεων περί τους 3.5k clients, σε χρονικό διάστημα 7 ωρών. Η CPU ξεπερνάει το 90% 3 ώρες μετά την

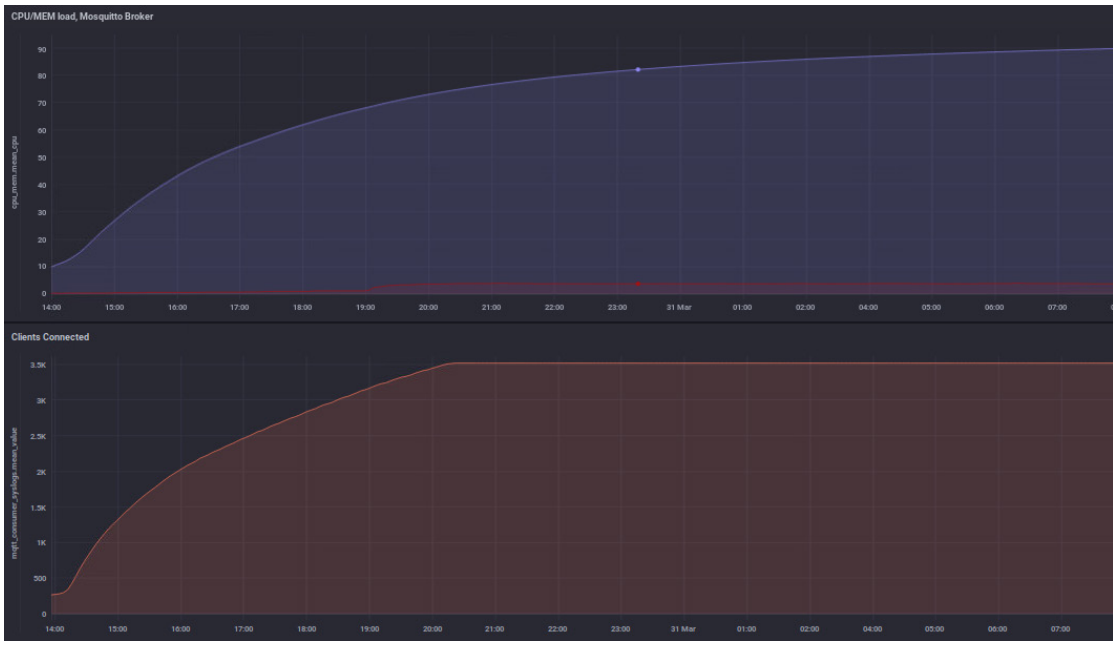
εκκίνηση του πειράματος.

**Διάγραμμα Μ2:**
**Spawning Delay = 1s, Publish Delay = 100ms**




Παρατηρείται μέγιστος αριθμός συνδέσεων περί τους 3.5k clients, σε χρονικό διάστημα 7 ωρών. Η CPU ξεπερνάει το 90% 4 ώρες μετά την έναρξη του πειράματος.

**Διάγραμμα Μ3:**
**Spawning Delay = 1s, Publish Delay = 500ms**



Παρατηρείται μέγιστος αριθμός συνδέσεων περί τους 3.5k clients, σε χρονικό διάστημα 7 ωρών. Η CPU ξεπερνάει το 90% 9 ώρες μετά την έναρξη του πειράματος.

**Διάγραμμα Μ4:**
**Spawning Delay = 1s, Publish Delay = 3000ms**


Παρατηρείται μέγιστος αριθμός συνδέσεων περί τους 3.5k clients, σε χρονικό διάστημα 6 ωρών. Η CPU δεν ξεπερνά το 20-25% καθ' όλη τη διάρκεια του πειράματος.

Στα παραπάνω διαγράμματα φαίνεται για πρώτη φορά η συμπεριφορά του Mosquitto Broker σε συνθήκες δικτυακού φόρτου. Μία πρώτη παρατήρηση που μπορεί να γίνει είναι πως είναι εμφανές ότι η εξάντληση των πόρων του broker γίνεται σε επίπεδο CPU, και δεν αφορά παρά ελάχιστα την μνήμη RAM. Η τιμή της CPU γρήγορα φτάνει μέχρι και 90%+, ενώ η μνήμη RAM κυμαίνεται στο 0-6%. Αυτό εξηγείται από το γεγονός ότι ο Mosquitto αποτελεί διεργασία ενός thread που εκτελείται από έναν μόνο πυρήνα - με άλλα λόγια διεργασία που απαιτεί την λειτουργία του πυρήνα CPU κατά κόρον.

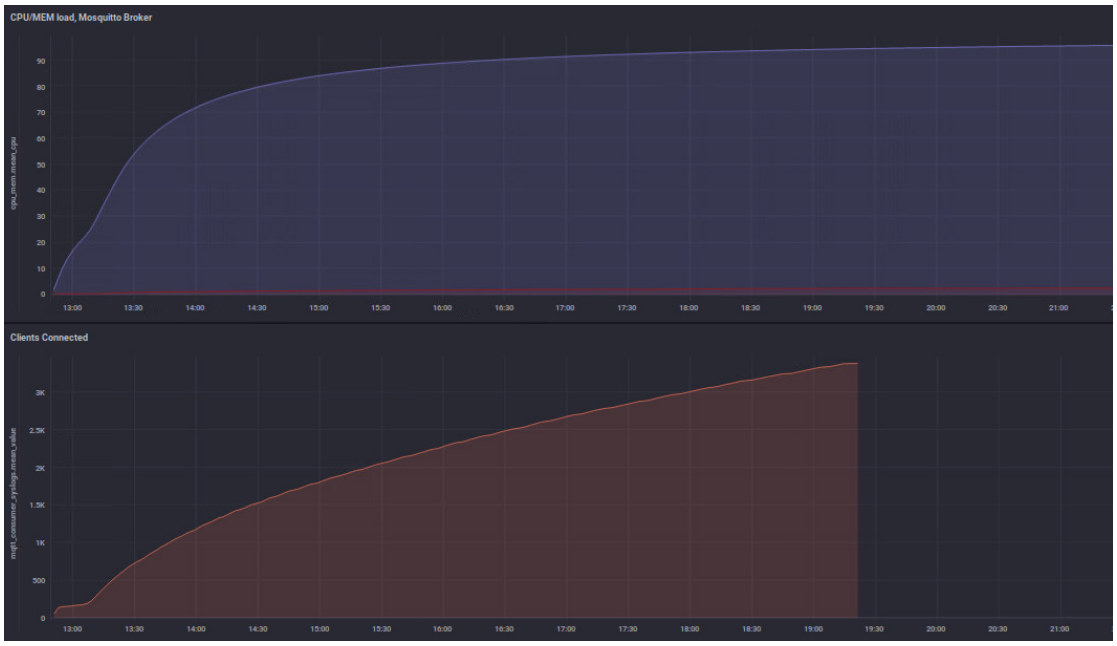
Επίσης σημειώνεται ότι για ίδιο ρυθμό σύνδεσης νέων clients - δηλαδή σταθερό Spawning Delay - όσο πιο μεγάλη καθυστέρηση υπάρχει μεταξύ των διαδοχικών publish των clients τόσο πιο σταδιακή είναι η

καταπόνηση του broker στη μονάδα του χρόνου, δηλαδή με αύξηση του Publish Delay υπάρχει καλύτερη ανταπόκριση του broker από πλευράς κατανάλωσης πόρων. Αυτό γίνεται προφανές μεταξύ των πειραμάτων M2 και M3, που από τις 3 ώρες που χρειάζεται για την καταπόνηση των πόρων του broker με Publish Delay = 500ms η αλλαγή του σε 3000ms οδηγεί σε αντίστοιχη εξάντληση της CPU σε 9 ώρες. Ο μέγιστος αριθμός ενεργών συνδέσεων δεν ξεπερνά τις 3.500 κατά προσέγγιση, και επιτυγχάνεται σε διάστημα περίπου 6-7 ωρών.

### *β. Spawning Delay = 0.5s*

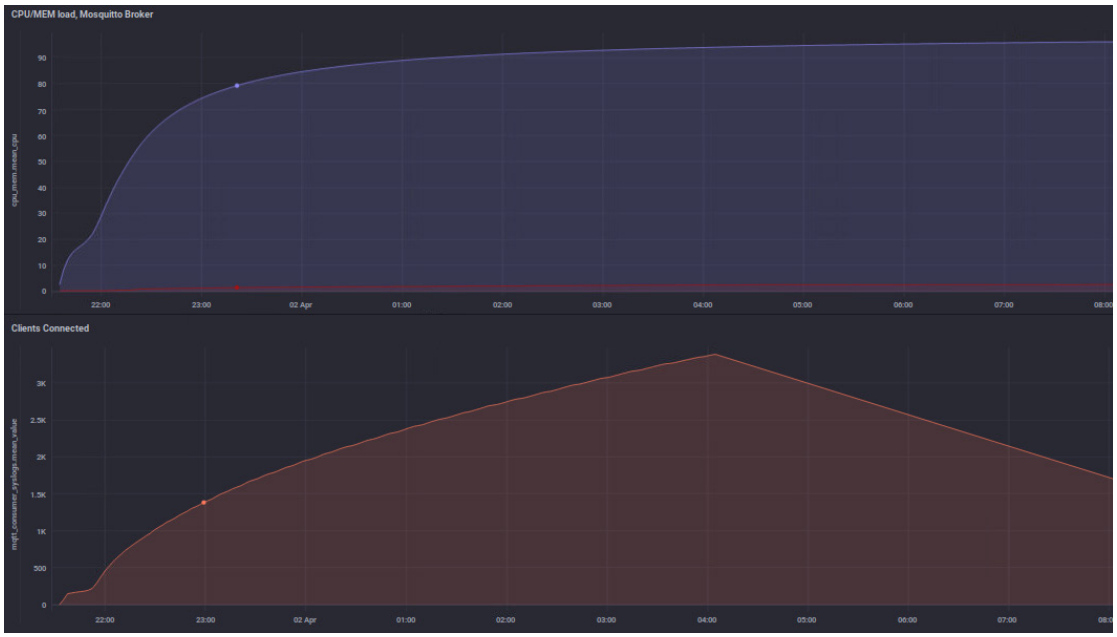
Οι παρακάτω γραφικές παραστάσεις απεικονίζουν τη κατανάλωση πόρων CPU/MEM στο μηχάνημα του Mosquitto Broker, για τις τιμές Publish Delay = 50/100/500/3000ms.

**Διάγραμμα M5:**
**Spawning Delay = 0.5s, Publish Delay = 50ms**



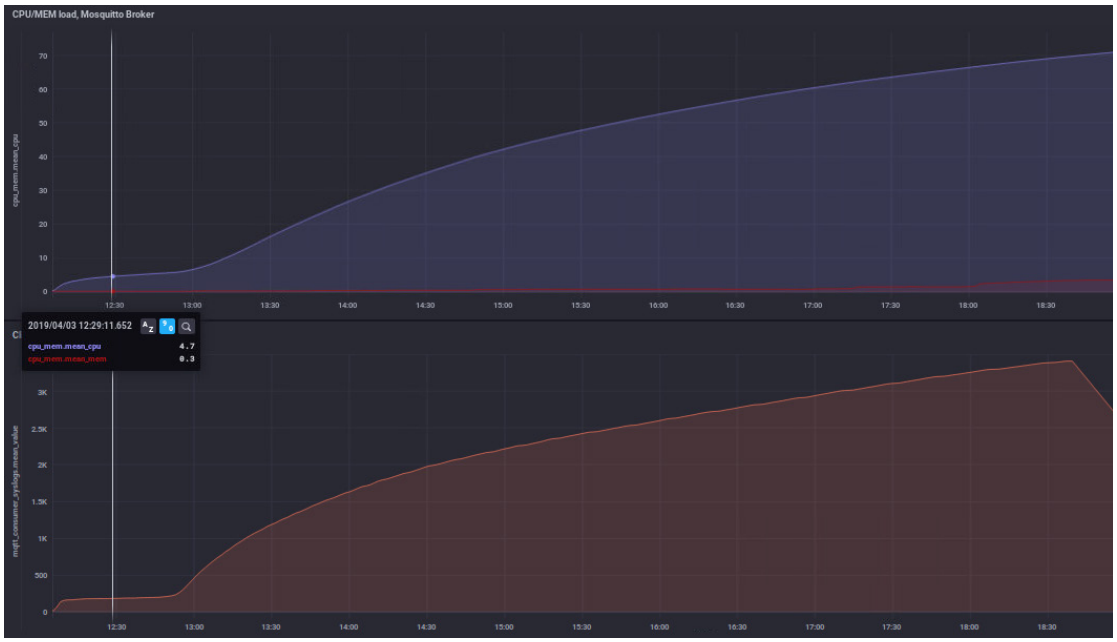

Παρατηρείται μέγιστος αριθμός συνδέσεων γύρω στις 3.5-3.8k clients. Η τιμή κατανάλωσης πόρων της CPU ξεπερνά το 90% μετά από 3 ώρες από την έναρξη του πειράματος.

**Διάγραμμα M6:**
**Spawning Delay = 0.5s, Publish Delay = 100ms**




Παρατηρείται μέγιστος αριθμός συνδέσεων γύρω στις 3.5-3.8k clients. Η τιμή κατανάλωσης πόρων της CPU ξεπερνά το 90% μετά από 5 ώρες από την έναρξη του πειράματος.

**Διάγραμμα M7:**
**Spawning Delay = 0.5s, Publish Delay = 500ms**



Παρατηρείται μέγιστος αριθμός συνδέσεων γύρω στις 3.5-3.8k clients. Η τιμή κατανάλωσης πόρων της CPU δεν ξεπερνά το 70% κατά τη διάρκεια του πειράματος.

**Διάγραμμα M8:**
**Spawning Delay = 0.5s, Publish Delay = 3000ms**


Παρατηρείται μέγιστος αριθμός συνδέσεων γύρω στις 3.5k clients. Η τιμή κατανάλωσης πόρων της CPU δεν ξεπερνά το 35% κατά τη διάρκεια του πειράματος.

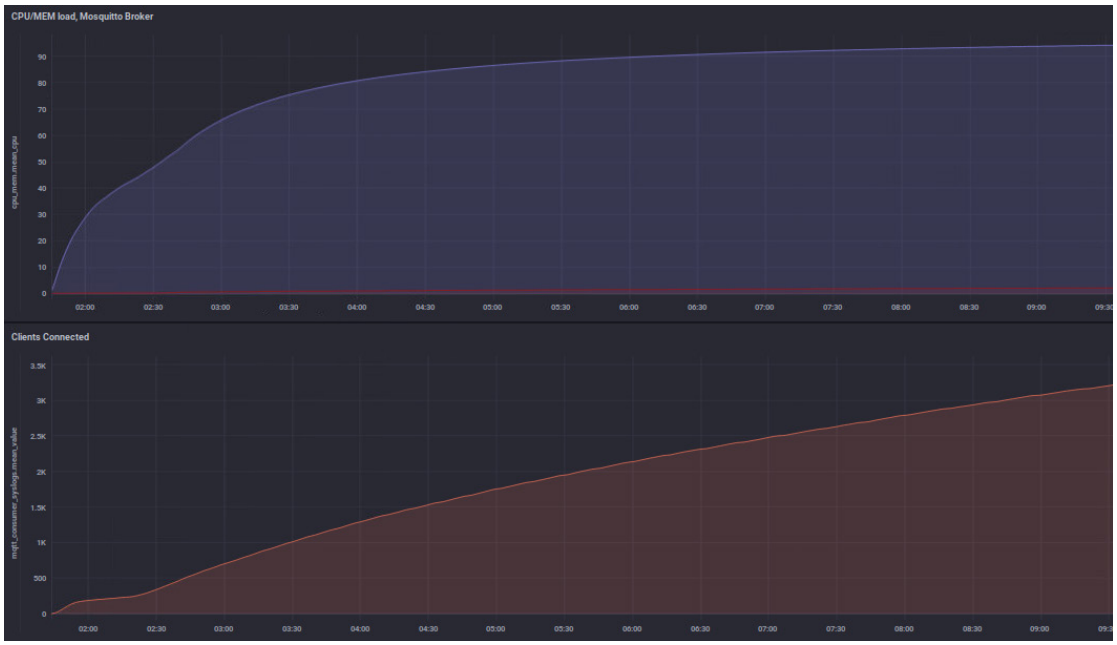
Αντίστοιχα με πρίν, και σε αυτή την τιμή του Spawning Delay έχουμε μείωση της κατανάλωσης πόρων σε αναλογία με την αύξηση του Publish Delay. Στο διάγραμμα M7 παρατηρείται μια καθυστέρηση στην αύξηση των συνδεδεμένων clients, η οποία ωστόσο ακολουθείται αντίστοιχα από την απόδοση του broker σε κατανάλωση CPU/MEM, κάτι που επιβεβαιώνει τον τρόπο με τον οποίο ανταποκρίνεται ο broker στις συνθήκες τις προσομοίωσης. Οι τιμές των CPU/MEM κυμαίνονται στα ίδια επίπεδα με πριν.

Και πάλι ο μέγιστος αριθμός ενεργών συνδέσεων είναι περίπου 3500-3800.

### *γ. Spawning Delay = 3s*

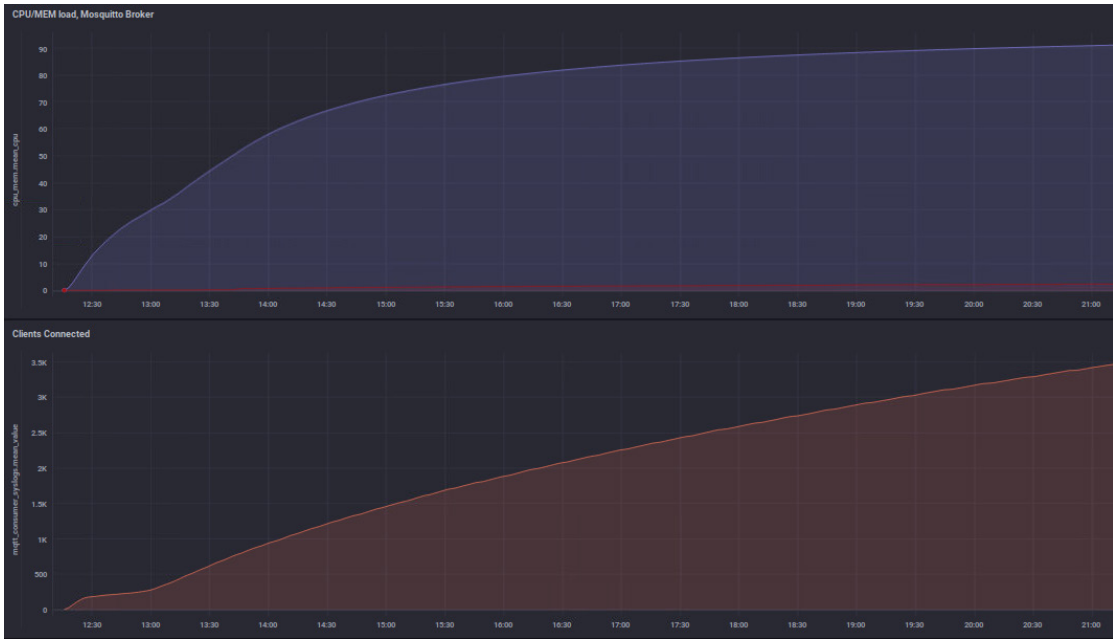
Οι παρακάτω γραφικές παραστάσεις απεικονίζουν τη κατανάλωση πόρων CPU/MEM στο μηχάνημα του Mosquitto Broker, για τις τιμές Publish Delay = 50/100/500/3000ms.

**Διάγραμμα M9:**
**Spawning Delay = 3s, Publish Delay = 50ms**



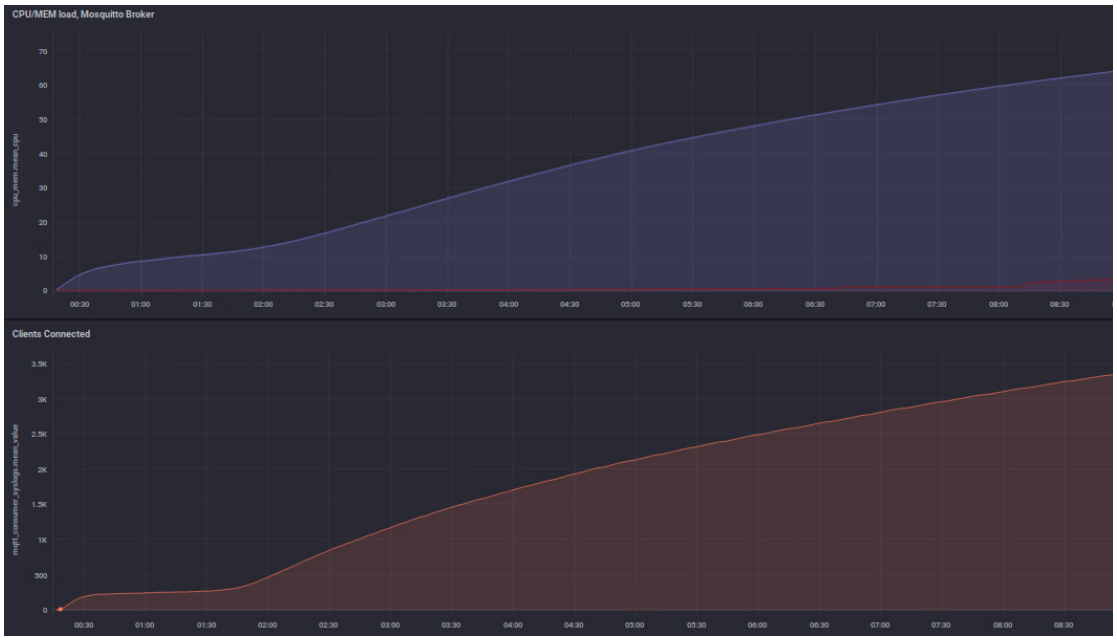

Παρατηρείται μέγιστος αριθμός συνδέσεων γύρω στις 3.0-3.5k clients. Η τιμή κατανάλωσης πόρων της CPU ξεπερνά το 90% μετά από 4.5 ώρες από την έναρξη του πειράματος.

**Διάγραμμα M10:**
**Spawning Delay = 3s, Publish Delay = 100ms**



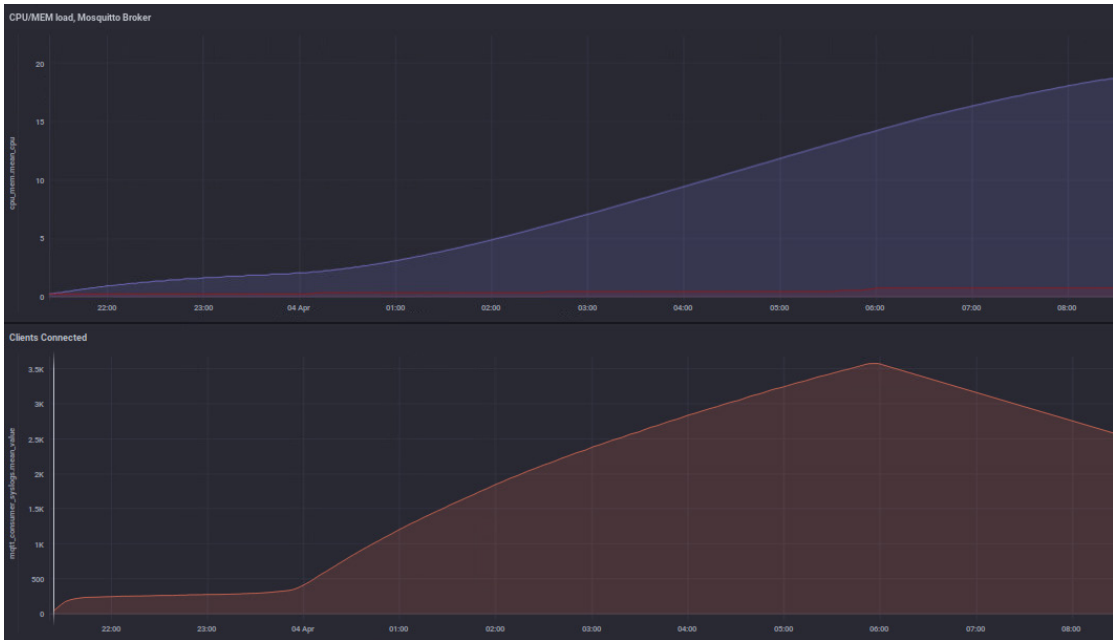

Παρατηρείται μέγιστος αριθμός συνδέσεων γύρω στις 3.5k clients. Η τιμή κατανάλωσης πόρων της CPU ξεπερνά το 90% μετά από 5.5 ώρες από την έναρξη του πειράματος.

**Διάγραμμα M11:**
**Spawning Delay = 3s, Publish Delay = 500ms**



Παρατηρείται μέγιστος αριθμός συνδέσεων γύρω στις 3-3.5k clients. Η τιμή κατανάλωσης πόρων της CPU δεν ξεπερνά το 70% καθ' όλη τη διάρκεια του πειράματος.

**Διάγραμμα M12:**
**Spawning Delay = 3s, Publish Delay = 3000ms**




Παρατηρείται μέγιστος αριθμός συνδέσεων γύρω στις 3.5k clients. Η τιμή κατανάλωσης πόρων της CPU δεν ξεπερνά το 20% καθ' όλη τη διάρκεια του πειράματος.

Και πάλι παρατηρείται η ίδια συμφωνία στην μεταβολή των γραφικών παραστάσεων, και μάλιστα με πρώτο δείγμα το διάγραμμα M11 και στη συνέχεια χαρακτηριστικά το διάγραμμα M12 που για μεγαλύτερη καθυστέρηση και στην δημιουργία νέων συνδέσεων και στην δημοσίευση μηνυμάτων ο broker ανταποκρίνεται με πολύ καλή απόδοση και αξιοποιεί τους διαθέσιμους πόρους σε μεγαλύτερο βάθος χρόνου.

Αξίζει επίσης να σημειωθεί πως **τα σημεία καμπής στις γραφικές παραστάσεις απεικόνισης των συνδεδεμένων client είναι τα σημεία στα οποία ο broker παύει να ανταποκρίνεται με συνέπεια στην δημιουργία νέων συνδέσεων**, δεδομένων των συνεχόμενων publish όλων των ήδη συνδεδεμένων clients. Συγκεκριμένα παρατηρήθηκε ότι υπό συνθήκες έντονου δικτυακού φόρτου ο broker δεν κρατούσε σταθερές τις ήδη υπάρχουσες συνδέσεις, με χαρακτηριστικό παράδειγμα το πρόγραμμα X

το οποίο για την παρακολούθηση των στατιστικών στοιχείων λειτουργίας του broker συνδέεται στα systopics αυτού για να διαβάζει τα δεδομένα που ανανεώνονται εκεί. Από κάποιο σημείο και μετά το πρόγραμμα δεν είχε τη δυνατότητα να διατηρήσει την σύνδεση με τον broker, και για αυτόν τον λόγο δημιουργήθηκε η αναγκαιότητα άλλης μεθόδου logging του broker. Με τον logger πλέον το πρόβλημα αυτό ξεπεράστηκε, μιας και γινόταν συνεχώς προσπάθεια ανανέωσης της σύνδεσης στα systopics, και επομένως το σημείο καμπής απεικονίζει το σημείο πλήρους αδυναμίας ανταπόκρισης του broker λόγω υπερβολικής κατανάλωσης πόρων, δίνοντας έτσι μια καλή εικόνα των ορίων του Mosquitto Broker σε συνθήκες αρκετά ισχυρού δικτυακού φόρτου. Αυτό γίνεται εμφανές και από το διάγραμμα της CPU η οποία σε αυτά τα σημεία έχει τιμή που κυμαίνεται πάνω του 90%. Δεδομένου αυτού, δημιουργείται η αναγκαιότητα να ελεγχθεί το ίδιο σενάριο τόσο με περισσότερους clients (δηλαδή προσθέτωντας VM αναπαραγωγής clients), όσο και με άλλη δομή broker, για την σύγκριση των αποτελεσμάτων εφαρμογής του ίδιου σεναρίου.

Για λόγους ευκολίας, στον επόμενο πίνακα παρατίθενται συγκεντρωτικά οι παρατηρήσεις των παραπάνω προσομοιώσεων. Στις δύο στήλες που απεικονίζουν χρόνο, όπου δεν υπάρχουν δεδομένα ( - ) εννοείται ότι ο broker συνέχιζε να είναι λειτουργικός μέχρι την διακοπή του πειράματος από τον διαχειριστή του δικτύου.

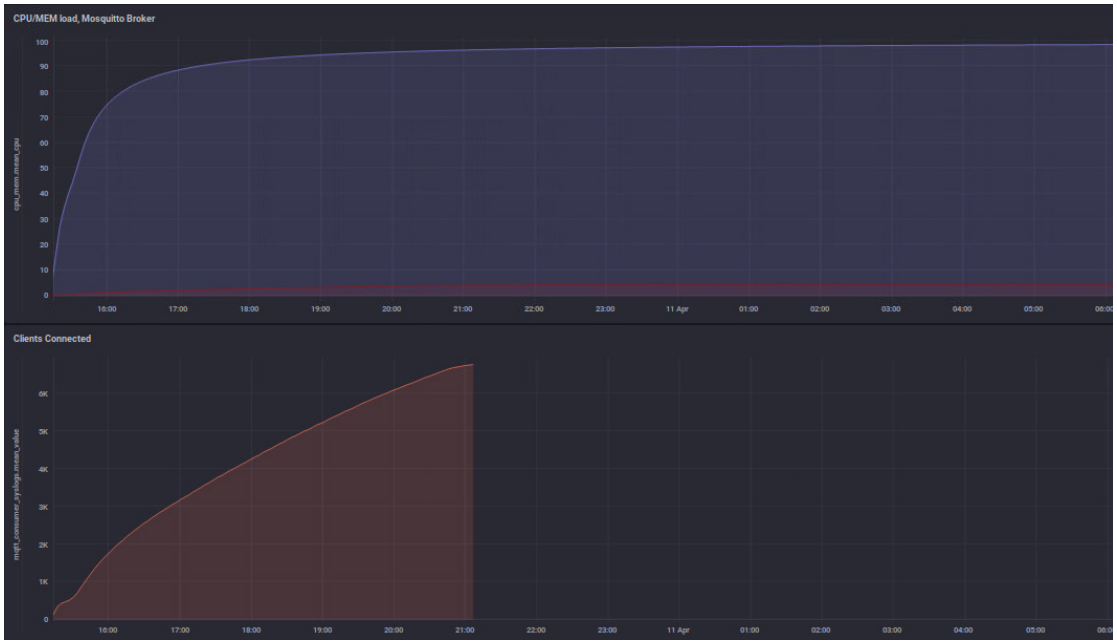
| **Προσομοίωση/ Spawning Delay/ Publish Delay** | | | **Maximum number of clients (approximately)** | **Time for CPU>90% (hours)** | **Time for non-responsive broker (hours)** |
|---|---|---|---|---|---|
| **M1** | **1s** | **50ms** | 3500 | 3 | 11 |
| **M2** | | **100ms** | 3500 | 4 | - |
| **M3** | | **500ms** | 3500 | 9 | - |
| **M4** | | **3000ms** | 3500 | - | - |
| **M5** | **0.5s** | **50ms** | 3500-3800 | 3 | 6.5 |
| **M6** | | **100ms** | 3500-3800 | 5 | 8 |
| **M7** | | **500ms** | 3500-3800 | - | - |
| **M8** | | **3000ms** | 3500 | - | - |
| **M9** | **3s** | **50ms** | 3000-3500 | 4.5 | - |
| **M10** | | **100ms** | 3500 | 5.5 | - |
| **M11** | | **500ms** | 3000-3500 | - | - |
| **M12** | | **3000ms** | 3500 | - | - |

*Πίνακας 2: Συγκεντρωτικός πίνακας μετρήσεων των προσομοιώσεων M1-M12 για τον Mosquitto Broker*

### *δ. Τρεις ακόμα προσομοιώσεις*

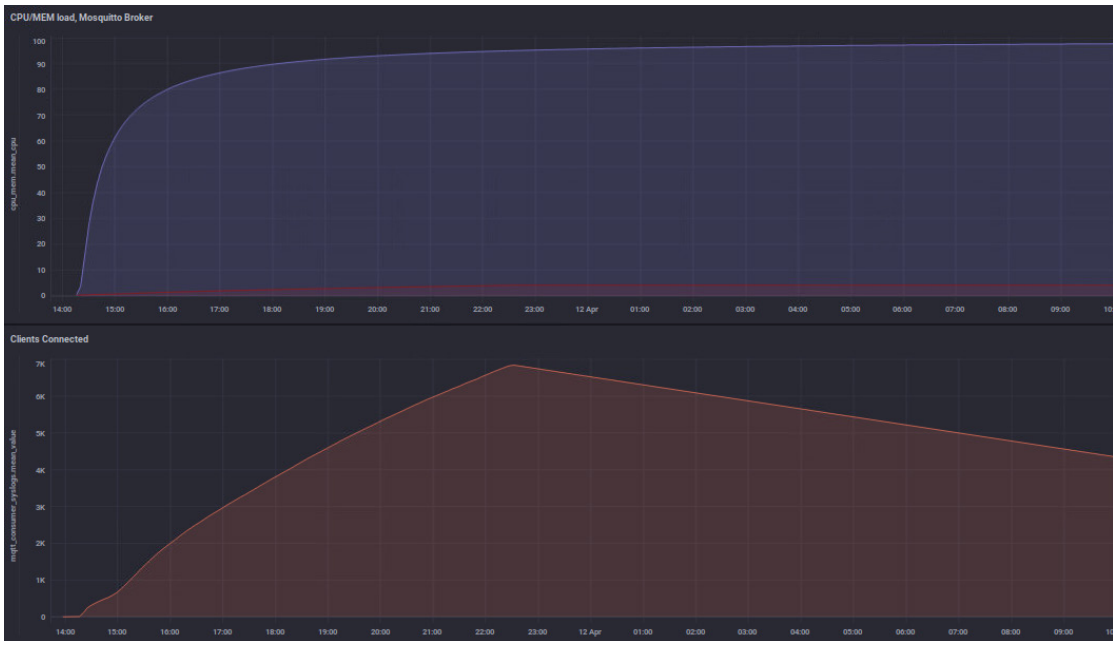
Οι παρακάτω γραφικές παραστάσεις ανταποκρίνονται στα σενάρια 13, 14, 15 του Πίνακα 1. Είναι τρία από τα προηγούμενα σενάρια προσομοίωσης, και συγκεκριμένα τα 2, 9 και 8, με δύο VM αναπαραγωγής νέων συνδέσεων για την παρατήρηση της ανταπόκρισης του Mosquitto Broker σε ακόμα μεγαλύτερο δικτυακό φόρτο.

**Διάγραμμα Μ13:**
**Spawning Delay = 1s, Publish Delay = 100ms**



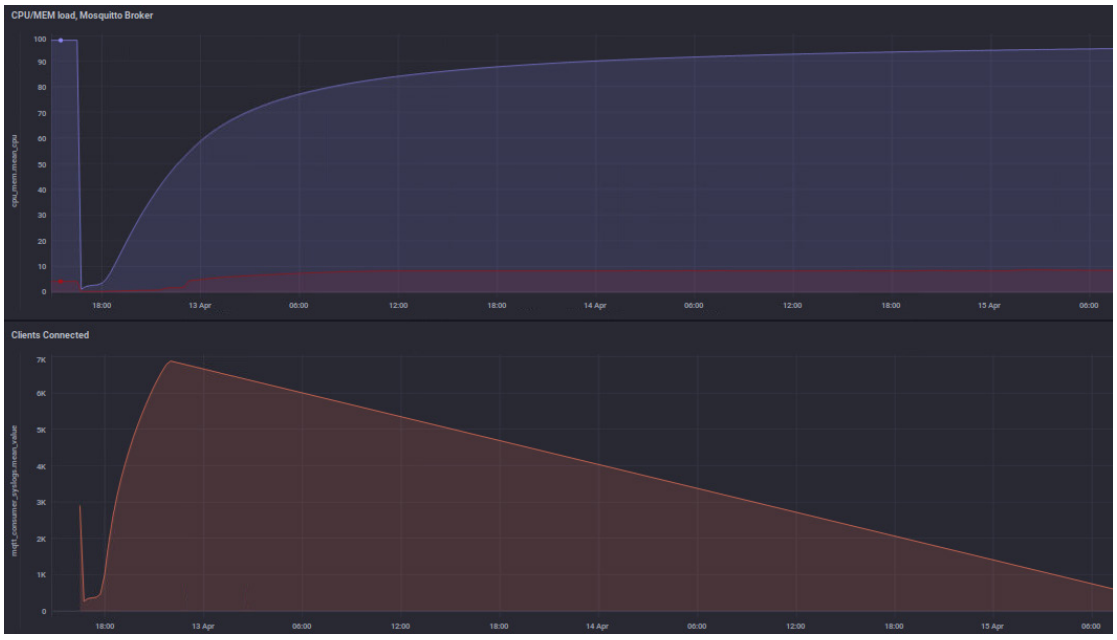

Παρατηρείται μέγιστος αριθμός συνδέσεων γύρω στις 6.5-7k clients, γεγονός αναμενόμενο μετά την προσθήκη δεύτερου VM δημιουργίας νέων client. Η τιμή κατανάλωσης πόρων της CPU ξεπερνά το 90% μετά από 3 ώρες από την έναρξη του πειράματος, ενώ μετά από 11 ώρες ο broker παύει να ανταποκρίνεται.

**Διάγραμμα M14:**
**Spawning Delay = 3s, Publish Delay = 50ms**




Παρατηρείται και πάλι μέγιστος αριθμός συνδέσεων γύρω στις 6.5-7k clients. Η τιμή κατανάλωσης πόρων της CPU ξεπερνά το 90% μετά από 4 ώρες από την έναρξη του πειράματος, ενώ μετά από 10 ώρες ο broker παύει να ανταποκρίνεται.

**Διάγραμμα M15:**
**Spawning Delay = 0.5s, Publish Delay = 3000ms**

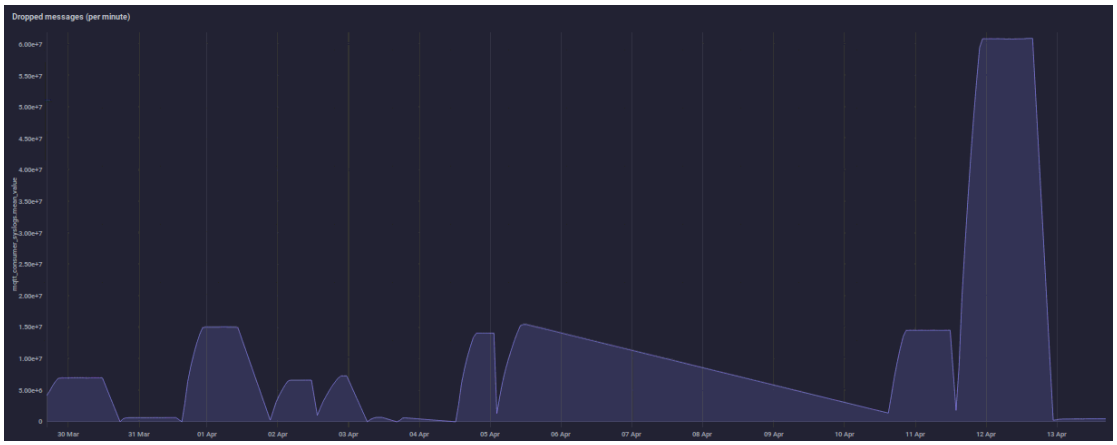


Παρατηρείται και πάλι μέγιστος αριθμός συνδέσεων γύρω στις 6.5-7k clients. Η τιμή κατανάλωσης πόρων της CPU ξεπερνά το 90% μετά από 2 ημέρες από την έναρξη του πειράματος. Η δημιουργία νέων client σταματά μετά από 6 ώρες (λόγω του μικρού Spawning Delay τα VM εξαντλούνται νωρίτερα) επομένως θεωρείται ότι ο broker συνεχίζει να λειτουργεί με όσες συνδέσεις δημιουργήθηκαν μέσα σε αυτό το διάστημα.

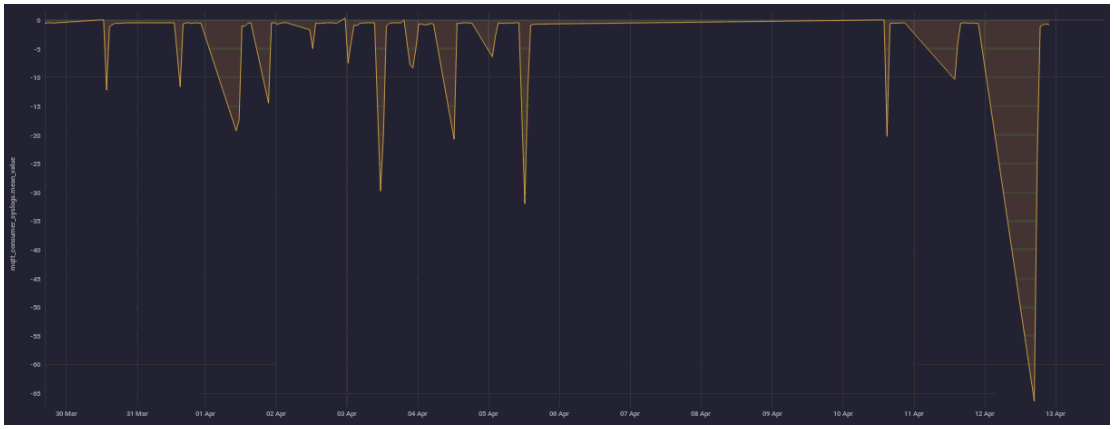
Συγκεντρωτικά λοιπόν, στο διάγραμμα M13 παρατηρείται ότι μέχρι το πλήθος των 3.5k clients η CPU έχει φτάσει ήδη μέχρι το 90%+, όπως συνέβη και στην προσομοίωση M2 που είχαμε τα ίδια μεγέθη Spawning Delay και Publish Delay. Από αυτό το σημείο και μετά, μέχρι και τους 7k clients, σημειώνεται αύξηση της μνήμης RAM από το ~2.5% που παρατηρήθηκε στην προσομοίωση 2 μέχρι το 5%, κάτι που δείχνει ότι με την εξάντληση της CPU γίνεται χρήση της RAM για να ανταποκριθεί ο broker στον ολοένα αυξανόμενο φόρτο. Στην προσομοίωση M2, ο broker δεν βγαίνει εκτός δικτύου, ενώ στην M13 αυτό φαίνεται να συμβαίνει, γεγονός που δείχνει ότι οι διπλάσιες συνδέσεις clients που επετεύχθηκαν, με συνεχόμενα publish στο πέρασμα του χρόνου καταφέρνουν να εξαντλήσουν τους πόρους του broker ώστε να μην ανταποκρίνεται πλέον.

Στο διάγραμμα M14 παρατηρείται αντίστοιχη συμπεριφορά με αυτή που περιγράφηκε παραπάνω, καθώς και εδώ ο broker μετά από 10 ώρες λειτουργίας αποσυνδέεται, σε αντίθεση με το πείραμα M9 με τις ίδιες παραμέτρους που αυτό δεν συμβαίνει. Στο διάγραμμα M15 παρατηρείται σταδιακή αύξηση της CPU, ακόμα και στον μέγιστο αριθμό συνδέσεων, κάτι που υποδηλώνει ότι ο φόρτος που προκαλείται στον Mosquitto Broker προέρχεται από τα συνεχόμενα publish των clients και όχι από το πλήθος τους. Καθώς προσεγγίζεται ο μέγιστος αριθμός ενεργών συνδέσεων, παρατηρείται αύξηση της RAM μέχρι το 10%, μαζί με την σταδιακή αύξηση της CPU.

Παρακάτω παρατίθονται οι γραφικές παραστάσεις των απωλεσθέντων μηνυμάτων (Dropped Messages, Διάγραμμα M16) και των απωλειών σύνδεσης (Disconnected Clients, Διάγραμμα M17), συνολικά για όλες τις προσομοιώσεις που αναφέρθηκαν παραπάνω (M1-M15).




**Διάγραμμα Μ16: Γραφική παράσταση απωλεσθέντων μηνυμάτων**




**Διάγραμμα Μ17: Γραφική παράσταση απωλειών συνδέσεων (disconnected clients)**

Σε αυτά τα δύο διαγράμματα φαίνεται ότι οι απώλειες τόσο των μηνυμάτων όσο και των ενεργών συνδέσεων κατά την εκτέλεση των προσομοιώσεων κινούνται σε σταθερά επίπεδα. Μεγαλύτερες αποσυνδέσεις ενεργών client εντοπίζονται στις προσομοιώσεις που συνέβησαν τις ημερομηνίες 3/4, 5/4, και 12/4 οι οποίες αντιστοιχούν στις προσομοιώσεις 7, 8 και 15. Και στις τρεις περιπτώσεις η τιμή του Spawning Delay ήταν 0.5s, γεγονός που υποδηλώνει αδυναμία του Mosquitto να ανταπεξέλθει άριστα στην διαχείριση πολλών ταυτόχρονων νέων συνδέσεων. Παρόλα αυτά, το πλήθος των συνδέσεων που απέτυχαν είναι σχετικά μικρό σε σχέση με το συνολικό πλήθος ενεργών clients.

Απαραίτητη παρατήρηση σε αυτό το σημείο είναι το γεγονός πως η

παρακολούθηση των χαμένων συνδέσεων, πέρα από την απεικόνιση του παραπάνω διαγράμματος, θεωρήθηκε απαραίτητο να συμπεριλάβει και το ποσοστό αποτυχίας συνδέσεων σε σύγκριση με το πλήθος των clients που δημιουργήθηκαν από τα scripts. Αυτή η παράμετρος ήταν δυνατόν να παρατηρηθεί μόνο μέσω της εντολής watch του linux terminal των VM, με την οποία γινόταν έλεγχος του πόσα client script ήταν ενεργά σε σύγκριση με το πόσα έδειχνε το ίδιο το client script ότι έχει δημιουργήσει. Είναι ασφαλές να ειπωθεί ότι περίπου το 1/3 αυτών των script όντως πέτυχαν να δημιουργήσουν ενεργή σύνδεση με τον broker, ποσοστό που αποτελεί μερικό δείκτη του δημιουργηθέντος δικτυακού φόρτου και της ικανότητας άμεσης ανταπόκρισης από μια non clustered δομή.

Για τα dropped messages του Διαγράμματος Μ16, φαίνεται ότι υπήρχαν απώλειες σε πακέτα σε όλες τις προσομοιώσεις που εκτελέσθηκαν. Το γεγονός αυτό είναι ένα ζήτημα που θα πρέπει να αντιμετωπιστεί, δεδομένου ότι η απώλεια μηνυμάτων μπορεί να επιφέρει απώλειες σοβαρών πληροφοριών για την λειτουργία του δικτύου στο σύνολό του.

### *4.2.2 Clustered implementation*

Έχοντας εξετάσει την ανταπόκριση του της non-clustered δομής μτ τον Mosquitto Broker στην προηγούμενη ενότητα, τώρα θα εκτελεστούν τα ίδια πειράματα στο ίδιο δίκτυο με clustering χρησιμοποιώντας τον EMQX Broker ως cluster με δύο ενεργά nodes και τον HAProxy Load Balancer, αρχιτεκτονική που περιγράφηκε στο Κεφάλαιο 3.2.2.β. Στην δομή του δικτύου με τον EMQX Broker εκτελέστηκαν οι 15 προσομοιώσεις, αντίστοιχες με αυτές της προηγούμενης ενότητας, και κάποιες ακόμα, προσαρμόζοντας τα βάρη του Load Balancer για πιο δίκαιη κατανομη του φόρτου μεταξύ των δύο nodes. Τα χαρακτηριστικά των προσομοιώσεων παρουσιάζονται συνοπτικά στον παρακάτω πίνακα:

| **EMQX Broker – Εκτέλεση Προσομοιώσεων** | | | | | |
|---|---|---|---|---|---|
| **Προσομοίωση/ Διάγραμμα** | | **Publish Delay (in ms)** | **Spawning Delay (in s)** | **Number of VMs** | **Weights (node1/node2)** |
| 1 / E1 | | 50 | 1 | 2 | - |
| 2 / E2 | (α) | 100 | 1 | 2 | - |
| | (β) | | | | 100/50 |
| 3 / E3 | (α) | 500 | 1 | 2 | - |
| | (β) | | | | 128/62 |
| 4 / E4 | | 3000 | 1 | 2 | - |
| 5 / E5 | | 50 | 0.5 | 2 | 128/62 |
| 6 / E6 | | 100 | 0.5 | 2 | - |
| 7 / E7 | | 500 | 0.5 | 2 | - |
| 8 / E8 | | 3000 | 0.5 | 2 | 249/180 |
| 9 / E9 | | 50 | 3 | 2 | 128/62 |
| 10 / E10 | | 100 | 3 | 2 | - |
| 11 / E11 | | 500 | 3 | 2 | 249/180 |
| 12 / E12 | | 3000 | 3 | 2 | 249/180 |
| 13 / E13 | | 100 | 1 | 4 | 249/180 |
| 14 / E14 | | 50 | 3 | 4 | 249/180 |
| 15 / E15 | | 3000 | 0.5 | 4 | 249/180 |

*Πίνακας 3: Πίνακας παραμέτρων προσομοιώσεων για τον EMQX Broker*

Για να γίνει πιο εύκολη η ανάλυση των πειραματικών αποτελεσμάτων, θα γίνει παρουσίαση αυτών ομαδοποιημένων βάσει της μεταβλητής Spawning Delay, η οποία παίρνει τις τιμές 0.5, 1, 3 και μετράται σε δευτερόλεπτα.

**Στα διαγράμματα που ακολουθούν απεικονίζονται:**

- **η κατανάλωση πόρων CPU για κάθε node του cluster**. Στα διαγράμματα φαίνονται 3 καμπύλες, οι οποίες απεικονίζουν το CPU load για 1/5/15 s. Ο EMQX δεν αποτελεί μία μεμονωμένη διεργασία αλλά πολλές που τρέχουν παράλληλα αξιοποιώντας όλη την CPU και όχι έναν μόνο πυρήνα όπως συνέβαινε στην περίπτωση του Mosquitto Broker. Επομένως στα διαγράμματα η τιμή που δίνεται για το CPU load εκφράζεται με μέτρο το ***1 * (number of cores),*** αντί για το ποσοστό %. Αυτή η έκφραση εξηγείται ως εξής: η τιμή 0 δείχνει ότι η CPU δεν αξιοποιείται καθόλου, ενώ η τιμή 1 εκφράζει την 100% χρήση των διαθέσιμων πόρων ενός πυρήνα. Αντίστοιχα, αν ο επεξεργαστής διαθέτει 2 πυρήνες, η τιμή 2 εκφράζει την 100% χρήση τους, αν οι πυρήνες είναι 4 η τιμή αυτή θα είναι 4 κ.ο.κ. Τιμή CPU load μεγαλύτερη του απόλυτου σημαίνει ότι ο φόρτος που έχει ανατεθεί στον επεξεργαστή είναι μεγαλύτερος από τις ικανότητές του, και κατά πάσα πιθανότητα πολλές διεργασίες να καθυστερούν ή να είναι σε ουρά αναμονής, με αποτέλεσμα να υπάρχει πιθανή απώλεια δεδομένων σε περιπτώσεις που απαιτείται 100% ανταπόκριση του συστήματος.
- **η χρησιμοποιούμενη μνήμη RAM για κάθε node του cluster**. Και σε αυτήν την περίπτωση, η απάντηση σε ερώτημα προς τον EMQX για την χρησιμοποιούμενη μνήμη, όπως είδαμε στο Κεφάλαιο 3.2.2.β, περιέχει πολλές πληροφορίες για την χρησιμοποιούμενη μνήμη. Στα διαγράμματα λαμβάνονται υπόψιν δύο από αυτές τις τιμές, η **used total (memory/total)** και η **total system memory (memory/system).** Η τιμή memory/total είναι η χρησιμοποιούμενη μνήμη τη δεδομένη χρονική στιγμή, ενώ η memory/system αναφέρεται στην δεσμευμένη για τον EMQX μνήμη από το σύστημα.
- **το πλήθος ενεργών συνδέσεων με τον broker (connected clients).**

Τα διαγράμματα απωλεσθέντων μηνυμάτων, απεσταλμένων και ληφθέντων, καθώς και οι υπόλοιπες πληροφορίες που λαμβάνονται από τα systopics δεν παρουσιάζονται χάριν συντομίας, ωστόσο σχολιάζονται οπου είναι απαραίτητο.

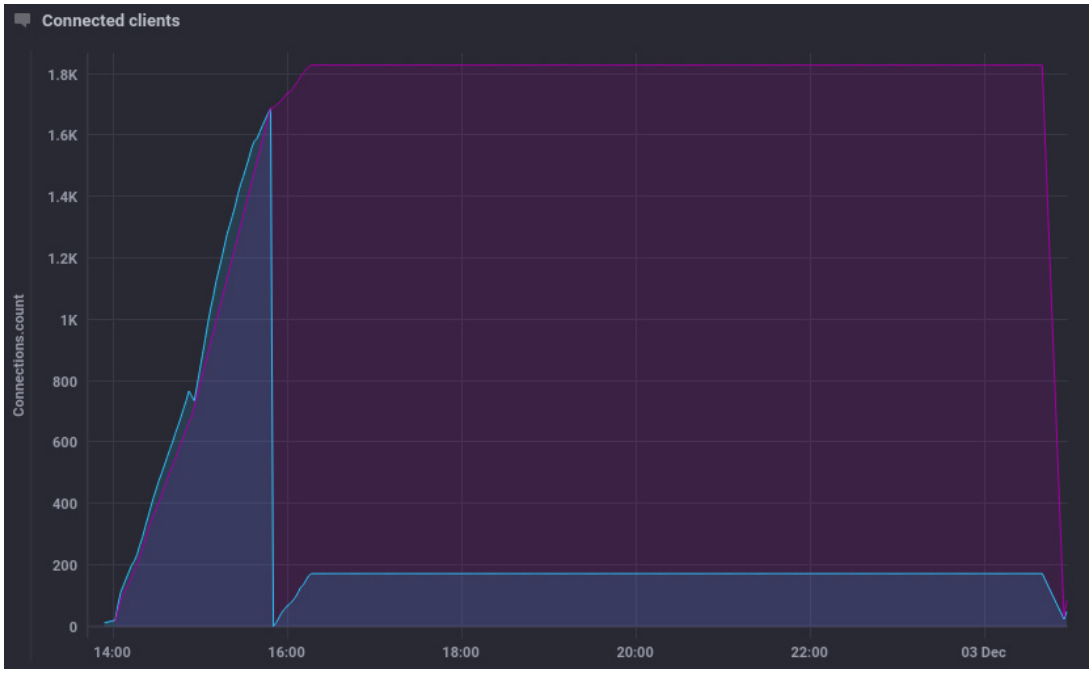
### *α. Spawning Delay = 1s*

Οι παρακάτω γραφικές παραστάσεις απεικονίζουν τη κατανάλωση πόρων CPU/MEM στο μηχάνημα του EMQX Broker, για τις τιμές Publish Delay = 50/100/500/3000ms.

**Διάγραμματα Ε1:**
**Spawning Delay = 1s, Publish Delay = 50ms**


*(α1) CPU load and Memory Consumption for node 172.16.8.203*


*(α2) CPU load and Memory Consumption for node 172.16.8.215*



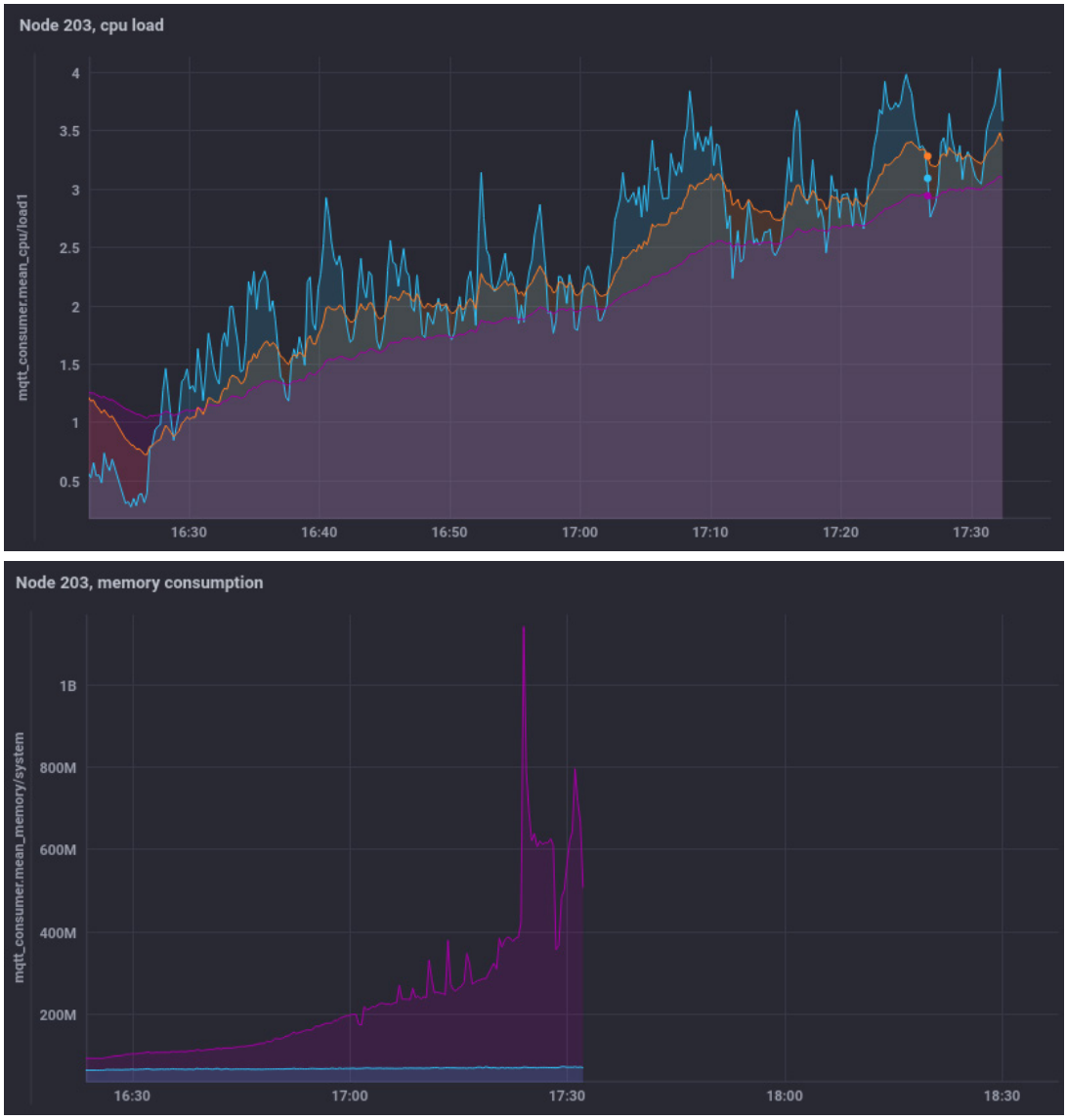

*(β) Connected clients for EMQX cluster*

Αυτή ήταν μια πρώτη δοκιμή των προσομοιώσεων στο δίκτυο με τον EMQX Broker. Στα διαγράμματα E1, φαίνεται ότι η κατανάλωση της CPU στο node 1 ξεπερνάει την τιμή 4, κάτι που όπως εξηγήθηκε παραπάνω σημαίνει ότι ο φόρτος που διαχειρίζεται ο επεξεργαστής του node είναι μεγαλύτερος από τις δυνατότητές του. Στο δεύτερο node η κατανάλωση της CPU είναι μέσα στα αποδοτικά όρια, προσεγγίζοντας το 2. Η μνήμη RAM και στα δύο node είναι σε φυσιολογικά επίπεδα, πραγματοποιώντας γρηγορότερη αύξηση κατά τη διάρκεια του πειράματος όταν οι τιμές της CPU πιάνουν τα πιο υψηλά τους επίπεδα.

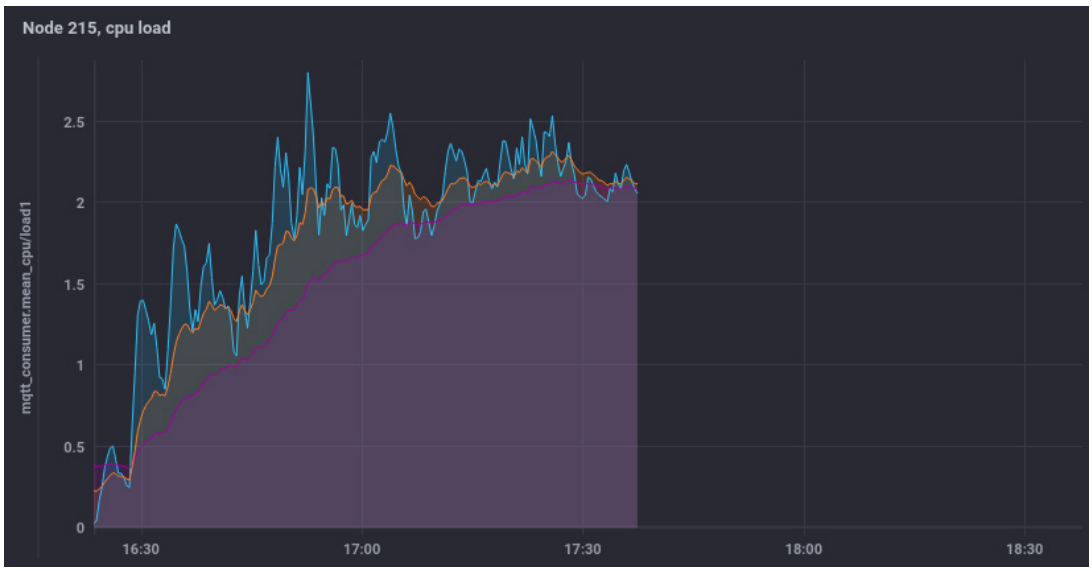
Το συνολικό πλήθος clients φτάνει τους 3.5k. Το πείραμα αυτό πραγματοποιήθηκε χωρίς βάρη στον load balancer, επομένως κάθε node δέχεται τον ίδιο φόρτο νέων συνδέσεων. Αυτό φαίνεται να επιβαρύνει αρκετά το node 2, το οποίο φαίνεται στα διαγράμματα να αποσυνδέεται από το δίκτυο κάποια στιγμή που η CPU ξεπερνάει τα αποδοτικά όρια και φτάνει σε πολύ μικρό χρονικό διάστημα μέχρι την τιμή 4. Από τη στιγμή που ξαναμπήκε το node 2 στο δίκτυο, η λειτουργια του βρίσκεται μέσα στα φυσιολογικά όρια.

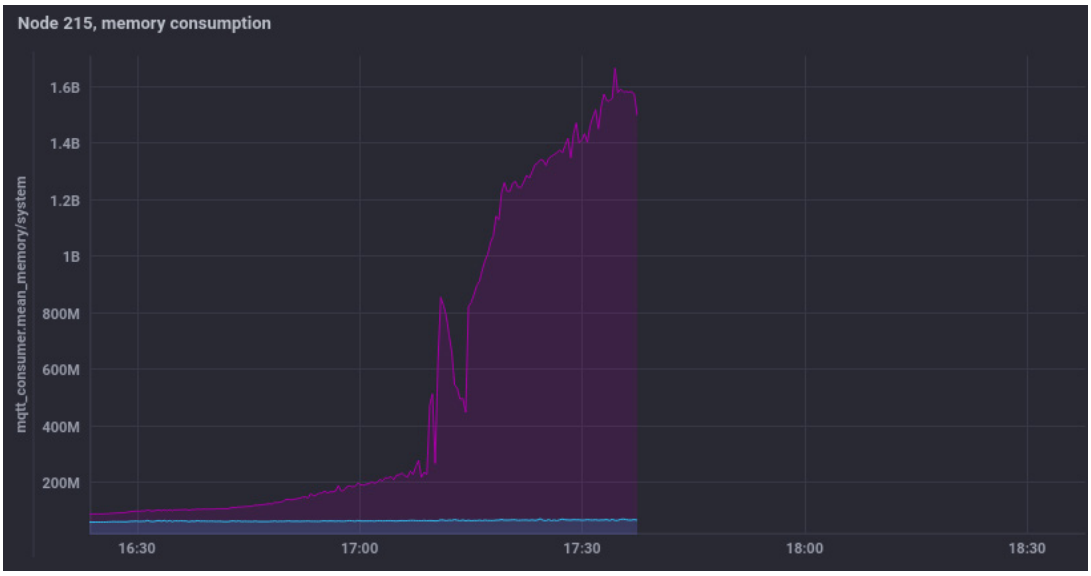
Να σημειωθεί επίσης πως στο συγκεκριμένο σενάριο προσομοίωσης δεν υπήρχε απώλεια μηνυμάτων.

**Διάγραμματα Ε2:**
**Spawning Delay = 1s, Publish Delay = 100ms**

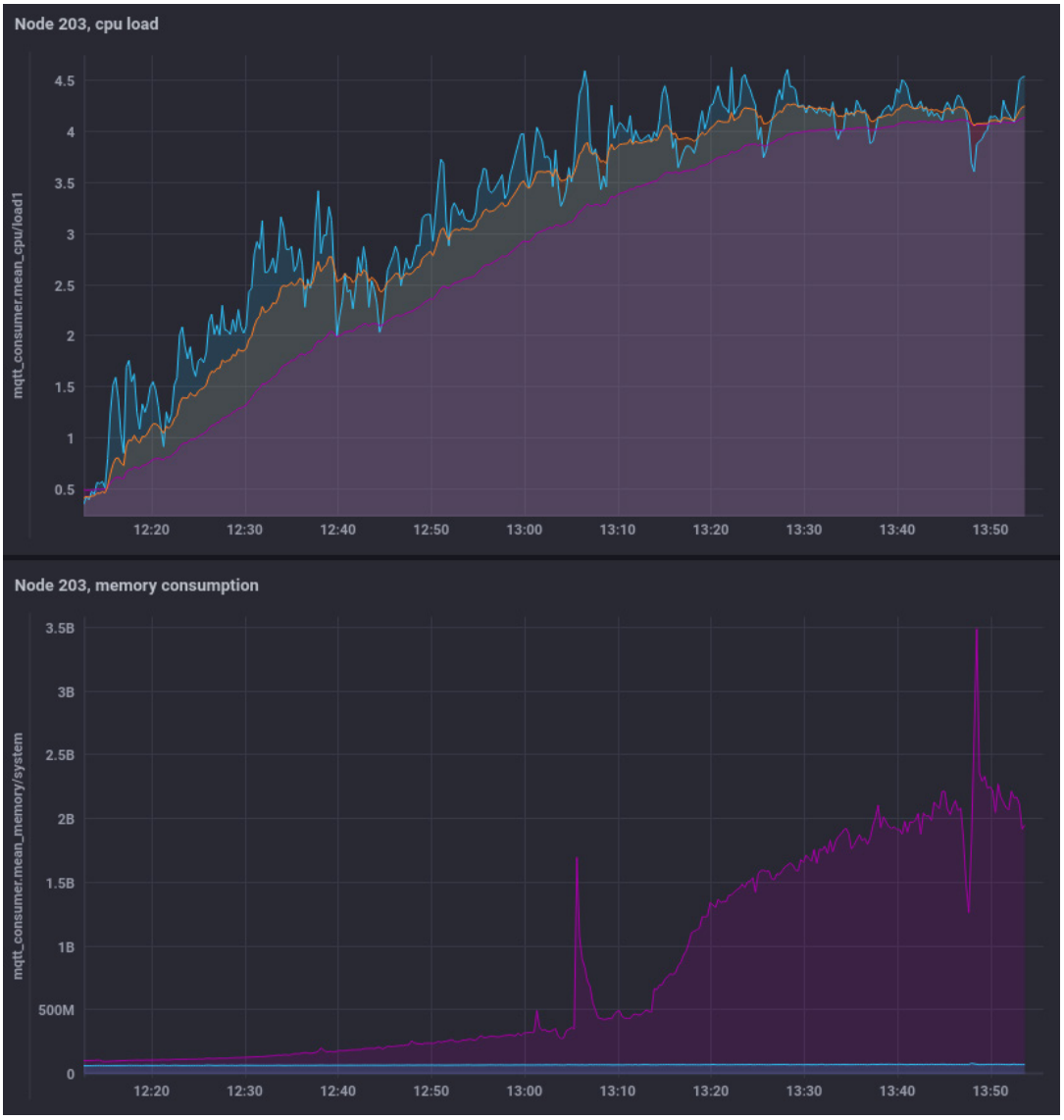



*(α1) CPU load and Memory Consumption for node 172.16.8.203*

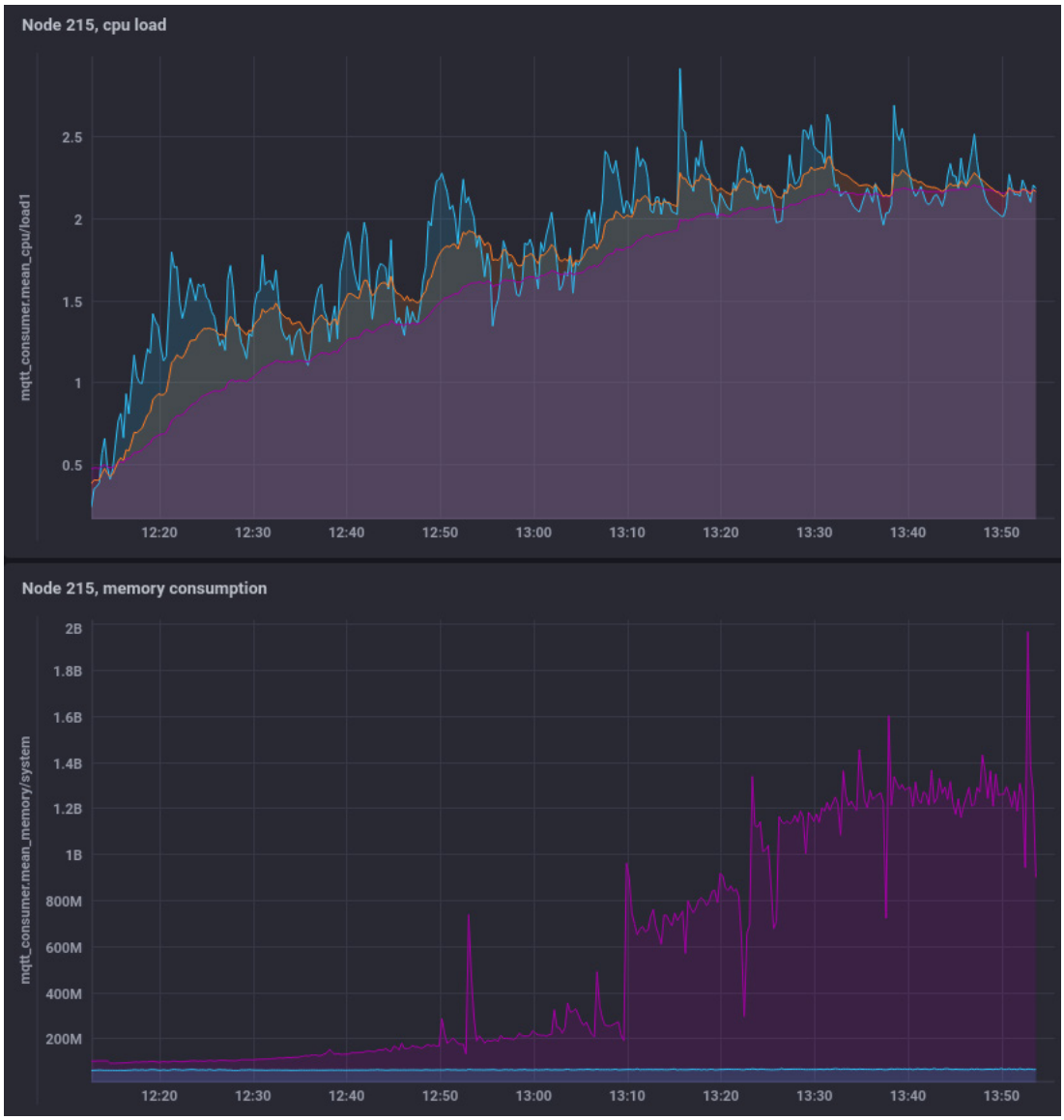




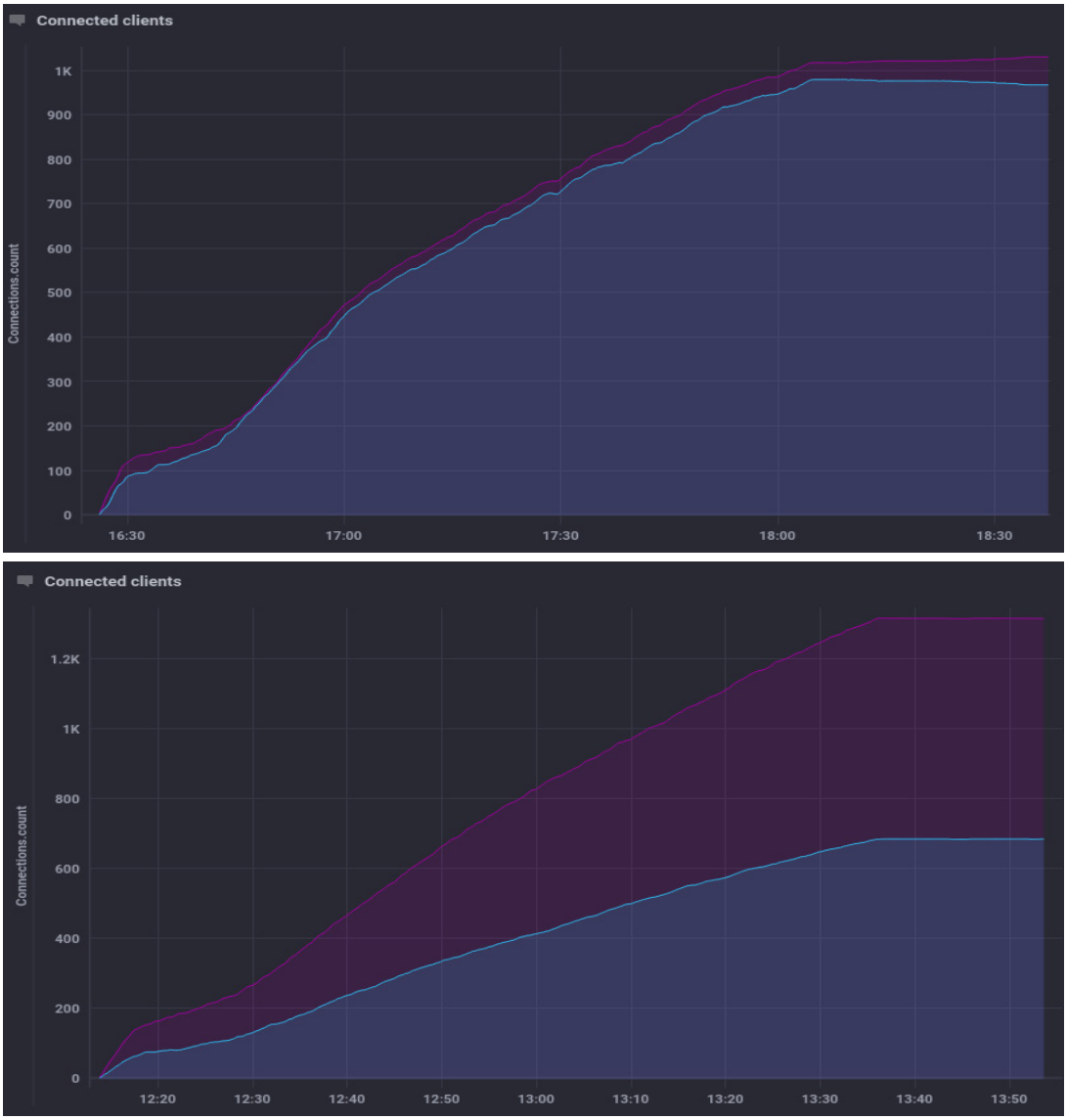

*(α2) CPU load and Memory Consumption for node 172.16.8.215*




*(β1) CPU load and Memory Consumption for node 172.16.8.203, with weighted load balancing*




*(β2) CPU load and Memory Consumption for node 172.16.8.215, with weighted load balancing*



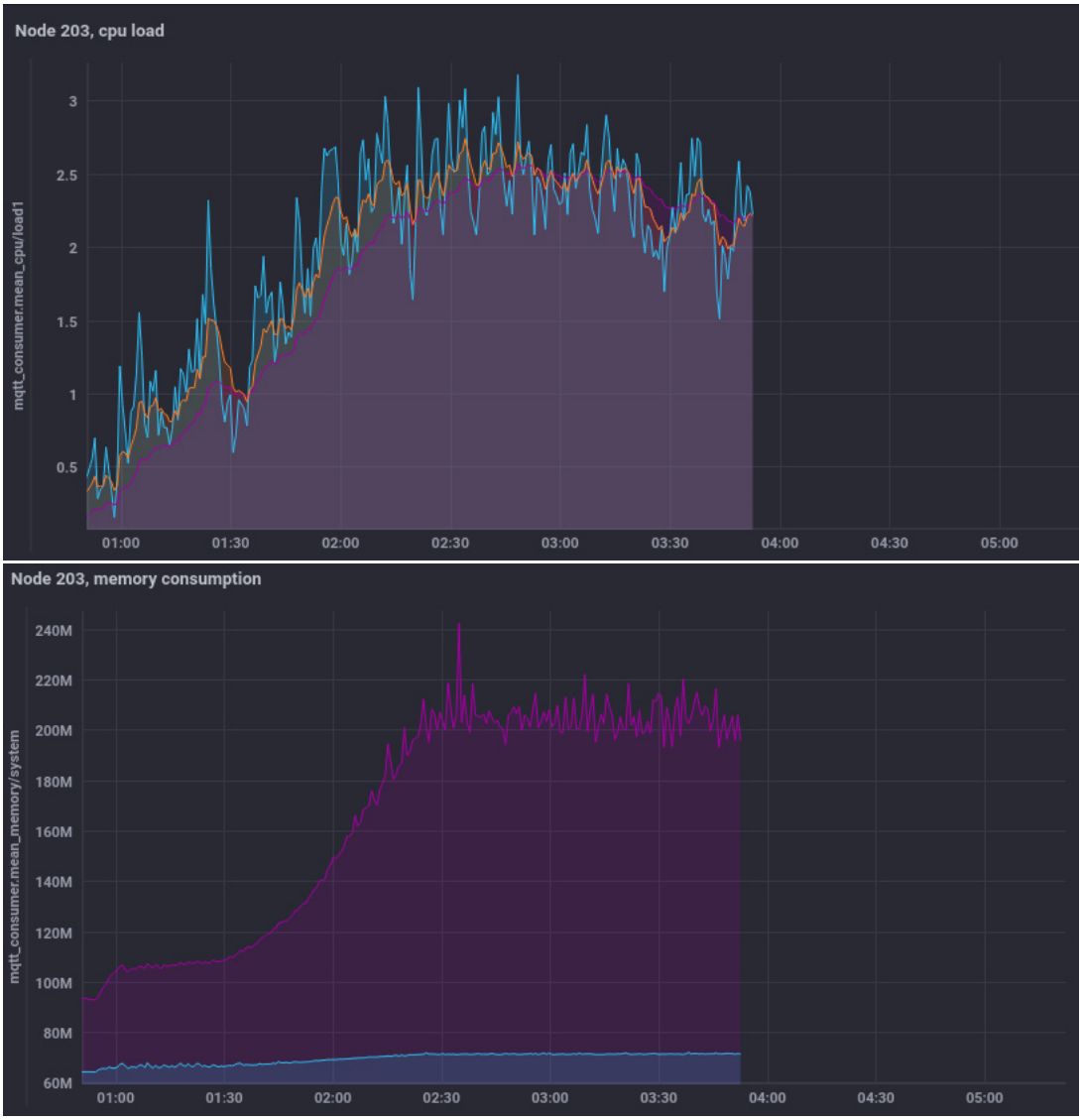

*(c) Connected clients for EMQX cluster with no weights and 100/50 weight*

Τα διαγράμματα Ε2 αποτελούν τη δεύτερη προσομοίωση, η οποία εκτελείται μία φορά χωρίς βάρη στον load balancer και τη δεύτερη μοιράζοντας το βάρος 100/50 στα δύο nodes, δηλαδή δρομολογώντας διπλάσιο φόρτο στο node 1 από ότι στο 2.

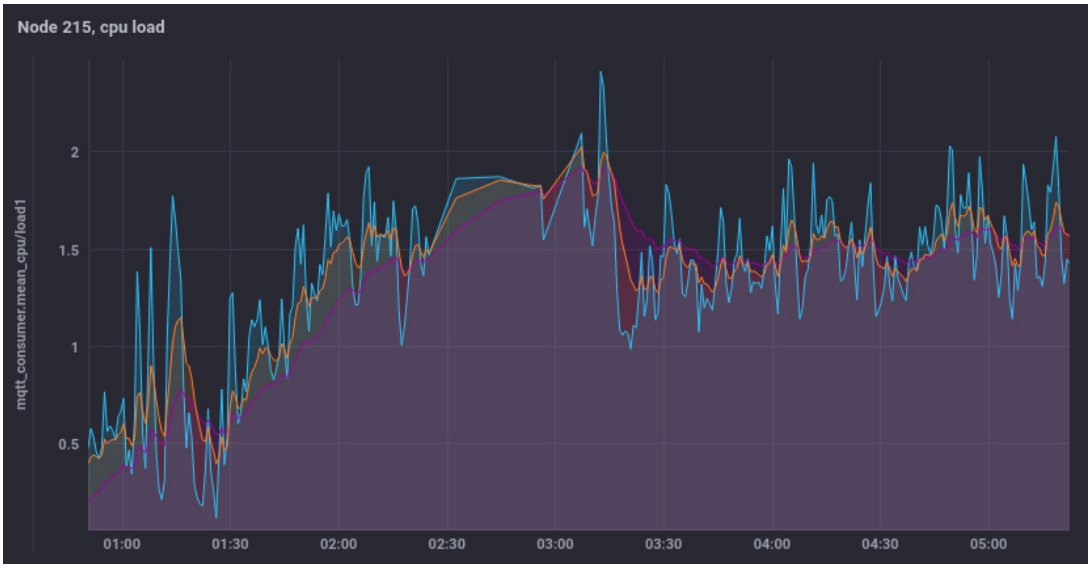
Στην πρώτη περίπτωση παρατηρείται σχεδόν πλήρης αξιοποίηση της CPU και στα δύο nodes, με την CPU στο node 2 να ξεπερνά ανά στιγμές το 2 υποδηλώνοντας υπέρβαση του ορίου φυσιολογικής λειτουργίας, και χαμηλό επίπεδο κατανάλωσης μνήμης RAM, με μεγαλύτερη ένταση στην RAM του node 2. Η απώλεια μηνυμάτων κατά τη διάρκεια του πειράματος ήταν ελάχιστη, και συγκεκριμένα χάθηκαν συνολικά 10 μηνύματα σύμφωνα με τα στοιχεία που λήφθηκαν από τον broker.

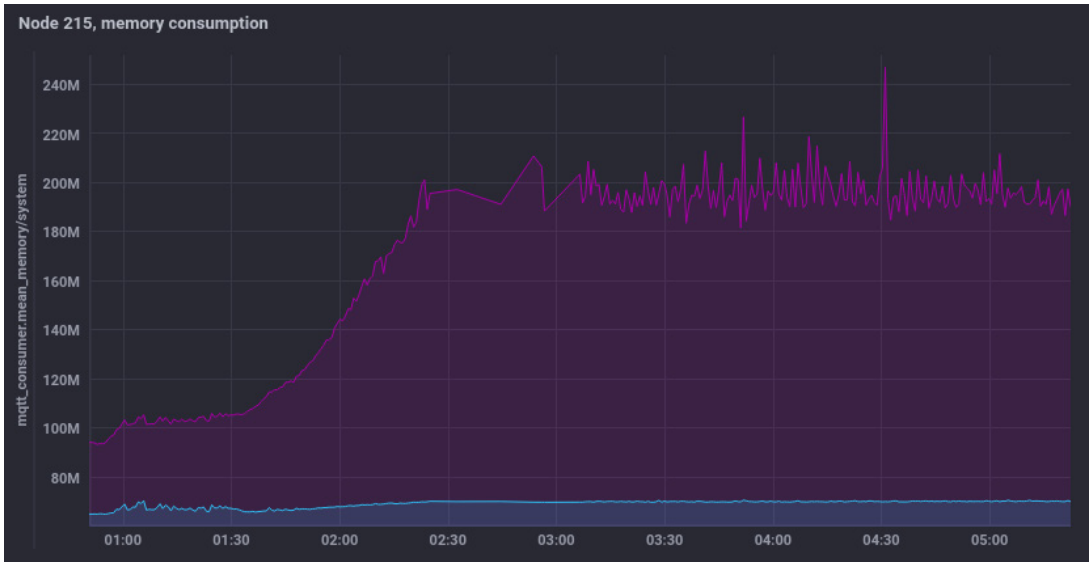
Μετά την προσθήκη βαρών στην δρομολόγηση των νέων συνδέσεων, όπως είναι λογικό, το node 1 είχε μεγαλύτερη κατανάλωση CPU και RAM σε σχέση με πριν, ενώ δεν υπήρχε κανένα χαμένο μήνυμα. Το node 2 έχει αντίστοιχη ανταπόκριση με πριν, με ελαφρώς χαμηλότερες τιμές για CPU και RAM. Παρατηρείται επίσης ότι συγκριτικά, στην εκτέλεση του πειράματος με βάρη επιτυγχάνεται ο ίδιος αριθμός clients στο δίκτυο στον μισό χρόνο από ότι στην εκτέλεση χωρίς βάρη. Ο συνολικός αριθμός clients είναι κοντά στο 2k.

**Διάγραμματα Ε3:**
**Spawning Delay = 1s, Publish Delay = 500ms**

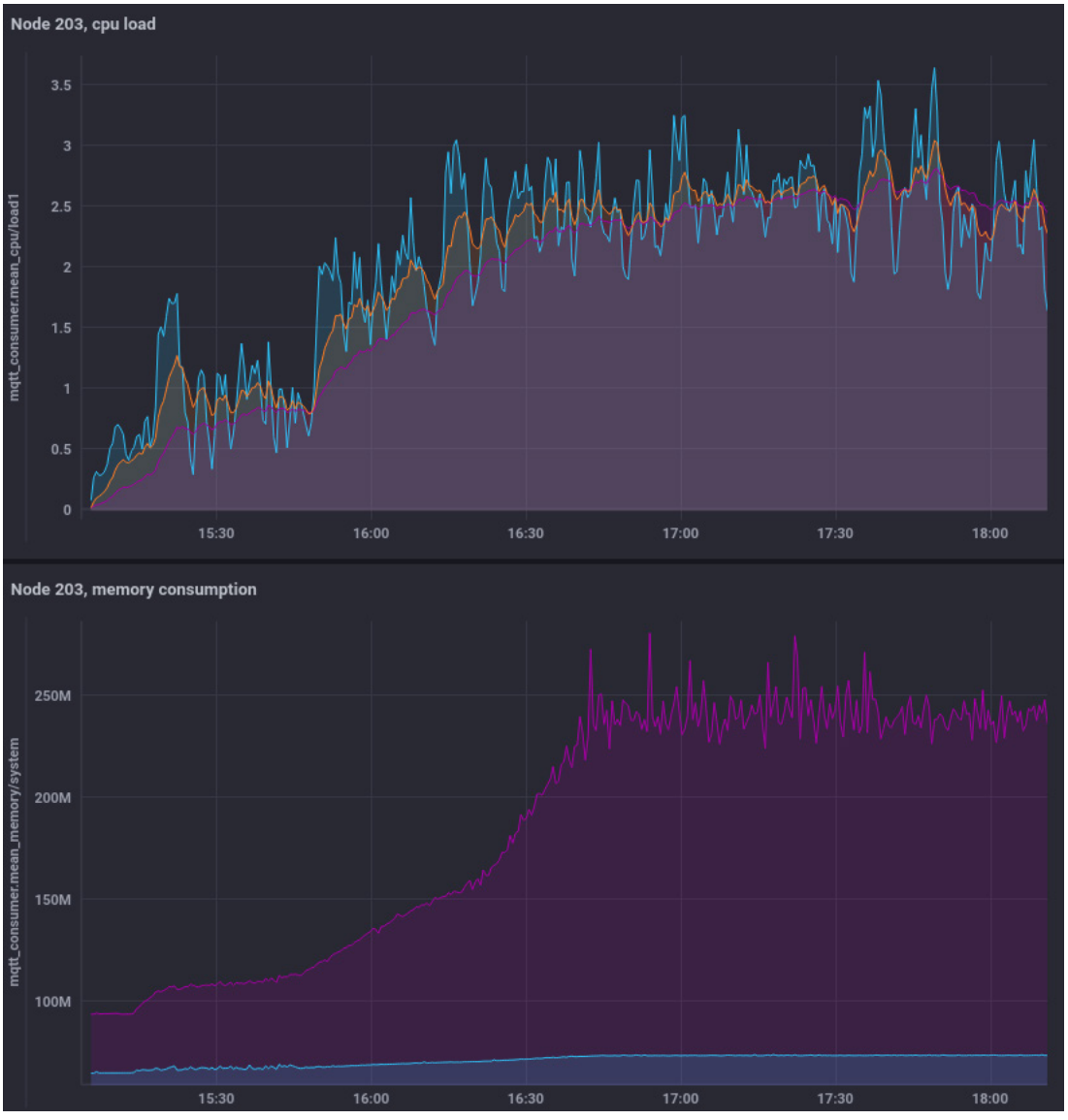



*(α1) CPU load and Memory Consumption for node 172.16.8.203*

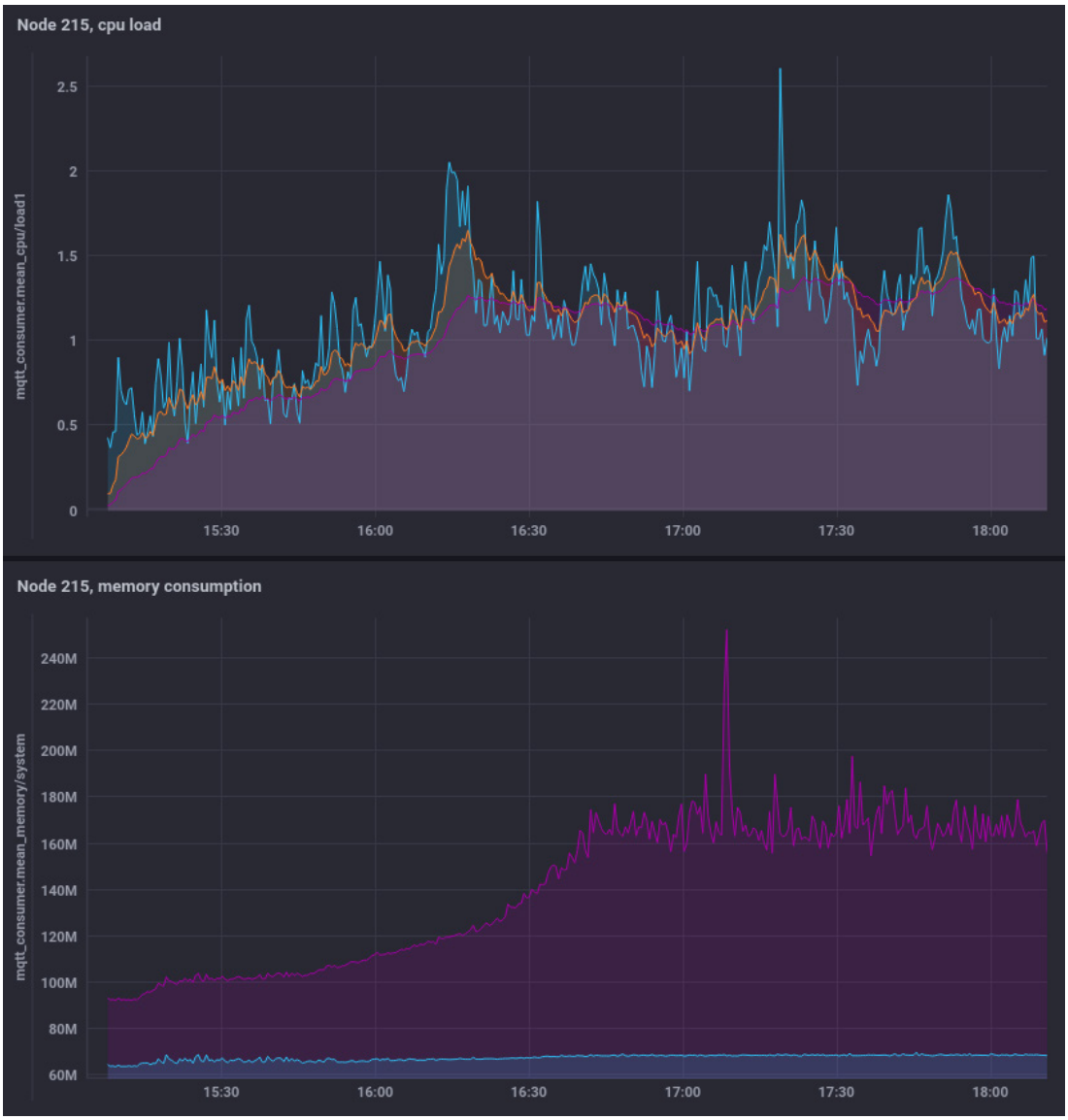




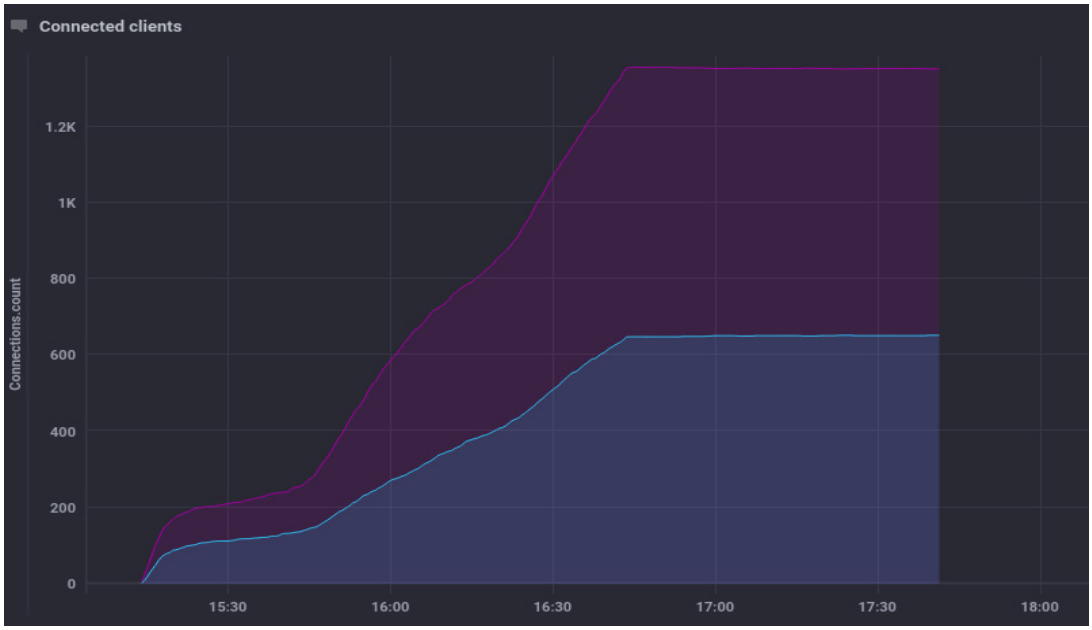

*(α2) CPU load and Memory Consumption for node 172.16.8.215*




*(β1) CPU load and Memory Consumption for node 172.16.8.203, with weighted load balancing*




*(β2) CPU load and Memory Consumption for node 172.16.8.215, with weighted load balancing*




*(c) Connected clients for EMQX cluster with no weights and 128/62 weight*

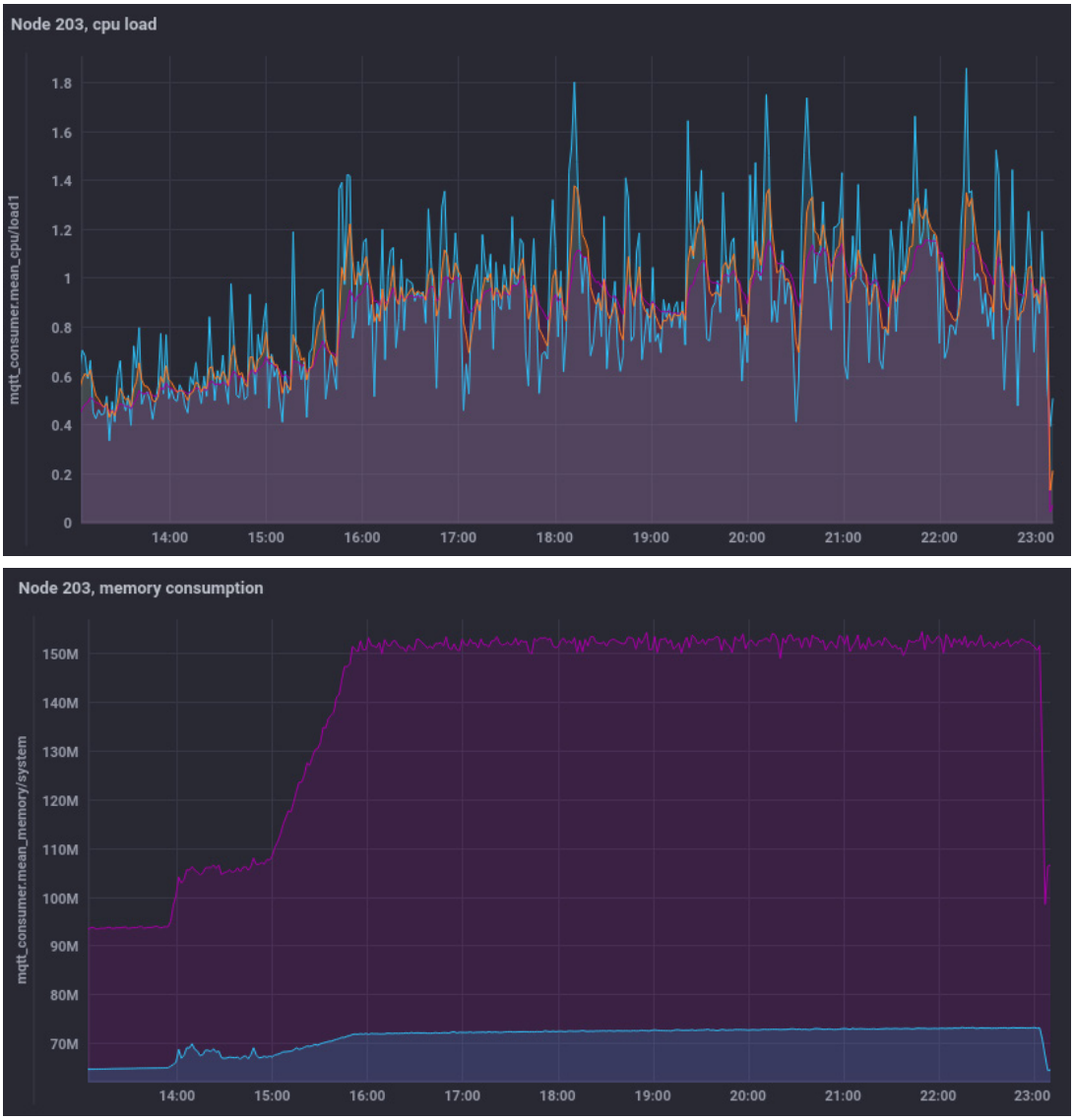
Τα διαγράμματα Ε3, όπως και τα Ε2, απεικονίζουν δύο προσομοιώσεις, μία χωρίς βάρη και μία με βάρη 128/62 μεταξύ των δύο nodes.

Στην πρώτη, χωρίς βάρη, λόγω της εισαγωγής μεγαλύτερης καθυστέρησης μεταξύ των διαδοχικών publish των clients, παρατηρείται αρκετά καλύτερη απόδοση και στα δύο nodes σε σχέση με τα πειράματα Ε1 και Ε2, με χαμηλότερες τιμές και για την CPU και για την μνήμη RAM, γεγονός που επιβεβαιώνει την υπόθεση ότι η αποστολή μηνυμάτων με μεγάλη συχνότητα επιβαρύνει τρομερά το δίκτυο. Επίσης, δεν υπήρχε απώλεια μηνυμάτων σε αυτό το πείραμα.

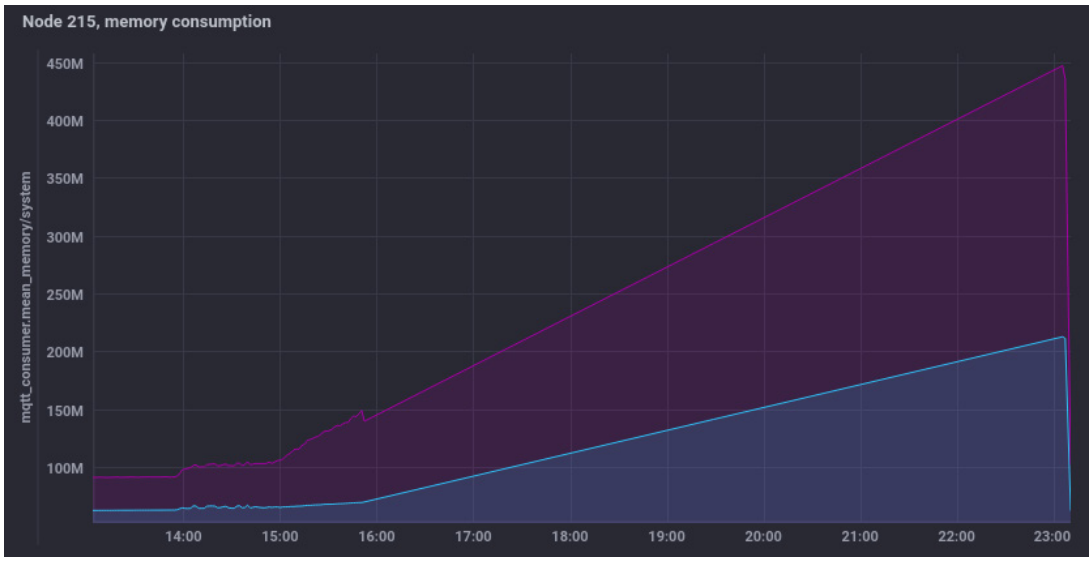
Με την προσθήκη βαρών, μετατοπίζεται όγκος πληροφορίας προς επεξεργασία στο node 1, και επομένως λογικά παρατηρείται μικρή αύξηση στην κατανάλωση πόρων εκεί και αντίστοιχη βελτίωση στους αξιοποιούμενους πόρους του node 2, ενώ υπάρχει μικρή απώλεια μηνυμάτων (35 dropped messages).

Ο συνολικός αριθμός clients και στις δύο περιπτώσεις είναι προσεγγιστικά 2k. Και πάλι παρατηρείται ότι στην εκτέλεση του πειράματος με βάρη στο load balancing ο μέγιστος αριθμός ενεργών clients επιτυγχάνεται σε μικρότερο χρόνικο διάστημα από ότι σε εκτέλεση χωρίς βάρη.

**Διάγραμματα Ε4:**
**Spawning Delay = 1s, Publish Delay = 3000ms**




*(α1) CPU load and Memory Consumption for node 172.16.8.203*




*(α2) CPU load and Memory Consumption for node 172.16.8.215*

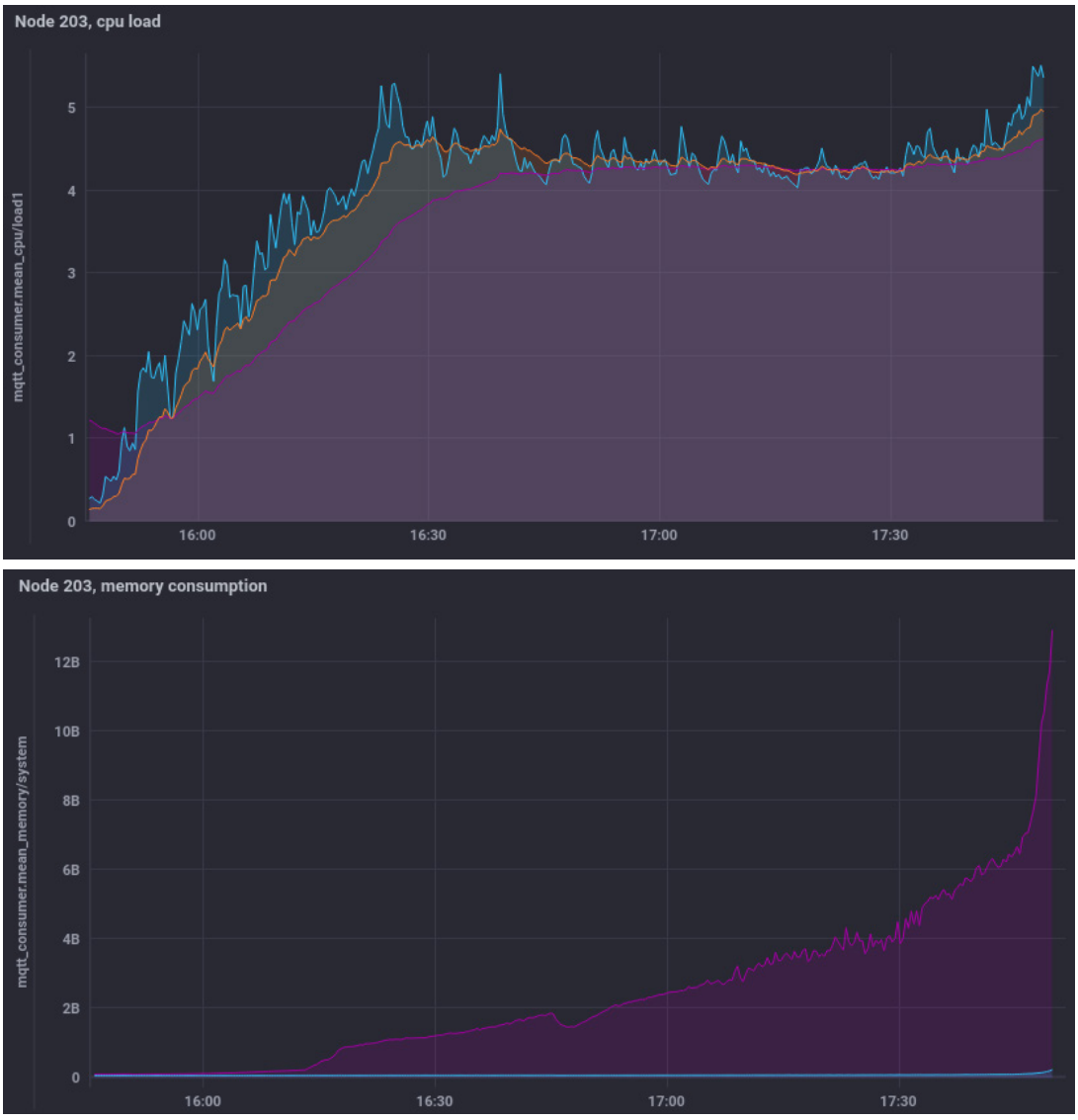

*(β) Connected clients for EMQX cluster*

Τα διαγράμματα Ε4, με Publish Delay = 3000ms, δηλαδή την μεγαλύτερη καθυστέρηση, παρουσιάζουν εξαιρετικά καλύτερη εικόνα για την CPU και την RAM, με τις τιμές των δύο να βρίσκονται κατά 50% κάτω από τις προηγούμενες προσομοιώσεις και στα δύο nodes, και καμία απώλεια μηνυμάτων. Το συνολικό πλήθος συνδεδεμένων clients είναι περίπου 2k.

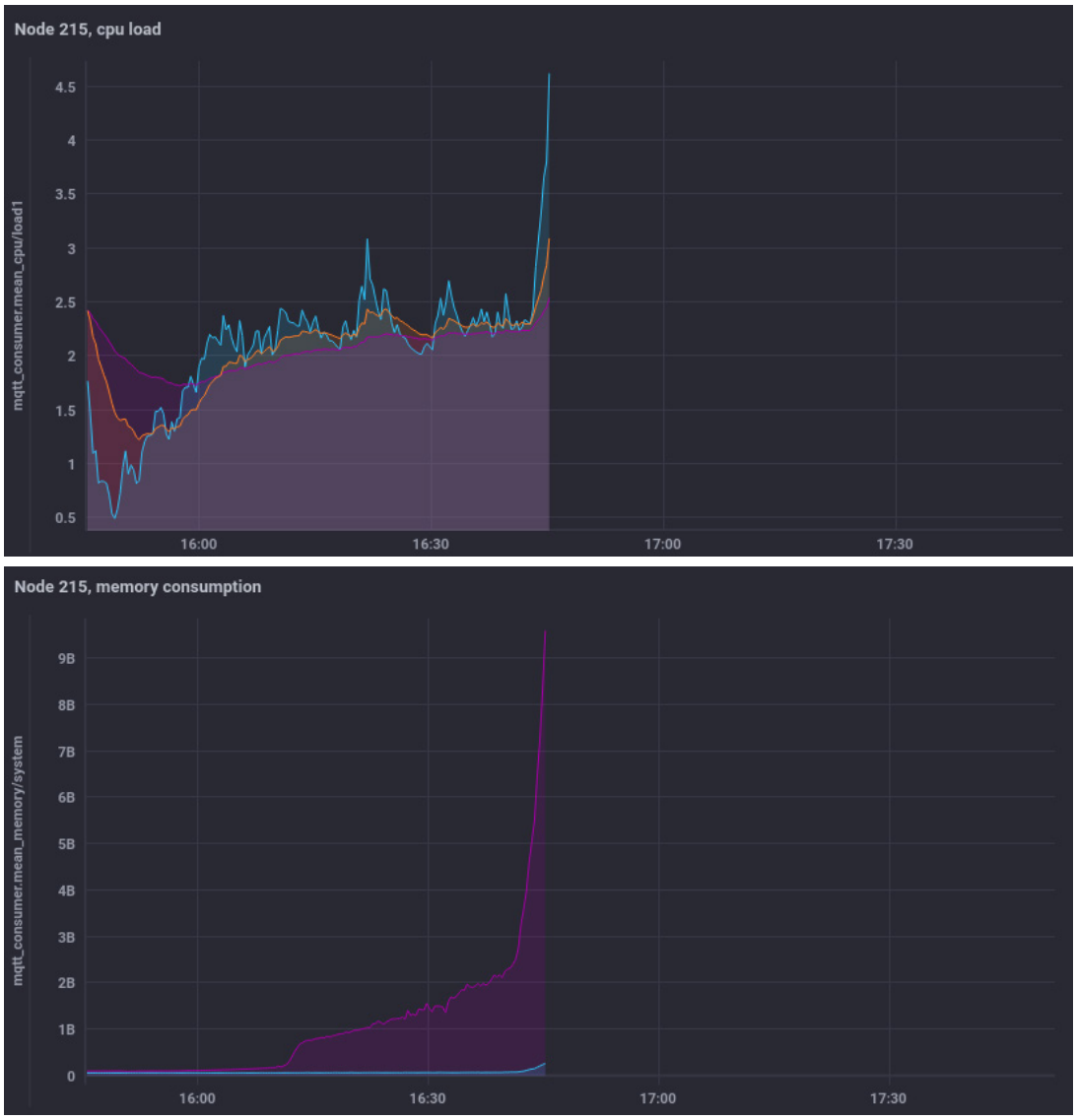
### *β. Spawning Delay = 0.5s*

Οι παρακάτω γραφικές παραστάσεις απεικονίζουν τη κατανάλωση πόρων CPU/MEM στο μηχάνημα του EMQX Broker, για τις τιμές Publish Delay = 50/100/500/3000ms.

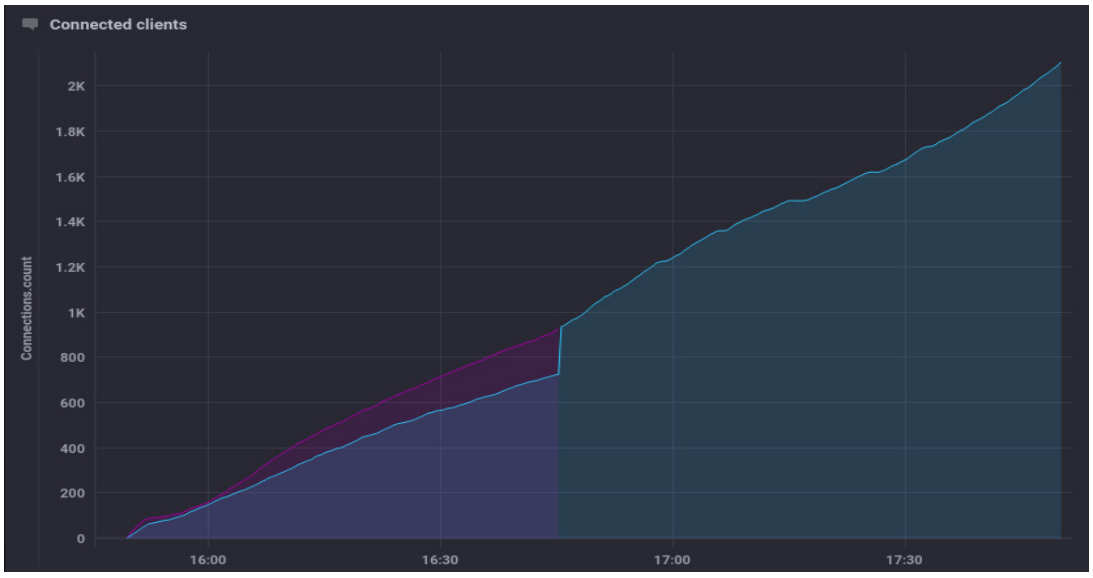
**Διάγραμματα Ε5:**
**Spawning Delay = 0.5s, Publish Delay = 50ms**




*(α1) CPU load and Memory Consumption for node 172.16.8.203*




*(α2) CPU load and Memory Consumption for node 172.16.8.215*



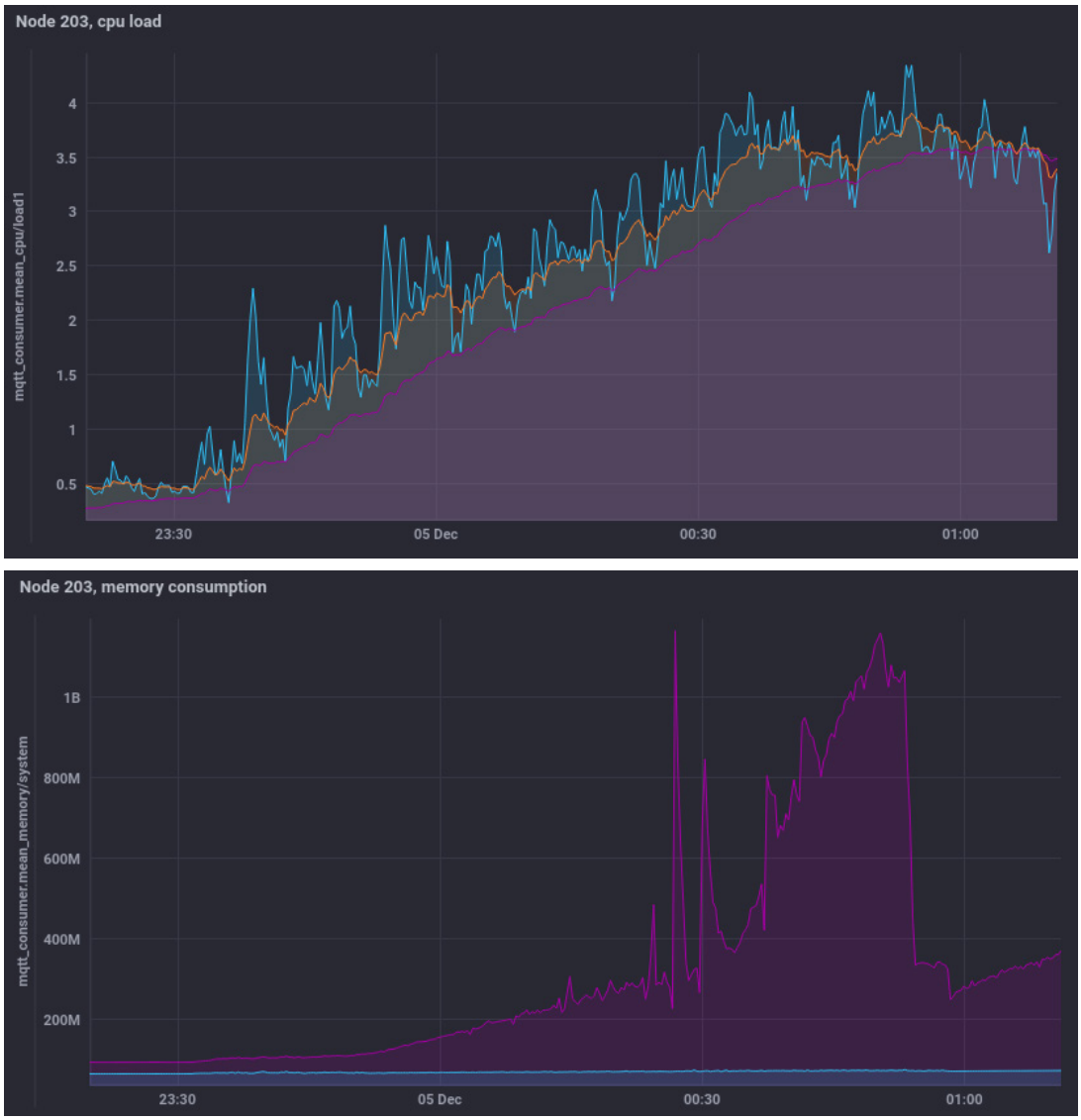

*(β) Clients connected for EMQX cluster*

Σε αυτή τη δεύτερη σειρά πειραμάτων, με το Spawning Delay να είναι στα 0.5s, παρατηρείται αρκετά επιβαρυμένη λειτουργία του EMQX Broker σε σχέση με την προηγουμενη τιμή Spawning Delay = 1s. Ιδιαίτερα σημαντική αύξηση παρουσιάζει η χρησιμοποιούμενη μνήμη RAM, κάτι που δείχνει ότι η διαχείριση μεγάλης συχνότητας νέων συνδέσεων με τον broker είναι γεγονός που δεσμεύει αρκετά την μνήμη.

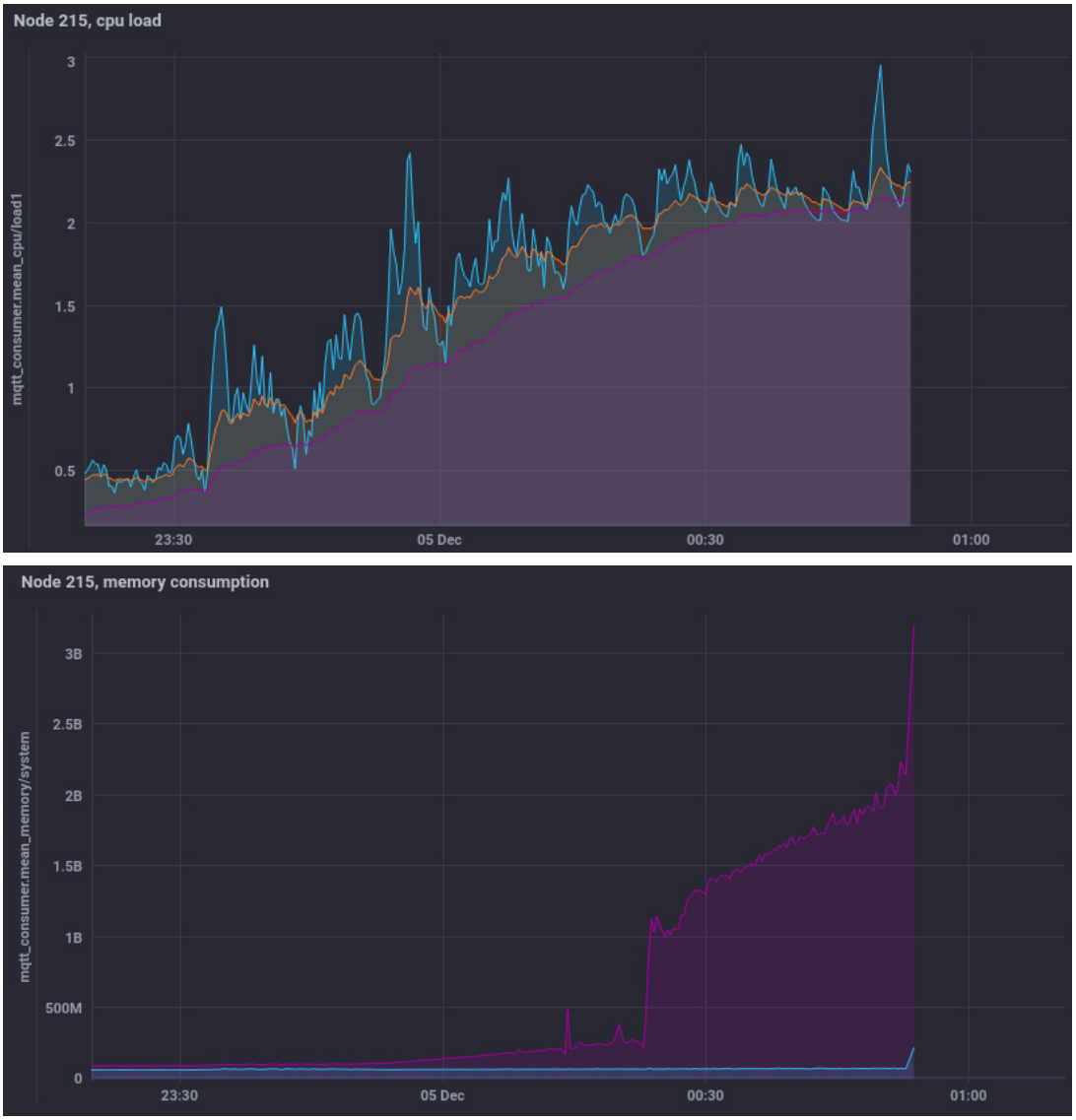
Στα διαγράμματα E5, στο node 1 η τιμή της CPU είναι ελαφρώς μεγαλύτερη του 4, δείχνοντας ήδη μια εικόνα επιβαρυμένης λειτουργίας, και η RAM παρουσιάζει εξαιρετικά απότομη αύξηση, ενώ στο node 2 η CPU κινείται γύρω από το 2.5 και μάλιστα σε μια ξαφνική αύξησή της στο 5 παρατηρείται διακοπή της λειτουργίας του node, κάτι που φαίνεται και από το διάγραμμα E5.β όπου η καμπύλη των συνδεδεμένων clients για το node διακόπτεται. Εκεί συμβαίνει μικρή απώλεια μηνυμάτων, μέχρι να δρομολογηθεί όλη η κίνηση προς το node 1, το οποίο από τη στιγμή αποσύνδεσης του δεύτερου node έχει CPU~=5, δείχνοντας έτσι την έντονη καταπόνηση των πόρων του για να ανταποκριθεί στην απότομη αλλαγή του δικτύου.

Το πλήθος των συνδεδεμένων clients είναι λίγο μεγαλύτερο του 2k.

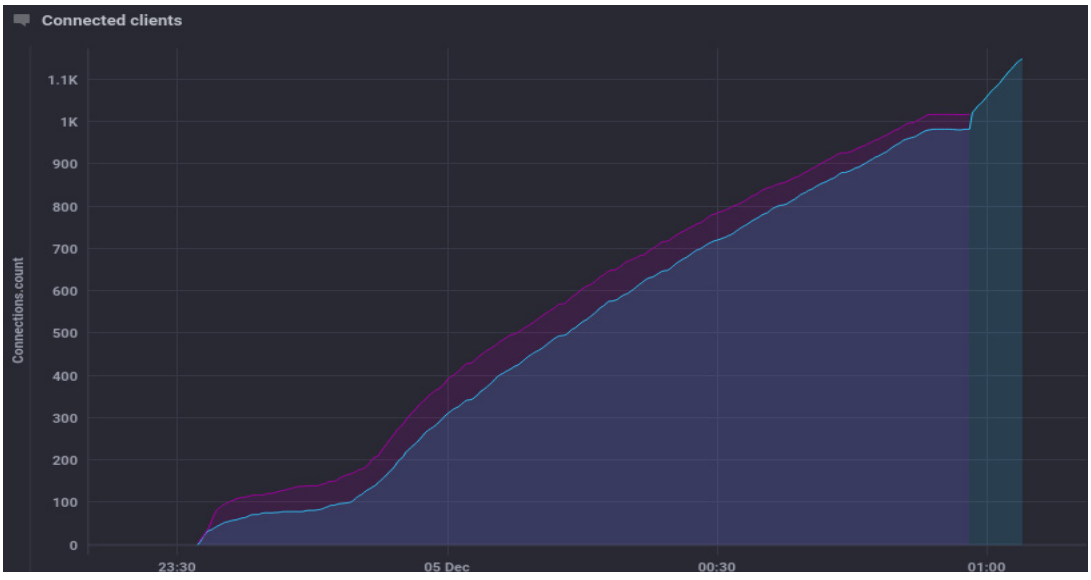
**Διάγραμματα Ε6:**
**Spawning Delay = 0.5s, Publish Delay = 100ms**




*(α1) CPU load and Memory Consumption for node 172.16.8.203*



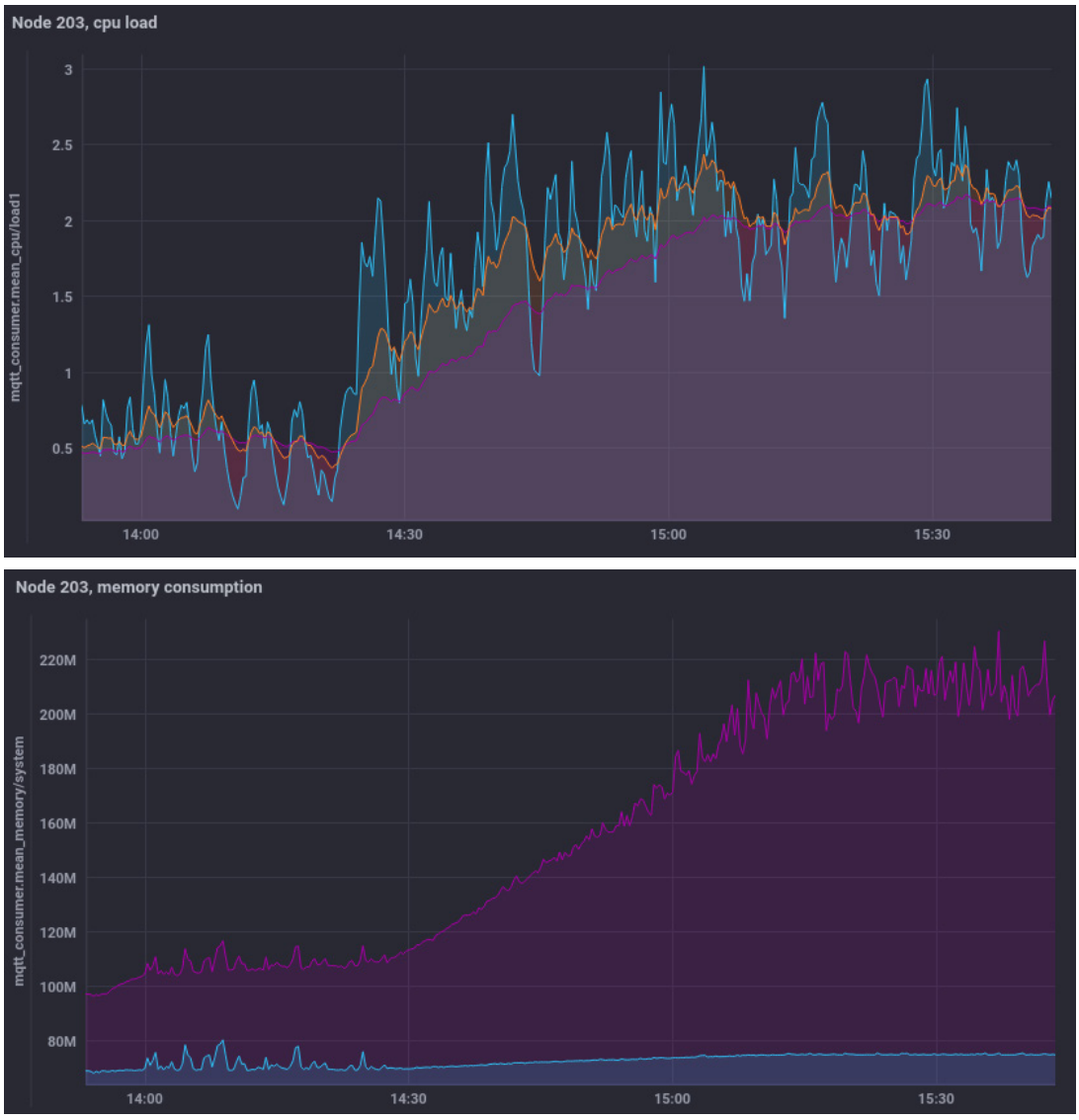

*(α2) CPU load and Memory Consumption for node 172.16.8.215*



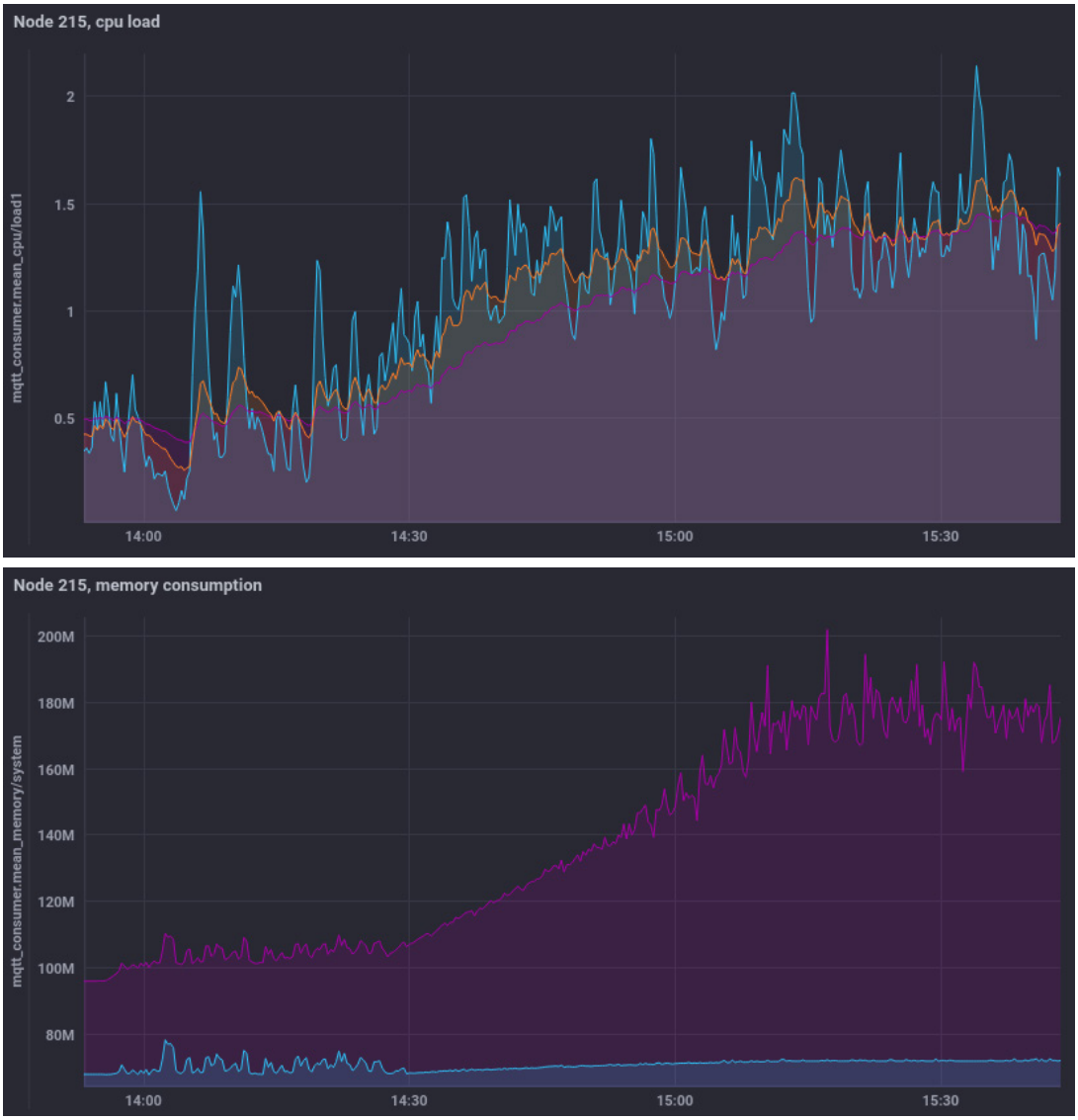

*(β) Clients connected for EMQX cluster*

Στα διαγράμματα Ε6, με Publish Delay = 100ms, η λειτουργία του broker κινείται στα φυσιολογικά επίπεδα πλήρους απασχόλησής του. Και αυτή η προσομοίωση δεν έχει βάρη. Η CPU στο node 1 δεν ξεπερνάει το 4, στο node 2 όσο αυξάνονται σημαντικά οι ενεργοί clients κυμαίνεται μεταξύ του 2-2.5, ενώ η RAM και στα δύο node αυξάνεται απότομα σε βάθος χρόνου στην προσομοίωση, ακολουθώντας την αυξανόμενη δημιουργία νέων συνδέσεων. Συνολικά οι συνδεδεμένοι clients ξεπέρασαν ελαφρώς τους 2k.

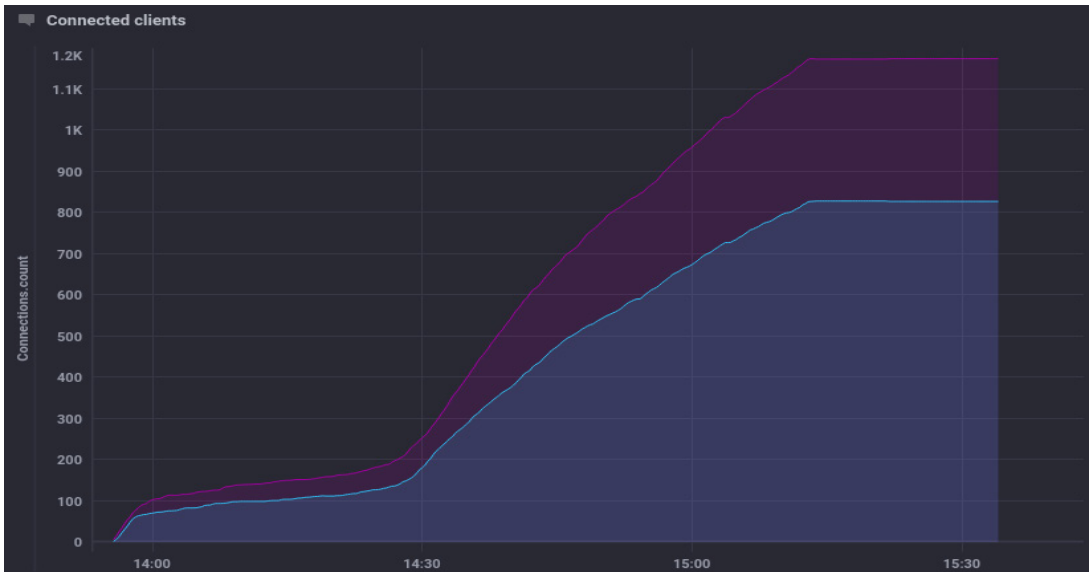
**Διάγραμματα Ε7:**
**Spawning Delay = 0.5s, Publish Delay = 500ms**




*(α1) CPU load and Memory Consumption for node 172.16.8.203*



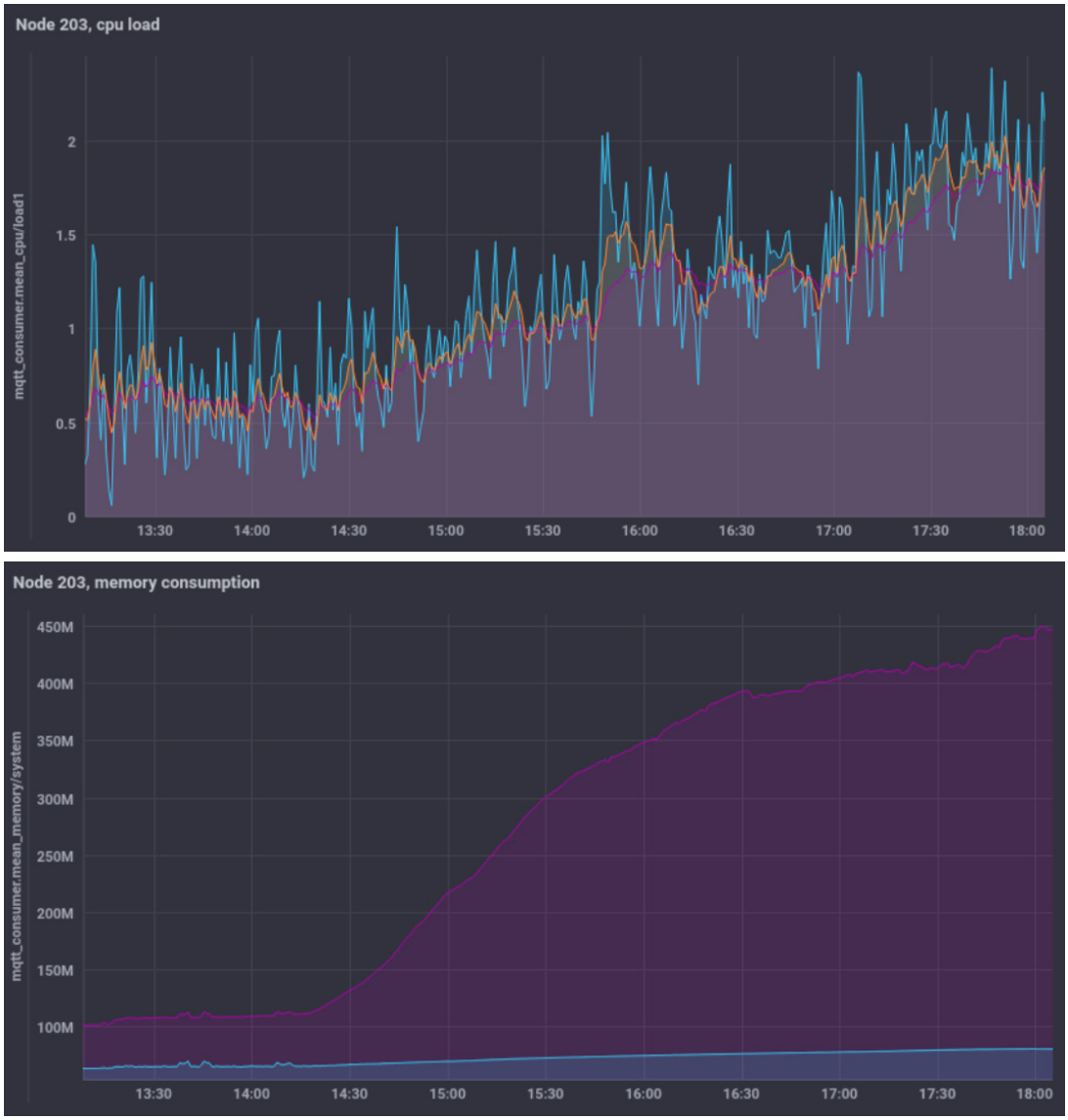

*(α2) CPU load and Memory Consumption for node 172.16.8.215*



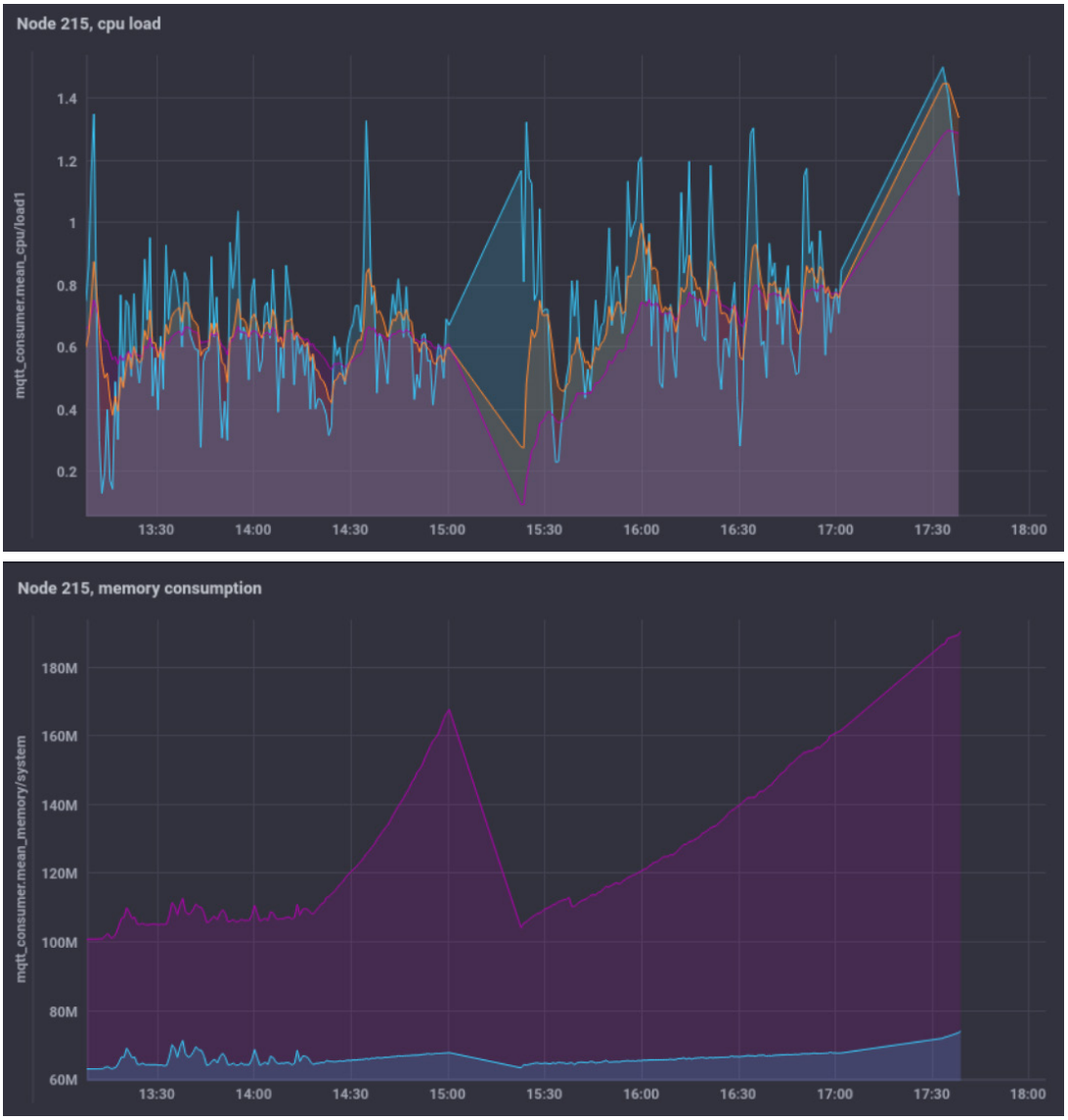

*(β) Clients connected for EMQX cluster*

Ακόμα περισσότερο στα διαγράμματα Ε7, και πάλι χωρίς βάρη στον load balancer, με ακόμα μεγαλύτερο Publish Delay = 500ms, οι τιμές CPU κινούνται κάτω από το μέγιστο όριό τους, δείχνοντας έτσι πολύ καλή ανταπόκριση του broker στη δεδομένη συνθήκη, και η RAM έχει μειωθεί σημαντικά σε σχέση με πρίν. Επίσης παρατηρείται αυξημένο πλήθος ενεργών συνδέσεων σε σχέση με τα προηγούμενα πειράματα, και στα δύο nodes αλλά ιδιαίτερα στο node 2 (2k+ ενεργοί clients). Να σημειωθεί επίσης πως δεν υπάρχουν χαμένα μηνύματα.

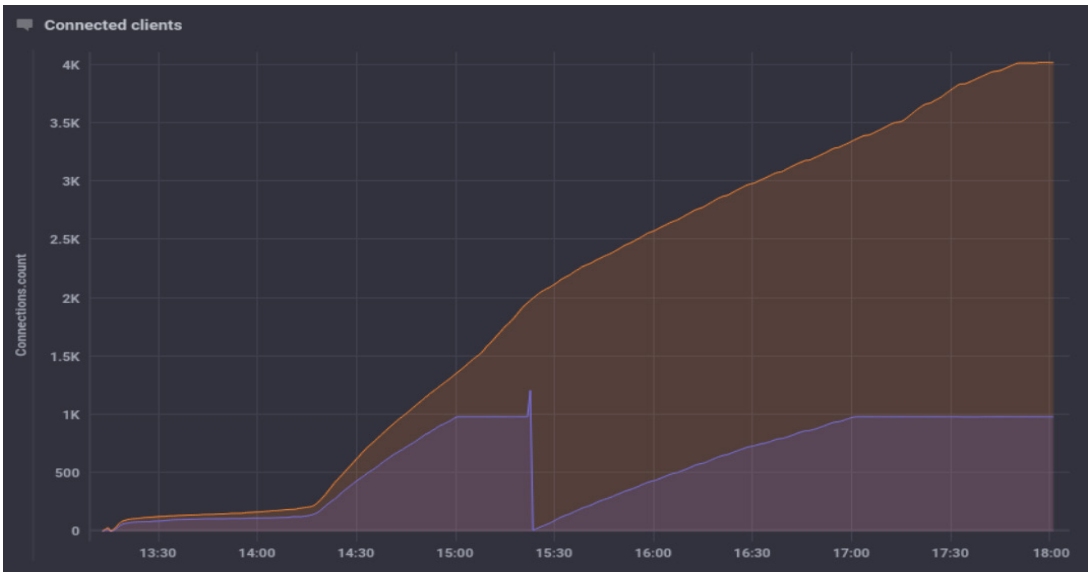
**Διάγραμματα Ε8:**
**Spawning Delay = 0.5s, Publish Delay = 3000ms**




*(α1) CPU load and Memory Consumption for node 172.16.8.203*




*(α2) CPU load and Memory Consumption for node 172.16.8.215*




*(β) Clients connected for EMQX cluster*

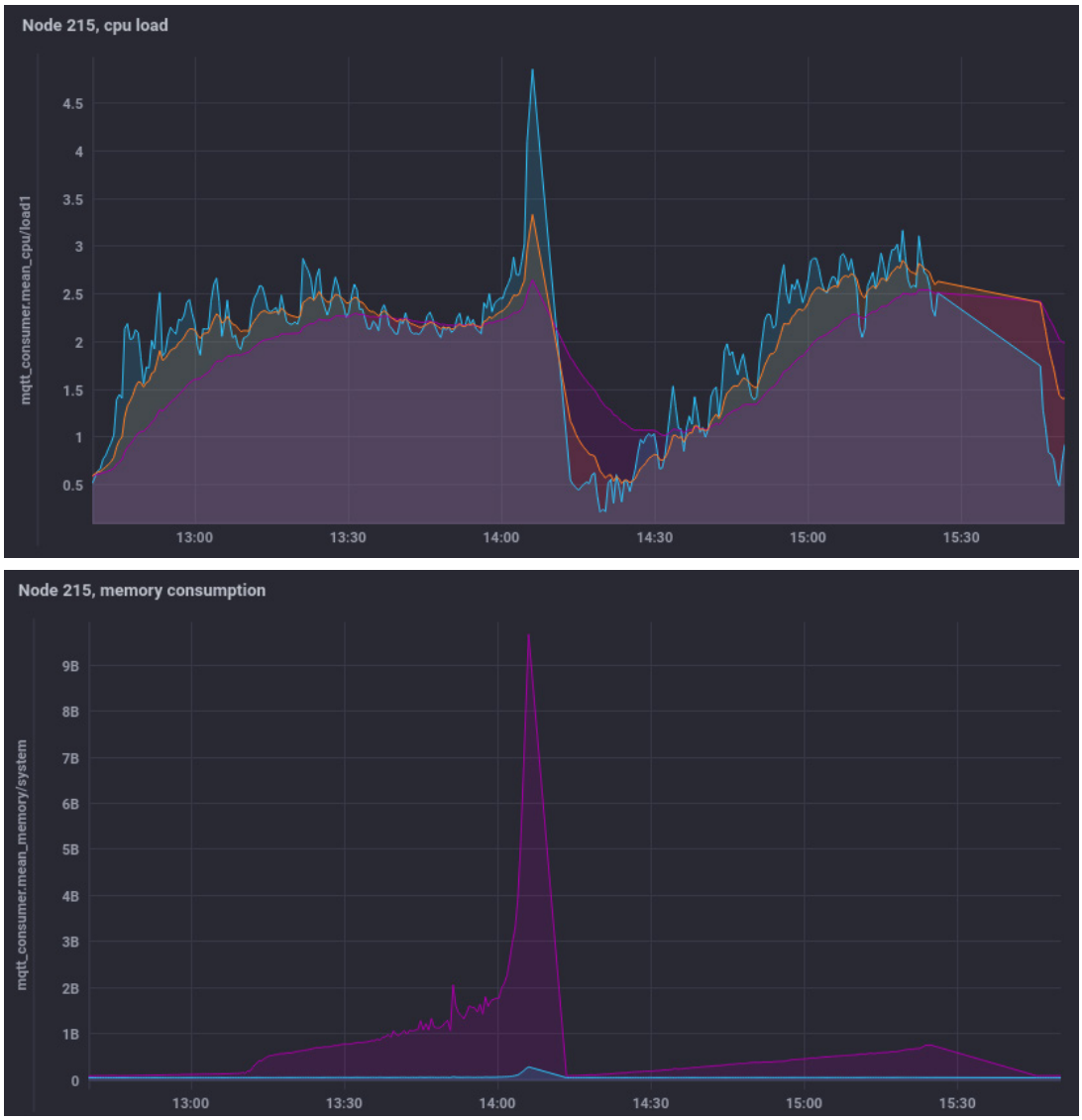
Τέλος, στα διαγράμματα E8, όπου έχει αυξηθεί το Publish Delay = 3000ms, τα επίπεδα CPU και RAM έχουν επανέλθει σε φυσιολογικά επίπεδα. Παρατηρείται αποσύνδεση του node 2 και επανασύνδεση, η οποία ωστόσο δεν οφείλεται σε ζήτημα φόρτου. Το πλήθος των clients σταδιακά φτάνει τους 4k. Δεν υπάρχει απώλεια μηνυμάτων ούτε σε αυτό το σενάριο.

### *γ. Spawning Delay = 3s*

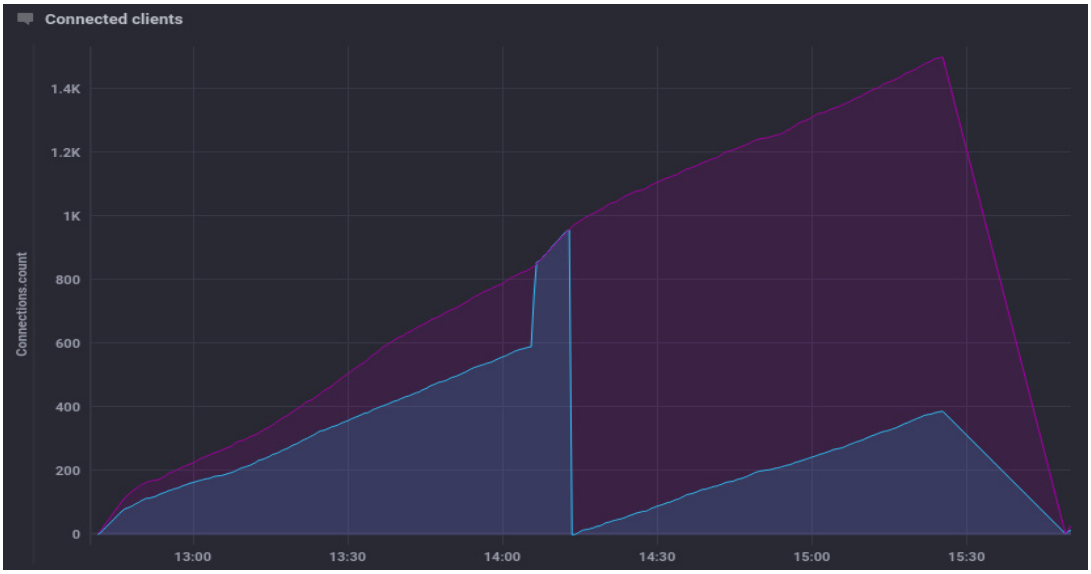
Οι παρακάτω γραφικές παραστάσεις απεικονίζουν τη κατανάλωση πόρων CPU/MEM στο μηχάνημα του EMQX Broker, για τις τιμές Publish Delay = 50/100/500/3000ms.

**Διάγραμματα Ε9:**
**Spawning Delay = 3s, Publish Delay = 50ms**


*(α1) CPU load and Memory Consumption for node 172.16.8.203*




*(α2) CPU load and Memory Consumption for node 172.16.8.215*

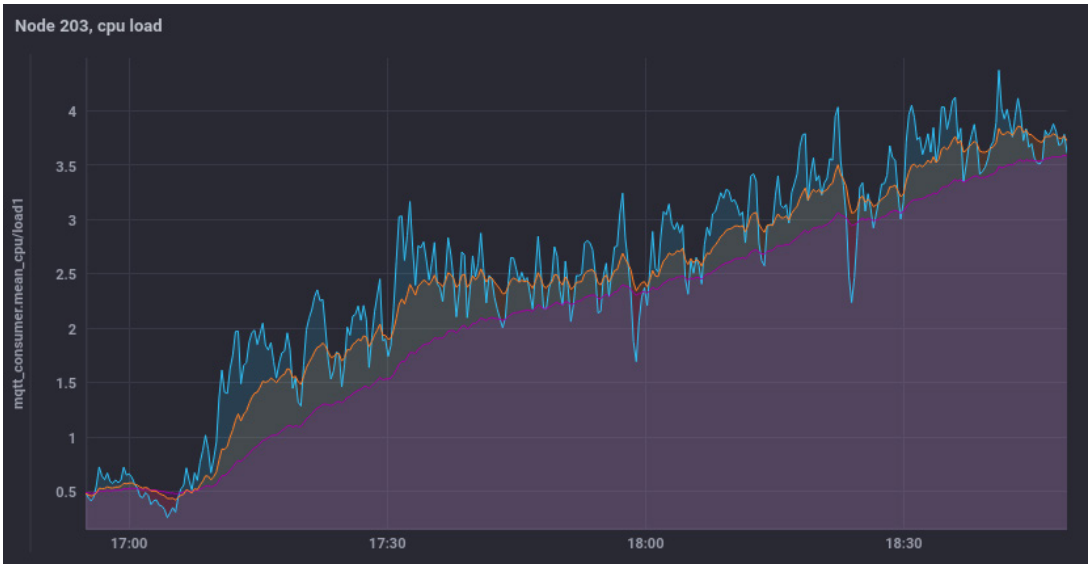


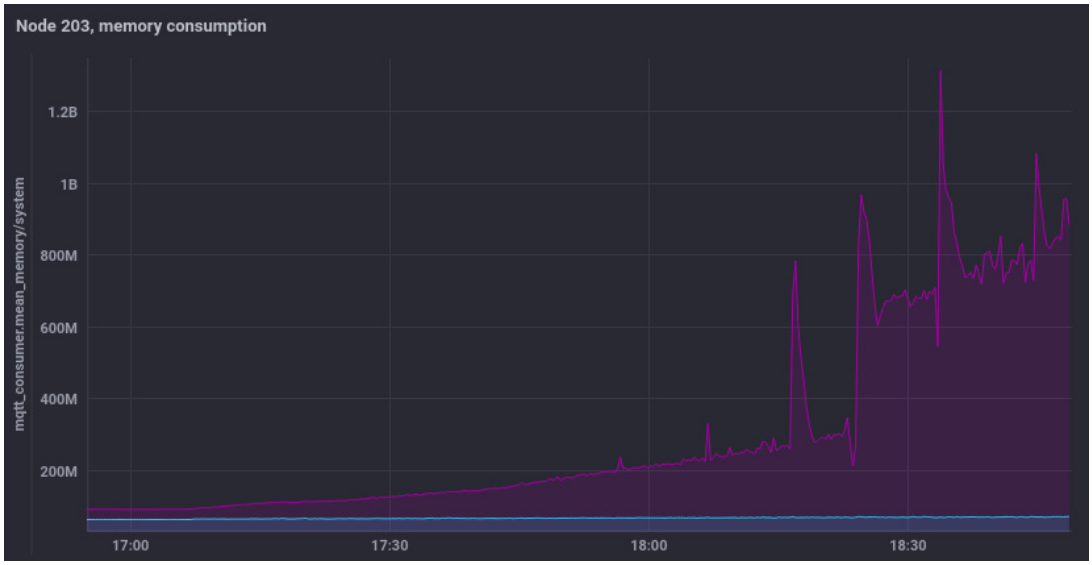

*(β) Clients connected for EMQX cluster*

Η τρίτη σειρά πειραμάτων στον EMQX Broker γίνεται με Spawning Delay = 3s. Όπως φαίνεται στα διαγράμματα Ε9, για πολύ μικρό Publish Delay = 50ms η CPU του broker επιβαρύνεται αρκετά, παίρνοντας τιμές κοντά στο 5 για το node 1 και κοντά στο 3 για το node 2. Η μνήμη RAM είναι επίσης σε υψηλά επίπεδα χρήσης, και μάλιστα στο node 2 κάποια στιγμή αυξάνεται πολύ απότομα και το node βγαίνει εκτός λειτουργίας, σημείο που αντίστοιχα στο node 1 αρχίζει να αυξάνεται επίσης η χρήση της RAM, δεδομένης της μετακίνησης όλου του προς διαχείριση φόρτου σε αυτό. Μετά από λίγο το δεύτερο node ξανασυνδέεται επιτυχώς στο δίκτυο, και ο load balancer δρομολογεί και πάλι προς αυτό νέους clients, επαναφέροντας την ισορροπία. Απώλεια μηνυμάτων εμφανίζεται μόνο τη στιγμή αποσύνδεσης του δεύτερου node.

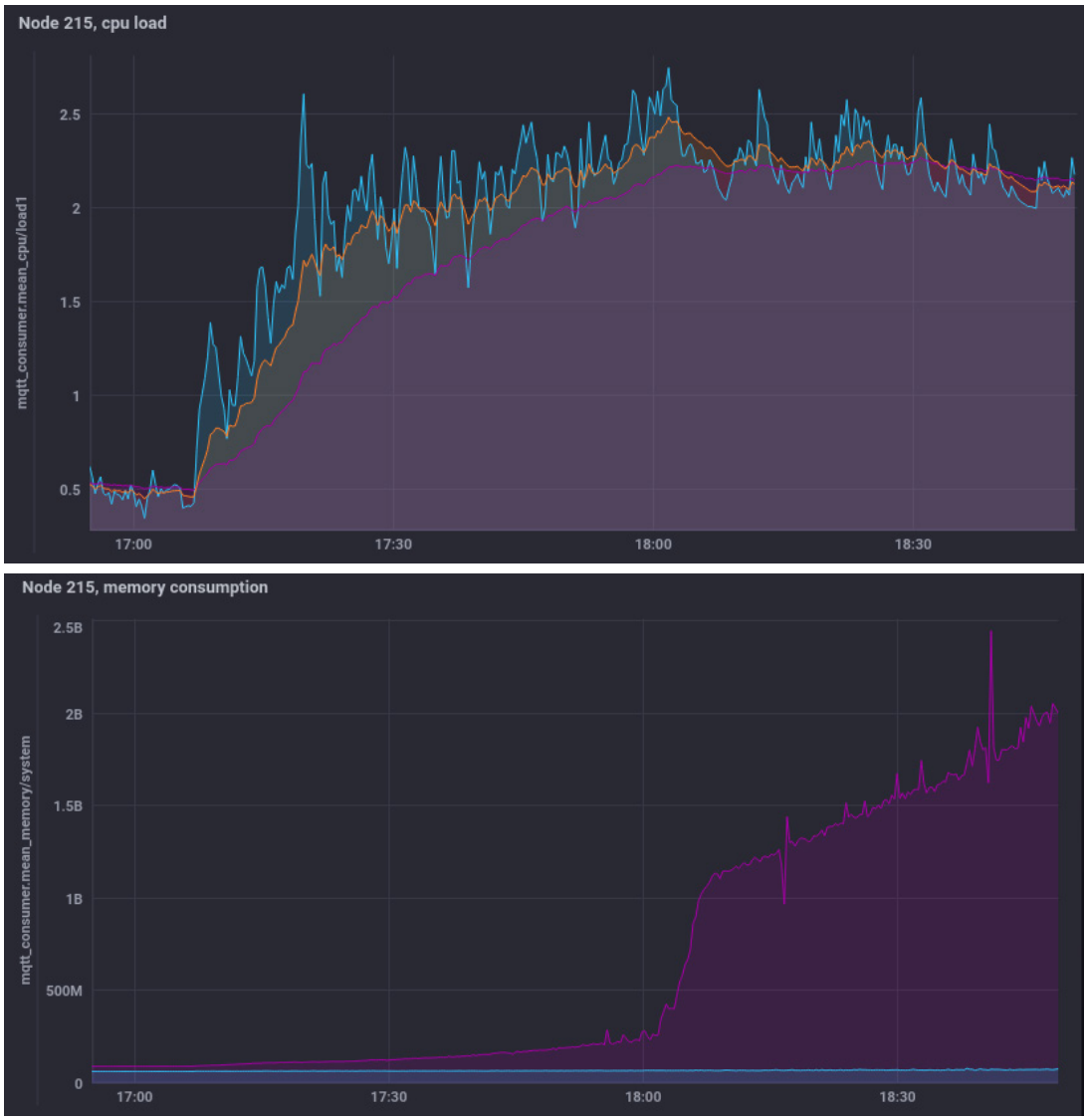
Το πλήθος των ενεργών clients κυμαίνεται στα 2k.

**Διάγραμματα Ε10:**
**Spawning Delay = 3s, Publish Delay = 100ms**

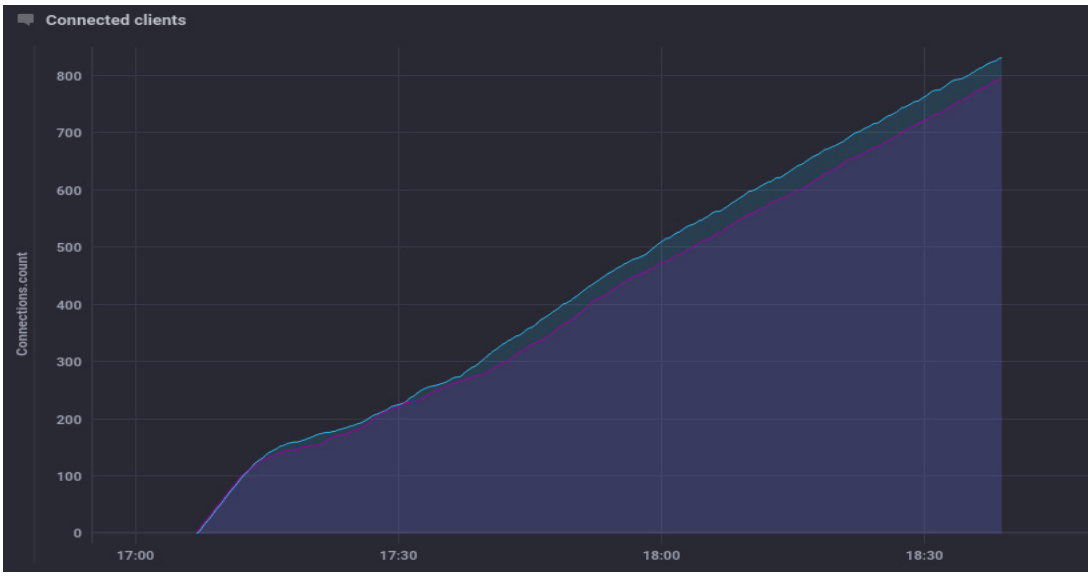





*(α1) CPU load and Memory Consumption for node 172.16.8.203*



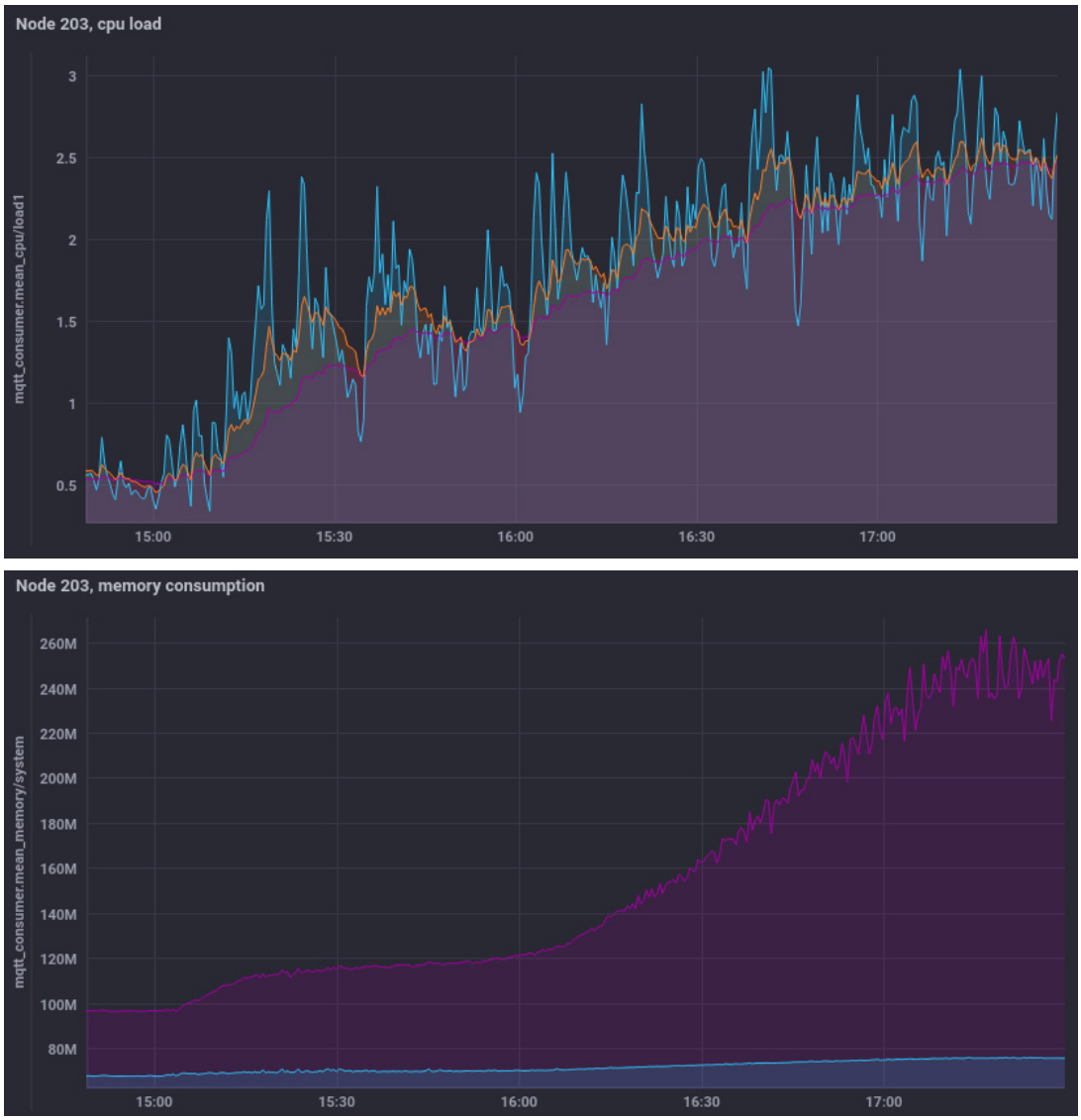

*(α2) CPU load and Memory Consumption for node 172.16.8.215*



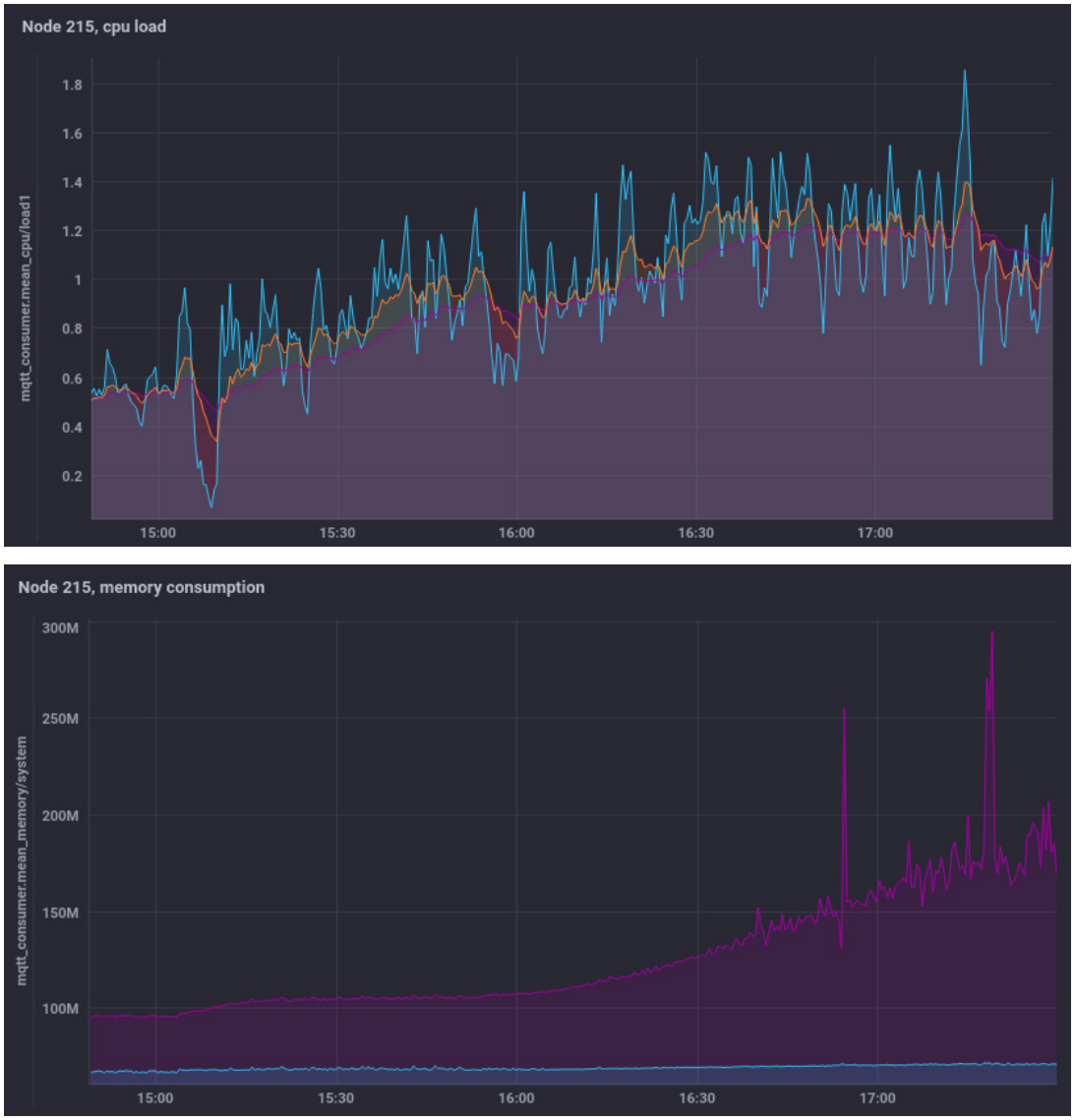

*(β) Clients connected for EMQX cluster*

Στα διαγράμματα E10 η λειτουργία του broker είναι μέσα στα φυσιολογικά πλαίσια, χωρίς κάποια ορατή δυσλειτουργία. Η CPU του node 2 ξεπερνά τις φυσιολογικές τιμές της και ανάλογα αυξάνεται και η χρήση της μνήμης RAM, ωστόσο δεν παύει να ανταποκρίνεται επαρκώς στο δίκτυο, χωρίς απώλειες μηνυμάτων. Οι ενεργοί clients δεν ξεπερνούν τους 2000.

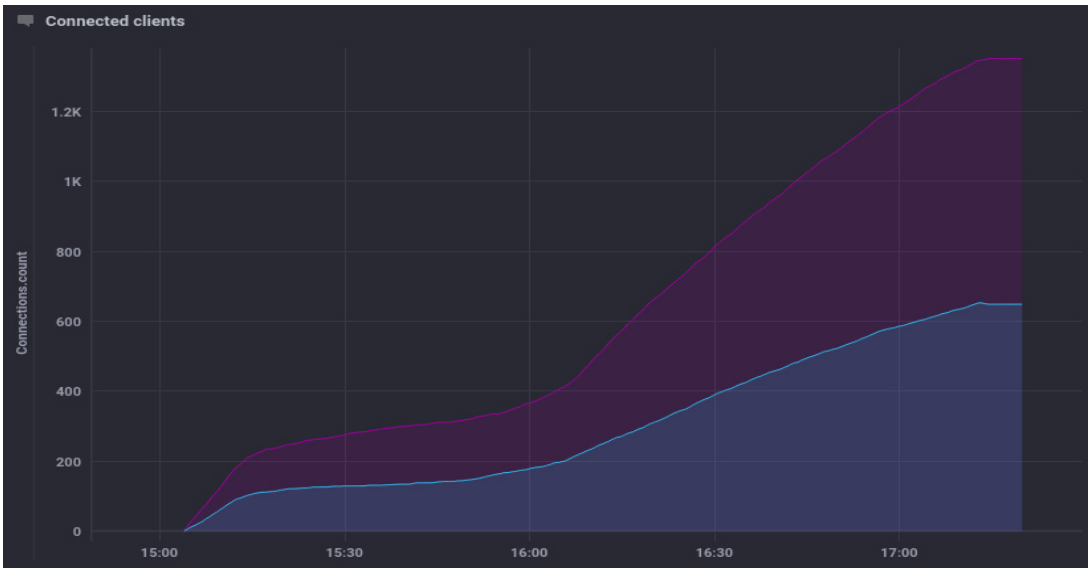
**Διάγραμματα E11:**
**Spawning Delay = 3s, Publish Delay = 500ms**




*(α1) CPU load and Memory Consumption for node 172.16.8.203*



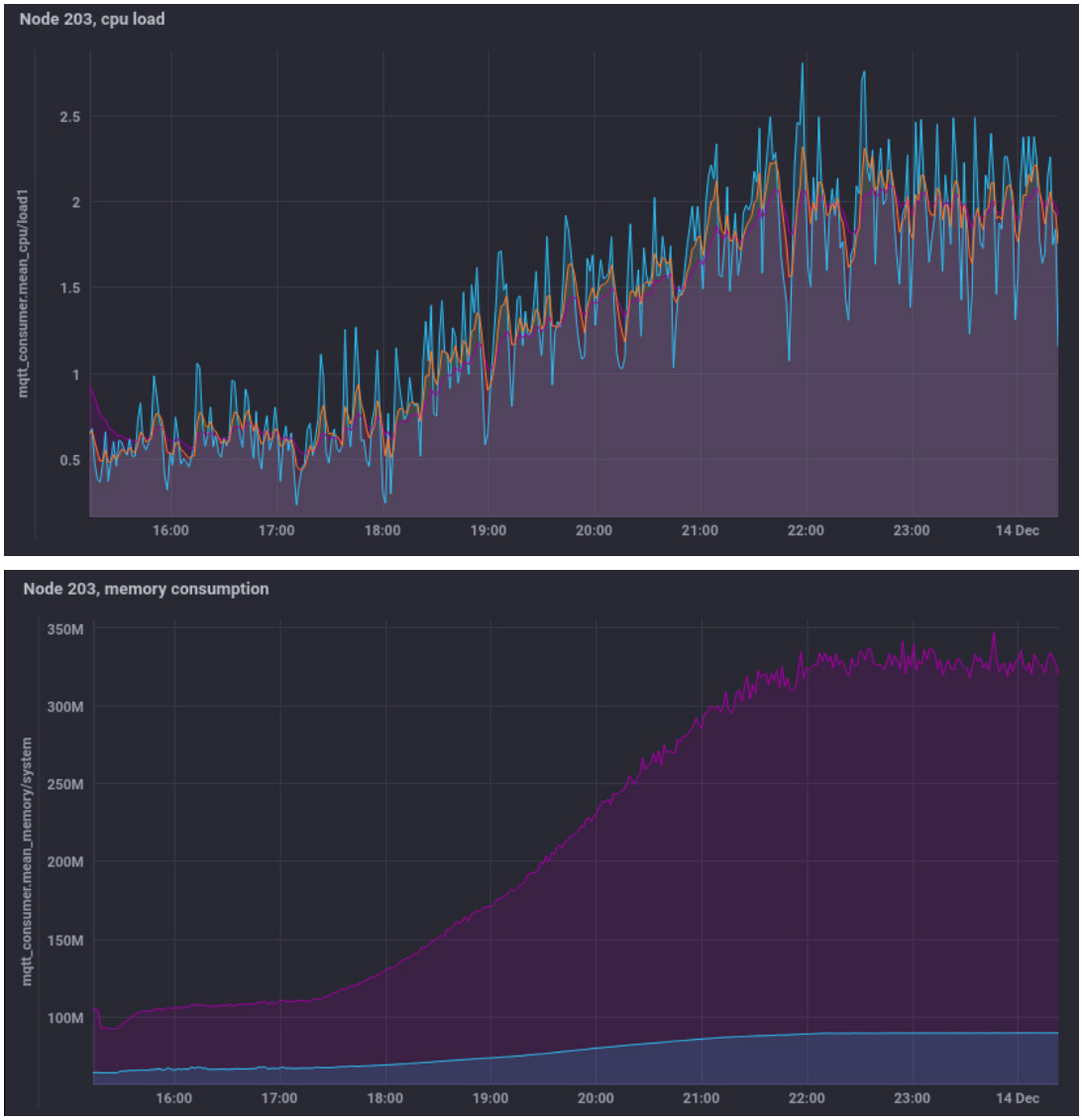

*(α2) CPU load and Memory Consumption for node 172.16.8.215*



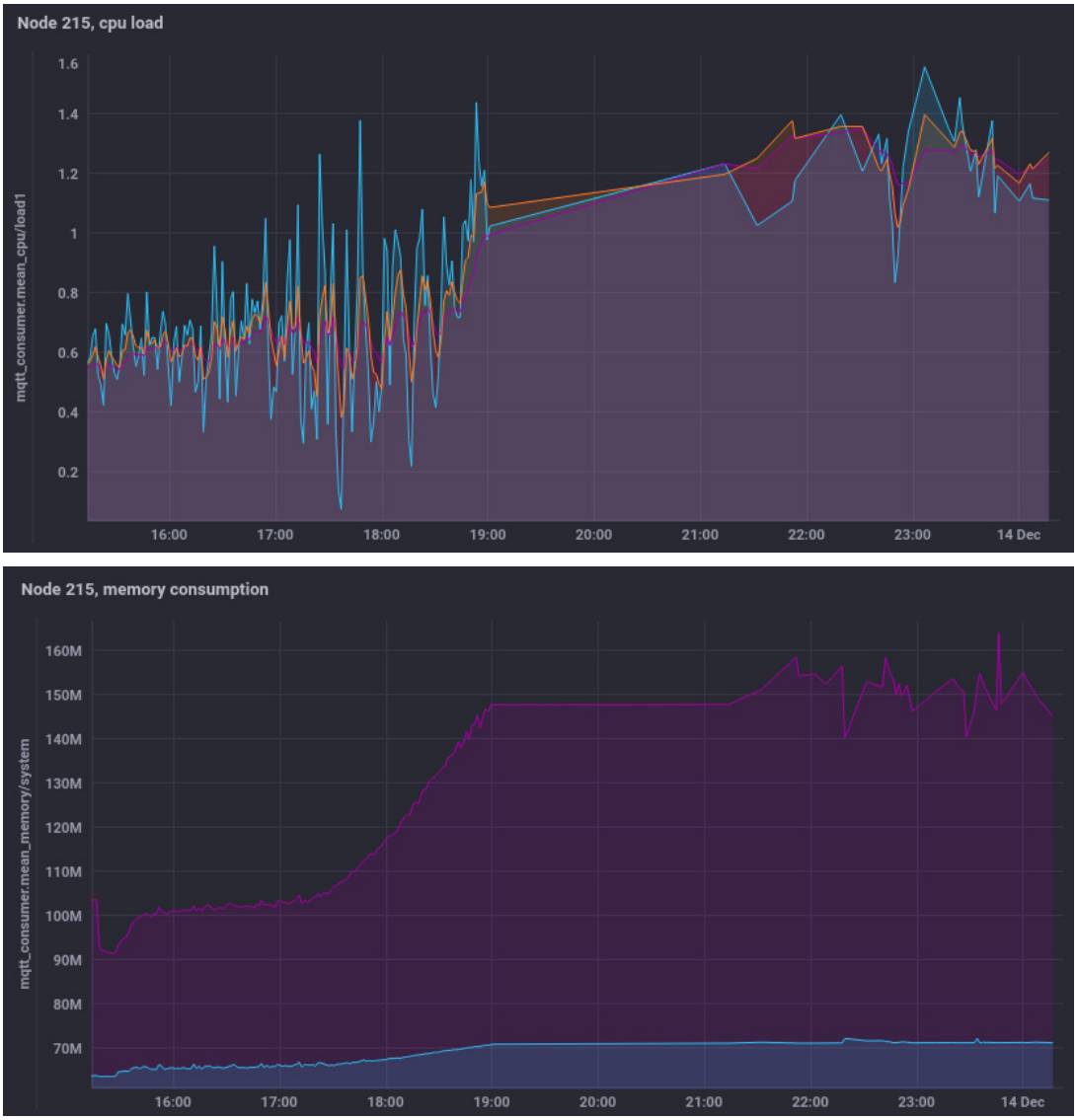

*(β) Clients connected for EMQX cluster*

Στο πείραμα E11, που γίνεται με βάρη στο load balancing (**όπως και όλες οι επόμενες προσομοιώσεις**), με Publish Delay = 500ms, η απόδοση του broker βελτιώνεται αισθητά και στην χρήση CPU και στην RAM, καθώς καμία τιμή δεν ξεπερνά τα φυσιολογικά όρια. Αυτό είναι λογικό δεδομένου ότι οι παράμετροι της προσομοίωσης Publish Delay και Spawning Delay έχουν αυξηθεί αρκετά, δίνοντας μεγαλύτερα χρονικά περιθώρια στην αντιμετώπιση κάθε νέας σύνδεσης από τον broker. Οι ενεργοί clients φτάνουν τους 2000.

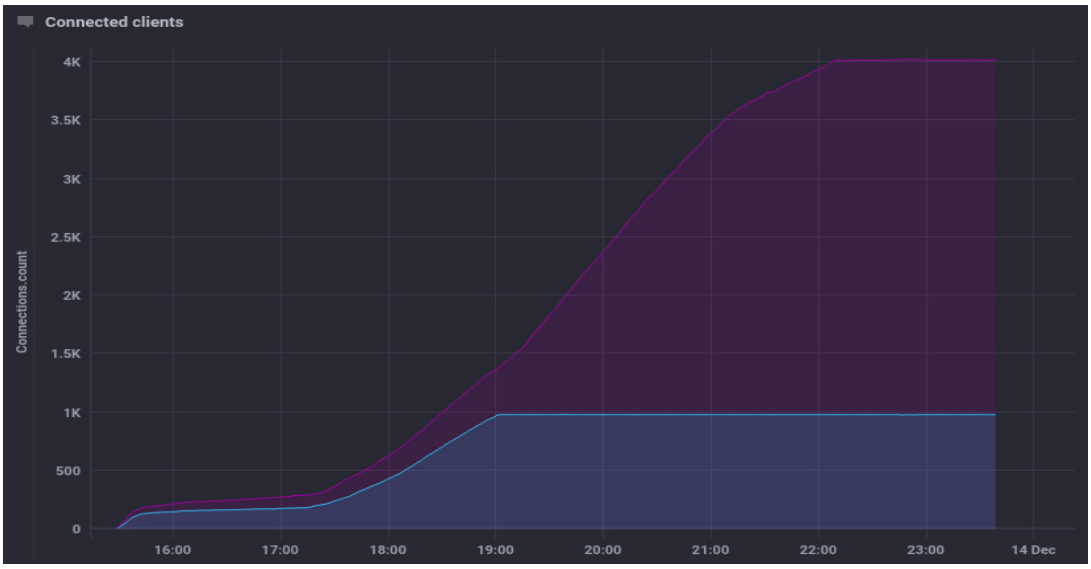
**Διάγραμματα E12:**
**Spawning Delay = 3s, Publish Delay = 3000ms**




*(α1) CPU load and Memory Consumption for node 172.16.8.203*




*(α2) CPU load and Memory Consumption for node 172.16.8.215*




*(β) Clients connected for EMQX cluster*

Στο τελευταίο πείραμα αυτής της ενότητας, επιβεβαιώνεται ότι όσο μειώνεται και το Publish Delay, με αρχή το πείραμα E10 και με ακόλουθα τα E11 και E12, ο broker λειτουργεί με όλο και καλύτερη απόδοση τόσο στα επίπεδα της CPU όσο και της RAM, χωρίς απώλειες συνδέσεων και μηνυμάτων. Το πλήθος των συνδεδεμένων clients φτάνει μέχρι και τους 5000.

Παρακάτω παρουσιάζονται συγκεντρωτικά οι μετρήσεις που θα ληφθούν υπόψιν στην εξαγωγή συμπερασμάτων.

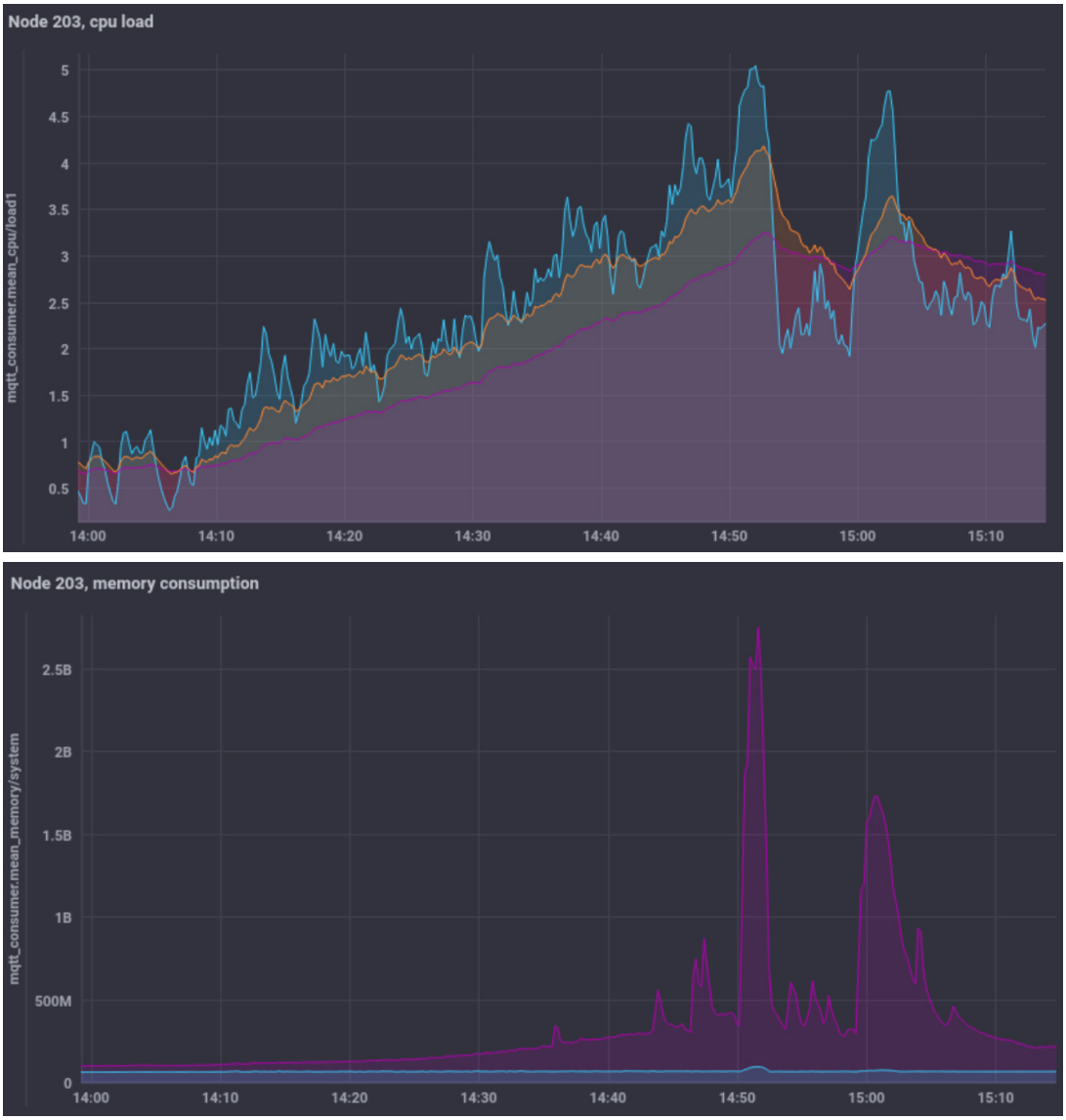
| **Προσομοίωση/Spawning Delay/ Publish Delay** | | | **Maximum number of clients (approximately)** | **Node 2 failure/ reconnection** |
|---|---|---|---|---|
| **E1** | **1s** | **50ms** | 3500 | ✓ / ✓ |
| **E2** | | **100ms** | 2000 | - |
| **E3** | | **500ms** | 2000 | - |
| **E4** | | **3000ms** | 2000 | - |
| **E5** | **0.5s** | **50ms** | 2000 | ✓ / - |
| **E6** | | **100ms** | 2000 | - |
| **E7** | | **500ms** | 2000 | - |
| **E8** | | **3000ms** | 4000 | ✓ / ✓ |
| **E9** | **3s** | **50ms** | 2000 | ✓ / ✓ |
| **E10** | | **100ms** | 2000 | - |
| **E11** | | **500ms** | 2000 | - |
| **E12** | | **3000ms** | 5000 | - |

*Πίνακας 4: Συγκεντρωτικός πίνακας μετρήσεων των προσομοιώσεων E1-E12 για τον EMQX Broker*

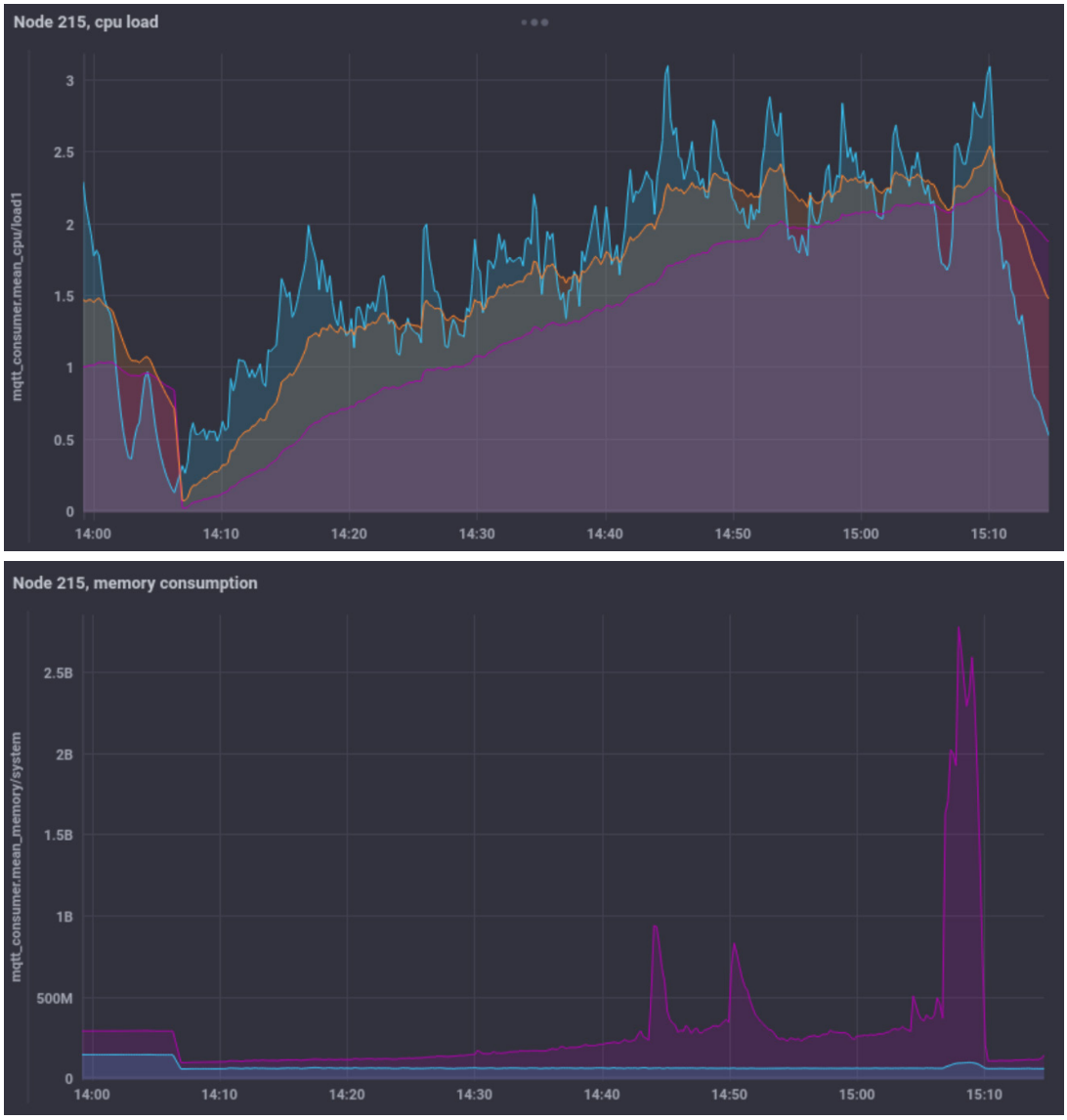
### *δ. Τρεις ακόμα προσομοιώσεις*

Οι παρακάτω γραφικές παραστάσεις ανταποκρίνονται στα σενάρια 13, 14, 15 του Πίνακα 2. Είναι τρία από τα προηγούμενα σενάρια προσομοίωσης, και συγκεκριμένα τα 2, 9 και 8, με τέσσερα VM αναπαραγωγής νέων συνδέσεων για την παρατήρηση της ανταπόκρισης του EMQX Broker σε ακόμα μεγαλύτερο δικτυακό φόρτο.

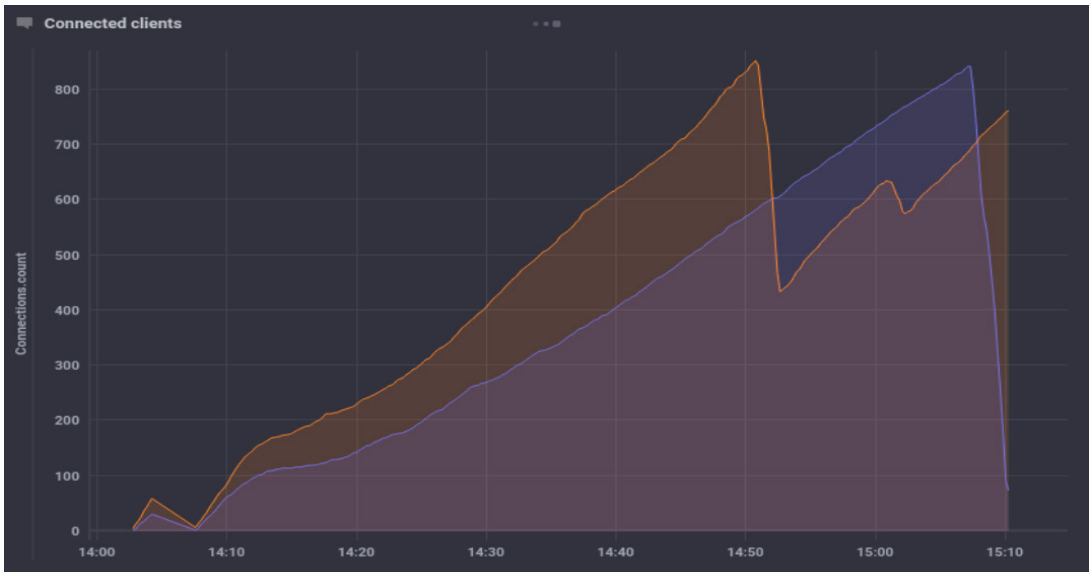
**Διάγραμματα E13:**
**Spawning Delay = 1s, Publish Delay = 100ms**




*(α1) CPU load and Memory Consumption for node 172.16.8.203*

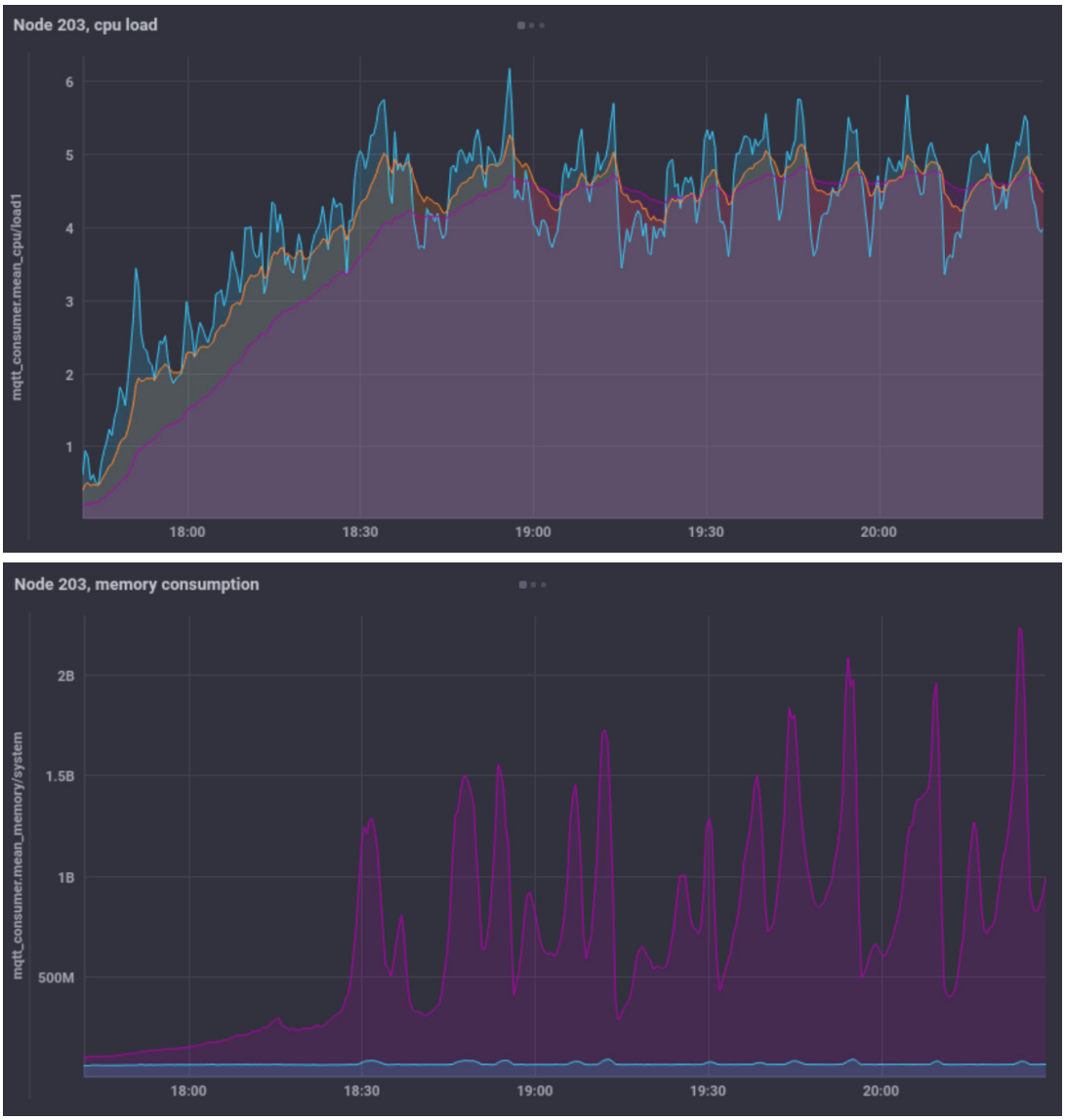



*(α2) CPU load and Memory Consumption for node 172.16.8.215*

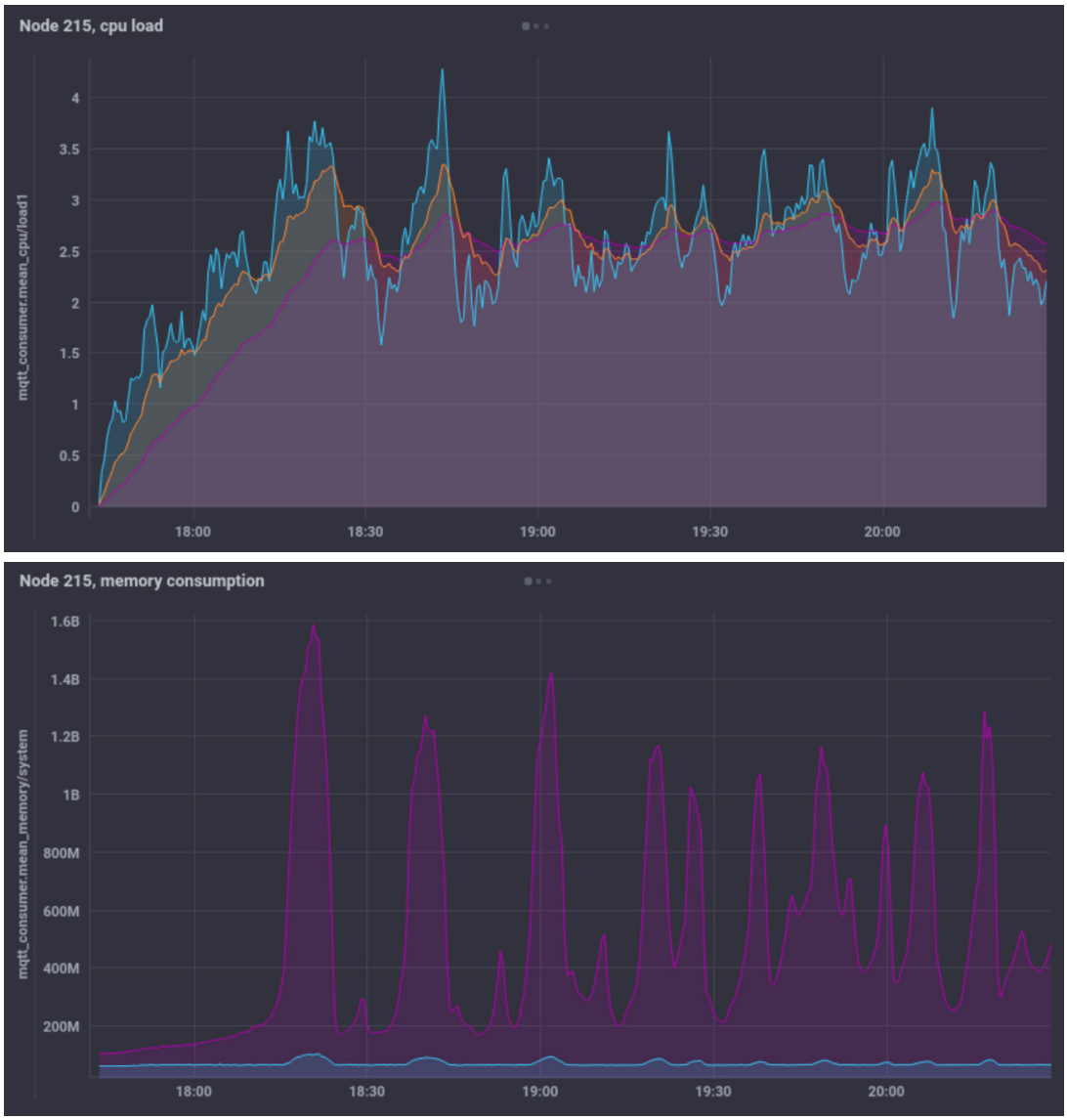



*(β) Clients connected for EMQX cluster*

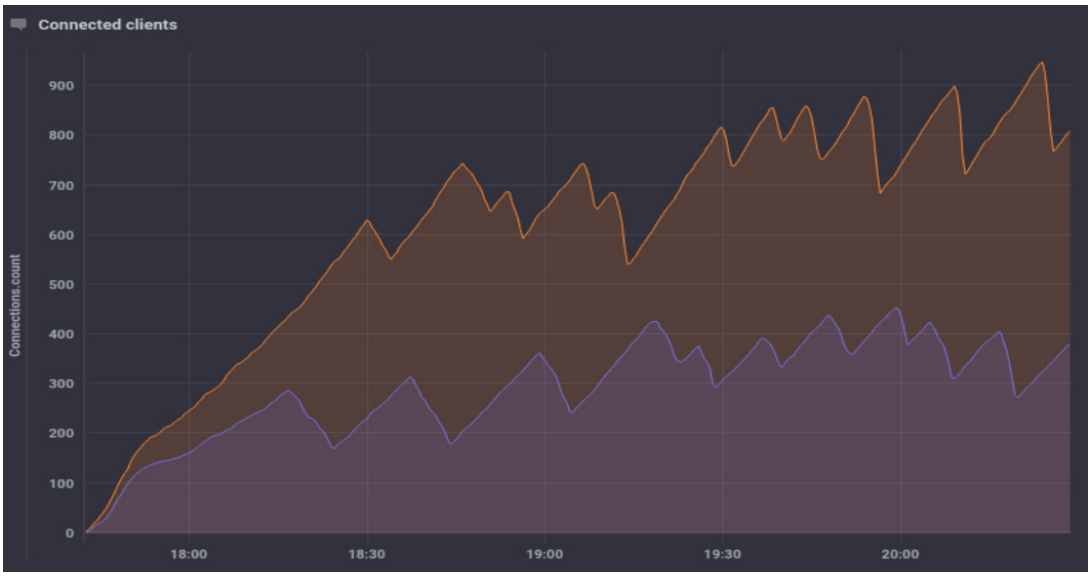
**Διάγραμματα E14:**
**Spawning Delay = 3s, Publish Delay = 50ms**




*(α1) CPU load and Memory Consumption for node 172.16.8.203*

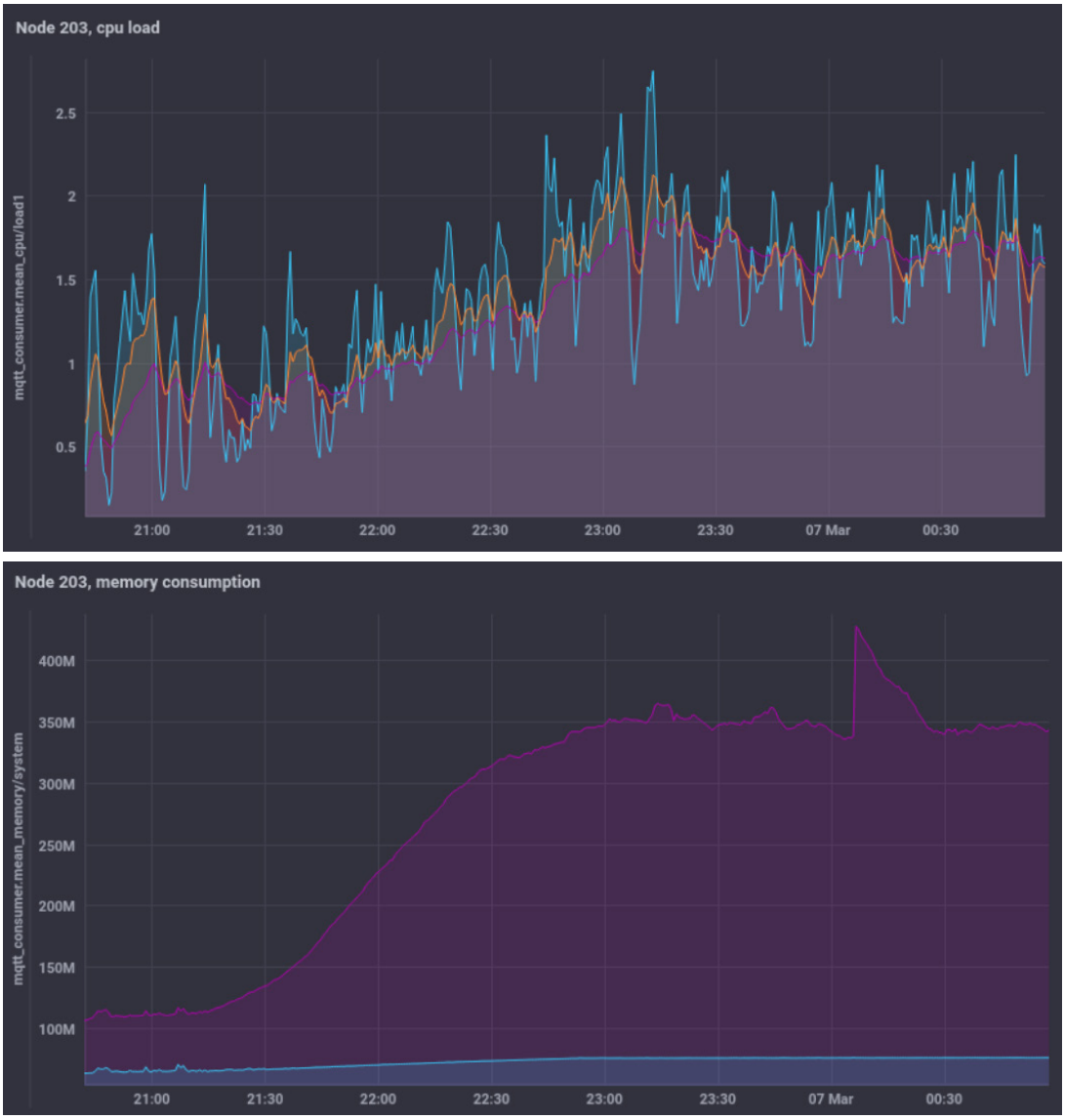



*(α2) CPU load and Memory Consumption for node 172.16.8.215*

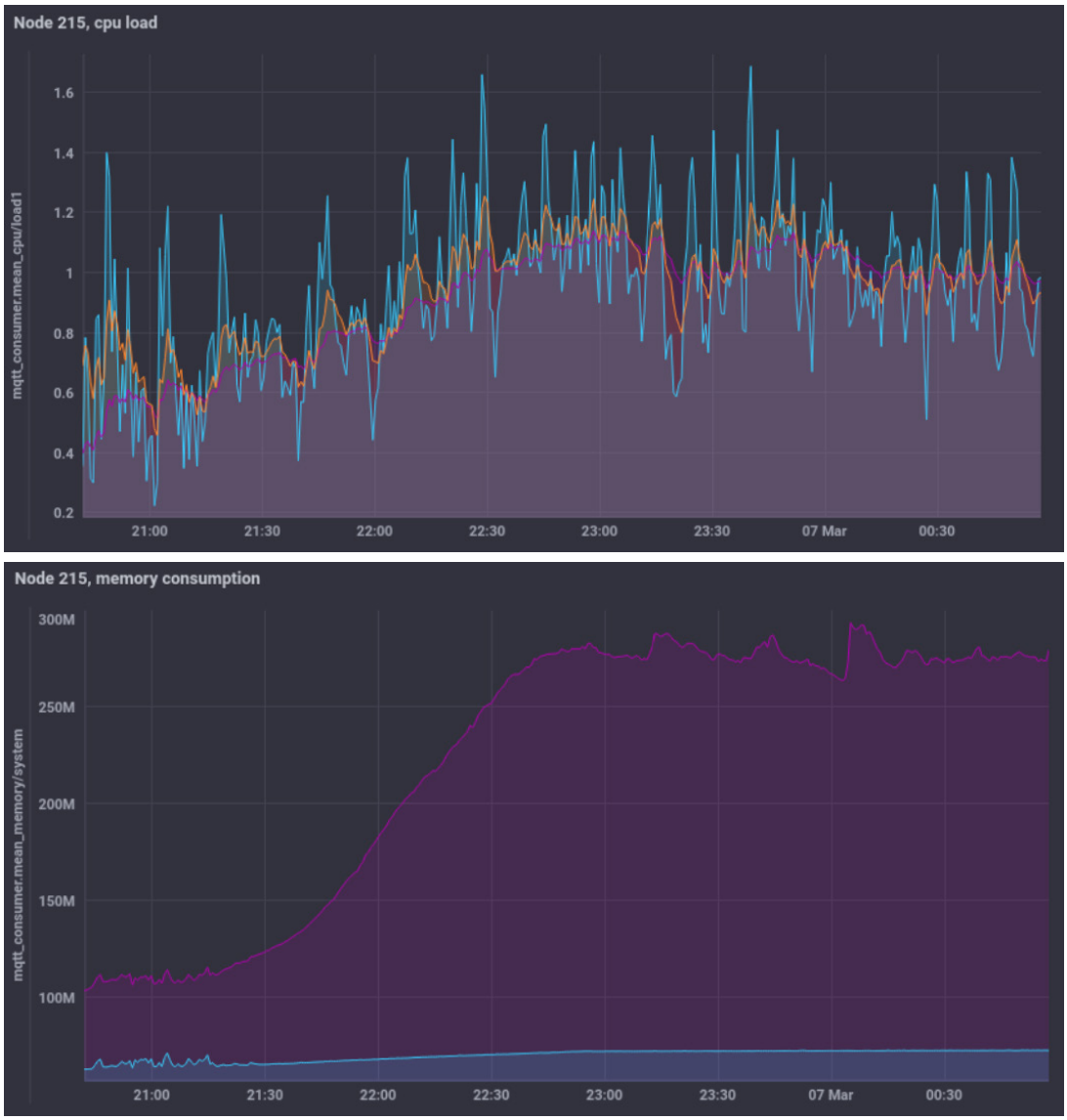



*(β) Clients connected for EMQX cluster*

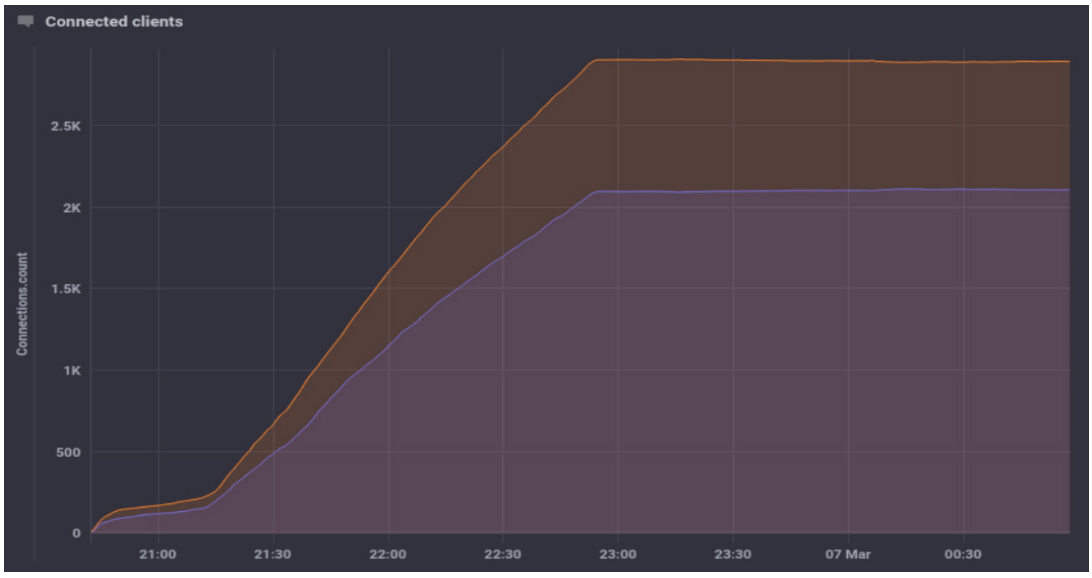
**Διάγραμματα E15:**
**Spawning Delay = 0.5s, Publish Delay = 3000ms**




*(α1) CPU load and Memory Consumption for node 172.16.8.203*




*(α2) CPU load and Memory Consumption for node 172.16.8.215*




*(β) Clients connected for EMQX cluster*

Τα τρία αυτά πειράματα δίνουν μία εικόνα για την ανταπόκριση του EMQX Broker στο εξαντλητικό σενάριο που έχει επιλεχθεί και πραγματοποιηθεί μέχρι τώρα, με μόνη διαφορά τον διπλασιασμό των ενεργών συνδέσεων που πραγματοποιούνται.

Η προσομοίωση E13, με Spawning Delay = 1s και Publish Delay = 100ms, δείχνει πολύ υψηλές τιμές CPU (για το node 1 κυμαίνεται στο 4 αλλά αρκετές φορές προσεγγίζει το 5, ενώ για το node 2 αντίστοιχα μεταβάλεται από το 2 μέχρι το 3), ενώ και η μνήμη RAM είναι σε αρκετά υψηλά επίπεδα, ιδιαίτερα αν συγκριθεί με το σενάριο E2 που είχε τις αντίστοιχες παραμέτρους. Επίσης παρατηρείται χαμηλό πλήθος clients (~1500-1600), κάτι που μπορεί να οφείλεται στη διαχείριση του φόρτου του δικτύου να μεταφράζεται ως αδυναμία ανταπόκρισης σε νέες συνδέσεις.

Η προσομοίωση E14, με Spawning Delay = 3s και Publish Delay = 50ms, δείχνει μία ακόμα πιο επιβαρυμένη εικόνα του broker (1000-1400 clients), με την CPU να ξεπερνά συχνά τα επίπεδα 5/3 για τα αντίστοιχα nodes, την μνήμη RAM στα σημεία που η CPU έχει αρκετά υψηλές τιμές να παρουσιάζει μεγάλες τιμές αντίστοιχα (και με μεγάλες διακυμάνσεις κατά τη διάρκεια της προσομοίωσης) και πλήθος συνδεδεμένων clients πολύ μικρό. Και πάλι μπορεί να σημειωθεί πως το πολύ μικρό Publish Delay είναι μια παράμετρος που επηρεάζει εξαιρετικά την λειτουργία του broker, ιδιαίτερα όταν αυξάνεται κατά το διπλάσιο το πλήθος των clients που αυτός πρέπει να εξυπηρετήσει.

Τέλος, στην προσομοίωση E15, με Spawning Delay = 0.5s και Publish Delay = 3000s, η λειτουργία του broker επανέρχεται σε απόλυτα φυσιολογικά επίπεδα, και αυξάνεται και σημαντικά το πλήθος των συνδέσεων που μπορεί να διαχειριστεί ταυτόχρονα (4-5k).

Αυτά ήταν και τα τελευταία πειράματα που έγιναν για την παρατήρηση της ανταπόκρισης του broker σε σενάρια υψηλού δικτυακού φόρτου. Στο επόμενο κεφάλαιο θα αναφερθούν συγκεντρωτικά τα αποτελέσματα που παρατηρήθηκαν και τα συμπεράσματα που προκύπτουν.

# Κεφάλαιο 5
# Συμπεράσματα

Σε αυτό το κεφάλαιο, θα γίνει μία σύνοψη των παρατηρήσεων που συγκεντρώθηκαν κατά την πειραματική διαδικασία με σκοπό την εξαγωγή συμπερασμάτων που αφορούν στη λειτουργία του εξυπηρετητή στο δίκτυο, και πώς η κάθε μία διαφορετική υλοποίησή του ευνοεί την αποδοτικότητα της συνολικής δομής.

## *5.1 Non clustered implementation*

Οι 15 προσομοιώσεις που πραγματοποιήθηκαν στο δίκτυο με τον non-clustered εξυπηρετητή, που υλοποιείται χρησιμοποιώντας το λογισμικό Mosquitto MQTT Broker, έδειξαν συγκεντρωτικά τα παρακάτω πορίσματα:

- ο Mosquitto Broker λειτουργεί σαν μοναδική διεργασία που υλοποιείται αξιοποιώντας έναν πυρήνα του μηχανήματος στο οποίο έχει εγκατασταθεί. Επομένως ο τρόπος λειτουργίας του επιβαρύνει την CPU του φυσικου μηχανήματος, και για αυτόν τον λόγο παρατηρείται σε όλα τα πειράματα υψηλό ποσοστό χρήσης του επεξεργαστή.
- η καμπύλη της CPU αποκτά μεγαλύτερη κλίση για μικρές τιμές της μεταβλητής Publish Delay, δηλαδή ο broker εξαντλεί τους διαθέσιμους πόρους όταν οι συνδεδεμένοι clients στέλνουν μηνύματα με πολύ μεγάλη συχνότητα. Το σενάριο αυτό που υλοποιήθηκε στα πειράματα μοιάζει αρκετά με την νοοτροπία μιας επίθεσης DDoS (distributed denial of service attack), αφού επιτυγχάνεται εξαιρετικά μεγάλος δικτυακός φόρτος από τις συσκευές του δικτύου IoT που στέλνουν μηνύματα με πολύ μεγάλες συχνότητες.
- η μνήμη RAM του broker δεν επηρεάζεται από την ένταση των σεναρίων προσομοίωσης, παρά μόνο σε πολύ μικρό βαθμό (0-10%). Τα σημεία που

παρατηρείται αύξηση στην κατανάλωση μνήμης είναι εκείνα που η CPU προσεγγίζει το 90%, χωρίς όμως η αύξηση αυτή να έχει αξιοσημείωτη διαφορά.

- ο broker αδυνατεί να διαχειριστεί νέες συνδέσεις και αποσυνδέεται από το δίκτυο, στο δεδομένο σενάριο δικτυακού φόρτου περίπου στους 3.5-4k ενεργούς clients, αριθμός που αποτελεί ένα αρκετά μικρό ποσοστό παρά την ένταση της προσομοίωσης. Σε σενάριο που προστέθηκε και δεύτερο VM για την δημιουργία νέων συνδέσεων, επιτεύχθηκε μέγιστο πλήθος ενεργών συνδέσεων κοντά στο 7k, που ωστόσο από τις 3.5k συνδέσεις και μετά υπήρχε μεγάλη απώλεια μηνυμάτων και αποτυχία συνδέσεων, με ταυτόχρονη εξάντληση των πόρων του εξυπηρετητή.
- μεγαλύτερο ποσοστό αποτυχίας νέων συνδέσεων/απώλειας ενεργών συνδέσεων εντοπίσθηκε στις προσομοιώσεις όπου η συχνότητα δημιουργίας νέων συνδέσεων ήταν αρκετά μεγάλη (Spawning Delay = 0.5s). Αυτο αναδεικνύει μία αδυναμία του broker να διαχειριστεί μεγάλο πλήθος ταυτόχρονων νέων συνδέσεων, χωρίς όμως τα ποσοστά αποτυχίας να είναι εξαιρετικά υψηλά.
- υπήρχε απώλεια απεσταλμένων μηνυμάτων σε όλα τα σενάρια που εκτελέσθηκαν.

Οι παραπάνω παρατηρήσεις οδηγούν στο συμπέρασμα ότι το λογισμικό Mosquitto Broker είναι ένα εξαιρετικά ελαφρύ λογισμικό για την υλοποίηση μιας non-clustered δομής broker σε ένα IoT δίκτυο, με δυνατότητες διαχείρισης μεγάλου πλήθους ταυτόχρονων συνδέσεων, το οποίο ωστόσο δεν ανταποκρίνεται αξιόπιστα σε ένα σενάριο εξαιρετικά μεγάλου δικτυακού φόρτου, όπως είναι αυτό που εφαρμόστηκε στις προσομοίωσεις που περιγράφηκαν. Το σενάριο αυτό είναι ένα πιθανό σενάριο επίθεσης (DDoS) σε δίκτυο IoT, όπως αναφέρθηκε παραπάνω, στο οποίο ένα δίκτυο σε περιβάλλον εργαστηριακό/βιομηχανικό, όπου απαιτείται αξιοπιστία και αρτιότητα στη λειτουργία του εξυπηρετητή και των λοιπών δομών της αρχιτεκτονικής, θα έπρεπε να μπορεί να ανταπεξέρχεται.

Ένα άλλο πολύ σημαντικό στοιχείο είναι η απώλεια απεσταλμένων μηνυμάτων που παρατηρήθηκε σε όλες τις προσομοιώσεις, γεγονός που δημιουργεί την αναγκαιότητα εφαρμογής μιας διαφορετικής δομής

εξυπηρετητή για αποφυγή απωλειών ανταλλασσόμενης πληροφορίας μέσα στο δίκτυο.

Αξίζει να αναφερθεί ξανά ότι οι παρατηρήσεις που γίνονται πάνω στην αξιοπιστία και την αποδοτικότητα του δικτύου οφείλονται στην παραδοχή της αναγκαιότητας ενός συστήματος χωρίς απώλειες μηνυμάτων και συνδέσεων και με γρήγορη και αξιόπιστη ανταπόκριση σε οποιαδήποτε δικτυακή συνθήκη, η οποία αποτελεί και την κύρια έρευνα σε αυτή τη διπλωματική. Τα χαρακτηριστικά αυτά δεν είναι απαραίτητα σε ένα μικρού εύρους δίκτυο IoT, ωστόσο είναι αναγκαία σε industrial εφαρμογές όπου είναι απαραίτητη η αξιοπιστία στον έλεγχο των συνδεδεμένων συσκευών και ακόμα και η παραμικρή απώλεια μηνυμάτων μπορεί να προκαλέσει σοβαρές βλάβες στο δίκτυο. Δεδομένου αυτού, η non clustered υλοποίηση ενός MQTT broker είναι αρκετά καλή λύση, και μάλιστα με ευέλικτο λογισμικό υλοποίησης (Mosquitto MQTT Broker), η οποία ωστόσο δεν βρέθηκε ικανοποιητικά αξιόπιστη σε συνθήκες μεγάλου δικτυακού φόρτου.

## *5.2 Clustered implementation*

Στην δομή του δικτύου με υλοποίηση του εξυπηρετητή σε cluster, χρησιμοποιώντας το λογισμικό EMQX Broker και τον HAProxy Load Balancer, πραγματοποιήθηκαν συνολικά 17 προσομοιώσεις, από τις οποίες συγκεντρώθηκαν οι παρακάτων παρατηρήσεις:

- ο EMQX Broker λειτουργεί χρησιμοποιώντας πολλά threads, και έτσι αξιοποιεί όλους τους πυρήνες που διατίθενται από το φυσικό μηχάνημα φιλοξενίας του εξυπηρετητή.
- παρατηρείται επιφόρτιση της CPU και στα δύο nodes, με το επίπεδο της 100% αξιοποίησης αυτής να ξεπερνάται κυρίως στα πειράματα όπου η συχνότητα αποστολης μηνυμάτων είναι πολύ μεγάλη (δηλαδή για μικρές τιμές του Publish Delay).
- παρατηρείται επιφόρτιση της χρησιμοποιούμενης από τον broker μνήμης RAM, σε φυσιολογικά επίπεδα. Οι έντονες διακυμάνσεις προκύπτουν στα σενάρια όπου η συχνότητα των νέων συνδέσεων είναι αρκετά μεγάλη (δηλαδή για μικρές τιμές του Spawning Delay).

- δεδομένου ότι το node 2 διαθέτει τους μισούς πόρους σε CPU και RAM από ότι το node 1, είναι αναμενόμενο να αποτελεί αυτό το σημείο αποτυχίας του δικτύου. Πράγματι, σε αρκετές προσομοιώσεις παρατηρείται αρκετά μεγάλη και ξαφνική αύξηση στην CPU του node αυτού, που ακολουθείται από αποσύνδεσή του από το δίκτυο. Σε αυτές τις περιπτώσεις ο load balancer δρομολογεί την κίνηση εκ νέου στο node 1, το οποίο αναλαμβάνει την διαχείριση των νέων συνδέσεων και των υπάρχοντων μέχρι την αποκατάσταση του node 2, το οποίο συνήθως επανεκκινεί και ξανασυνδέεται με παρέμβαση του διαχειριστή του δικτύου.
- στο σύνολο των προσομοιώσεων, οι μόνες απώλειες μηνυμάτων και ενεργών συνδέσεων που παρατηρήθηκαν εντοπίσθηκαν στις χρονικές στιγμές αποσύνδεσης του node 2. Το μέγεθος των απωλειών αυτών ήταν αμελητέο.
- ο Load Balancer, λειτουργώντας σαν εξωτερικό σημείο επικοινωνίας ανάμεσα στον broker και τις συσκευές, βοήθησε στην δρομολόγηση του δικτυακού φόρτου κατ' επιλογή. Συγκεκριμένα, κάποιες προσομοιώσεις έγιναν χωρίς βάρη, και όλες οι υπόλοιπες με βάρη που μοίραζαν δίκαια το φορτίο στα δύο nodes, ανάλογα με τις δυνατότητες του καθενός. Η χρησιμότητά του στην δομή του δικτύου ήταν εμφανής και μη διαπραγματεύσιμη, αφού η εναλλαγή δρομολόγησης ανάμεσα στα δύο nodes ανάλογα με την κατάσταση του καθενός έπαιξε σημαντικό ρόλο στην διαρκή ανταποκρισιμότητα του broker.
- το πλήθος των ενεργών συνδέσεων εξαρτάται σε μεγάλο βαθμό από τις παραμέτρους της εκάστοτε προσομοίωσης. Μεγαλύτερος αριθμός ταυτόχρονων ενεργών συνδέσεων επιτεύχθηκε σε περιπτώσεις που το Publish Delay ήταν αρκετά μεγάλο (3000ms), ακόμα και αν το Spawning Delay είναι πολύ μικρό.

Συγκριτικά λοιπόν, μεταξύ των υλοποιήσεων non-clustered server (με τον Mosquitto MQTT Broker) και clustered server (με τον EMQX Broker), η clustered υλοποίηση υπερτερούσε διότι:

- παρουσίασε πολύ καλύτερη διαχείριση των δεδομένων πόρων των μηχανημάτων που φιλοξένησαν τα nodes του broker,

- είχε ελάχιστες απώλειες μηνυμάτων και ενεργών συνδέσεων,
- έδειξε άμεση ανταπόκριση - με τη βοήθεια του load balancer - στην διαχείριση της αποτυχίας κάποιου node του server λόγω αυξημένης κίνησης, μέχρι την αποκατάσταση του δικτύου. Η χρήση του load balancer με βάρη, μάλιστα, έπαιξε σημαντικό ρόλο στη συχνότητα αποτυχίας του node 2, μιας και οι τιμές των βαρών προσαρμόστηκαν με τέτοιο τρόπο έτσι ώστε να κατανέμεται δίκαια το φορτίο σε κάθε node ανάλογα με τις δυνατότητές του.
- δεν παρουσίασε σε καμία προσομοίωση πλήρη αποσύνδεση του broker από το δίκτυο (no single point of failure).
- παρόλο που μπορεί κανείς να παρατηρήσει μεγαλύτερο πλήθος ενεργών συνδέσεων στα πειράματα του non-clustered συστήματος σε σχέση με αυτά που έγιναν στην clustered δομή, η διαφορά αυτή μπορεί να εξηγηθεί βλέποντας την διάρκεια των εκάστοτε πειραμάτων. Συγκεκριμένα, τα αποτελέσματα που λήφθηκαν στις προσομοιώσεις στον cluster παρατηρήθηκαν σε πολύ μικρότερα χρονικά διαστήματα από τα αντίστοιχα στον non-clustered broker, γεγονός που φανερώνει μεγάλη διαφορά στην αμεσότητα ανταπόκρισης του cluster σε μεγάλο δικτυακό φόρτο, χαρακτηριστικό που μαζί με την αξιοπιστία στην μεταβίβαση της πληροφορίας κάνει μια τέτοια δομή broker ιδανική για δίκτυα σε περιβάλλοντα με μεγάλες απαιτήσεις.

Όλα τα παραπάνω οδηγούν στο συμπέρασμα ότι ένας clustered broker αποτελεί μια πολύ αποδοτική και άριστα εξυπηρετική λύση για υλοποίηση server σε ένα δίκτυο IoT με αρκετά μεγάλες απαιτήσεις στη διαχείριση του δικτυακού φόρτου, όπως ένα δίκτυο σε εργαστηριακό ή βιομηχανικό περιβάλλον, μιας και παρά τις αλλαγές στην κίνηση που προκλήθηκαν, ήταν πάντα ενεργός και ανταποκρίνονταν στις απαιτήσεις του περιβάλλοντος (no single point of failure). Είναι βέβαια σημαντικό να αναγνωριστεί ότι σε συνθήκες που είναι απαραίτητη η πολύ συχνή αποστολή μηνυμάτων μεταξύ των συσκευών, ο clustered broker παρουσιάζει καλύτερη συμπεριφορά για μικρότερο πλήθος ενεργών συνδέσεων, χωρίς αυτό να σημαίνει ότι δεν ανταποκρίνεται ικανοποιητικά όταν αυτό δεν συμβαίνει. Τέλος, να αναφερθεί ξανά ότι η συχνότητα αποστολής μηνυμάτων σε μοναδικό topic

είναι ένα σενάριο που δεν θα συμβεί εύκολα σε ένα δίκτυο IoT (worst case scenario), αποτελεί ωστόσο μία τόσο έντονη συνθήκη που η ανταπόκριση του broker σε αυτή χωρίς σοβαρές απώλειες πληροφορίας εγγυάται την εύρυθμη και άριστη λειτουργία του σε οποιοδήποτε άλλο σενάριο.

# Κεφάλαιο 6
# Προβλήματα και Προεκτάσεις

Η διαδικασία εκπόνησης της συγκεκριμένης διπλωματικής έφερε στο φως διάφορα ζητήματα, τόσο τεχνικά προβλήματα που αφορούν στην υλοποίησή της, όσο και προεκτάσεις άξιες περαιτέρω διερεύνσης, που ωστόσο δεν ήταν δυνατό να απαντηθούν στα πλαίσια της εργασίας. Σε αυτό το κεφάλαιο θα γίνει αναφορά στα ζητήματα αυτά.

## *6.1 Προβλήματα*

Η υλοποίηση ενός δικτύου IoT με την δομή που αναλύθηκε στα προηγούμενα κεφάλαια προκαλεί τους παρακάτω προβληματισμούς σε σχέση με τη λειτουργία του:

- Υπήρξαν πρακτικά όρια στον δικτυακό φόρτο που ήταν δυνατό να δημιουργηθεί. Τα client spawning VM είχαν πεπερασμένες δυνατότητες στην δημιουργία πολλών client object ταυτόχρονα. Επιπλέον αυτό συνέβαινε και με συγκεκριμένη ταχύτητα, η οποία παρότι ορισμένη (Spawning Delay) με την υπερβολική κατανάλωση των πόρων των μηχανημάτων δεν μπορούσε να διατηρηθεί στο ακέραιο.
- Η υλοποίηση του client spawning, που όπως αναφέρθηκε έγινε με ένα σύνολο scripts που το καθένα ήταν γραμμένο σε διαφορετική γλώσσα και επιτελούσε διαφορετικό ρόλο, δημιουργεί διάφορα ζητήματα συμβατότητας, καθώς καθιστά αναγκαία την ενημέρωση όλων των VM στον ίδιο βαθμό με τα ίδια πακέτα. Αποτρέπει έτσι την χρήση των scripts αυτών σαν μία ενιαία πρόταση για δημιουργία clients objects,

καθώς ο ορισμός των παραμέτρων γίνεται με διαφορετικό τρόπο σε κάθε κομμάτι της υλοποίησης.

- για την παρακολούθηση της συμπεριφοράς του Mosquitto MQTT Broker αρχικά χρησιμοποιήθηκε το λογισμικό MQTTBox, καθώς αντίθετα με τον EMQX Broker, ο Mosquitto δεν διαθέτει διαδικτυακή πλατφόρμα (dashboard) για εύκολη εποπτεία της λειτουργίας του. Το λογισμικό αυτό ωστόσο φάνηκε ότι δεν μπορούσε να διατηρήσει σταθερή σύνδεση με τον broker, κάτι που κατά πάσα πιθανότητα οφείλεται στην αδυναμία του broker να διαχειριστεί αξιόπιστα πολλές συνδέσεις που απαιτούν συνεχή αποστολή μηνυμάτων ταυτόχρονα, όταν το δίκτυο έχει μεγάλη κίνηση. Δημιουργήθηκε έτσι η ανάγκη διερεύνησης και σχεδιασμού μιας λύσης για το logging της διαδικασίας.
- Άλλο ένα πρόβλημα που παρατηρήθηκε, και που αναφέρθηκε και στο Κεφάλαιο 4, είναι η συμπεριφορά της InfluxDB στην αναπαράσταση του "κενού χρόνου" μεταξύ δύο μετρήσεων. Αυτό συνέβη όταν ο broker δεν έστελνε πλέον πληροφορίες στην Influx λόγω εξαιρετικά μεγάλου φόρτου, με αποτέλεσμα μετά τον τερματισμό της προσομοίωσης η τελευταία τιμή να είναι κάποια συγκεκριμένη, που θα ενώνονταν γραμμικά με την πρώτη τιμή της επόμενης προσομοίωσης. Ως αποτέλεσμα τα διαγράμματα του Chronograf ήταν ενιαία, και δεν ήταν δυνατό να ληφθεί εικόνα των CPU/MEM για τον broker στην διαδικασία τερματισμού των προσομοιώσεων.

## *6.2 Προεκτάσεις*

Βάσει των προβλημάτων που αναφέρθηκαν στην προηγούμενη ενότητα, θα μπορούσαν να γίνουν κάποιες προτάσεις για βελτίωση της συγκεκριμένης εργασίας, οι οποίες συζητήθηκαν αλλά δεν ήταν δυνατό να πραγματοποιηθούν στα πλαίσια της διπλωματικής εργασίας.

- τα πειράματα που έγιναν είχαν ως βασική παράμετρο την αποστολή μηνυμάτων από τις συσκευές του δικτύου σε ένα κοινό topic (που θα μπορούσε να αποτελεί κάποιο topic ελέγχου λειτουργίας του δικτύου) **χωρίς καμία κρυπτογράφηση** της πληροφορίας που ανταλλάσεται. Τόσο ο Mosquitto Broker, όσο και ο EMQX Broker (αλλά και όλα τα διαθέσιμα λογισμικά broker) προσφέρουν λύσεις για ασφαλείς συνδέσεις των clients με τον broker, η πιο κοινή από τίς οποίες είναι η χρήση του openssl για την σύναψη συνδέσεων μέσω SSL (Sequre Sockets Layer). Αυτό σημαίνει ότι κάθε client και ο broker θα χρησιμοποιούν στην χειραψία σύνδεσης πιστοποιητικά υπογεγραμμένα από το ίδιο Certificate Authority. Η σύνδεση χωρίς καμία κρυπτογράφηση που χρησιμοποιείται θεωρείται μεγάλο κενό ασφάλειας σε ένα δίκτυο, και αποτελεί και ένα από τα μεγαλύτερα ελλείματα της τεχνολογίας του Διαδικτύου των Πραγμάτων μέχρι στιγμής, καθώς είναι δύσκολη η ικανοποιητική προστασία χιλιάδων συσκευών τόσο λόγω μεγάλου πλήθους τους όσο και εξαιτίας των διαφορετικών προδιαγραφών και δυνατοτήτων τους.
- το περιεχόμενο (payload) του μηνύματος που έστελναν οι clients κατά τη διάρκεια των προσομοιώσεων για δημιουργία δικτυακού φόρτου κρίθηκε αμελητέο, και έτσι ήταν πολύ μικρού μεγέθους. Οποιοδήποτε μήνυμα με μεγαλύτερο payload από το χρησιμοποιούμενο εκτιμάται ότι θα προκαλέσει εξάντληση των πόρων στον broker αρκετά νωρίτερα, μεταβάλλοντας έτσι τα όρια εύρυθμης λειτουργίας του σε σχέση και με τις υπόλοιπες παραμέτρους. Σε συνέχεια του παραπάνω προβληματισμού, στη δεδομένη υλοποίηση που παρουσιάστηκε, η χρήση πιστοποιητικών για ασφαλή σύνδεση των clients με τον broker θα προσέθετε κατά πολύ στο payload των μηνυμάτων, δημιουργώντας πολύ μεγαλύτερο φόρτο στο δίκτυο κατά την συνεχή ανταλλαγή μηνυμάτων. Επομένως η διεξαγωγή των πειραμάτων με διαφορετικά payload μηνυμάτων θα

ξεκαθάριζε αρκετά τα όρια εύρυθμης και αξιόπιστης λειτουργίας του δικτύου.

- Στα πλαίσια εξερεύνησης των ορίων της ομαλής λειτουργίας του broker σε ένα δίκτυο IoT, θα είχε μεγάλη σημασία η διεξαγωγή πειραμάτων άλλων σεναρίων δικτυακού φόρτου, γιατί η διερεύνηση της ανταπόκρισης του broker σε συνθήκες εκτός του worst case scenario πιθανόν να αποκαλύψει άλλες αδυναμίες που θα βοηθήσουν στον σχεδιασμό μιας βέλτιστης λύσης για περιβάλλον εργαστηριακό/παραγωγής με μεγάλες απαιτήσεις.
- Ίσως καθαρότερη εικόνα θα προέκυπτε αν τα πειράματα γίνονταν εισάγοντας τον παράγοντα του non-spacial clustering, δηλαδή τα nodes του cluster να μην είναι μέλη του ίδιου δικτύου.
- Όσο αφορά το logging, όπως αναφέρθηκε και στην ενότητα με τα προβλήματα, λόγω της αδυναμίας προγραμμάτων παρακολούθησης του broker (όπως το MQTTBox) να διατηρήσουν σταθερή σύνδεση με τον broker όταν αυτός πρέπει να ανταποκριθεί σε συνθήκες μεγάλου φόρτου, δημιουργείται η ανάγκη μιας πρότασης για logging της πειραματικής διαδικασίας μέσω των systopics του ίδιου του broker, η οποία θα βασίζεται στις δυνατότητες του λογισμικού του broker. Στην περίπτωση του cluster, και συγκεκριμένα της υλοποίησης που παρουσιάστηκε σε αυτήν την εργασία, άλλη μία πιθανή προέκταση θα μπορούσε να είναι η υλοποίηση ενός ολοκληρωμένου προγράμματος logging, αντί των δύο διαφορετικών που χρησιμοποιήθηκαν στην περίπτωση του EMQX Broker, για την λήψη δεδομένων μέσω των systopics. Για να γίνει αυτό θα πρέπει να δωθεί περισσότερη έμφαση στο dashboard του EMQX, στο οποίο πέρα από τις πληροφορίες για το δίκτυο υπάρχουν και λεπτομέρειες για την κατανάλωση πόρων των nodes. Επομένως αν η λήψη των πληροφοριών αυτών γινόταν μέσω του dashboard, δεν θα ήταν αναγκαία η υλοποίηση του emq.py ως service στα δύο nodes του broker.

# Κεφάλαιο 7
# Παράρτημα κώδικα

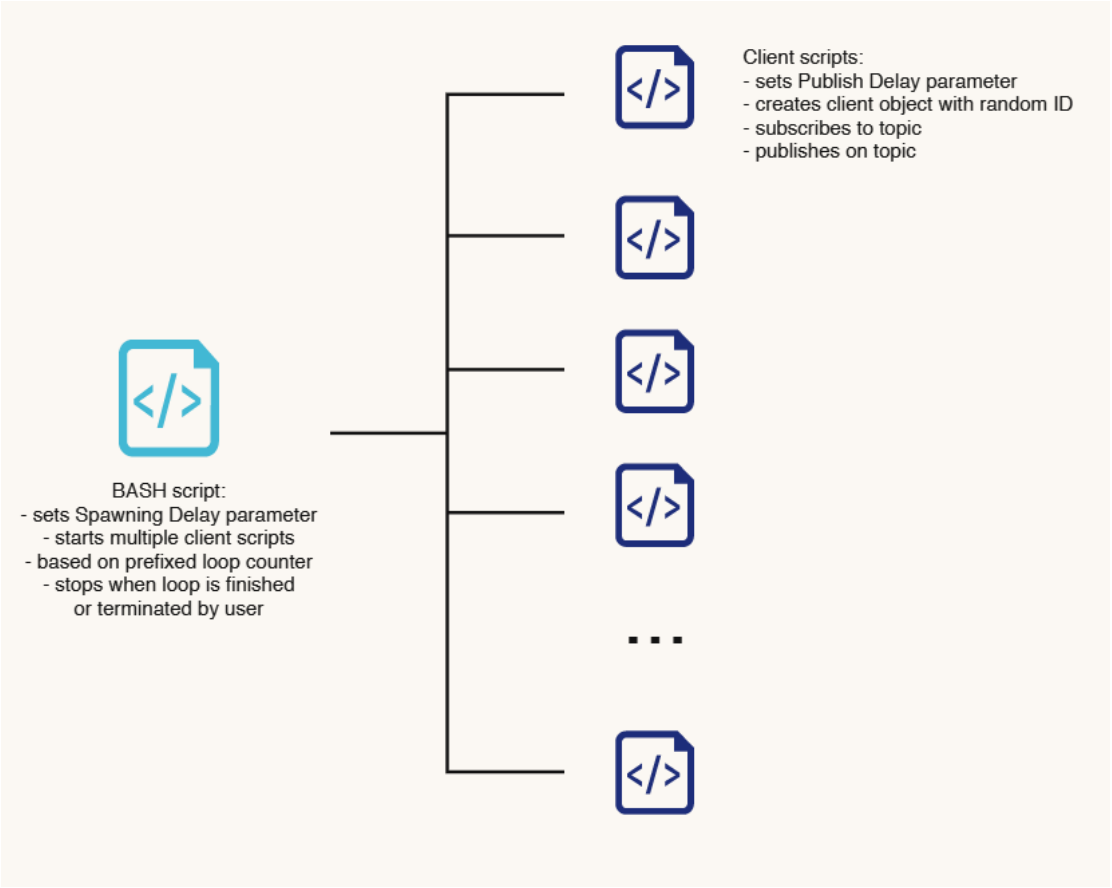
Σε αυτό το κεφάλαιο παρατίθεται ο κώδικας που χρησιμοποιήθηκε για την αναπαραγωγή συνθηκών δικτύου πολλών συσκευών IoT και ο κώδικας των logger που έκαναν συλλογή πληροφοριών όσον αφορά τον broker.

Ο κώδικας για τα VM προσομοίωσης συσκευών IoT είναι κοινός τόσο για την υλοποίηση του Mosquitto Broker, όσο και για τον EMQX Broker.

## *7.1 Client spawning*

Όπως αναφέρθηκε σε προηγούμενο κεφάλαιο, για την δημιουργία των νέων client χρησιμοποιήθηκε ένα βασικό C script, το **pub_client.c**, το οποίο εκτελείται παράλληλα πολλές φορές (όσες και το επιθυμητό πλήθος clients) από ένα BASH script, το **bash.sh**. Το BASH script επίσης καλεί άλλα δύο Python scripts, των οποίων η λειτουργία είναι:

- το **start_bash.py,** εκκινεί έναν MQTT Client με το όνομα Control, που κάνει publish στο topic start_timestamp όταν ξεκινήσει η δημιουργία των νέων clients με ένα μήνυμα σε μορφή JSON που περιέχει τα tags num_client (τελικό πλήθος συσκευών που θα δημιουργηθούν), current_client (πλήθος συσκευών αυτή τη στιγμή), testcase (αναγνωριστικό προσομοίωσης). Σε αυτό το topic ακούει ο logger του Mosquitto Broker, και έτσι ενεργοποιείται και ξεκινά τη συλλογή πληροφοριών.
- το **check.py**, εκκινεί έναν MQTT Client με το όνομα Control, ο οποίος πάλι κάνει publish στο topic start_timestamp και στέλνει ένα μήνυμα σε μορφή JSON, ανάλογο με πριν, για την διαρκή ενημέρωση του τρέχοντος αριθμού clients.




*Εικόνα 7.1: Αναπαράσταση της διαδοχικής εκκίνησης των προγραμμάτων για την προσομοίωση των clients του δικτύου*

## bash.sh

```
#!/bin/bash

pidArr=()              #Will keep the PIDs of the children in here
n=10000;

sigint() {             #Trap for Ctrl-C to terminate the process
    echo "Terminating script";
    echo "Terminating child processes...";
    sudo pkill -9 -f pub_out.out;
    for pid in ${pidArr[@]}
    do
        (echo "Killing process: $pid");
        (kill -9 $pid 2> /dev/null);
        if [[ $? -eq 0 ]]
        then
            echo "Killed"
        else
            echo "Process was already finished"
        fi
    done

    exit 0
}
trap 'sigint' INT      # setup trap for Ctrl-C

echo "Starting script. My PID is $$"
for cnt in {1..100..10}          #loop estimating total number of clients, stated by admin
do
    if [[ cnt -eq 1 ]]
    then
        m=$(($n*$cnt))
    else
        cnt=$((cnt-1));
        m=$(($n*$cnt))
    fi
    python3 start_bash.py $m;          # start_bash.py notifies logger to start monitoring,
                                       # passes total number of clients simulated
    echo 'Test with' $m 'clients starting';
    z=0;p=1;                           # z is the id number of each client
    for id in $(seq $m)
    do
        sleep .5;                      # SPAWNING DELAY parameter
        (sudo ./pub_out.out &>/dev/null &);      #simulation of each client object initiated
        echo 'client' $id 'is up';
        z=$((z+1));
        pidArr+=($!)
        if [[ $z -eq $p*100 ]]
        then
            python3 check.py $m $z;            # every 100 client objects logger is notified
            p=$((p+1));
        fi
        if [[ id -eq m ]]
        then
            sudo pkill -9 -f pub_out.out;      # Termination of client objects when requestes total is reached
            echo "Killing script and subprocesses";
        fi
    done
done
echo 'Done.'
wait
```

Η εκτέλεση του προηγούμενου προγράμματος γίνεται από το command line του VM με την εντολή:

**./bash.sh**

### *pub_client.c*

```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4   #include "MQTTClient.h"
5   #include <time.h>
6   #include "/home/ubuntu/Eclipse-Paho-MQTT-C-1.3.0-Linux/include/MQTTClient.h"
7   #define ADDRESS    "tcp://172.16.8.206:1883"          // broker ip address:port
8   #define TOPIC      "MQTT Examples"                    // topic name
9   #define PAYLOAD    "1234567"                          // payload
10  #define QOS        1                                  // quality of service
11  #define TIMEOUT    10000L
12
13  void delay(int milliseconds)                          // function delay sets the PUBLISH DELAY
14  {
15     long pause;
16     clock_t now,then;
17
18     pause = milliseconds*(CLOCKS_PER_SEC/1000);
19     now = then = clock();
20     while( (now-then) < pause )
21        now = clock();
22  }
23
24
25  volatile MQTTClient_deliveryToken deliveredtoken;
26
27  void delivered(void *context, MQTTClient_deliveryToken dt)
28  {
29     deliveredtoken = dt;
30
31  }
32
33  int msgarrvd(void *context, char *topicName, int topicLen, MQTTClient_message *message)
34  {                                                     // function that sets the payload
35     int i;
36     char* payloadptr;
37     payloadptr = message->payload;
38     MQTTClient_freeMessage(&message);
39     MQTTClient_free(topicName);
40     return 1;
41  }
42
43  void connlost(void *context, char *cause)             // function that notifies when connection is lost
44  {
45     printf("\nConnection lost\n");
46     printf("     cause: %s\n", cause);
47  }
48
49  int main(int argc, char* argv[])
50  {
51     srand(time(NULL));                                 // Initialization, should only be called once.
```

```
52          int id = rand();                                // rand() is used as shown to provide unique random
53          char str[10];                                   // client ID from a large set of integers
54          sprintf(str, "%d", id);
55          const char * CLIENTID;
56          char * r;
57          r  = str;
58          CLIENTID = r;
59
60          MQTTClient client;                              // initialization of client object
61          MQTTClient_connectOptions conn_opts = MQTTClient_connectOptions_initializer;
62          MQTTClient_message pubmsg = MQTTClient_message_initializer;
63          MQTTClient_deliveryToken token;
64          int rc;
65
66          MQTTClient_create(&client, ADDRESS, CLIENTID,       // creation of client object
67             MQTTCLIENT_PERSISTENCE_NONE, NULL);
68          conn_opts.keepAliveInterval = 20;
69          conn_opts.cleansession = 1;
70
71          MQTTClient_setCallbacks(client, NULL, connlost, msgarrvd, delivered);
72
73          if ((rc = MQTTClient_connect(client, &conn_opts)) != MQTTCLIENT_SUCCESS)
74          {
75             printf("Failed to connect, return code %d\n", rc);
76             exit(EXIT_FAILURE);
77          }
78
79          MQTTClient_subscribe(client, TOPIC, QOS);           // subscribe to topic
80
81          while(1){                                           // publish to topic
82                delay(3000);                                  // PUBLISH DELAY parameter
83                pubmsg.payload = PAYLOAD;
84                pubmsg.payloadlen = strlen(PAYLOAD);
85                pubmsg.qos = QOS;
86                pubmsg.retained = 0;
87                MQTTClient_publishMessage(client, TOPIC, &pubmsg, &token);
88          }
89
90
91          rc = MQTTClient_waitForCompletion(client, token, TIMEOUT);
92          MQTTClient_disconnect(client, 10000);               // client object disconnect and destroy
93          MQTTClient_destroy(&client);
94
95          return rc;
96       }
```

Το παραπάνω script, όντας γραμμένο σε C, πρέπει να γίνει compile και να συνδεθεί με τις απαραίτητες βιβλιοθήκες της Eclipse Paho. Αυτό γίνεται με την εκτέλεση της παρακάτω εντολής, η οποία δημιουργεί το εκτελέσιμο αρχείο pub_out.out:

***gcc -o pub_out.out pub_client.c -L /home/ubuntu/Eclipse-Paho-MQTT-C-1.3.0-Linux/lib/ -l paho-mqtt3c***

Στη συνέχεια η εκτέλεσή του γίνεται στη γραμμή 46 του bash.sh, με την εντολή:

**(sudo ./pub_out.out &> /dev/null &);**

η οποία πέρα από την εκτέλεσή του καθορίζει και ότι η έξοδος του προγράμματος (στη συγκεκριμένη περίπτωση τα μηνύματα επιβεβαίωσης σύνδεσης που έχουν οριστεί) αντί για την οθόνη θα πηγαίνει στο /dev/null, μία διεύθυνση που στα συστήματα Linux είναι καθορισμένη να διαγράφει οτιδήποτε αποστέλλεται εκεί.

Τέλος η πολλαπλή εκτέλεση του pub_out.out τερματίζεται είτε χειροκίνητα από τον διαχειριστή της προσομοίωσης ή στο τέλος αυτής με τις εντολές στις γραμμές 10 και 57 του bash.sh αντίστοιχα.

### *start_bash.py*

```
1   import paho.mqtt.client as paho
2   import sys
3   import json
4   import time
5   import os
6
7   broker=”172.16.8.206”                  # broker ip address
8   port=1883                              # broker port
9
10  print(‘Control Client is up’)
11  num = sys.argv[1]                      # script takes as input total number of clients in the simulation
12  p=0
13  msg= json.dumps({‘num_client’: num,’current_client’: p, ‘testcase’: “T11”})      # json with info sent to the logger
14                                                  # for comparing and detecting actual number of active clients
15  client=paho.Client(“Control”)          # initialization of client object
16  client.connect(broker,port)
17  client.subscribe(“start_timestamp”, qos=2)      # quality of service level 2, control messages cannot be lost
18  client.publish(‘start_timestamp’, msg, qos=2)
19  time.sleep(2)
20  client.disconnect()
```

### *check.py*

```
1   import paho.mqtt.client as paho
2   import sys
3   import json
4   import time
5   import os
6
7   broker=”172.16.8.206”                  # broker ip address
8   port=1883                              # broker port
9
10  print(‘Control Client is up’)
11  num = sys.argv[1]                      # script takes as input total number of clients in the simulation
12  c_num = sys.argv[2]                    # and current number of clients already initiated
13  msg= json.dumps({‘num_client’: num, ‘current_client’: c_num, ‘testcase’: “T11”})      # json for the logger
```

```
client=paho.Client("Control")            # initialization of client object
client.connect(broker,port)
client.subscribe("start_timestamp", qos=2)
client.publish('start_timestamp', msg, qos=2)
time.sleep(2)
client.disconnect()
```

## 7.2 Loggers

### 7.2.1 Mosquitto Logger

### log.py

```
import time
import sys
import subprocess
import paho.mqtt.client as paho
import json
from influxdb import InfluxDBClient

try:                                          # trap set for keyboard interrupt
   from StringIO import StringIO
except ImportError:
   from io import StringIO

myList= []


def on_message(client, userdata, message):    # callback function for initiating logger
                                              # when json from start_bash.py arrives
   global myList
   myList = []
   n = json.loads(message.payload)
   num = n['num_client']
   c_num = n['current_client']
   testc = n['testcase']
   myList= [num, c_num, testc]
   print(myList)
   pass


def get_cpumem(pid):                          # function that retrieves CPU/MEM statistics with ps command and pid
   d = [i for i in subprocess.getoutput("ps -aux").split("\n")
      if i.split()[1] == str(pid)]
   return (float(d[0].split()[2]), float(d[0].split()[3])) if d else None     # returns parsed metrics


def writeToInfluxDB():                        # function that forms json with metrics and sends them to InfluxDB
   global dbConn
   if (get_cpumem(sys.argv[1]) == None):
      print("no such process")
      exit(1)
   x,y = get_cpumem(sys.argv[1])

   try:
```

```
43          dbConn = InfluxDBClient("10.0.100.21""8086", "athena", "@cosmote@", "mqtt_tests")   # Influx login
44      credentials
45          print("Connected to InfluxDB.")
46        except:
47          print("Could not connect to InfluxDB.")
48          exit(1)
49
50          json_body = [                       # Creating json with the relevant information
51            {
52                "measurement": "cpu_mem",
53                "tags": {
54                        "number_of_clients": int(myList[0]),
55                        "current_number_clients": int(myList[1]),
56                        "testcase": str(myList[2]),
57                      },
58                "fields": {
59                        "cpu": float(x),
60                        "mem": float(y),
61                        }
62                }
63
64              ]
65
66        dbConn.write_points(json_body)
67        dbConn.close()
68
69      def main():
70        global myList
71        client=paho.Client("logger")                    # initialization of logger client object
72        client.connect('94.70.239.222',1883)
73        client.on_message=on_message                    # callback function initiated
74        client.subscribe("start_timestamp", qos=1)
75        client.loop_start()
76        while True:
77          time.sleep(5)
78          if not myList:
79            print('Nothing to log')
80          else:
81            writeToInfluxDB()                           # write to InfluxDB
82
83        client.loop_stop()
84        client.disconnect()
85
86      print('Starting logger')
87      main()
```

Ο logger εκτελείται με την εντολή:

**python3 log.py**

στο μηχάνημα του Mosquitto MQTT Broker.

### 7.2.2 EMQX Logger

#### emq.py

```
#!/usr/bin/python3
import json
import os
import re
import tempfile
import paho.mqtt.client as mqtt
import time
from influxdb import InfluxDBClient

scratchdir = tempfile.mkdtemp()          # create temporary directory to store data

try:                                     # trap set for keyboard interrupt
  from StringIO import StringIO
except ImportError:
  from io import StringIO


def parse_requests(filename):            # function that parses response from emqx command line queries
  f = open(filename,"r")                 # takes tempfile as input
  ndict = {}
  for x in f.readlines():
    if "cpu" or "memory" in x:           # gets values related to cpu/mem from the query response
      ndict[x.split(" ")[0]]= ((x.split(":")[1]).rstrip('\r\n')).replace(' ', '')
  for key, value in ndict.items():
    ndict[key] = float(value)
  return json.dumps(ndict)               # returns values in json dictionary format


def get_cpumem(query, scratchdir):       # function that makes queries through emqx command line
  fileobj = tempfile.NamedTemporaryFile(
      prefix=("%s" % query), suffix="-load",
      dir=scratchdir, delete=False)
  filename = fileobj.name                # initialization of tempfile to store query response

  os.system('emqx_ctl vm %s > %s' % (query, filename))     # query takes place

  return parse_requests(filename)        # returns response in json dictionary format, created by parse_requests


def main():
  client=mqtt.Client("logger_node203")           # client object logger
  client.connect('172.16.8.203',1883)
  client.subscribe("emq_node203", qos=1)
  client.loop_start()
  while True:
    time.sleep(1)                                # requests metrics every 1 second
    client.publish('emq_node203', get_cpumem('load', scratchdir))     # cpu load query for broker
    client.publish('emq_node203', get_cpumem('memory', scratchdir))           # mem query for broker
  client.loop_stop()
  client.disconnect()

main()
```

Το script αυτό αποτελεί τη μισή υλοποίηση του logger για τον EMQX, και τρέχει ως service σε κάθε node του broker, συλλέγοντας πληροφορίες για την κατανάλωση CPU και MEM μέσω της γραμμής εντολών του EMQX:

**emqx_ctl vm load**
**emqx_ctl vm memory**

Οι πληροφορίες αυτές στη συνέχεια επεξεργάζονται κατάλληλα από την συνάρτηση ***parse_requests(filename)*** και αποστέλλονται στα topics emq_node203 και emq_node215 που παρακολουθούνται από την InfluxDB.

Ο κώδικας είναι αυτός που χρησιμοποιείται στο node 172.16.8.203, για το δεύτερο node ισχύουν οι ανάλογες τροποποιήσεις στα ονόματα των διευθύνσεων και των topics.

### *curl.py*

```
1   import requests
2   import pycurl
3   import json
4   import paho.mqtt.client as mqtt
5   import time
6   import subprocess
7   import sys
8   from influxdb import InfluxDBClient
9
10
11  try:                                    # trap set for keyboard interrupt
12      from StringIO import StringIO
13  except ImportError:
14      from io import StringIO
15
16
17  myList = []
18
19
20  def on_message(client, userdata, message):   # callback function to initiate logger when notified from start_bash.py
21      global myList
22      myList = []
23      n = json.loads(message.payload)
24      num = n['num_client']
25      c_num = n['current_client']
26      testc = n['testcase']
27      myList= [num, c_num, testc]
28      print(myList)
29      pass
30
31
32  def curl_call(url):                 # function that makes curl requests to EMQX dashboard
33      c = pycurl.Curl()
34      c.setopt(c.URL, 'http://172.16.8.203:8081/api/v4/' + url)   # connects to dashpoard api
35      c.setopt(pycurl.USERPWD, 'appsec' + ':' + 'MjkxMzg5NjAzNDIzNjI1OTA5MDE4MzkxMDc1NzM2MTI1N-
36  DE')                                # credentials for successful connection
```

```
37      try:
38          rb = c.perform_rs()
39      except pycurl.error:
40          print('An error occurred.')
41      c.close()
42      return rb                       # returns requestes metrics in json format
43
44
45  def main():
46      global myList
47      client=mqtt.Client("logger")
48      client.connect('172.16.8.203',1883)
49      client.on_message=on_message
50      client.subscribe("start_timestamp", qos=2)
51      client.loop_start()
52      while True:
53          time.sleep(5)
54          if not myList:              # myList initiated by callback in order to start logging
55              print('Nothing to log')
56          else:
57              client.publish("telegraf/monitor_stats", curl_call("stats"), qos=1)          # perform curl requests
58              client.publish("telegraf/monitor_metrics", curl_call("metrics"), qos=1)
59      client.loop_stop()
60      client.disconnect()
61
62
63  print('Starting logger')
64  main()
```

Το παραπάνω script εκτελείται στο node 172.16.8.203 με την εντολή:

python3 curl.py

και πραγματοποιεί τα curl requests (μέσω της βιβλιοθήκης PyCurl) προς τον broker για συλλογή δεδομένων που προέρχονται από τα systopics αυτού. Τα δεδομένα που στέλνει ο broker ως απάντηση αποστέλλονται στην InfluxDB και από εκεί επιλέγεται ποια από αυτά θα παρασταθούν γραφικά.

### 7.2.3 Configuration files

#### mosquitto.conf

```
# Place your local configuration in /etc/mosquitto/conf/d/
#
# A full description od the configuration file is at
# /usr/share/doc/mosquitto/examples/mosquitto.conf.example

pid_file /var/run/mosquitto.pid

persistence true
persistence_location /var/lib/mosquitto

include_dir /etc/mosquitto/conf.d

sys_interval 1              # time for refreshing information in syslog topics

max_connections -1          # set max connections number to unlimited
log_dest syslog topic       # send logging information to syslog topics
```

#### telegraf.conf

```
# # Read metrics from MQTT topic(s)
 [[inputs.mqtt_consumer]]
 name_suffix = "_syslogs"
#   ## MQTT broker URLs to be used. The format should be scheme://host:port,
#   ## schema can be tcp, ssl, or ws.
  servers = ["tcp://10.0.100.14:1883"]
#
#   ## QoS policy for messages
#   ##   0 = at most once
#   ##   1 = at least once
#   ##   2 = exactly once
  qos = 1
#
#   ## Connection timeout for initial connection in seconds
  connection_timeout = "30s"
#   ## Topics to subscribe to
#   topics = [
#     "telegraf/host01/cpu",
#     "telegraf/+/mem",
#     "sensors/#",
#   ]
  topics = [
          "$SYS/broker/clients/connected",
          "$SYS/broker/load/connections/1min",
          "$SYS/broker/subscriptions/count",
          "$SYS/broker/messages/publish/dropped",
          "$SYS/broker/clients/disconnected",
          "SYS/broker/clients/total",
          "$SYS/broker/load/publish/dropped/1min",
          "$SYS/broker/load/publish/received/1min",
          "$SYS/broker/load/sockets/1min",
          "$SYS/broker/messages/inflight",
          "$SYS/broker/load/publish/sent/1min",
```

```
34              "$SYS/broker/load/messages/received/1min",
35              "$SYS/broker/messages/sent",
36              "$SYS/broker/load/messages/sent/1min"
37              ]
38
39  #
40  #   # if true, messages that can't be delivered while the subscriber is offline
41  #   # will be delivered when it comes back (such as on service restart).
42  #   # NOTE: if true, client_id MUST be set
43    persistent_session = true
44  #   # If empty, a random client ID will be generated.
45    client_id = "test_telegraf3"
```

Να σημειωθεί ότι το παραπάνω κομμάτι του telegraf.conf δεν αποτελεί ολόκληρο το αρχείο, παρά μόνο το σημείο που χρειαζόταν κατάλληλη διαμόρφωση για την παρακολούθηση του Mosquitto Broker. Αντίστοιχες αλλαγές (όπως ip addresses, ports, credentials) έγιναν και στην περίπτωση του EMQX Broker, και συγκεκριμένα τα topics δηλώθηκαν ως εξής:

```
1       topics = [
2               "emq_node203/#",
3               "emq_node215/#",
4               "telegraf/monitor_stats/#",
5               "telegraf/monitor_metrics/#",
6               "start_timestamp/#"
7               ]
```

### *emq.service*

```
1       [Unit]
2       Description= Check emqx cpu and memory usage on node
3       After=multi-user.target
4
5       [Service]
6       Type=idle
7       ExecStart=/usr/bin/python3 /home/ubuntu/running/emq.py
8       WorkingDirectory=~
9       Restart=on-failure
10
11      [Install]
12      WantedBy=multi-user.target
```

Παραπάνω φαίνεται το αρχείο που δημιουργήθηκε στη διεύθυνση /lib/systemd/system/ με σκοπό να μετατρέψει το script emq.py σε service. Μετά την αποθήκευση του αρχείου, χρειάστηκε να εκτελεστούν οι παρακάτω εντολές.

**sudo systemctl daemon-reload**

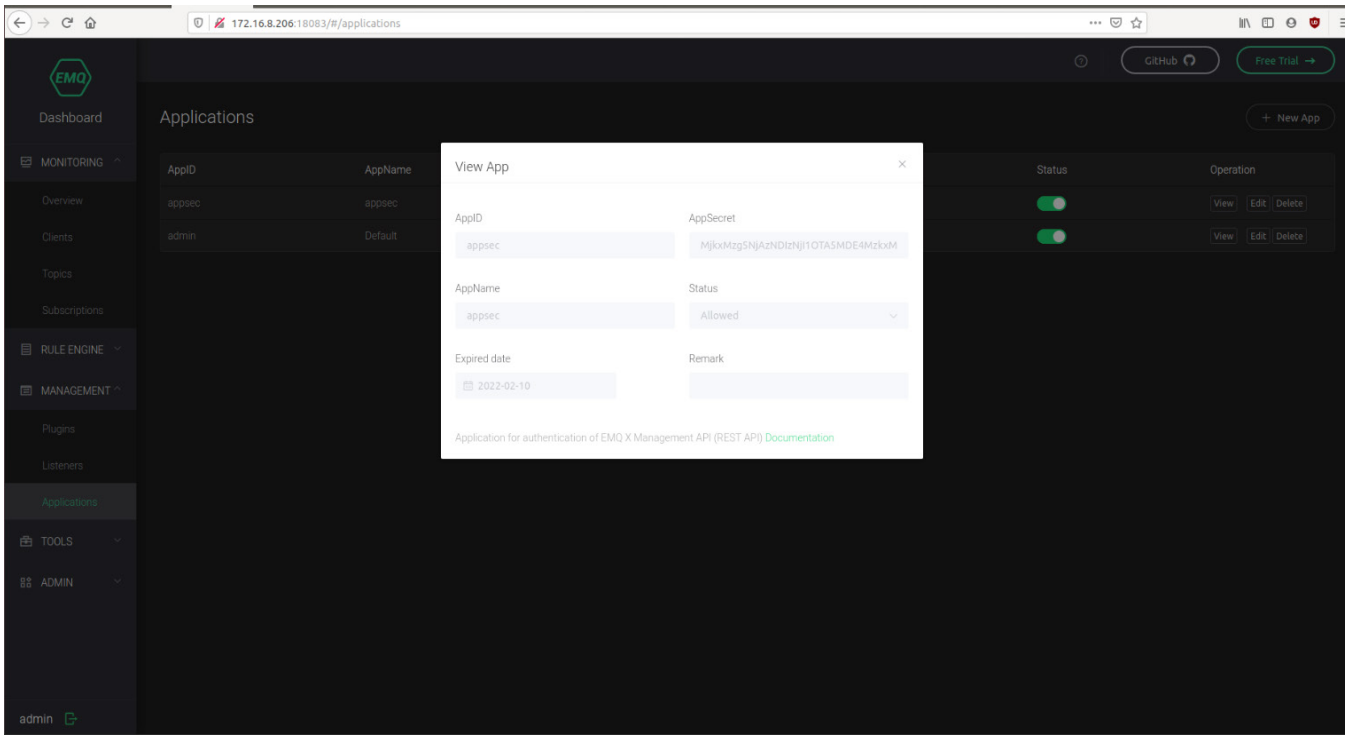
**sudo systemctl enable emq.service**

**sudo systemctl start emq.service**

Το emq.py με αυτή τη διαδικασία εκτελείται στο background συνεχώς, μέχρι ο διαχειριστής να παρέμβει στη λειτουργία του.

### *CuRL Request - EMQX Dashboad Permission*

Για την σύνδεση του curl.py με τον broker και την αποστολή του curl request χρειάζεται σύνδεση με ειδικό ζεύγος username/password, το οποίο αποδίδεται από τον διαχειριστή μέσω του EMQX dashboard στην ενότητα Management/Applications. Μια απεικόνιση του δημιουργηθέντος ζεύγους κλειδιών που χρησιμοποιήθηκε φαίνεται παρακάτω.



## *haproxy.conf*

```
1   global
2       log /dev/log    local0
3       log /dev/log    local1 notice
4       chroot /var/lib/haproxy
5       stats socket /run/haproxy/admin.sock mode 660 level admin expose-fd listeners
6       stats timeout 30s
7       user haproxy
8       group haproxy
9       daemon
10
11      # Default SSL material locations
12      ca-base /etc/ssl/certs
13      crt-base /etc/ssl/private
14
15      # Default ciphers to use on SSL-enabled listening sockets.
16      # For more information, see ciphers(1SSL). This list is from:
17      #  https://hynek.me/articles/hardening-your-web-servers-ssl-ciphers/
18      # An alternative list with additional directives can be obtained from
19      #  https://mozilla.github.io/server-side-tls/ssl-config-generator/?server=haproxy
20      ssl-default-bind-ciphers ECDH+AESGCM:DH+AESGCM:ECDH+AES256:DH+AES256:EC-
21  DH+AES128:DH+AES:RSA+AESGCM:RSA+AES:!aNULL:!MD5:!DSS
22      ssl-default-bind-options no-sslv3
23
24  defaults
25      log     global
26      mode    tcp
27      option  tcplog
28      option  dontlognull
29      timeout connect 5000
30      timeout client  50000
31      timeout server  50000
32
33  frontend emqx_tcp                   # set frontend for load balancer on 172.16.8.206:1883
34      bind *:1883
35      default_backend emqx_tcp_back
36
37  frontend emqx_dashboard             # set emqx common dashboard on 172.16.8.206:18083
38      bind *:18083
39      default_backend emqx_dashboard_back
40
41  backend emqx_tcp_back               # set emqx backend
42      balance roundrobin              # select load balancing algorithm
43      mode tcp
44      server emqx1 172.16.8.203:1883 check weight 250     # name cluster nodes, ports, choose weights
45      server emqx2 172.16.8.215:1883 check weight 180
46
47  backend emqx_dashboard_back
48      balance roundrobin
49      mode tcp
50      server emqx1 172.16.8.203:18083 check
51      server emqx2 172.16.8.215:18083 check
```

# Βιβλιογραφία


***Internet of Things***

- https://en.wikipedia.org/wiki/Internet_of_things
- https://www.sas.com/el_gr/insights/big-data/internet-of-things.html

***MQTT***

- http://mqtt.org/
- https://en.wikipedia.org/wiki/MQTT
- http://www.steves-internet-guide.com/into-mqtt-python-client/

***Mosquitto MQTT Broker***

- https://mosquitto.org/
- https://github.com/eclipse/mosquitto
- http://www.steves-internet-guide.com/mosquitto-broker/
- https://www.digitalocean.com/community/tutorials/how-to-install-and-secure-the-mosquitto-mqtt-messaging-broker-on-ubuntu-16-04

***EMQX Broker***

- https://www.emqx.io/
- https://docs.emqx.io/broker/latest/en/
- https://github.com/emqx/emqx

***Clustering***

- https://en.wikipedia.org/wiki/Cluster_analysis
- https://www.hivemq.com/blog/clustering-mqtt-introduction-benefits/

- https://docs.emqx.io/broker/v3/en/cluster.html

***HAProxy Load Balancer***

- https://en.wikipedia.org/wiki/Load_balancing_(computing)
- http://www.haproxy.org/
- https://www.haproxy.com/blog/the-four-essential-sections-of-an-haproxy-configuration/?fbclid=IwAR2kVUnFpJ92PsZ-Cu-pOyA4jGobxy6teCO8dCXO1JPUs3N8EJrA5sJlLnjQ
- https://cbonte.github.io/haproxy-dconv/1.8/configuration.html?fbclid=IwAR3xSNObs4ylZ_r39MNSMx1B9KWB1aQUZIAS-tYdS15zWH2qRT3qmILOY5g0#4-maxconn

***Other MQTT Brokers Researched***

- https://www.hivemq.com/
- https://vernemq.com/
- https://www.cloudmqtt.com/
- http://www.steves-internet-guide.com/mqtt-hosting-brokers-and-servers/
- https://en.wikipedia.org/wiki/Comparison_of_MQTT_implementations

***TICK Stack***

- https://www.influxdata.com/time-series-platform/
- https://www.thoughtworks.com/radar/platforms/tick-stack
- https://www.influxdata.com/blog/introduction-to-influxdatas-influxdb-and-tick-stack/
- https://docs.influxdata.com/influxdb/v1.7/administration/config/
- https://en.wikipedia.org/wiki/Database

***Other***


- https://en.wikipedia.org/wiki/JSON
- http://www.scalagent.com/IMG/pdf/Benchmark_MQTT_servers-v1-1.pdf
- http://pycurl.io/docs/latest/index.html
- https://docs.emqx.io/broker/v3/en/rest.html?fbclid=IwAR1qgybzWN-2LNYN328t-U6GcPGhvbxlbPoMnzMeSbRwPBvaYTUc7ss4RgkE
- M. H. Miraz, M. Ali, P. S. Excell and R. Picking, "A review on Internet of Things (IoT), Internet of Everything (IoE) and Internet of Nano Things (IoNT)," 2015 Internet Technologies and Applications (ITA), Wrexham, 2015, pp. 219-224.
- Benchmark of MQTT Servers, http://www.scalagent.com/IMG/pdf/JoramMQ_MQTT_white_paper-v1-2.pdf