Ατομική Διπλωματική Εργασία

Σχεδιασμός και Υλοποίηση Συστατικών Διαχείρισης Δεδομένων στην Αρχιτεκτονική Triabase

Σταυρούλλα Κούμου

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Ιούνιος 2021

ΠΑΝΕΠΙΣΤΗΜΙΟ ΚΥΠΡΟΥ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Σχεδιασμός και Υλοποίηση Συστατικών Διαχείρισης Δεδομένων στην Αρχιτεκτονική Triabase

Σταυρούλλα Κούμου

Επιβλέπων Καθηγητής
Δημήτρης Ζεϊναλιπούρ

Η Ατομική Διπλωματική Εργασία υποβλήθηκε προς μερική εκπλήρωση των απαιτήσεων απόκτησης του πτυχίου Πληροφορικής του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου

Ιούνιος 2021

Ευχαριστίες

Με την ολοκλήρωση της παρούσας διπλωματικής εργασίας οφείλω να ευχαριστήσω όσα άτομα με συνέβαλαν στην επίτευξη του στόχου μου.

Αρχικά, θα ήθελα να ευχαριστήσω τον κ. Δημήτρη Ζεϊναλιπούρ, καθηγητή του Τμήματος Πληροφορικής του Πανεπιστημίου Κύπρου και επιβλέποντα της εργασίας αυτής. Παρ' όλες τις πρωτοφανείς προκλήσεις που κληθήκαμε να αντιμετωπίσουμε εν καιρώ πανδημίας, με τις τακτικές διαδικτυακές συναντήσεις που είχαμε και τη συνεχής καθοδήγηση που μου πρόσφερε τον τελευταίο χρόνο, αισίως τα καταφέραμε. Επιπλέον, θα ήθελα να τον ευχαριστήσω για το χρόνο και ενέργεια που επένδυσε σε εμένα, αφού κατά τη διάρκεια της συνεργασίας μας απέκτησα πολλές γνώσεις και δεξιότητες που σαφώς και θα χρειαστώ στην μετέπειτα επαγγελματική μου σταδιοδρομία.

Στη συνέχεια, θα ήθελα να ευχαριστήσω την οικογένεια μου, η οποία με στήριζε οικονομικά καθ' όλη τη διάρκεια των προπτυχιακών μου σπουδών και ήταν πάντα στο πλευρό μου όταν τη χρειαζόμουν.

Ακόμα, ένα μεγάλο ευχαριστώ στην κ. Κωνσταντίνα Χρίστου, ειδική ψυχολόγο-ψυχοθεραπεύτρια και σύμβουλο μου, για την καθοδήγηση και τις συμβουλές της, όπως και την καλύτερη μου φίλη Ελένη Ιωακείμ, προπτυχιακή φοιτήτρια του Πανεπιστημίου Κύπρου, για την συνεχή υποστήριξη της.

Τέλος, ευχαριστώ τόσο τον Ηρόδοτο Δημητρίου, απόφοιτο του Πανεπιστημίου Κύπρου, όσο και τον Πάνο Δρακάτο, διδακτορικό φοιτητή του Πανεπιστημίου Κύπρου, για τη συμβολή τους στην έρευνα που διεξάχθηκε στα πλαίσια της παρούσας διπλωματικής εργασίας.

Η διπλωματική αυτή βασίζεται στη γνώση που έχω αποκτήσει από τη συμμετοχή μου στα παρακάτω έγκριτα άρθρα κατά τη διάρκεια των προπτυχιακών σπουδών μου.

- *"Triastore: A Web 3.0 Blockchain Datastore for Massive IoT Workloads", Panagiotis Drakatos, Erodotos Demetriou, Stavroulla Koumou, Andreas Konstantinidis, Demetrios Zeinalipour-Yazti, The 22nd IEEE International Conference on Mobile Data Management (MDM'21), IEEE Press, 6 pages, June 15 - June 18, 2021, Toronto, Canada, 2021. (in-press)*

- *"Towards a Blockchain Database for Massive IoT Workloads", Panagiotis Drakatos, Erodotos Demetriou, Stavroulla Koumou, Andreas Konstantinidis, Demetrios Zeinalipour-Yazti, Proceedings of the 3rd Intl. Workshop on Blockchain and Data Management (BlockDM'21), IEEE Press, pp. 4 pages, pp. 76-79, DOI: 10.1109/ICDEW53142.2021.00021, April 19, 2021, Chania, Crete, Greece, 2021.*

Περίληψη

Με την τεράστια εξέλιξη στον τομέα του *Internet of Things* (*IoT*) ο όγκος δεδομένων που παράγουν οι ευρέως, πλέον, υιοθετημένες έξυπνες συσκευές όλο και αυξάνεται, και μαζί του αυξάνονται και οι ανάγκες/απαιτήσεις των συστημάτων αυτών. Ως αποτέλεσμα αυτού, οι υφιστάμενες λύσης αποθήκευσης, επεξεργασίας, διαχείρισης και διαμοιρασμού δεδομένων παρουσιάζουν διάφορα προβλήματα ή/και αδυνατούν να αντεπεξέλθουν. Συγκεκριμένα, η χρήση των υπάρχων υποδομών για την εξασφάλιση της απαιτούμενης ιδιωτικότητας, διαφάνειας, αλλά και (υψηλής) επίδοσης δεν αποτελεί βιώσιμη λύση λόγω, τόσο του κεντρικοποιημένου χαρακτήρα, όσο και του μεγάλου κόστους συντήρησης τέτοιων συμβατικών συστημάτων.

Στόχος της παρούσας διπλωματικής εργασίας είναι η υλοποίηση και αποτίμηση μιας πρώτης έκδοσης των επιπέδων εφαρμογής (*Application Layer*) και αποθήκευσης (*Storage Layer*) του συστήματος *Triabase*, το οποίο χρησιμοποιεί την νεοφυή και αρκετά υποσχόμενη τεχνολογία αλυσίδας συστοιχιών (*Blockchain*) αναζητώντας λύσεις στα προαναφερθέντα προβλήματα. Το σύστημα αυτό αποτελεί μια ιδιωτική αλυσίδα συστοιχιών με δικαιώματα (*Private Permissioned Blockchain*) για *IoT* δεδομένα τα οποία αποθηκεύονται στο Blockchain στη συμπαγή μορφή των μοντέλων μηχανικής μάθησης. Ειδικότερα, χρησιμοποιώντας τεχνικές μηχανικής μάθησης στα άκρα (*Edge*), το σύστημα αποσκοπεί να μειώσει τον όγκο δεδομένων που μεταφέρονται (από του αισθητήρες των *IoT* συστημάτων) προς αποθήκευση (στο *Blockchain*), έτσι ώστε να υπερβεί τα θέματα χαμηλής επίδοσης που αντιμετωπίζουν οι κατά τα άλλα ασφαλείς και διαφανείς υπάρχον τεχνολογίες *Blockchain*.

Το σύστημα *Triabase* χωρίζεται σε τρία επίπεδα: το επίπεδο άκρων (*Edge Layer*), το επίπεδο εφαρμογής (*Application Layer*) και το επίπεδο αποθήκευσης (*Storage Layer*) που αποτελείτε από ένα δίκτυο αλυσίδας συστοιχιών (*Blockchain*). Στο επίπεδο ακρών γίνεται η παραγωγή δεδομένων και η σύμπτυξη τους σε μοντέλα μηχανικής μάθησης, τα οποία στη συνέχεια προωθούνται (από εξουσιοδοτημένους χρήστες/συσκευές) στο επίπεδο εφαρμογής. Το επίπεδο εφαρμογής, αφού επεξεργαστεί κατάλληλα τα μοντέλα, τα αποστέλλει στο *Blockchain* (επίπεδο αποθήκευσης) για αποθήκευση. Επίσης, το

επίπεδο εφαρμογής, υλοποιεί μηχανισμούς ανάκτησης και διαχείρισης των δεδομένων που βρίσκονται αποθηκευμένα στο σύστημα, μέσω ενός γραφικού περιβάλλοντος υποβολής επερωτήσεων. Το τελευταίο επίπεδο ασχολείται με τη διανομή και αποθήκευση δεδομένων σε ένα κατανεμημένο σύστημα αλυσίδας συστοιχιών (*Blockchain*), και (σε μεταγενέστερο στάδιο) παρέχει διάφορους μηχανισμούς ευρετηρίασης δεδομένων.

Στα πλαίσια της παρούσας δουλειάς, έγιναν πειράματα που αφορούν τόσο την αποθήκευση, όσο και την ανάκτηση μοντέλων μηχανικής μάθησης από το επίπεδο αποθήκευσης του *Triabase* μέσω του επιπέδου εφαρμογής του. Πιο συγκεκριμένα, διεξάχθηκαν πειράματα αξιολόγησης της επίδοσης του συστήματος για σκοπούς ρύθμισης διαφόρων εσωτερικών παραμέτρων των επιπέδων του. Επίσης, πραγματοποιήθηκε έλεγχος συμβατότητας διαφόρων τύπων και μεγεθών αρχείων που χρησιμοποιούνται για την σειριοποίηση μοντέλων μηχανικής μάθησης με το επίπεδο αποθήκευσης του συστήματος.

Τα αποτελέσματα της πειραματικής αξιολόγησης ήταν αρκετά θετικά. Μέσω αυτών φάνηκε πως το σύστημα *Triabase* μπορεί να υποστηρίξει την αποθήκευση μοντέλων μηχανικής μάθησης που παράγονται από τις πλείστες πλατφόρμες μηχανικής μάθησης σήμερα. Ακόμα, η ρύθμιση των εσωτερικών παραμέτρων του συστήματος ήταν επιτυχής.

Περιεχόμενα

Ευχαριστίες ..... **iii**

Περίληψη ..... **v**

Περιεχόμενα ..... **vii**

Κεφάλαιο 1 Εισαγωγή **1**

1.1 Παρακίνηση 1
1.2 Πρόβλημα 2
1.3 Συνεισφορά 4
1.4 Περίγραμμα εργασίας 7

Κεφάλαιο 2 Σχετική Δουλειά **9**

2.1 Κλασικά IoT Pipelines 10
2.2 InfluxDB: Μια Βάση Δεδομένων για Χρονοσειρές 11
2.2.1 Εγγραφές Δεδομένων / Points 11
2.2.2 Εσωτερικοί Μηχανισμοί της Βάσης 12
2.3 Τεχνική Ομοσπονδιακής Μάθησης 14
2.4 Αλυσίδες Συστοιχιών 14
2.4.1 Είδη Αλυσίδων Συστοιχιών 15
2.5 Ερευνητικά Άρθρα 17
2.5.1 Decaying Telco Big Data with Data Postdiction 17
2.5.2 CAPER: A Cross-Application Permissioned Blockchain 19
2.5.3 BlockchainDB: A Shared Database on Blockchains 21
2.5.4 AnyLog: A Grand Unification of the Internet of Things 23

Κεφάλαιο 3 Αρχιτεκτονική Triabase **27**

3.1 Επίπεδα Συστήματος Triabase 27
3.1.1 Επίπεδο Ακρών 27
3.1.2 Επίπεδο Εφαρμογής 29
3.1.3 Επίπεδο Αποθήκευσης 31
3.2 Ροή Εκτέλεσης / Δεδομένων 33

Κεφάλαιο 4 Συστατικό Αποθήκευσης 35

4.1 Απαιτήσεις Επιπέδου Αποθήκευσης 36
4.2 Περιορισμοί Αλυσίδας Συστοιχιών 37
4.3 Σχεδιασμός Συστατικού Αποθήκευσης 38
4.3.1 Σελιδοποίηση Δεδομένων 38
4.3.2 Δομές Δεδομένων 39
4.3.3 World State Database 41
4.4 Υλοποίηση Smart Contract 41

Κεφάλαιο 5 Συστατικό Διεπαφής 48

5.1 Απαιτήσεις Επιπέδου Εφαρμογής 49
5.2 Περιορισμοί 49
5.3 Σχεδιασμός Συστατικού Διεπαφής 50
5.3.1 Αρχιτεκτονική REST 50
5.3.2 Σχήμα Δεδομένων Διεπαφής 51
5.3.3 API Επιπέδου Εφαρμογής 53
5.3.4 Επιπρόσθετες Λειτουργίες Διακομιστή 54
5.4 Υλοποίηση Διακομιστή REST 55
5.4.1 Κύριες Συναρτήσεις 55
5.4.2 Ενεργοποίηση Επιπρόσθετων Λειτουργιών Διακομιστή 60
5.4.3 Παραδείγματα Χρήσης Διεπαφής 61

Κεφάλαιο 6 Πειραματική Αξιολόγηση 67

6.1 Πειραματική Διάταξη 68
6.2 Δεδομένα Αξιολόγησης 69
6.3 Διαδικασία Διεξαγωγής Πειραμάτων 70
6.3.1 Εισαγωγή Δεδομένων 71
6.3.2 Ανάκτηση Δεδομένων 71
6.4 Αποτίμηση Αποτελεσμάτων 72
6.4.1 Μέγεθος Σελίδας Δεδομένων 72
6.4.2 Μοντέλα Μεγάλου Μεγέθους 75
6.4.3 Ρυθμός Αιτημάτων Εισόδου 76
6.4.4 Συμπίεση Δεδομένων 77

Κεφάλαιο 7 Συμπεράσματα & Μελλοντική Δουλειά 79

7.1 Συμπεράσματα 79
7.2 Μελλοντική δουλειά 82

Βιβλιογραφία 84

Παράρτημα Α 1

Παράρτημα Β 1

Κεφάλαιο 1

Εισαγωγή

1.1 Παρακίνηση 1

1.2 Πρόβλημα 2

1.3 Συνεισφορά 4

1.4 Περίγραμμα εργασίας 7

1.1 Παρακίνηση

Στις μέρες μας, παρατηρείτε μια ραγδαία ανάπτυξη στον τομέα του *Internet of Things* (*IoT*). Ειδικότερα, μέχρι το 2022 υπολογίζεται πως πάνω από 500 έξυπνες συσκευές θα κατέχονται από το κάθε αναπτυγμένο νοικοκυριό. Αυτό οφείλεται στην μεγάλη πρόοδο διαφόρων άλλων τεχνολογιών, όπως για παράδειγμα το 5G και το WiFi 6. Με την υιοθέτηση των έξυπνων συσκευών να κάνει την καθημερινότητά μας όλο και πιο εύκολη, κανείς δεν αρνείται τις θετικές επιδράσεις των καινοτόμων εφαρμογών του *IoT* στην καθημερινότητα μας. Ωστόσο, η συνεχής ανάπτυξη του τομέα έχει εγείρει διάφορα ερωτήματα και προκλήσεις που έχουν κερδίσει το ενδιαφέρων των εμπλεκόμενων ερευνητικών κοινοτήτων, αφού οι ευκαιρίες για έρευνα είναι πολλές (και πολλά υποσχόμενες). Μερικές εξ' αυτών είναι η ενσωμάτωση τεχνικών μηχανικής μάθησης (*machine learning*), επαυξημένης πραγματικότητας (*augmented reality*), συνεργατικής συγκριτικής αξιολόγησης (*cooperative benchmarking*), κλπ, στις έξυπνες συσκευές του μέλλοντος.

Μέχρι στιγμής, κάθε αρχιτεκτονική ενός *IoT* συστήματος περιλαμβάνει μια κοινόχρηστη (από τις συσκευές του συστήματος) βάση δεδομένων- *Relational* (*SQL*),

Document (*MongoDB*), *Temporal* (*InfluxDB*), κλπ. Η βάση αυτή χρησιμοποιείτε για την αποθήκευση και διαχείριση της παραγόμενης πληροφορίας, αλλά επιτρέπει και την μετέπειτα επεξεργασία των δεδομένων των συσκευών. Συνεπώς, η κοινόχρηστη βάση δεδομένων τέτοιων συστημάτων λειτουργεί ως γέφυρα μεταξύ του επιπέδου γένεσης των δεδομένων στα άκρα (*Edge*) και του επιπέδου ανάλυσης τους σε πραγματικό χρόνο.

Παράλληλα, η υιοθέτηση τεχνικών *machine learning* σε διάφορους τομείς της καθημερινότητας μας είναι αρκετά συνηθισμένη. Λαμβάνοντας υπόψη τη δυνατότητα αναγνώρισης και πρόβλεψης μοτίβων σε δεδομένα, και, κατά συνέπεια, την βελτιστοποίηση της επεξεργασίας ή/και αποθήκευσής τους, δεν αποτελεί πλέον ασυνήθιστο φαινόμενο το γεγονός πως σχεδόν όλοι οι σύγχρονοι επεξεργαστές (ανεξαρτήτως συσκευής εφαρμογής) διαθέτουν ειδική μονάδα επεξεργασίας *machine learning* μοντέλων.

Επιπλέον, αύξηση παρατηρείτε στο ερευνητικό και βιομηχανικό ενδιαφέρον για την αναδυόμενη τεχνολογία *Blockchain*. Διάφοροι τομείς που επιθυμούν να εντάξουν τον αποκεντρωμένο και διαφανή χαρακτήρα της τεχνολογίας αλυσίδας συστοιχιών (*Blockchain*) την έχουν πλέον ενσωματώσει στα συστήματα τους. Στους τομείς αυτούς ενδέχεται μελλοντικά να εντάσσεται και ο τομέας του *IoT*, για τον οποίο οι υπάρχον εφαρμογές αλυσίδων συστοιχιών είναι επί το πλείστο ερευνητικής φύσεως.

1.2 Πρόβλημα

Στρέφοντας την προσοχή μας στο επερχόμενο *Web 3.0*, μπορούμε να εντοπίσουμε αρκετές μελλοντικές εφαρμογές οι οποίες δεν μπορούν να υποστηριχτούν από τις υπάρχον υποδομές και τεχνολογίες (συστήματα διαχείρισης δεδομένων), λόγω των πρωτοφανών αναγκών τους.

Δύο χαρακτηρίστηκα παραδείγματα εντοπίζονται στις οι έξυπνες πόλεις (*smart cities*) του μέλλοντος. Παίρνοντας την διαχείριση των μεγάλων τηλεπικοινωνιακών δεδομένων [1][2] ως πρώτο παράδειγμα εφαρμογής, το σύστημα φιλοδοξεί να καλύψει τις ανάγκες διαφάνειας και αμεταβλητότητας (*immutability*) των δεδομένων αυτών, τις οποίες

δημιουργούν κυρίως οι απαιτούμενοι (δια νόμου) έλεγχοι συμμόρφωσης (*monitoring and compliance*). Ένα ακόμα ενδιαφέρον παράδειγμα εφαρμογής αποτελεί η συνεργατική (*collaborative*) συγκέντρωση δεδομένων κυκλοφοριακής συμφόρησης στα άκρα (*Edge*), για σκοπούς αποτελεσματικού προγραμματισμού διακίνησης (*public transit planning*) μεγάλης κλίμακας- με τη χρήση μοντέλων μηχανικής μάθησης που προκύπτουν από τα δεδομένα αυτά. Οι ανάγκες που εντοπίζονται στο δεύτερο αυτό σενάριο είναι, ξανά, η διαφάνεια και αμεταβλητότητα των δεδομένων. Και στις δύο περιπτώσεις, το σύστημα *Triabase* φιλοδοξεί να αποτελέσει μια αποδοτική και κλιμακώσιμη λύση/επιλογή.

Επιπλέον, επιστρέφοντας στο παρόν, υπάρχουν και τα προβλήματα που ήδη αντιμετωπίζουμε. Στα προβλήματα αυτά περιλαμβάνονται η κλιμακωσιμότητα, η συνέπεια (*consistency*), και η διαφάνεια (*transparency*) των δεδομένων και διαδικασιών στα δίκτυα των αποκεντρωμένων συστημάτων σήμερα.

Όπως γνωρίζουμε, οι συμβατικές υποδομές αποθήκευσης και επεξεργασίας δεδομένων έχουν πεπερασμένο χώρο αποθήκευσης και εύρος υπολογιστικών πόρων, κάτι το οποίο δημιουργεί πολλαπλά προβλήματα σε εφαρμογές με ροές εισόδου (δεδομένων) υψηλού ρυθμού – *data ingestion rate* – (π.χ., *IoT*). Ειδικότερα, η επεξεργασία δεδομένων είναι αρκετά χρονοβόρα και δύσκολη. Συνεχίζοντας, ο κεντρικοποιημένος χαρακτήρας των προαναφερθείς υποδομών εγείρει περεταίρω προβλήματα, αφού μια μονολιθική υποδομή αποτελεί τόσο μοναδικό σημείο αποτυχίας του συστήματος, αλλά και ελκυστικό σημείο κακόβουλων επιθέσεων που αποσκοπούν την καταπάτηση της ιδιωτικότητας και ακεραιότητας των δεδομένων που συσσωρεύονται σε αυτά. Επίσης, εν μέρει καταπάτηση της ιδιωτικότητας των (ενδεχομένως) ευαίσθητων δεδομένων που παράγουν τα άκρα του δικτύου (*Edge*) αποτελεί και η μεταφορά τους σε μια κεντρική δομή.

Παράδειγμα συμβατικής υποδομής αποτελεί μια συστοιχία διακομιστών σε ένα κέντρο δεδομένων (*data center*) στην οποία όλες οι έξυπνες συσκευές ενός *IoT* συστήματος αποστέλλουν τα τοπικά τους δεδομένα με σκοπό την εκπαίδευση ενός *machine learning* μοντέλου. Σε μια τέτοια υποδομή η συγκεκριμένη διαδικασία (εκπαίδευση) χρειάζεται μερικές, έως και πάμπολλες, ώρες (ανάλογα με τον όγκο των δεδομένων εκπαίδευσης), λόγω του κεντρικοποιημένου χαρακτήρα της.

Για σκοπούς αποφυγής μεταφοράς αυτούσιων δεδομένων από τα άκρα (*Edge*) σε πιο ισχυρά (υπολογιστικά) επίπεδα του συστήματος, θα μπορούσε να γίνει χρήση της τεχνικής ομοσπονδιακής μάθησης (*Federated Learning*). Η γενική ιδέα στο *Federated Learning* είναι η εκπαίδευση πολλαπλών τοπικών μοντέλων μηχανικής μάθησης στα άκρα (*Edge*), και στη συνέχεια η συσσωμάτωση τους σε ένα και μόνο καθολικό μοντέλο. Επομένως, λόγω του ότι η τεχνική αυτή δεν απαιτεί καμία μεταφορά αυτούσιων δεδομένων (γίνεται μεταφορά μοντέλων μόνο), εκτός από την εξασφάλιση της ασφάλειας των δεδομένων, μειώνεται επίσης και ο φόρτος του δικτύου του συστήματος.

Συνεχίζοντας, για σκοπούς αποφυγής αποθήκευσης δεδομένων σε κεντρικοποιημένες υποδομές, μπορεί να γίνει χρήση της τεχνολογίας *Blockchain*, που εγγυάται την συνέπεια και διαφάνεια των δεδομένων των συστημάτων που την υιοθετούν. Η τεχνολογία αυτή ωστόσο δεν είναι ιδιαίτερα κλιμακώσιμη, κυρίως λόγω των υπολογιστικά ακριβών πρωτοκόλλων (ομοφωνίας) που χρησιμοποιεί. Οι υφιστάμενες πλατφόρμες *Blockchain* πάσχουν από χαμηλή ρυθμαπόδοση (*throughput*) και μεγάλες καθυστερήσεις (*latency*) κατά την αποθήκευση και επαναφορά δεδομένων. Συγκεκριμένα, στο δίκτυο οικονομικών συναλλαγών *Bitcoin*, όπου και υλοποιήθηκε το πρώτο *Blockchain*, το *throughput* και το *latency* είναι 7 *Transactions/Second* και 10 λεπτά, αντίστοιχα.

Στα πλαίσια αυτής της διπλωματικής εργασίας ερευνήθηκε η πιθανότητα συνδυασμού των τεχνολογιών *Blockchain* και *machine learning* για την αποτελεσματική αποθήκευση και διαχείριση *machine learning* μοντέλων που δύναται να παράγονται σε *IoT* συστήματα. Πιο συγκεκριμένα, η παρούσα δουλειά υλοποιεί μέρος του συστήματος *Triabase* που παρουσιάζεται στη συνέχεια.

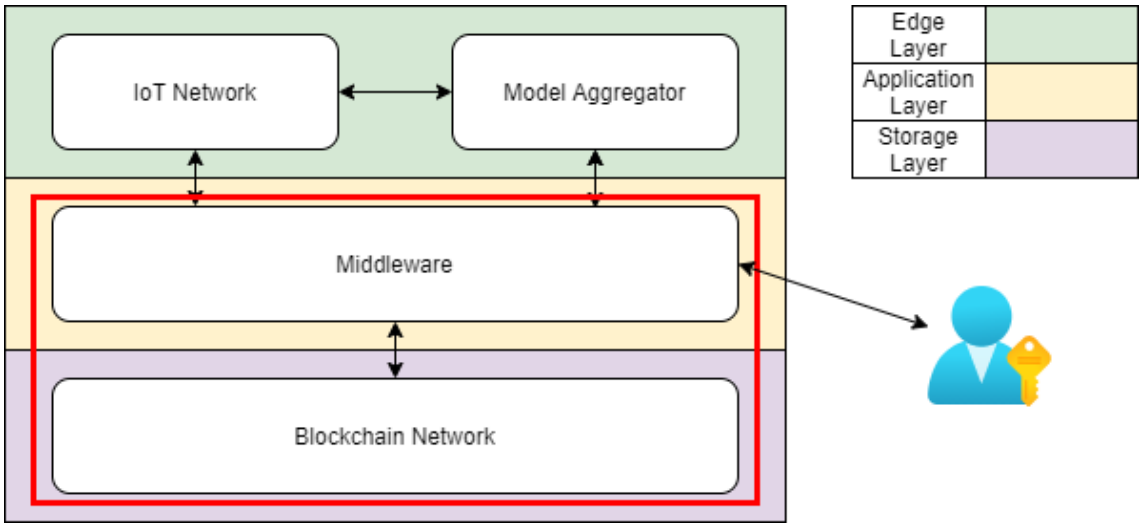
1.3 Συνεισφορά

Στα πλαίσια της εργασίας αυτής γίνεται υλοποίηση και αποτίμηση δύο εκ των συστατικών που ενσωματώνονται στην αρχιτεκτονική του συστήματος *Triabase*. Το όνομα *Triabase* αποτελεί μίξη των λέξεων «Τρία» και «Base», που παραπέμπουν στο

Web 3.0 και στις Βάσεις Δεδομένων, τονίζοντας έτσι την πρωτοποριακή αρχιτεκτονική που προτείνεται.

Σε γενικές γραμμές, το *Triabase* αποτελεί ένα σύστημα Βάσης Δεδομένων για *IoT* δεδομένα το οποίο χρησιμοποιεί ομοσπονδιακή μηχανική μάθηση (*Federated Learning*) για εκπαίδευση μοντέλων μηχανικής μάθησης στα άκρα (*Edge*) και μια ιδιωτική αλυσίδα συστοιχιών με δικαιώματα (*Permissioned Blockchain*) για την αποθήκευση/διαχείριση των μοντέλων αυτών. Σκοπός της αρχιτεκτονικής αυτής, είναι η παροχή ενός ασφαλούς και κλιμακώσιμου συστήματος διαχείρισης δεδομένων για τον τομέα του *IoT*, βελτιώνοντας τις επιδώσεις της τεχνολογίας *Blockchain* μέσω του συνδυασμού της με την τεχνική μηχανικής μάθησης *Federated Learning*.

Η αρχιτεκτονική του συστήματος *Triabase* φαίνεται στο Σχήμα 1.1.



Σχήμα 1.1 : Η αρχιτεκτονική του συστήματος *Triabase*. Σε κόκκινο πλαίσιο τα επίπεδα με τα οποία ασχολείται αυτή η διπλωματική εργασία.

Το *Triabase* χωρίζεται σε τρία μέρη. Το επίπεδο άκρων (*Edge Layer*), το επίπεδο εφαρμογής (*Application Layer*) και το επίπεδο αποθήκευσης (*Storage Layer*) που αποτελείτε από ένα δίκτυο αλυσίδας συστοιχιών (*Blockchain*). Η ροή των δεδομένων γίνεται από το *Edge Layer*, όπου παράγονται, προς το *Storage Layer*, όπου αποθηκεύονται (στο Σχήμα 1.1, από πάνω προς τα κάτω).

Στο *Edge Layer* υλοποιείται το κομμάτι του *Federated Learning*. Ειδικότερα, οι έξυπνες συσκευές (του *IoT* δικτύου που χρησιμοποιεί το *Triabase* ως βάση δεδομένων του) εκπαιδεύουν τοπικά μοντέλα μηχανικής μάθησης και τα προωθούν στο *Application*

Layer όπου θα προχωρήσουν μετέπειτα για αποθήκευση στο *Storage Layer*. Παράλληλα, ο *Model Aggregator* ειδοποιείτε πως η τοπική μηχανική μάθηση (*local training*) έχει ολοκληρωθεί, έτσι ώστε να προχωρήσει στη δημιουργία ενός καθολικού μοντέλου μηχανικής μάθησης μέσω συσσωμάτωσης των πολλαπλών τοπικών (*global training*).

Η επιλογή αποθήκευσης των τοπικών μοντέλων στο *Blockchain* γίνεται για λόγους ασφάλειας, αφού τα άκυρα μοντέλα (από μη εξουσιοδοτημένες/κακόβουλες πηγές/συσκευές) θα απορριφθούν από τους κόμβους του δικτύου *Blockchain*. Έτσι, εξασφαλίζεται η εγκυρότητα του καθολικού μοντέλου, αφού ο (έμπιστος) *Model Aggregator* δεν θα λάβει ψεύτικα ή/και παραποιημένα δεδομένα (τοπικά μοντέλα).

Το *Application Layer* του συστήματος ουσιαστικά αποτελεί τον ενδιάμεσο μεταξύ των *Edge* και *Storage Layers*. Το επίπεδο αυτό χρησιμοποιείτε ως συνδετικός κρίκος του ανώτατου (*Edge*) και κατώτατου (*Storage*) επιπέδου. Πιο συγκεκριμένα, το ανώτατο και κατώτατο επίπεδο παραμένουν ανεξάρτητα και το συστατικό διεπαφής που υλοποιείτε στο *Application Layer* φροντίζει για την μεταξύ τους αλληλεπίδραση. Αυτό επιτυγχάνεται μέσω τυποποίησης και υλοποίησης διαφόρων μεθόδων επικοινωνίας με το *Storage Layer*.

Συνεχίζοντας, το *Storage Layer* του συστήματος, αποτελείτε από ένα *Blockchain Network* όπως αυτό φαίνεται στο Σχήμα 1.1. Το επίπεδο υπό αναφορά φροντίζει για την ασφαλή, αμετάβλητη και διαφανή αποθήκευση των δεδομένων του συστήματος στην συστοιχία κόμβων που το αποτελεί. Περαιτέρω πληροφορίες σχετικά με το πώς αυτό επιτυγχάνεται παραθέτονται στα Κεφάλαια 2 και 3, όπου περιγράφεται η τεχνολογία *Blockchain* και η αρχιτεκτονική *Triabase*, αντίστοιχα, με μεγαλύτερη λεπτομέρεια.

Μέσω της πιο πάνω αρχιτεκτονικής τριών επιπέδων, το σύστημα *Triabase* επιχειρεί να εξασφαλίσει την αμεταβλητότητα, αλλά και τη διαφάνεια των διεργασιών και δεδομένων του, χρησιμοποιώντας την τεχνολογία *Blockchain*. Η επιτυχής ολοκλήρωση των προαναφερόμενων στόχων επιδιώκεται (με αλγοριθμικές τεχνικές) με όσο το δυνατό αποτελεσματικότερη αξιοποίηση των πόρων του, μειώνοντας έτσι τις ενεργειακές του ανάγκες. Επίσης, η (μελλοντική) υποστήριξη πληθώρας επερωτημάτων και μηχανισμών ευρετηρίασης (θα) εγγυάται την υψηλή απόδοσή του συστήματος. Σημαντικό σημείο προς αναφορά αποτελεί και η διασφάλιση της ιδιωτικότητας των

δεδομένων του συστήματος, κάτι που στο *Triabase* γίνεται μέσω της τεχνικής του *Federated Learning*.

Κλείνοντας, να αναφέρουμε πως στην παρούσα έκδοση του συστήματος, μέρος του *Edge Layer* και το *Storage Layer* αξιοποιούν τις πλατφόρμες *TensorFlow* και *Hyperledger Fabric*, αντίστοιχα, ενώ το *Application Layer* αποτελεί ένας REST server με τη χρήση του *Node.js runtime environment*. Επιπλέον, το *Application Layer* προσφέρει την δυνατότητα χειρονακτικής πρόσβασης και διαχείρισης των δεδομένων του συστήματος (από εξουσιοδοτημένους διαχειριστές του συστήματος), αφού η διεπαφή που υλοποιεί είναι κατανοητή και φιλική προς τον χρήστη.

Περισσότερες λεπτομέρειες για το *Triabase* δίνονται στο Κεφάλαιο 3, αλλά και στο ακαδημαϊκό άρθρο [3] το οποίο παρουσιάζει το άμεσα συσχετιζόμενο σύστημα *Triastore*. Οι διαφορές των *Triabase* και *Triastore* δεν επηρεάζουν την παρούσα δουλειά, ωστόσο, να αναφέρουμε πως το *Triastore* αποτελεί, ουσιαστικά, τον πρόγονο (ή/και μια πρώτη έκδοση) του *Triabase*, προσφέροντας μόλις τις θεμελιώδεις λειτουργίες του συστήματος- και επομένως, τεχνικά, δεν νοείται ως μια «βάση δεδομένων». Στην εργασία αυτή ο όρος *Triastore* μπορεί να χρησιμοποιηθεί εναλλακτικά του *Triabase*, αφού η δουλειά που έγινε αφορά κάποιες από τις επικαλυπτόμενες θεμελιώδεις λειτουργίες των δύο αυτών συστημάτων αυτών (αποθήκευση και ανάκτηση δεδομένων).

1.4 Περίγραμμα εργασίας

Η παρούσα διπλωματική εργασία αποτελείτε από εφτά κεφάλαια. Αυτά είναι η «Εισαγωγή», η «Σχετική Δουλειά», η «Αρχιτεκτονική *Triabase*», το «Συστατικό Αποθήκευσης», το «Συστατικό Διεπαφής», η «Πειραματική Αποτίμηση» και τα «Συμπεράσματα».

Σε αυτό το κεφάλαιο αναφέρθηκαν οι λόγοι που ώθησαν στην διεξαγωγή έρευνας στο συγκεκριμένο πεδίο και τα κύρια προβλήματα που αντιμετωπίζονται σε αυτό. Επίσης, έγινε μια εισαγωγική περιγραφή της αρχιτεκτονικής του συστήματος *Triabase* που επιχειρεί να αντιμετωπίσει τα προαναφερθέντα ζητήματα.

Στο δεύτερο κεφάλαιο ακολουθεί περιγραφή σχετικών δουλειών που έχουν γίνει στο πεδίο ενασχόλησης της εργασίας αυτής και των βασικών εννοιών και τεχνολογιών που αξιοποιούνται, τόσο από το σύστημα *Triabase*, όσο και από κάποιες από τις σχετικές δουλειές που αναφέρονται.

Στο τρίτο κεφάλαιο δίνεται μια πιο λεπτομερής περιγραφή της αρχιτεκτονικής του συστήματος *Triabase* και του τρόπου αλληλεπίδρασης των επιπέδων της τελευταίας έκδοσης του.

Το τέταρτο κεφάλαιο επικεντρώνεται στην περιγραφή του συστατικού αποθήκευσης, που αποτελεί μέρος του *Storage Layer* του συστήματος. Σε αυτό το σημείο περιγράφεται η υλοποίηση κάποιων εκ μεθόδων που (θα) καθιστούν την αξιοποίηση ενός *Blockchain* δικτύου ως μια βάση δεδομένων εφικτή.

Το πέμπτο κεφάλαιο εμβαθύνει στα χαρακτηριστικά του συστατικού διεπαφής, που αποτελεί μέρος του *Application Layer*. Στο κεφάλαιο αυτό γίνεται επίσης αιτιολόγηση των αποφάσεων που λήφθηκαν κατά την φάση υλοποίησης του συγκεκριμένου συστατικού.

Η πειραματική αποτίμηση ωφελείας των συστατικών αποθήκευσης και διεπαφής παρουσιάζεται στο έκτο κεφάλαιο. Εδώ περιγράφονται οι παράμετροι που αξιολογήθηκαν και ρυθμίστηκαν μέσω της διεξαγωγής πειραμάτων, όπως και η φύση των δεδομένων, αλλά και η πειραματική διάταξη, που χρησιμοποιήθηκαν.

Τέλος, στο κεφάλαιο επτά γίνεται αναφορά στα γενικά συμπεράσματα που προκύπτουν από την παρούσα διπλωματική εργασία, όπως και παράθεση κάποιων εκ των πιθανών μελλοντικών ερευνητικών κατευθύνσεων για περεταίρω ανάπτυξη του συστήματος *Triabase*.

Κεφάλαιο 2

Σχετική Δουλειά

2.1 Κλασικά IoT Pipelines 10

2.2 InfluxDB: Μια Βάση Δεδομένων για Χρονοσειρές 11

2.2.1 Βασικές Έννοιες 11

2.2.2 Εσωτερικοί Μηχανισμοί της Βάσης 12

2.3 Τεχνική Ομοσπονδιακής Μάθησης 14

2.4 Αλυσίδες Συστοιχιών 14

2.4.1 Είδη Αλυσίδων Συστοιχιών 15

2.5 Ερευνητικά Άρθρα 17

2.5.1 Decaying Telco Big Data with Data Postdiction 17

2.5.2 CAPER: A Cross-Application Permissioned Blockchain 19

2.5.3 BlockchainDB: A Shared Database on Blockchains 21

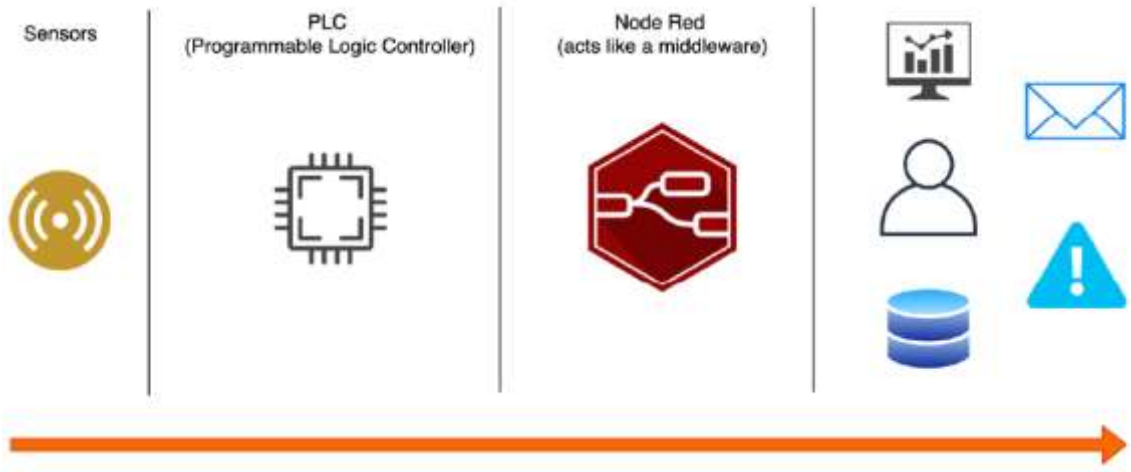
2.5.4 AnyLog: A Grand Unification of the Internet of Things 23

Σε αυτό το κεφάλαιο θα παρουσιαστούν εν συντομία διάφορες τεχνολογίες και ακαδημαϊκά άρθρα τα οποία μελετήθηκαν στα πλαίσια της παρούσας διπλωματικής εργασίας.

2.1 Κλασικά IoT Pipelines

Σε αυτή την υπο-ενότητα εξετάζεται ο τρόπος διαχείρισης δεδομένων σε διάφορα υφιστάμενα *IoT* συστήματα.

Όπως φαίνεται και στο Σχήμα 2.1, το κλασσικό *pipeline* που χρησιμοποιείτε για τη διαχείριση βιομηχανικών *IoT* δεδομένων μπορεί να χωριστεί σε τέσσερα κύρια στρώματα. Το πρώτο στρώμα αποτελείτε από τους αισθητήρες των συσκευών που συμμετέχουν στο *IoT* σύστημα, οι οποίοι παράγουν δεδομένα που προωθούνται στο δεύτερο στρώμα. Εκεί (στο δεύτερο στρώμα), τα δεδομένα ελέγχονται και οργανώνονται (πιθανόν) από κάποιο *Rasberry Pi* ή *Arduino* [4][5] που αναλαμβάνουν το ρόλο του *Programmable Logic Controller* (*PLC*). Στο τρίτο στρώμα, συνήθως γίνεται χρήση ενδιάμεσων λογισμικών (*middleware*), για σκοπούς αυτοματοποίησης διαφόρων διεργασιών που απαιτούνται πριν την αποθήκευση των δεδομένων στο τέταρτο στρώμα του *pipeline*. Το εργαλείο *Node-Red* [6] αποτελεί παράδειγμα τέτοιου (ενδιάμεσου) λογισμικού. Τέλος, όσο αφορά τις υποδομές αποθήκευσης του στρώματος τέσσερα, παραδείγματα αυτών αποτελούν μια βάση δεδομένων, κάποιος *main-server* (για σκοπούς παροχής ειδοποιήσεων στον χρήστη), ή/και κάποια *front-end* εφαρμογή παρουσίασης στατιστικών στοιχείων.



Σχήμα 2.1 : *Industrial IoT Architecture Example*. [4]

2.2 InfluxDB: Μια Βάση Δεδομένων για Χρονοσειρές

Τυπικά παραδείγματα δεδομένων χρονοσειρών στις μέρες μας αποτελούν οι τιμές των μετοχών στο χρηματιστήριο, τα δεδομένα που παράγονται σε συστήματα έξυπνων συσκευών (*IoT*), και πολλά άλλα. Δεδομένου αυτού, στα πλαίσια της παρούσας διπλωματικής εργασίας συμπεριλαμβάνεται και η μελέτη της *InfluxDB* [7], μιας σχετικά καινούριας βάσης δεδομένων ανοικτού κώδικα (*open source*) που δύναται να χρησιμοποιείται για την αποθήκευση δεδομένων χρονοσειρών από συμβατικά *IoT* συστήματα στις μέρες μας.

2.2.1 Εγγραφές Δεδομένων / Points

Η βασική δομή δεδομένων που χρησιμοποιείται στην *InfluxDB* είναι το *Point*, το οποίο αντιπροσωπεύει μια εγγραφή δεδομένων. Ένα *Point* αποτελείτε από τέσσερα συστατικά, ένα *Measurement*, μια ομάδα από *Tags* και *Fields*, και μια χρονοσφραγίδα που το ταυτοποιεί μοναδικά.

- *Measurement*: Χρησιμοποιείτε για την ομαδοποίηση δεδομένων κάτω από μια κοινή περιγραφή. Έχοντας στο μυαλό μας τις παραδοσιακές σχεσιακές βάσης δεδομένων, θα μπορούσαμε να θεωρήσουμε ένα *Measurement* ως έναν πίνακα.
- *Tags* & *Fields*: Χρησιμοποιούνται με αντίστοιχο τρόπο όπως οι στήλες των πινάκων των παραδοσιακών σχεσιακών βάσεων δεδομένων. Η διαφορά των δύο εντοπίζεται στο ότι τα *Tags* χρησιμοποιούνται για τη δημιουργία ευρετηρίων, ενώ τα *Fields* όχι.

```
name: census
-----------------
time                  butterflies honeybees   location    scientist
2015-08-18T00:00:00Z  1           30          1           perpetua
```

Σχήμα 2.2 : Παράδειγμα *Point* στην *InfluxDB*.

Στο Σχήμα 2.2 δίνεται ένα παράδειγμα *Point* στην *InfluxDB*, όπου το ‘census’ είναι το *Measurement* και τα ‘butterflies’, ‘honeybees’, ‘location’ και ‘scientist’ είναι το

σύνολο των *Tags* & *Fields* του συγκεκριμένου *Point*. Η χρονοσφραγίδα του *Point* φαίνεται κάτω από το ‘time’.

2.2.2 Εσωτερικοί Μηχανισμοί της Βάσης

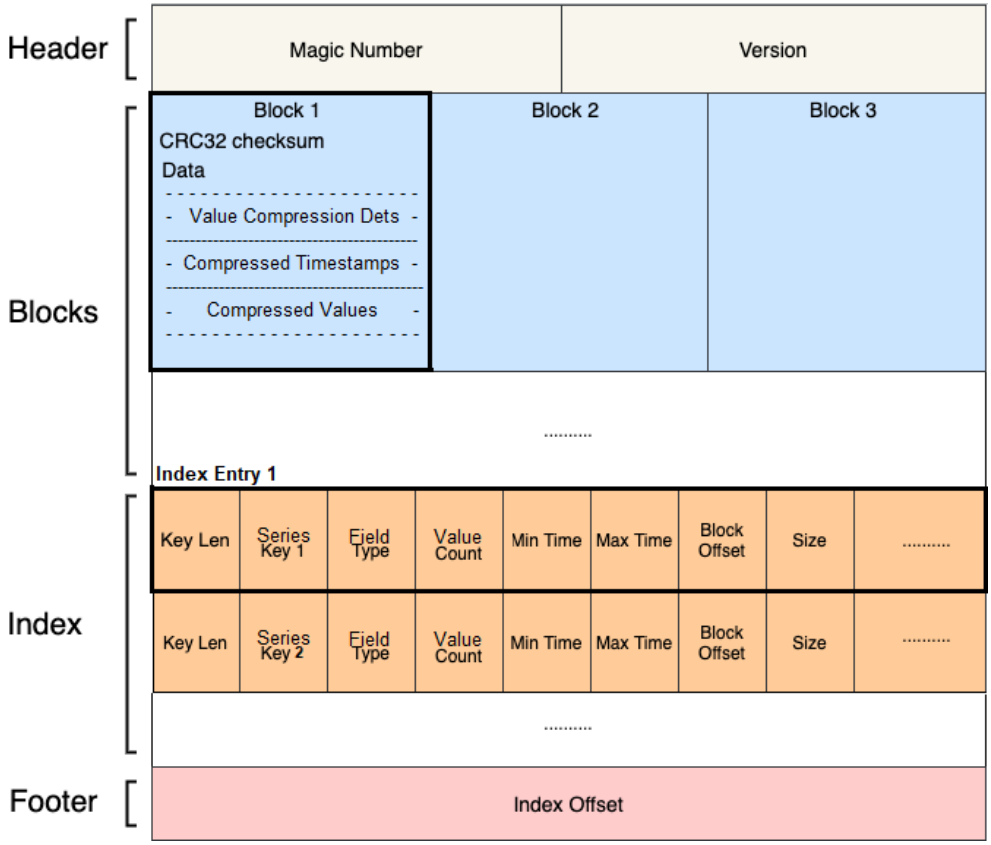
Όπως σε κάθε βάση δεδομένων, το *storage engine* της *InfluxDB* αποτελείτε από διάφορα επιμέρους συστατικά. Τα κυριότερα συστατικά που εντοπίζονται στην *InfluxDB* είναι η μνήμη *Cache*, το *Write Ahead Log*, τα αρχεία *TSM* και τα ευρετήρια *TSI* [7].

Η μνήμη *Cache* είναι ένας σχετικά μικρής χωρητικότητας *volatile* αποθηκευτικός χώρος που χρησιμοποιείτε για την βελτίωση της απόδοσης του συστήματος κατά την εκτέλεση επερωτημάτων (*queries*). Στη μνήμη αυτή βρίσκονται δεδομένα που διαβάστηκαν ή εγγράφηκαν πρόσφατα στην βάση δεδομένων. Τα δεδομένα αυτά, σε συνδυασμό με άλλα που υπάρχουν στο δίσκο, χρησιμοποιούνται για την απάντηση των νέων επερωτημάτων που υποβάλλονται στο σύστημα. Σημαντικό σημείο αποτελεί το ότι, κατά την απάντηση ενός επερωτήματος, τα δεδομένα που ανακτώνται από τον δίσκο αντιγράφονται στην *Cache* για να αποφευχθούν τυχόν ασυνέπειες λόγω των ταυτόχρονων (με την απάντηση του επερωτήματος) εγγραφών δεδομένων που λαμβάνουν χώρα.

Με τον όρο *Write Ahead Log* (*WAL*) [8] αναφερόμαστε στην στρατηγική ενημέρωσης δεδομένων της *InfluxDB*. Συγκεκριμένα, η στρατηγική *WAL* ορίζει την χρήση ενός *log* αρχείου το οποίο ενημερώνεται με τις αλλαγές που γίνονται στα δεδομένα πριν αυτές γίνουν *commit* (μονιμοποιηθούν) στη δευτερεύουσα μνήμη (δίσκο). Η χρήση του πρωτοκόλλου αυτού γίνεται για σκοπούς ανάκαμψης/επαναφοράς (*recovery*) στις περιπτώσεις αποτυχίας της βάσης δεδομένων πριν προλάβουν να γίνουν τα απαραίτητα *commits* στο δίσκο. Σε αυτές τις περιπτώσεις, ανατρέχεται το log αρχείο και γίνονται οι κατάλληλες εγγραφές/διαγραφές δεδομένων από το δίσκο μέχρι η βάση δεδομένων να βρεθεί σε μια συνεπή κατάσταση. Να σημειωθεί ότι το ίδιο το *log* αρχείο μονιμοποιείται στο δίσκο σε τακτά χρονικά διαστήματα.

Όσο αφορά τα αρχεία *TSM*, τα αρχεία αυτά χρησιμοποιούνται για την αποθήκευση των δεδομένων στην ειδική δομή των *Time Structured Merge Trees* [7]. Ειδικότερα, τα δεδομένα ομαδοποιούνται σε blocks σύμφωνα με το *Measurement*, τα *Tags* και το *Field*

τους (το *Series Key* τους), όπου και ταξινομούνται με βάση την χρονοσφραγίδα τους. Να αναφέρουμε ακόμα ότι, με σκοπό την γρηγορότερη πρόσβαση στα δεδομένα, τα *TSM* αρχεία εμπεριέχουν ευρετήρια με δείκτες προς τα σχετικά *blocks*. Την ανατομία ενός *TSM* αρχείου περιγράφει διαγραμματικά το Σχήμα 2.3.



Σχήμα 2.3: Η Δομή ενός *TSM* αρχείου.

Τέλος, έχουμε τα ευρετήρια *TSI* (*Time Series Indexes*). Τα συγκεκριμένα ευρετήρια χρησιμοποιούνται μόνο όταν επιλεγεί το *tsi1* ως η μέθοδος ευρετηρίασης του συστήματος. Εμβαθύνοντας, η επιλογή *tsi1* επιτρέπει την επέκταση του ευρετηρίου του συστήματος στην δευτερεύουσα μνήμη. Αυτό δεν υποστηριζόταν προηγουμένως από την καθαρά *in-memory* στρατηγική ευρετηρίασης της *InfluxDB*, προκαλώντας διάφορα προβλήματα, μέχρι και χάσιμο δεδομένων, αφού ο αριθμός των *Series Keys* που μπορούσε να συγκρατηθεί ήταν ανάλογος του μεγέθους της κύριας μνήμης της μηχανής [7].

2.3 Τεχνική Ομοσπονδιακής Μάθησης

Ως Ομοσπονδιακή/Συνεργατική Μάθηση (*Federated Learning*) [9][10] ονομάζεται η μηχανική μάθηση η οποία γίνεται με κατανεμημένο/αποκεντρωμένο τρόπο σε περιβάλλοντα επικοινωνούντων κόμβων (δίκτυα). Κύρια ιδέα της τεχνικής αυτής αποτελεί η εκπαίδευση τοπικών μοντέλων μηχανικής μάθησης στα άκρα (*machine learning on the Edge*) και, ακολούθως, η συσσωμάτωση τους από κάποιο συντονιστή (*Coordinator*) για τη δημιουργία του καθολικού μοντέλου του συστήματος.

Επιγραμματικά ο αλγόριθμος της τεχνικής Ομοσπονδιακής Μάθησης:

1. Ένας *Coordinator* δημιουργεί το σκελετό του μοντέλου μηχανικής μάθησης που θα εκπαιδευτεί, και αποστέλλει αντίγραφα του στους κόμβους των άκρων του δικτύου (*Edge*).
2. Οι κόμβοι στα άκρα εκπαιδεύουν το μοντέλο με τα τοπικά τους δεδομένα (δημιουργώντας τοπικά μοντέλα).
3. Τα τοπικά μοντέλα αποστέλλονται στον *Coordinator*, ο οποίος αναλαμβάνει την συσσωμάτωση τους σε ένα καθολικό μοντέλο που αντικατοπτρίζει ολόκληρο το δίκτυο.

Ως αποτέλεσμα της μη-ανταλλαγής αυτούσιων δεδομένων (αλλά μόνο μοντέλων) στο δίκτυο που εφαρμόζεται, επιτυγχάνεται η διασφάλιση της ιδιωτικότητας και ασφάλειας της πληροφορίας, αλλά και η αποφόρτιση του δικτύου.

2.4 Αλυσίδες Συστοιχιών

Η πρώτη εφαρμογή της τεχνολογίας αλυσίδων συστοιχιών (*Blockchain*) παρουσιάζεται στο *Bitcoin* [11][12], το κρυπτονόμισμα που έπαιξε τον ρόλο του εναρκτήριου λακτίσματος στην επανάσταση του τομέα κρυπτονομισμάτων, περίπου δεκατρία χρόνια πριν. Με απλούς ορισμούς, η τεχνολογία αυτή είναι ένα αποκεντρωμένο και κατανεμημένο (σε ένα *Peer-to-Peer* δίκτυο) κατάστιχο (*ledger*) στο οποίο καταγράφονται όλες οι αλλαγές που συμβαίνουν σε έναν πόρο με αμετάβλητο τρόπο.

Ειδικότερα, οι αλλαγές αυτές (επονομαζόμενες ως *συναλλαγές*/*transactions*) επιτυγχάνουν την μόνιμη τροποποίηση ενός πόρου, μόνο εν κοινή συναινέσει των κόμβων του δικτύου *Blockchain* όπως αυτή ορίζεται από το πρωτόκολλο ομοφωνίας (*consensus protocol*) [13][14] του. Η αμεταβλητότητα και, εν συνέπεια, η ασφάλεια που εγγυείται η χρήση του *Blockchain* οφείλεται στον υπολογισμό *digests*, τα οποία αποθηκεύονται στο κατάστιχο μαζί με τον πόρο που επιδέχτηκε την σχετική αλλαγή.
Σχετικά με το κατάστιχο του δικτύου *Blockchain*, να αναφέρουμε πως αποτελείτε από πολλαπλά κρυπτογραφημένα αρχεία (*blocks*) τα οποία συνδέονται μεταξύ τους, σχηματίζοντας μια χρονολογική αλυσίδα (*blockchain*). Η έννοια της αλυσίδας δημιουργείται λόγω του ότι τα *blocks* συνδέονται μεταξύ τους με δείκτες (*pointers*) και εμπεριέχουν μεταδεδομένα (συγκεκριμένα, το *digest*) του αμέσως προηγούμενου (χρονολογικά) *block* στην αλυσίδα. Αυτό ενισχύει την αμεταβλητότητα που χαρακτηρίζει την τεχνολογία, αφού η όποια κακόβουλη μεταποίηση δεδομένων του οποιουδήποτε *block* του καταστίχου μπορεί να εντοπιστεί με απλούς ελέγχους στα δυο είδη *digests* που χρησιμοποιούνται.

2.4.1 Είδη Αλυσίδων Συστοιχιών

Το πιο διαδεδομένο είδος *Blockchain* αποτελούν οι Δημόσιες Αλυσίδες Συστοιχιών (*Public Blockchains*) [15][16][17], οι οποίες είναι ανοικτές προς το ευρύ κοινό με αποτέλεσμα ο οποιοσδήποτε να έχει δικαίωμα συμμετοχής σε αυτές. Στα σημαντικότερα χαρακτηριστικά των δημόσιων αλυσίδων (συστοιχιών) περιλαμβάνονται η πλήρης ανωνυμία των μελών τους, όπως και ο αποκεντρωμένος χαρακτήρας τους, αφού κανένα μέλος δεν κατέχει διοικητικά προνόμια στο δίκτυο. Τα μειονεκτήματα, ωστόσο, των δημόσιων αλυσίδων είναι αρκετά αισθητά. Συγκεκριμένα, η δυσκολία κλιμακωσιμότητας και οι κακές επιδόσεις τους αντισταθμίζουν τα προαναφερθέντα πλεονεκτήματα τους. Αυτό συμβαίνει διότι, για την εξασφάλιση της ασφαλούς λειτουργίας του συστήματος [16], γίνεται χρήση πιθανοτικών αλγορίθμων ομοφωνίας (*consensus*) των οποίων η εφαρμογή αυξάνει το υπολογιστικό, και κατ' επέκταση ενεργειακό, κόστος λειτουργίας του δικτύου. Εν συνέπεια, υπάρχει μεγάλη σπατάλη ενέργειας και το δίκτυο είναι αρκετά αργό [18]. [4]

Ένα άλλο είδος Blockchain αποτελούν οι Ιδιωτικές Αλυσίδες Συστοιχιών (*Private Blockchains*) [15][16][17]. Αυτή η κατηγορία αλυσίδων συστοιχιών αναφέρεται σε

Blockchains όπου η πρόσβαση και συμμετοχή είναι ελεγχόμενη από τους διαχειριστές του δικτύου. Το γεγονός αυτό στερεί στα μέλη του δικτύου την ανωνυμία τους, ωστόσο, λόγω της εμπιστοσύνης που υπάρχει μεταξύ των μελών, επιτρέπεται η χρήση αλγορίθμων ομοφωνίας (*consensus*) μικρότερου υπολογιστικού και ενεργειακού κόστους σε σχέση με τις Δημόσιες Αλυσίδες Συστοιχιών. Ως εκ τούτου, οι Ιδιωτικές Αλυσίδες Συστοιχιών είναι πιο κλιμακώσιμες και έχουν καλύτερες επιδώσεις από τις Δημόσιες. [4]

Μια τρίτη κατηγορία αλυσίδων συστοιχιών είναι οι αλυσίδες με δικαιώματα (*Permissioned Blockchain*) [15][16][17]. Αρκετά παραπλήσιες με τις Ιδιωτικές Αλυσίδες Συστοιχιών, οι αλυσίδες (συστοιχιών) δικαιωμάτων ενσωματώνουν την έννοια του *access control*. Είναι, δηλαδή, ειδικά διαμορφωμένα συστήματα, στα οποία γίνονται συνεχείς έλεγχοι των δικαιωμάτων των μελών τους. Για παράδειγμα, συγκεκριμένοι χρήστες του δικτύου που ανήκουν στον οργανισμό Χ (που είναι εγγεγραμμένος στο δίκτυο) δύναται να έχουν διαφορετική εξουσιοδότηση (μόνο να γράφουν/διαβάζουν από το *ledger*, ή και τα δύο). Στις πλείστες περιπτώσεις οι Ιδιωτικές Αλυσίδες Συστοιχιών είναι και αλυσίδες δικαιωμάτων, αφού οι δύο έννοιες είναι αλληλένδετες. [4]

Στην τέταρτη, και τελευταία, κατηγορία *Blockchain* ανήκουν οι αλυσίδες (συστοιχιών) των οποίων η χρήση γίνεται χωρίς την χρήση άδειας (*Permissionless Blockchains*) [15][16][17]. Σε αυτές τις αλυσίδες δεν υπάρχουν περιορισμοί πρόσβασης και συμμετοχής. Στις πλείστες περιπτώσεις οι Δημόσιες Αλυσίδες Συστοιχιών είναι και αλυσίδες χωρίς χρήση άδειας, αφού οι δύο έννοιες είναι, και σε αυτή την περίπτωση, αλληλένδετες [4]. Παραδείγματα τέτοιων αλυσίδων αποτελούν το *Bitcoin* [11][12] και το *Ethereum* [19].

2.5 Ερευνητικά Άρθρα

2.5.1 Decaying Telco Big Data with Data Postdiction

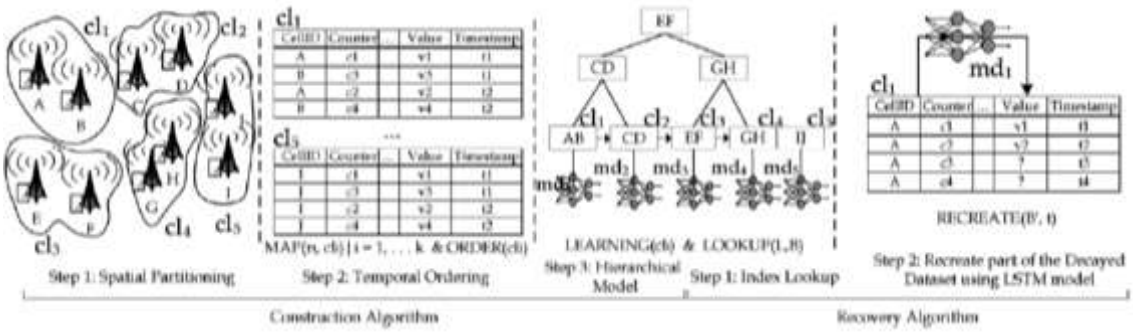
Χρησιμοποιώντας το παράδειγμα μιας εταιρείας τηλεπικοινωνιών, σε αυτή την υποενότητα παρουσιάζεται η έννοια της αποσύνθεσης δεδομένων (*data decay*) η οποία φιλοδοξεί να βελτιώσει τα διάφορα προβλήματα που αντιμετωπίζουν οργανισμοί που πραγματεύονται με μεγάλα δεδομένα (*Big Data*).

Αρχίζοντας, να αναφέρουμε πως ο όγκος των δεδομένων που παράγετε ημερησίως σε μια μεγάλη εταιρεία τηλεπικοινωνιών φτάνει μέχρι και τα 5TB [20]. Σε περιπτώσεις όπως αυτή, η αποθήκευση δεδομένων σαφώς και δεν αποτελεί εύκολο εγχείρημα. Ειδικότερα, δεν δύναται η χρήση παραδοσιακών υποδομών συννέφου (*cloud*), αφού, πρώτον, η αποθήκευση και η πρόσβαση στα δεδομένα έχει μεγάλο υπολογιστικό κόστος, και, δεύτερον, υπάρχουν διάφορα θέματα ασφάλειας και ιδιωτικότητας. Τέλος, ένα άλλο θέμα που τίθεται, είναι η δύσκολη ή/και αδύνατη παροχή αντιγράφων ασφαλείας των δεδομένων τέτοιων εταιριών, λόγω του τεραστίου όγκου τους.

Παρακινημένοι από τις προαναφερθέντες προκλήσεις, οι συγγραφείς της επιστημονικής δημοσίευσης «*Decaying Telco Big Data with Data Postdiction*» [20] προτείνουν έναν πρωτοποριακό τρόπο αποσύνθεσης δεδομένων για την αύξηση της επίδοσης τέτοιων συστημάτων, αφού όμως πρώτα εξετάσουν διάφορες υφιστάμενες προσεγγίσεις. Συγκεκριμένα, εξετάστηκαν οι προσεγγίσεις της συμπίεσης δεδομένων (*COMPRESSION*), αποθήκευσης κάθε δεύτερου δείγματος μόνο (*SAMPLING*), και λήψης δειγμάτων με ομοιόμορφη κατανομή (*RANDOM*), σε σύγκριση με την πρόταση των συγγραφέων, που φέρει το όνομα *TBD-DP* (*Telco Big Data – Data Postdiction*) [20].

Η γενική ιδέα του *TBD-DP* περιλαμβάνει την χρήση μοντέλων μηχανικής μάθησης και πάλι για λόγους συρρίκνωσης του όγκου δεδομένων που παράγεται σε τέτοια συστήματα. Αναλυτικότερα, προτείνεται η εκπαίδευση μοντέλων μηχανικής μάθησης πάνω στα παλαιότερα δεδομένα του συστήματος (που σπανίως/δεν χρησιμοποιούνται πλέον), και η αποθήκευση των μοντέλων αυτών, αντί των αυτούσιων δεδομένων. Όσο αφορά την έννοια του «*Postdiction*», ο όρος αυτός εμφανίζεται κατά την «ανάγνωση» δεδομένων από τα μοντέλα μηχανικής μάθησης, όπου, ουσιαστικά, γίνονται προβλέψεις

(στο παρελθόν…). Ακόμα, να αναφέρουμε ότι, λόγω του είδους των δεδομένων (χρονοσειρές [21]), χρησιμοποιήθηκαν μοντέλα νευρωνικών δικτύων τύπου *LSTM* (*Long Short-Term Memory*) [22]. Ιδιαίτερο χαρακτηριστικό των μοντέλων αυτών αποτελεί η παράμετρος «*decay factor*» που ρυθμίζει το ποσοστό πληροφορίας που συγκρατείται από το μοντέλο κατά την εκπαίδευσή του.



Σχήμα 2.4 : Τα εννοιολογικά βήματα των αλγορίθμων κατασκευής μοντέλων και ανάκτησης δεδομένων. [20]

Για σκοπούς περεταίρω κατανόησης του τρόπο εφαρμογής του *TBD-DP*, ακολουθεί σύντομη αναφορά στους αλγορίθμους κατασκευής των *LSTM* μοντέλων και του τρόπου ανάκτησης δεδομένων από αυτά.

Η κατασκευή των μοντέλων χωρίζεται σε τέσσερα βήματα [20]:

1. *Decaying Pre-processing*: Καθορισμός του *decay factor* (f).
2. *Spatial Partitioning*: Διαχωρισμός και ομαδοποίηση (*clustering*) των τηλεπικοινωνιακών κόμβων βάση της τοποθεσίας τους.
3. *Temporal Ordering*: Ταξινόμηση των δεδομένων βάση της χρονοσφραγίδας τους.
4. *Hierarchical Model*: Εκπαίδευση των μοντέλων (ένα μοντέλο ανά *cluster*), και ιεραρχική τοποθέτηση τους σε ένα *DP-Tree* για ευκολότερη και γρηγορότερη ανάκτηση.

Η ανάκτηση δεδομένων από τα εκπαιδευμένα μοντέλα αποτελεί μια απλή διαδικασία δύο βημάτων. Πρώτον, βρίσκουμε το μοντέλο που αποθηκεύει τα δεδομένα που θέλουμε ανατρέχοντας το *DP-Tree* που δημιουργήθηκε κατά την κατασκευή των

μοντέλων. Δεύτερον, χρησιμοποιούμε το μοντέλο για την ανάκτηση των σχετικών πληροφοριών.

2.5.2 CAPER: A Cross-Application Permissioned Blockchain

Το εργαλείο *CAPER* [23] αναπτύχθηκε με σκοπό την υποστήριξη πολλαπλών εφαρμογών (*Applications*) σε *Blockchain* περιβάλλοντα, με τον όρο *Application* να αναφέρεται στην επιχειρηματική λογική που ακολουθούν οι συμμετέχοντες του δικτύου *Blockchain*.

Για περεταίρω κατανόηση της έννοιας του *Application*, μπορούμε να αναφερθούμε στο σενάριο ενός εργοστασίου που είναι μέλος μιας αλυσίδας ανεφοδιασμού (*Supply Chain*) στην οποία χρησιμοποιούνται τεχνολογίες *Blockchain* για την καταγραφή των δοσοληψιών (*transactions*) που γίνονται καθημερινά [23]. Για την ομαλή λειτουργία του εργοστασίου αυτού, χρειάζεται να τρέχουν παράλληλα διάφορες εφαρμογές που αντιστοιχούν στις διεργασίες που λαμβάνουν χώρα (π.χ., εφαρμογές προμηθευτών, πελατών, αποθέματος). Επιπρόσθετα, οι εφαρμογές αυτές μπορεί να τροποποιούν μόνο δικά τους δεδομένα (*local level transactions*) ή να έχουν την δυνατότητα να τροποποιούν και δεδομένα άλλων εφαρμογών του δικτύου (*global level transactions*).

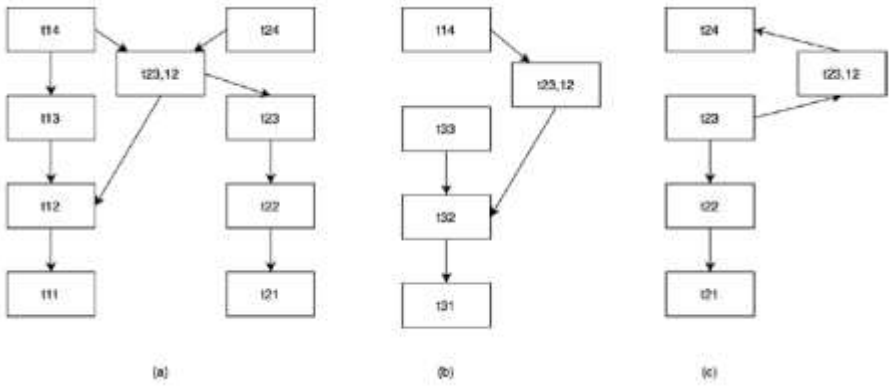
Το πιο πάνω παράδειγμα αποτελεί ένα πιθανό σενάριο χρήσης του *CAPER* το οποίο επιχειρεί να αντιμετωπίσει τις προκλήσεις που εγείρει η ύπαρξη των *local* και *global level transactions*. Συγκεκριμένα, τα δεδομένα της κάθε εφαρμογής στη γενική περίπτωση πρέπει να διατηρούνται ξεχωριστά (*local level*), ωστόσο, στις περιπτώσεις που δύο ή περισσότερες εφαρμογές επικοινωνούν (τροποποιώντας τα δεδομένα η μια της άλλης) πρέπει η ορατότητα των δεδομένων της κάθε μιας να προσαρμόζεται κατάλληλα (*global level*).

Δεδομένης φύση του προβλήματος που αντιμετωπίζει το *CAPER*, το ίδιο το εργαλείο εντάσσεται στην κατηγορία των *Permissioned Blockchains*.

Εμβαθύνοντας στον τρόπο λειτουργίας του *CAPER*, να αναφερθεί ότι η υλοποίηση των διαφόρων εφαρμογών στο εργαλείο γίνεται με τη χρήση ξεχωριστού *Blockchain* για την κάθε μια εξ' αυτές, ενώ σε κάθε block δοσοληψιών/συναλλαγών (*transactions*) υπάρχει μόνο μια δοσοληψία για σκοπούς επίδοσης. Επιπλέον, όλες οι δοσοληψίες σε όλα τα *Blockchains* του συστήματος πρέπει να είναι σε σειρά, με βάση τη χρονοσφραγίδα τους,

και οι δοσοληψίες μεταξύ εφαρμογών (*Cross-Application Transactions*) [23] οφείλουν να εμφανίζονται σε όλα τα εμπλεκόμενα *Blockchains* και να συνδέονται μεταξύ τους με κρυπτογραφικούς συνδέσμους *hash*. [4]

Για την υλοποίηση των συνδέσμων μεταξύ παράλληλων *Blockchains* οι δημιουργοί του *CAPER* χρησιμοποίησαν κατευθυνόμενους μη-κυκλικούς γράφους (*Directed Acyclic Graphs - DAGs*) με τον τρόπο που αναπαρίσταται στο πιο κάτω σχήμα.



Σχήμα 2.5 : Παράλληλα *Blockchain* με τη χρήση *DAG*. [4]

Στο Σχήμα 2.5 βλέπουμε το *cross-application transaction* t23,12 να εμφανίζεται στα *Blockchains* όλων των εμπλεκομένων εφαρμογών.

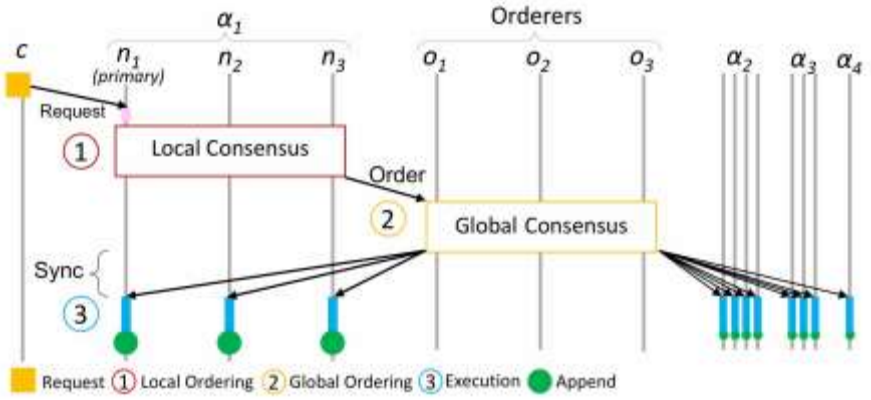
Συνεχίζοντας, στο σχετικό άρθρο πρότασης του *CAPER* εξετάζονται τρεις αλγόριθμοι ομοφωνίας (*consensus*), για σκοπούς αύξηση της απόδοσης του συστήματος- ο *Global Consensus Using Orders,* ο *Hierarchical Global Consensus*, και ο *One Level Global Consensus*.

Πιο κάτω περιγράφεται ο αλγόριθμος *Global Consensus Using Orders*, ο οποίος μέσω πειραματικής αξιολόγησης αποδείχθηκε και ως ο πιο αποτελεσματικός:

1. Αρχικά ο *Client* αποστείλει αίτημα (*request*) που αφορά κάποια από τις διαθέσιμες εφαρμογές στον επονομαζόμενο *Primary Peer* της συγκεκριμένης εφαρμογής (κάθε εφαρμογή έχει τον δικό της *Primary Peer*). Τότε, ένας τοπικός αλγόριθμος ομοφωνίας (*local consensus*) χρησιμοποιείτε για να αποφασιστεί το αποτέλεσμα του αιτήματος από τα *Peers* στα οποία τρέχει η εφαρμογή.

2. Στη συνέχεια, ένα νέο αίτημα (με το αποτέλεσμα) δημιουργείται και προωθείται στους *Orderers* του συστήματος, οι οποίοι αναλαμβάνουν τη δουλειά του *global consensus*. Αποτέλεσμα της διαδικασίας του *global consensus* είναι η δημιουργία ενός *block* δοσοληψιών.

3. Στην τρίτη φάση, το *block* δοσοληψιών διανέμεται σε όλα τα *Peers* του δικτύου, όπου και εκτελούνται πάνω στους σχετικούς πόρους (ο κάθε *Peer* το εκτελεί στο δικό του αντίγραφο του *ledger*). Τέλος, το *block* προστίθεται στο *ledger* (του κάθε *Peer*).



Σχήμα 2.6 : *Global Consensus using a Set of Orderers*. [23]

2.5.3 BlockchainDB: A Shared Database on Blockchains

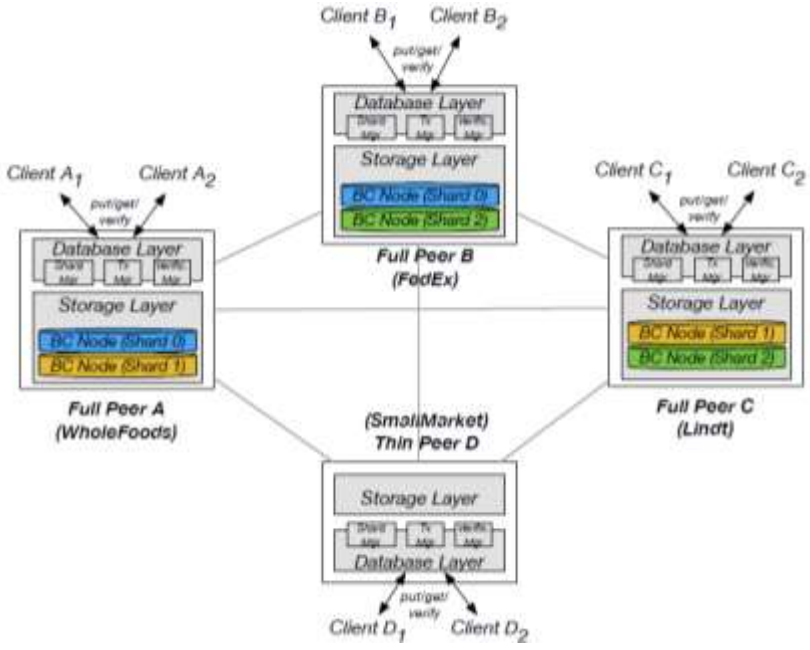
Η *BlockchainDB* [24] αποτελεί μια βάση δεδομένων η οποία αξιοποιεί αλυσίδες συστοιχιών για την αποθήκευση των δεδομένων της. Ειδικότερα, η αρχιτεκτονική της *BlockchainDB* αποτελείτε από το επίπεδο πελάτη (*Client Layer*), το επίπεδο βάσης δεδομένων (*Database Layer*) και το επίπεδο αποθήκευσης (*Storage Layer*), των οποίων τα σημαντικότερα χαρακτηριστικά συνοψίζονται ως ακολούθως:

- *Storage Layer*: Αποτελείτε από μία, ή περισσότερες, αλυσίδες συστοιχιών (*Blockchains*), και χρησιμοποιείται για την αποθήκευση των δεδομένων του συστήματος. Συγκεκριμένα, κάθε *Blockchain* στο επίπεδο αυτό αντιπροσωπεύει ένα *shard* δεδομένων του συστήματος. Αυτό αποσκοπεί στην εφαρμογή των ευρέως γνωστών τεχνικών *sharding* και *replication* στο σύστημα,

εκμεταλλευόμενοι του αποκεντρωμένου χαρακτήρα και της αντιγραφής (*replication*) δεδομένων που υπάρχουν στην τεχνολογία *Blockchain*.

- *Database Layer*: Αποτελεί το επίπεδο υλοποίησης διαφόρων παραδοσιακών χαρακτηριστικών μιας βάσης δεδομένων, και συνιστάται από δύο συστατικά-τον *Shard Manager* και τον *Transaction Manager*. Ο *Shard Manager* αποφασίζει ποιά *shards* πρέπει να χρησιμοποιηθούν κατά την εξυπηρέτηση επερωτημάτων, όπως και τον τρόπο διαχείρισης των *shards* του συστήματος. Το συστατικό του *Transaction Manager* υλοποιεί διάφορες προσεγγίσεις επίτευξης συνέπειας δεδομένων (*eventual* και *sequential consistency*) του συστήματος, ο οποίες μπορούν να εφαρμοστούν κατόπιν επιλογής κάποιου διαχειριστή.
- *Client Layer*: Παρέχει μια φιλική, προς το χρήστη, διεπιφάνεια αλληλεπίδρασης με το *Database Layer*, με σκοπό τη απόκρυψη της πολυπλοκότητας του από τους πελάτες του συστήματος.

Συνεχίζοντας, οι συγγραφείς της σχετικής επιστημονικής δημοσίευσης [24] περιγράφουν τις τρείς βασικές λειτουργίες που υποστηρίζει το σύστημα *BlockchainDB*. Οι λειτουργίες αυτές, όπως φαίνεται και στη συνέχεια, αντιστοιχούν στις επιλογές διαχείρισης δεδομένων που παρέχονται στον χρήστη στο *Client Layer* και χρησιμοποιούν τη λογική την αφηρημένη (*abstract*) έννοια των *Shared Tables* που αντιστοιχούν στους απλούς πίνακες των συμβατικών βάσεων δεδομένων.



Σχήμα 2.7 : Ένα τυπικό *BlockchainDB* δίκτυο.

Οι λειτουργίες του συστήματος *BlockchainDB*:

- *get(t, k)* → *v*: Δοθέντος του ονόματος ενός *Shared Table* (t) και ενός κλειδιού/ταυτοποιητή (k), γίνεται ανάκτηση του πόρου που ανήκει στο t και ταυτοποιείται από το k.
- *put(t, k, v)* → *void*: Δοθέντος του ονόματος του *Shared Table* που επιθυμούμε να τοποθετηθούν τα δεδομένα του v με τον ταυτοποιητή k, γίνεται η αποθήκευση τους στον t. Η κλήση της μεθόδου αυτής είναι ασύγχρονη.
- *verify()* → *bool*: Επιστρέφει True μόνο εάν η αμέσως προηγούμενη μέθοδος που χρησιμοποιήθηκε (*put* ή *get*) ολοκληρώθηκε με επιτυχία. Ειδικότερα, η λειτουργία *put* θεωρείτε επιτυχής εάν όντως τα δεδομένα έχουν αποθηκευτεί στο *Shared Table* για το οποία προορίζονταν, ενώ η λειτουργία *get* είναι επιτυχής εάν τα δεδομένα που ανακτήθηκαν δεν έχουν παραποιηθεί από κάποιο κόμβο του δικτύου (γίνεται επαλήθευση ακεραιότητας).

2.5.4 AnyLog: A Grand Unification of the Internet of Things

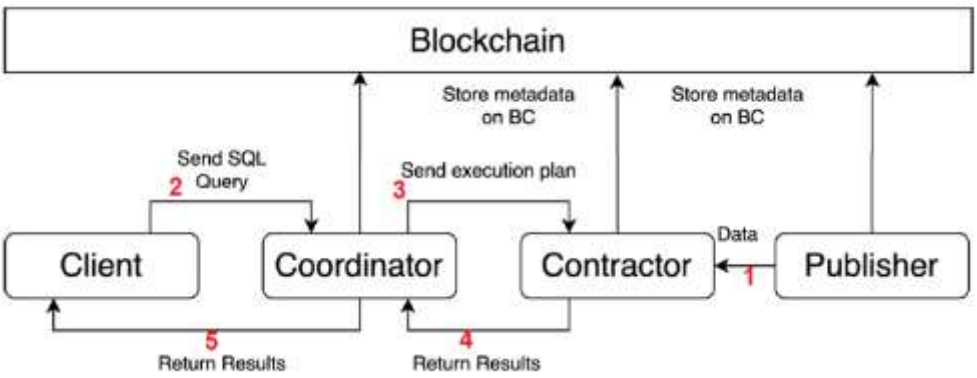
Με την χρήση των *IoT* συσκευών να εξαπλώνεται με ταχείους ρυθμούς στις μέρες μας, ο όγκος των δεδομένων που παράγονται καθημερινά όλο και πληθαίνει. Τα δεδομένα αυτά πρέπει να αποθηκευτούν, να διανεμηθούν και να επεξεργαστούν καταλλήλως, κάτι το οποίο είναι δύσκολο (εάν όχι αδύνατο), αφού η υφιστάμενη προσέγγιση χρήσης παραδοσιακών *SQL* βάσεων δεδομένων δεν επιφέρει τα επιθυμητά αποτελέσματα ή/και επιδώσεις. Για την αντιμετώπιση του προβλήματος αυτού έχει προταθεί το εργαλείο *AnyLog* [25].

Το *AnyLog* είναι ένα *Public*/*Permissionless Blockchain* το οποίο σχεδιάστηκε συγκεκριμένα για τη διαχείριση δεδομένων του IoT τομέα. Εν ολίγοις, το *AnyLog* αποτελεί ένα ανοικτό/δημόσιο δίκτυο οργανωμένης αποθήκευσης δεδομένων έξυπνων συσκευών, οι οποίες συνεισφέρουν στο δίκτυο έναντι κάποιας ανταμοιβής (για λόγους παρακίνησης προς συμμετοχή και συνεισφορά τους).

Η αρχιτεκτονική του *AnyLog* βασίζεται σε διάφορα είδη κόμβων και δομών δεδομένων. Ακολουθεί σύντομη περιγραφή των πιο σημαντικών οντοτήτων του εργαλείου αυτού [4]:

- *Clients*: Κόμβοι που παράγουν και καταναλώνουν *SQL* επερωτήματα και τις απαντήσεις αυτών, αντίστοιχα
- *Coordinators*: Κόμβοι που παραλαμβάνουν, βελτιστοποιούν και ετοιμάζουν ένα πλάνο εκτέλεσης (*execution plan*) για επερωτήματα, το οποίο προωθούν στους *Contractors* για εκτέλεση.
- *Contractors*: Κόμβοι που αποθηκεύουν δεδομένα που τους αποστέλλονται από τους *Publishers* και εκτελούν τα πλάνα εκτέλεσης επερωτημάτων που λαμβάνουν από τους *Coordinators*.
- *Publishers*: Κόμβοι που παράγουν *IoT* δεδομένα και τα προωθούν στους *Contractors* για αποθήκευση
- *Blockchain*: Δομή δεδομένων που φυλάει μεταδομένα των δοσοληψιών (transactions), ως επίσης τις ρυθμίσεις του δικτύου

Η ροή δεδομένων μεταξύ των οντοτήτων του συστήματος απεικονίζεται στο ακόλουθο διάγραμμα.

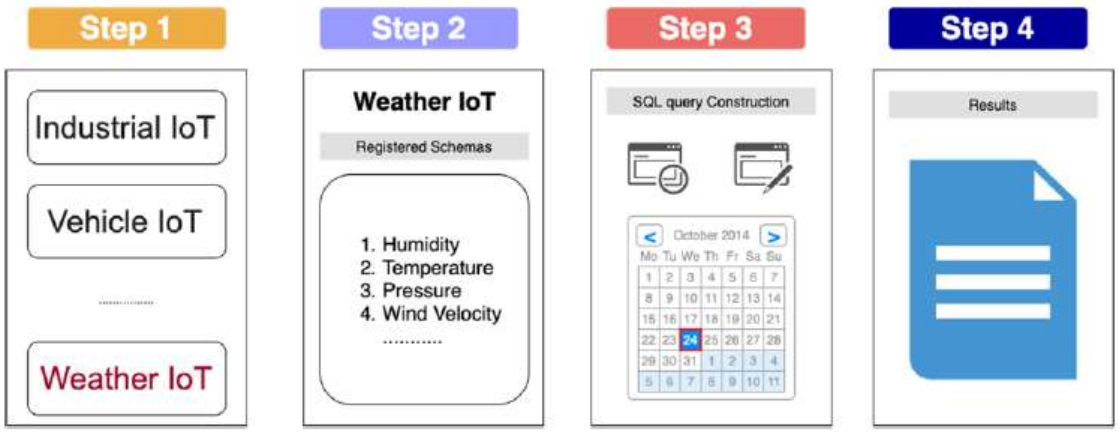


Σχήμα 2.8 : AnyLog Execution Flow. [4]

1. Αρχίζοντας, οι *Publishers* αποστέλλουν τα δεδομένα που παράγουν οι *IoT* συσκευές στους *Contractors*, οι οποίοι αποθηκεύουν τα δεδομένα αυτά.
2. Όταν ένας *Client* το θελήσει, υποβάλλει κάποιο επερώτημα σε έναν από τους *Coordinators*.
3. Ο *Coordinator*, παράγει το πλάνο εκτέλεσης για το συγκεκριμένο επερώτημα.
4. Το πλάνο εκτέλεσης προωθείτε σε, και εκτελείται από, όλους τους σχετικούς *Contractors* για να υπολογιστεί η απάντηση του επερωτήματος. Τα αποτελέσματα που παράγονται επιστρέφονται στους *Coordinators*.
5. Οι *Coordinators* ομαδοποιούν τα δεδομένα που έχουν παραλάβει και αποστέλλουν το παραγόμενο αποτέλεσμα στους *Clients*.
6. Τέλος, οι *Clients* καταθέτουν κάποιο ποσό στους *Coordinators* για τις υπηρεσίες τους. Οι *Coordinators* στη συνέχεια αναλαμβάνουν να ανταμείψουν τους *Contractors*, και αυτοί (*Contractors*) με την σειρά τους *Publishers*.

Καθόλη τη διάρκεια της πιο πάνω διαδικασίας, γίνεται αποθήκευση μεταδεδομένων στο *Blockchain*.

Επιπρόσθετα, πολύ σημαντικό συστατικό του *AnyLog* αποτελεί η ειδικά διαμορφωμένη διεπιφάνεια υποβολής επερωτήσεων την οποία προσφέρει στους χρήστες (*Clients*) του. Συγκεκριμένα, η διεπιφάνεια χρησιμοποιείτε στο δεύτερο βήμα της διαδικασίας που περιγράφηκε μόλις πιο πάνω. Στο Σχήμα 2.9 φαίνεται πιο αναλυτικά ο τρόπος υποβολής επερωτημάτων μέσω της διεπιφάνειας του εργαλείου *AnyLog*.



Σχήμα 2.9 : Διαδικασία κατασκευής query. [4]

Σύμφωνα με το Σχήμα 2.9, η δημιουργία και εκτέλεση ενός επερωτήματος χωρίζεται σε τέσσερα βήματα (από την οπτική του χρήστη). Αρχικά ο χρήστης (*Client*) καλείτε να επιλέξει τον *IoT* τομέα με τον οποίο σχετίζεται το επερώτημα που θέλει να δημιουργήσει, και, στη συνέχεια, να επιλέξει τη δομή δεδομένων που τον εκφράζει. Στο βήμα τρία, το σύστημα υποβοηθά το χρήστη στο σχηματισμό του επερωτήματος του στη γλώσσα SQL, το οποίο υποβάλλεται για εκτέλεση. Στο τέταρτο βήμα, τα αποτελέσματα του επερωτήματος επιστρέφονται στο χρήστη για κατανάλωση. [4]

Κεφάλαιο 3

Αρχιτεκτονική Triabase

3.1 Επίπεδα Συστήματος Triabase 27

3.1.1 Επίπεδο Ακρών 27

3.1.2 Επίπεδο Εφαρμογής 29

3.1.3 Επίπεδο Αποθήκευσης 31

3.2 Ροή Εκτέλεσης / Δεδομένων 33

Επιθυμώντας την περεταίρω κατανόηση της αρχιτεκτονικής του συστήματος *Triabase* από τον αναγνώστη, στο παρόν κεφάλαιο αναλύονται οι βασικές έννοιες, ο σκοπός ύπαρξης και ο τρόπος λειτουργίας των τριών επιπέδων που το αποτελούν, όπως και οι μηχανισμοί που καθιστούν την μεταξύ τους αλληλεπίδραση εφικτή.

3.1 Επίπεδα Συστήματος Triabase

Σε αυτή την υπο-ενότητα περιγράφονται τα τρία επίπεδα του συστήματος *Triabase* (Ακρών/*Edge*, Εφαρμογής/*Application*, Αποθήκευσης/*Storage*).

3.1.1 Επίπεδο Ακρών

Αρχίζοντας, να αναφέρουμε πως το επίπεδο των ακρών (*Edge Layer*) αποτελεί το σημείο επιστράτευσης τεχνολογιών μηχανικής μάθησης (συγκεκριμένα *Federated Learning*) για την επίτευξη των στόχων του συστήματος (*Triabase*). Σημαντικό

στοιχείο του επιπέδου αυτού, αποτελεί και η μοναδική αξιοποίηση της τεχνολογίας αλυσίδων συστοιχιών (*Blockchain*) κατά την διαδικασία της προαναφερθείσας ομοσπονδιακής μάθησης (*Federated Learning*). Σκοπός της, η εξασφάλιση της ποιότητας (ακρίβειας - *accuracy*) του καθολικού μοντέλου μηχανικής μάθησης του *IoT* συστήματος που χρησιμοποιεί το *Triabase*.

Ειδικότερα, η παρουσία της τεχνολογίας *Blockchain* κατά την εκτέλεση των βημάτων του αλγορίθμου ομοσπονδιακής μάθησης γίνεται μέσω της χρήσης του αλγόριθμου ομοφωνίας *Proof-of-Federated-Learning* (*PoFL*), όπως αυτός προτάθηκε στις σχετικές δημοσιεύσεις [3][26].

Κύριες ιδέες του *PoFL* αποτελούν:

- Η χρήση ενός *Blockchain* για την αποθήκευση των ενδιάμεσων αποτελεσμάτων της διαδικασίας ομοσπονδιακής μάθησης στο σύστημα.

- Η επιβράβευση των κόμβων που συνεισφέρουν σημαντικά στην αύξηση ακρίβειας (*accuracy*) του καθολικού μοντέλου.

- Η εξασφάλιση της αξιοπιστίας του *Model Aggregator* μέσω της επιλογής του κόμβου με την πολυτιμότερη προσφορά (στο προηγούμενο γύρο εκπαίδευσης/*training round*) για να εκτελέσει την διαδικασίας συσσωμάτωσης των τοπικών μοντέλων.

Ακολούθως, παρατίθενται τα βήματα της διαδικασίας ομοσπονδιακής μάθησης με χρήση του αλγορίθμου *PoFL*:

1. Οι έξυπνες συσκευές (κόμβοι του δικτύου) ανακτούν την τελευταία έκδοση του καθολικού μοντέλου του συστήματος από το *Blockchain*.

2. Οι έξυπνες συσκευές εκπαιδεύουν το καθολικό μοντέλο στα (τοπικά) τους δεδομένα, μετατρέποντας το έτσι σε τοπικό μοντέλο (για την κάθε συσκευή).

3. Ο *Model Aggregator*, όπως αυτός προκύπτει από τη συνεισφορά του στον προηγούμενο γύρο εκπαίδευσης, φροντίζει για την εκκίνηση (και ολοκλήρωση) της διαδικασίας αποθήκευσης των τοπικών μοντέλων στο *Blockchain*.

4. Ο *Model Aggregator* χρησιμοποιεί τα «έγκυρα» τοπικά μοντέλα για τη δημιουργία της νέας έκδοσης του καθολικού μοντέλου του συστήματος, και ακολούθως αποθηκεύει το μοντέλο στο *Blockchain*.

5. Οι έξυπνες συσκευές των οποίων τα τοπικά μοντέλα συνεισέφεραν στην βελτίωση της ακρίβειας του καθολικού μοντέλου επιβραβεύονται με νομίσματα εκπαίδευσης (*training coins*).

Η πιο πάνω διαδικασία αποτελεί ένα «γύρο εκτέλεσης» (*training round*) και επαναλαμβάνεται κάθε φορά που παράγονται αρκετά (τοπικά) δεδομένα για να είναι εφικτή η τοπική εκπαίδευση (π.χ., σε συγκεκριμένα τακτά χρονικά διαστήματα).

Συμπληρωματικά, για περεταίρω κατανόηση του πιο πάνω αλγορίθμου δίδονται τα πιο κάτω σημεία:

- Ως ο *Model Aggregator* του πρώτου γύρου εκπαίδευσης επιλέγεται ο κόμβος/συσκευή του οποίου το τοπικό μοντέλο (πρώτου γύρου) έχει τη μεγαλύτερη ακρίβεια.

- Ως «έγκυρα» θεωρούνται τα μοντέλα που αποθηκεύτηκαν με επιτυχία στο *Blockchain*, στο βήμα 3 του αλγορίθμου.

- Στο βήμα 5, επιβραβεύονται μόνο οι συσκευές των οποίων τα τοπικά μοντέλα αποδείχτηκαν «έγκυρα».

3.1.2 Επίπεδο Εφαρμογής

Συνεχίζοντας, το επίπεδο εφαρμογής (*Application Layer*) φέρει τα καθήκοντα του ενδιάμεσου μεταξύ του επίπεδου ακρών (*Edge Layer*) και του επίπεδου αποθήκευσης (*Storage Layer*).

Η χρήση ενός ενδιάμεσου επιπέδου, εμπνευσμένη από την Αρχιτεκτονική Κλεψύδρας (*Hourglass Architecture*) του Διαδικτύου [27], αποσκοπεί κυρίως στην μεγιστοποίηση της διαλειτουργικότητας (*interoperability*) των επιπέδων *Edge* και *Storage*, υιοθετώντας την υπόθεση της ελάχιστης κοινής/συμβατής λειτουργίας (*least common*

functionality) των επιπέδων αυτών. Ως εκ τούτου, έχουμε την μεγιστοποίηση του αριθμού πλατφόρμων/τεχνολογιών *Federated Learning* και *Blockchain* που μπορούν να χρησιμοποιηθούν κατά την υλοποίηση των επίπεδων που προηγούνται και επέρχονται του *Application Layer* (*Edge* και *Storage*, αντίστοιχα). Μοναδική προϋπόθεση είναι η συμβατότητα των πλατφόρμων/τεχνολογιών των δύο επιπέδων αυτών με το *Application Layer*.

Επιπρόσθετο πλεονέκτημα της χρήσης του ενδιάμεσου επιπέδου εφαρμογής αποτελεί και η απόκρυψη της πολυπλοκότητας της επικοινωνίας με το επίπεδο αποθήκευσης. Το γεγονός αυτό, προσδίδει την δυνατότητα παροχής της επιλογής χειροκίνητης (*manual*) διαχείρισης των δεδομένων του *Storage Layer* στους εξουσιοδοτημένους χρήστες του συστήματος. Εν ολίγης, το *Application Layer* δρα, επίσης, ως επίπεδο αφαίρεσης (*abstraction layer*) που επιτρέπει την κατανόηση και, μετέπειτα, χρήση των μεθόδων αλληλεπίδρασης του επίπεδου αποθήκευσης από το μέσο άνθρωπο (*human understandable and usable*).

Όπως περεταίρω αναλύεται και στο Κεφάλαιο 5 (Συστατικό Διεπαφής), στην παρούσα έκδοση του συστήματος *Triabase* στο επίπεδο εφαρμογής βρίσκεται ένας *REST* διακομιστής (*server*), που υλοποιεί δύο κύριες λειτουργίες (μεταξύ και άλλων βοηθητικών). Την πρώτη εξ' αυτών αποτελεί η παραλαβή (από το *Edge Layer*) των δεδομένων των μοντέλων προς αποθήκευση και η (μετατροπή και) προώθηση τους στη μορφή δοσοληψιών στο υποκείμενο *Blockchain* (στο *Storage Layer*). Η δεύτερη κύρια λειτουργία που υλοποιείται αφορά την ανάκτηση των δεδομένων ενός μοντέλου από το *Blockchain*. Συγκεκριμένα, κατόπιν της επιτυχούς εκτέλεσης των σχετικών επερωτημάτων (*queries*) για τη συλλογή τους, γίνεται επαναφορά των δεδομένων (του αιτούμενου μοντέλου) στην αρχική τους κατάσταση και αποστολή τους στον αιτούμενο.

Σε αυτό το σημείο, να διευκρινιστεί ότι ο διακομιστής προσφέρει ακριβώς την ίδια διεπαφή και στις έξυπνες συσκευές και στους εξουσιοδοτημένους χρήστες του συστήματος (δεν παρέχετε μηχανισμός διαχωρισμού/αναγνώρισης), εξού και η εισαγωγή του όρου του «αιτητή».

Τέλος, ενδιαφέρον σημείο προς εξέταση αποτελεί και η επιλογή αξιοποίησης της τεχνικής των *tokens* στο σύστημα. Αυτό γίνεται για λόγους διαφύλαξης της ασφάλειας

των δεδομένων του επιπέδου αποθήκευσης, μέσω της επιβεβαίωσης της εξουσιοδότησης χρήσης των υπηρεσιών του *Application Layer* από τον αιτητή. Ειδικότερα, εκτός από τα δεδομένα/στοιχεία ταυτοποίησης του μοντέλου για σκοπούς αποθήκευσης/ανάκτησης του, στα αιτήματα παροχής υπηρεσιών που δύναται προς εξυπηρέτηση πρέπει να συμπεριλαμβάνεται το *token* του αιτητή. Η εγκυρότητα του *token* αυτού ελέγχεται, και η εκτέλεση του αιτήματος λαμβάνει χώρα μόνο εάν επιβεβαιωθεί (η εγκυρότητα).

Ο τρόπος εξασφάλισης κάποιου *token* από τον αιτητή δεν έχει υλοποιηθεί στα πλαίσια της παρούσας διπλωματικής εργασίας. Στην τρέχον έκδοση του *Triabase*, υποθέτουμε ότι οι εξουσιοδοτημένοι αιτητές έχουν την δυνατότητα να επικοινωνήσουν άμεσα (*directly*) με το (*Permissioned*) *Blockchain* του *Storage Layer*, και να αποθηκεύσουν στο κατάστιχο (*ledger*) του συστήματος μια συμβολοσειρά την οποία εφεξής επιθυμούν να αποτελεί το (έγκυρο) *token* τους. Παρ' όλο που δεν αποτελεί βέλτιστη λύση, η ασφάλεια του συστήματος δεν διακυβεύεται από την προαναφερθείς προσέγγιση, αφού, ούτος ή άλλως, μόνο οι εξουσιοδοτημένοι χρήστες μπορούν να επικοινωνήσουν άμεσα με το *Blockchain* και να εξασφαλίσουν επιτυχώς το *token* που επιθυμούν.

3.1.3 Επίπεδο Αποθήκευσης

Το χαμηλότερο επίπεδο του *Triabase* αποτελεί το επίπεδο αποθήκευσης (*Storage Layer*), που φροντίζει για την ασφαλή και διαφανή αποθήκευση και τροποποίηση των δεδομένων του συστήματος. Κύριο χαρακτηριστικό του *Storage Layer* είναι η χρήση τεχνολογιών *Blockchain* που διασφαλίζει τα προαναφερθέντα.

Εμβαθύνοντας, στην παρούσα έκδοση του συστήματος η υλοποίηση του *Storage Layer* γίνεται με τη χρήση του *Hyperledger Fabric*, ένα *open source framework* για την δημιουργία επαγγελματικών *Private Permissioned Blockchain* εφαρμογών [28][29].

Ακολουθεί σύντομη περιγραφή του ρόλου των θεμελιωδών οντοτήτων [28] του *Hyperledger Fabric Blockchain* δικτύου που έχει αναπτυχθεί (στο επίπεδο αποθήκευσης) για το σύστημα *Triabase* στα πλαίσια της διπλωματικής εργασίας [4]:

- *Organizations* / Οργανισμοί: Οι οργανισμοί που συμμετέχουν στο δίκτυο. Κάθε οργανισμός έχει υπό την αιγίδα του ένα ή περισσότερα *Peers* τα οποία διαχειρίζεται, και συμμετέχει σε ένα ή περισσότερα *Channels*.

- *Channels* / Κανάλια: Κάθε *Channel* έχει το δικό του κατάστιχο (*ledger*) και *Smart Contracts*. Ακόμα, τα *Channels* είναι πλήρως ανεξάρτητα. Δηλαδή, τα *Smart Contracts* και το κατάστιχο (οι πόροι) ενός *Channel* δεν είναι ορατά/προσβάσιμα από άλλα *Channels*.

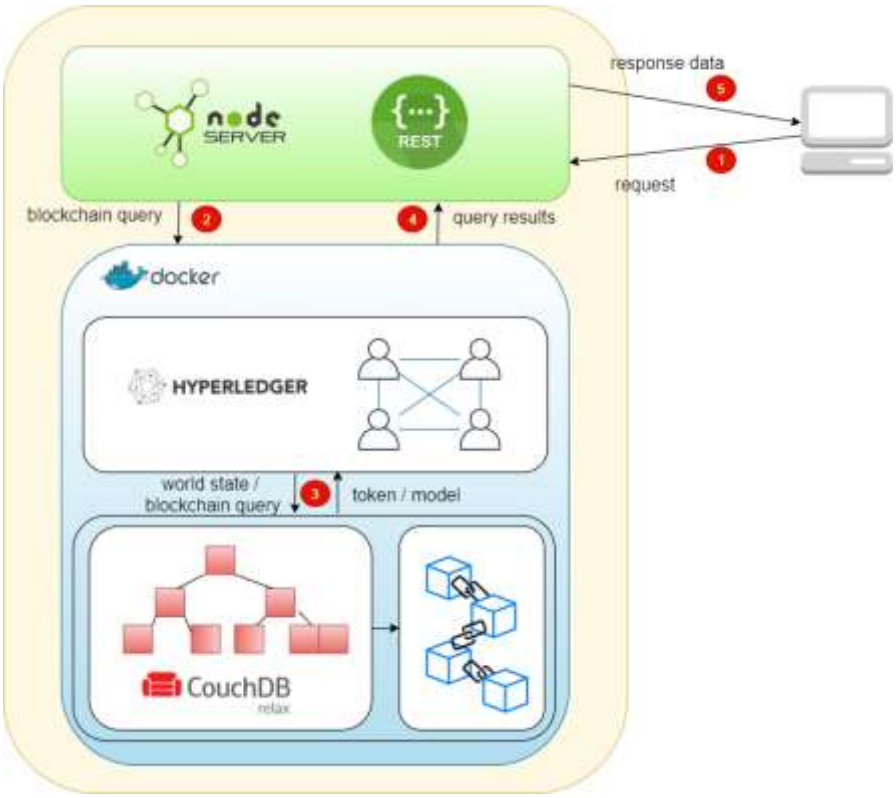
- *Smart Contracts* / Συμβόλαια: Προγράμματα τα οποία παρέχουν μεθόδους δημιουργίας, τροποποίησης ή/και διαγράφης πόρων στο κατάστιχο του *Channel* στο οποίο ανήκουν. Ως δοσοληψία (*transaction*) ονομάζεται η κλήση μιας εκ των μεθόδων αυτών.

- *Peers* / Ομότιμοι Κόμβοι: Το σύνολο των κόμβων του *Peer-to-Peer* δικτύου που αποτελεί το δίκτυο *Blockchain*. Ο κάθε *Peer* ανήκει σε ένα *Organization*, συμμετέχει σε ένα ή περισσότερα *Channels*, και έχει το δικό του τοπικό αντίγραφο των καταστίχων των *Channels* στα οποία συμμετέχει.

- *Endorsing Peers*: Υποσύνολο των *Peers* το οποίο αναλαμβάνει τον έλεγχο εξουσιοδότησης των χρηστών που εκδίδουν (*issue*) τις δοσοληψίες που δύναται να τροποποιήσουν μόνιμα τα δεδομένα του συστήματος.

- *Ordering Service* / Υπηρεσία Σειριοποίησης: Υποσύνολο των *Peers* που αναλαμβάνει την οργάνωση των δοσοληψιών σε *blocks* με βάση τον αλγόριθμο ομοφωνίας (*consensus*) που χρησιμοποιείται στο δίκτυο. Οι *Peers* αυτοί ονομάζονται και *Orderers*.

Συνεχίζοντας, για την αξιοποίηση του *Blockchain* δικτύου ως βάση δεδομένων του συστήματος *Triabase*, γίνεται χρήση των *Smart Contracts*. Συγκεκριμένα, το *Smart Contract* που υλοποιήθηκε στα πλαίσια της παρούσας δουλειάς παρέχει μηχανισμούς αποθήκευσης και ανάκτησης δεδομένων των μοντέλων που αποθηκεύονται στο κατάστιχο των *Peers* του δικτύου. Μια πιο λεπτομερής ανάλυση του κώδικα του *Smart Contract* αυτού δίνεται στο Κεφάλαιο 4 (Συστατικό Αποθήκευσης).



Σχήμα 3.1 : Ροή εκτέλεσης κατά την εξυπηρέτηση αιτημάτων.

3.2 Ροή Εκτέλεσης / Δεδομένων

Έχοντας δει μια υψηλού επιπέδου περιγραφή των διαφόρων στρωμάτων του συστήματος *Triabase*, σε αυτή την υπο-ενότητα αναλύεται η ροή εκτέλεσης/δεδομένων στο σύστημα κατά τις λειτουργίες αποθήκευσης και ανάκτησης δεδομένων.

Όπως φαίνεται και στο Σχήμα 3.1, η διαδικασία εξυπηρέτησης αιτημάτων είναι η ακόλουθη (η πιο κάτω αρίθμηση αντιστοιχεί στα νούμερα του σχήματος):

1. Ο αιτητής αποστέλλει το αίτημα του, μαζί με τα απαραίτητα δεδομένα για την διεκπεραίωση του, στο κατάλληλο *endpoint* του *REST server* (στο επιπέδου εφαρμογής).

2. Σύμφωνα με το είδος του αιτήματος, ο *server* καλεί τις αντίστοιχες μεθόδους του *Smart Contract* που βρίσκετε στο *Blockchain* (του επιπέδου αποθήκευσης), προωθώντας του τα δεδομένα που λήφθηκαν από τον αιτητή.

3. Κατά την εκτέλεση του κώδικα του *Smart Contract*, γίνεται ενημέρωση (*update*) του, ή ανάγνωση από το, *World State Database* ή/και των αυτούσιων αρχείων της αλυσίδας *block* δεδομένων (*blockchain*).

4. Με την επιτυχής εκτέλεση του κώδικα του *Smart Contract*, στον *server* επιστρέφονται τα δεδομένα που ζητήθηκαν (σε περιπτώσεις ανάκτησης) ή/και κάποιος κωδικός (επιτυχίας/αποτυχίας) που σηματοδοτεί την ολοκλήρωση της εκτέλεσης (σε περιπτώσεις ενημέρωσης).

5. Ο *server* προωθεί στον αιτητή τα αποτελέσματα του βήματος 4.

Κλείνοντας, να αναφέρουμε πως το *World State Database* αποθηκεύει την πιο πρόσφατη κατάσταση των πόρων του καταστίχου, ενώ στα αυτούσια αρχεία είναι καταγεγραμμένες όλες οι αλλαγές που έγιναν σε κάθε πόρο με το πέρασμα του χρόνου. Η χρήση του *World State Database* γίνεται για λόγους επίδοσης (και είναι υποχρεωτική στο *Hyperledger Fabric framework*), ενώ τα αυτούσια αρχεία χρησιμοποιούνται μόνο όταν χρειαστεί πρόσβαση σε παλαιότερες «εκδόσεις» κάποιου πόρου (ιστορικά δεδομένα). Η παροχή υπηρεσιών που αφορούν ιστορικά δεδομένα δεν αποτελεί μέρος της δουλειάς που έγινε στα πλαίσια της διπλωματικής εργασίας υπό αναφορά, ωστόσο, εντάσσετε στις πιθανές μελλοντικές επεκτάσεις που παραθέτονται στο υπο-κεφάλαιο 7.2.

Κεφάλαιο 4

Συστατικό Αποθήκευσης

4.1 Απαιτήσεις Επιπέδου Αποθήκευσης 36

4.2 Περιορισμοί Αλυσίδας Συστοιχιών 37

4.3 Σχεδιασμός Συστατικού Αποθήκευσης 38

4.3.1 Σελιδοποίηση Δεδομένων 38

4.3.2 Δομές Δεδομένων 39

4.3.3 World State Database 41

4.4 Υλοποίηση Smart Contract 41

Αυτό το κεφάλαιο επικεντρώνεται στην ανάλυση των λειτουργικών απαιτήσεων που λήφθηκαν υπόψη κατά την υλοποίηση του *Smart Contract* του *Blockchain* που χρησιμοποιείται στο επίπεδο αποθήκευσης. Επιπλέον, γίνεται ανάλυση των αποφάσεων που λήφθηκαν κατά τη φάση σχεδιασμού για την ικανοποίηση των απαιτήσεων αυτών, ενώ δίνονται περεταίρω λεπτομέρειες για βασικά στοιχεία της υλοποίησης.

Πριν προχωρήσουμε, για σκοπούς πληρότητας, να αναφέρουμε ότι οι λόγοι (απαιτήσεις) επιλογής της χρήσης ενός *Permissioned Blockchain* στο επίπεδο αποθήκευσης έχουν ήδη (άμεσα και έμμεσα) αναφερθεί σε προηγούμενα κεφάλαια (1 και 3). Περιληπτικά, ωστόσο, να σημειώσουμε ξανά πως οι ανάγκες διασφάλισης της ασφάλειας και διαφάνειας των δεδομένων (που εγγυείται το κατανεμημένο και αυστηρώς ελεγχόμενο περιβάλλον των *Permissioned Blockchains*) αποτελούν την αιτία επιλογής της τεχνολογίας αυτής. Όπως θα φανεί και στη συνέχεια, όμως, η χρήση της επιφέρει αρκετούς περιορισμούς στον σχεδιασμό και υλοποίηση του συστατικού αποθήκευσης του συστήματος.

4.1 Απαιτήσεις Επιπέδου Αποθήκευσης

Για την εξυπηρέτηση των αναγκών του συστήματος *Triabase*, στην ουσία, χρειάζεται η χρήση μιας βάσης δεδομένων μοντέλων μηχανικής μάθησης (*machine learning model database*). Μια τέτοια βάση δεδομένων πρέπει να μπορεί να διαχειρίζεται (με τον ίδιο τρόπο) όλων των ειδών μοντέλα, ανεξαρτήτως της αρχιτεκτονικής, των βαρών και των όποιων άλλων ιδιαιτεροτήτων δύναται να έχουν, κάτι που αποτελεί ιδιαίτερη πρόκληση.

Στην περίπτωση του *Triabase*, τα μοντέλα μηχανικής μάθησης προωθούνται (από το *Edge Layer*) στα *Application* και *Storage Layers* αφού έχουν πρώτα σειριοποιηθεί. Η διαδικασία σειριοποίησης (*export*) στα διάφορα *machine learning frameworks* σήμερα έχει ως αποτέλεσμα την παραγωγή ενός, ή περισσοτέρων, αρχείων με σχετικά δεδομένα. Τα αρχεία αυτά περιέχουν δεδομένα που περιγράφουν την αρχιτεκτονική, τα βάρη, σημαντικά *checkpoints* εκπαίδευσης (*training checkpoints*) ή/και δεδομένα αρχικοποίησης του μοντέλου. Επομένως, αυτά τα αρχεία καλείται να μπορεί να διαχειρίζεται το συστατικό αποθήκευσης που περιγράφεται στην παρούσα ενότητα.

Οι πιο δημοφιλείς τύποι αρχείων σειριοποίησης συνοψίζονται στον πιο κάτω πίνακα, βάση του *framework*/*module* μηχανικής μάθησης εν χρήση:

Machine Learning Framework/Module	**File Formats Involved**
Darknet	.cfg, .weights
PyTorch	.pt, .pth
Tensorflow Standard (v1,v2)	.h5, .pb, .txt, .index, .data-XXXXX-of-XXXXX
Tensorflow JS (TFjs)	.bin, .json
Tensorflow Lite (TFLite)	.tflite
Python Serialization Libraries (e.g pickle, joblib, dill)	any

Πίνακας 4.1 : Τύποι αρχείων σειριοποίησης μοντέλων.

Συνεχίζοντας, να επισημανθεί πως πρόκληση αποτελεί και το μέγεθος των σειριοποιημένων μοντέλων. Συγκεκριμένα, το συνολικό μέγεθος των αρχείων σειριοποίησης κυμαίνεται από μερικά MB μέχρι πολλαπλά GB [30], η διαχείριση των

οποίων σαφώς και πρέπει να ληφθεί υπόψη στις λειτουργικές απαιτήσεις του συστατικού αποθήκευσης.

Τέλος, δοθέντας της αρχιτεκτονικής του συστήματος *Triabase*, μια άλλου είδους πρόκληση αποτελεί και η διασφάλιση της ασφάλειας και ιδιωτικότητας των δεδομένων των μοντέλων που αποθηκεύονται στο *Blockchain* που χρησιμοποιεί. Ειδικότερα, απαιτείται η υλοποίηση κάποιου (συμπληρωματικού) μηχανισμού ελέγχου εξουσιοδότησης χρήσης των υπηρεσιών του συστήματος. Η ανάγκη αυτή προκύπτει λόγω της εισαγωγής ενός ενδιάμεσου επιπέδου (*Application Layer*) μεταξύ των αιτητών παροχής υπηρεσιών και του *Storage Layer*. Ως εκ τούτου, ο έλεγχος εξουσιοδότησης που προσφέρει το *Hyperledger Fabric* (*Permissioned Blockchain*) *framework* δεν είναι αρκετός, αφού η επικοινωνία αιτητή – *Blockchain* γίνεται μέσω του *REST server* του *Application Layer*, ο οποίος είναι πάντα εξουσιοδοτημένος και ενωμένος (*connected*) στο δίκτυο *Blockchain*. Επομένως, χωρίς τη χρήση κάποιου επιπρόσθετου μηχανισμού ελέγχου εξουσιοδότησης, οποιοσδήποτε χρήστης/συσκευή καλέσει τις υπηρεσίες του *Application Layer* έχει το δικαίωμα τροποποίησης των δεδομένων του συστήματος.

4.2 Περιορισμοί Αλυσίδας Συστοιχιών

Προχωρώντας, σε αυτή την υπο-ενότητα παρουσιάζονται οι διάφοροι περιορισμοί που επιβάλλει η χρήση τεχνολογιών *Blockchain* στο σύστημα.

Ο πρώτος περιορισμός εντοπίζεται στη δομή των δεδομένων (*data structure*) προς αποθήκευση. Ειδικότερα, στο *Hyperledger Fabric framework*, όπως και στις πλείστες πλατφόρμες τεχνολογιών *Blockchain*, εφαρμόζεται το *key-value storage paradigm* για την αποθήκευση και, μετέπειτα, εντοπισμό/ανάκτηση των πόρων των ενεργών καταστίχων. Επομένως, η μεθοδολογία αποθήκευσης των δεδομένων των μοντέλων μηχανικής μάθησης πρέπει να προσαρμοστεί ανάλογα.

Ο δεύτερος περιορισμός έγκειται στον τρόπο αποστολής δεδομένων στο *Smart Contract* του *Blockchain* εν χρήση. Ως ένα ακόμα πρόγραμμα (αν και με ιδιαίτερη εφαρμογή), τα *Smart Contracts* αποτελούνται από μεθόδους με υπογραφές (*function*

signatures), οι οποίες περιλαμβάνουν συγκεκριμένους προκαθορισμένους τύπους δεδομένων. Ως εκ τούτου, και δεδομένου του ότι οι τύποι αρχείων σειριοποίησης είναι πολλαπλοί, δεν δύναται να παρέχεται μια μέθοδος για κάθε πιθανό τύπο αρχείου στο *Smart Contract* του συστήματος. Αντιθέτως, παρατηρούμε μια ανάγκη για γενίκευση.

Κλείνοντας, ο τρίτος περιορισμός παρουσιάζεται στο μέγιστο μέγεθος των δεδομένων ανά δοσοληψία (*maximum transaction payload size*). Συγκεκριμένα, το μέγιστο μέγεθος δεδομένων ανά δοσοληψία που μπορεί να υποστηρίξει το *Hyperledger Fabric framework* έχει αναφερθεί [31] πως φτάνει περίπου μέχρι τα 90 MB. Παρ' όλα αυτά, μέσω διαφόρων δοκιμών που έγιναν στα πλαίσια της παρούσας δουλειάς, το *trade-off* μεταξύ κύρια μνήμης *RAM* και μεγέθους δεδομένων ανά δοσοληψία είναι αρκετά μεγάλο (μικρές αυξήσεις στο μεγέθους επιφέρουν δραματικές αυξήσεις χρήσης υπολογιστικών πόρων - *RAM*). Επομένως, για τη δημιουργία μιας βιώσιμης λύσης στο θέμα υπό αναφορά (αποθήκευση μεγάλων μοντέλων), πρέπει να γίνει αξιοποίηση σχετικά μικρών μεγεθών δεδομένων ανά δοσοληψία.

4.3 Σχεδιασμός Συστατικού Αποθήκευσης

Σε αυτό το υπο-κεφάλαιο αναλύονται οι διάφορες τεχνικές και δομές δεδομένων που επιλέγηκαν κατά τη φάση σχεδιασμού του συστατικού αποθήκευσης, λαμβάνοντας υπόψη τις απαιτήσεις και τους περιορισμούς που αναφέρθηκαν πιο πάνω.

Επιπλέον, να σημειωθεί πως το παρόν υπο-κεφάλαιο δεν εμβαθύνει στην χρήση των *tokens*. Η τεχνική αυτή φέρει τον ρόλο του απαιτούμενου συμπληρωματικού μηχανισμού ελέγχου εξουσιοδότησης χρήσης του συστήματος *Triabase* και έχει ήδη αναλυθεί στις τελευταίες δύο παραγράφους του υπο-κεφαλαίου 3.1.2.

4.3.1 Σελιδοποίηση Δεδομένων

Δεδομένου του μεγάλου μεγέθους των μοντέλων μηχανικής μάθησης, αλλά και των θεμάτων που παρουσιάζονται στα σενάρια χρήσης μεγάλου όγκου δεδομένων ανά δοσοληψία, αποφασίστηκε η υιοθέτηση της τεχνικής της σελιδοποίησης (*Paging*). Πιο

συγκεκριμένα, τα δεδομένα ενός μοντέλου αποθηκεύονται στο δίκτυο *Blockchain* με τη χρήση πολλαπλών δοσοληψιών συγκεκριμένου όγκου δεδομένων (*page/payload size*).

Η χρήση σελιδοποίησης, όπως φαίνεται και στο κεφάλαιο 6 (Πειραματική Αποτίμηση), όντως επιτρέπει την επιτυχή αποθήκευση μεγάλων μοντέλων στο δίκτυο. Ωστόσο, εκτός από την προσθήκη επιπλέον overhead, η προσέγγιση αυτή εισάγει νέα σημεία αποτυχίας (*points of failure*) κατά την εκτέλεση των μεθόδων που παρέχει το *Smart Contract* του δικτύου. Ειδικότερα, με τη χρήση σελιδοποίησης διάφορες διαδικασίες διαχείρισης δεδομένων δεν είναι πλέον ατομικές (*atomic*). Αποτέλεσμα αυτού αποτελεί η πιθανότητα εύρεσης του δικτύου σε ασυνεπή κατάσταση *(inconsistent state*), έχοντας κατεστραμμένα δεδομένα (*corrupted data*).

Παραδείγματα εκτελέσεων που μπορεί να επιφέρουν καταστροφή δεδομένων (*data corruption*) αποτελούν η αποθήκευση δεδομένων και η διαγραφή δεδομένων. Εάν αποτύχει η διεκπεραίωση έστω και μια δοσοληψίας αποθήκευσης/διαγραφής, τότε δύναται να υπάρχουν ατελή (*incomplete*) ή/και απρόσιτα (*unreachable*) δεδομένα στο δίκτυο. Για την εκκαθάριση τέτοιων δεδομένων παρέχεται η μέθοδος *Clean-Up* που συζητείτε σε μετέπειτα στάδιο.

4.3.2 Δομές Δεδομένων

Στα Σχήματα 4.1 και 4.2 που ακολουθούν παρουσιάζονται οι δομές που επιλέγηκαν για την αποθήκευση δεδομένων στο δίκτυο *Blockchain*.

```
type MetaData struct {
    ModelID              string `json:"model_id"`
    Tag1                 string `json:"tag1"`
    Tag2                 string `json:"tag2"`
    SerializationEncoding string `json:"serialization_encoding"`
    PageNumber           int    `json:"page_number"`
}
```

Σχήμα 4.1 : Δομή μεταδεδομένων γενικής σελίδας.

```
type GenericPage struct {
    ModelID   string `json:"model_id"`
    PageID    int    `json:"page_id"`
    PageBytes int    `json:"page_bytes"`
    Data      string `json:"data"`
    Digest    string `json:"digest"`
}
```

Σχήμα 4.2 : Δομή δεδομένων γενικής σελίδας.

Όπως φαίνεται στο Σχήμα 4.1, τα μεταδιδόμενα ενός μοντέλου (*MetaData*) περιλαμβάνουν τον ταυτοποιητή του (*ModelID*), δύο περιγραφικές ετικέτες (Tag1, Tag2), την κωδικοποίηση σειριοποίησης του (*SerializationEncoding*), και το πλήθος σελίδων του μοντέλου (*PageNumber*).

Ακολούθως, όπως βλέπουμε στο Σχήμα 4.2, τα δεδομένα του μοντέλου διαιρούνται σε πολλαπλές γενικές σελίδες (*GenericPage*). Μια γενική σελίδα περιλαμβάνει τον ταυτοποιητή του μοντέλου στο οποίο ανήκει η σελίδα (*ModelID*), τον ταυτοποιητή/αριθμό της σελίδας (*PageID*), το μέγεθος της σελίδας (*PageBytes*), τα σειριοποιημένα δεδομένα που περιέχει (*Data*), και το *digest* των δεδομένων αυτών (*Digest*).

Όπως διαπιστώνουμε από τη γενική μορφή των *GenericPages*, το επίπεδο αποθήκευσης διαχειρίζεται τα δεδομένα των μοντέλων χωρίς επίγνωση του είδους τους (αρχιτεκτονική μοντέλου, βάρη, *framework* προέλευσης, κλπ). Ο τρόπος αποτελεσματικής αξιοποίησης της γενίκευσης αυτής, αλλά και της ανάγκης ύπαρξης του *Digest* μελετάται στο κεφάλαιο 5 (Συστατικό Διεπαφής).

Επιπρόσθετα, να αναφέρουμε ότι ο διαχωρισμός των δεδομένων και μεταδεδομένων ενός μοντέλου προτιμήθηκε, τόσο για λόγους εξοικονόμησης χώρου, όσο και για σκοπούς αύξησης της επίδοσης του συστήματος. Εμβαθύνοντας, η χρήση μιας εγγραφής μεταδεδομένων επιτρέπει την αποδοτική ενημέρωση των δεδομένων ενός μοντέλου. Ειδικότερα, η διαδικασία ενημέρωσης που έχει υλοποιηθεί αποτελεί μια απλή αντικατάσταση (*overwrite*) των αναγκαίων σελίδων, χωρίς κάποια διαγραφή *trail over* σελίδων (σε περιπτώσεις συρρίκνωσης του όγκου δεδομένων ενός μοντέλου). Η προσέγγιση αυτή δεν επιφέρει προβλήματα ορθότητας κατά την ανάγνωση δεδομένων,

αφού ο μηχανισμός ανάκτησης που παρέχεται επιστρέφει μόνο έγκυρες σελίδες. Ως «έγκυρες» θεωρούνται οι πρώτες N σελίδες του μοντέλου, όπου το N ισούται του *PageNumber* (στο το Σχήμα 4.1), όπως αυτό προέκυψε από την τελευταία ενημέρωση του. Επιπλέον, για σκοπούς διαγραφής των *trail over* σελίδων, παρέχεται η διαδικασία *Clean-Up*, κατά την οποία διαγράφονται οι άχρηστες σελίδες αυτές. Στην παρούσα φάση, η διαδικασία αυτή (*Clean-Up*) προσφέρεται μόνο για χειροκίνητη (*manual*) χρήση.

4.3.3 World State Database

Όπως αναφέρθηκε στο υπο-κεφάλαιο 3.2 (Ροή Εκτέλεσης/Δεδομένων), στο *Hyperledger Fabric framework* η χρήση ενός *World State Database* είναι υποχρεωτική. Επίσης, οι βάσεις δεδομένων που υποστηρίζει το *framework* είναι συγκεκριμένες. Οι βάσεις αυτές είναι η *LevelDB* και η *CouchDB*.

Για τους σκοπούς της παρούσας δουλειάς έγινε επιλογή της *CouchDB*, λόγω της ανάγκης χρήσης των πλούσιων επερωτημάτων (*rich queries*) που υποστηρίζει, κάτι το οποίο δεν υπάρχει στην *LevelDB*. Η *LevelDB* αποτελεί ένα, σχετικά, απλό *key-value store* [32].

4.4 Υλοποίηση Smart Contract

Συνεχίζοντας, στην παρούσα ενότητα παρουσιάζονται οι μέθοδοι του *Smart Contract* του δικτύου, οι οποίες αποφασίστηκαν με βάση όσα έχουν προαναφερθεί στο υπο-κεφάλαιο Σχεδίαση.

Ακολουθούν οι κύριες μέθοδοι που υλοποιήθηκαν, όπως και αποσπάσματα (*snippets*) από τον αντίστοιχο κώδικα του *Smart Contract* που βρίσκεται στο Παράρτημα «Β». Να σημειωθεί πως ο κώδικας των αποσπασμάτων είναι γραμμένος σε *Golang*, και πως ο ακριβής τρόπος αξιοποίησης των παρακάτω μεθόδων θα παρουσιαστεί στο κεφάλαιο 5 (Συστατικό Διεπαφής).

- ***CreateClientToken*** (Σχήμα 4.3): Χρησιμοποιείται για την αποθήκευση ενός *token* στο κατάστιχο. Το *token* λαμβάνεται ως είσοδος της μεθόδου αυτής.

- ***CheckClientToken*** (Σχήμα 4.3): Χρησιμοποιείται για τον έλεγχο ύπαρξης ενός *token* στο κατάστιχο, επιστρέφοντας *True* ή *False* ανάλογα.

```
func (t *SimpleChaincode) CreateClientToken(ctx contractapi.TransactionContextInterface,
    token string) (string, error) {

    err := ctx.GetStub().PutState(token, []byte{'t', 'o', 'k', 'e', 'n'})
    if err != nil {
        return "", err
    }

    return token, nil
}

func (t *SimpleChaincode) CheckClientToken(ctx contractapi.TransactionContextInterface,
    token string) (bool, error) {
    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return false, fmt.Errorf("failed to retrieve token %s from world state. %v", token, err)
    }

    return tokenBytes != nil, nil
}
```

Σχήμα 4.3 : Κώδικας *CreateClientToken, CheckClientToken*.

- ***SubmitMeta, SubmitPage*** (Σχήμα 4.4): Χρησιμοποιούνται για την αποθήκευση μιας εγγραφής μεταδεδομένων, ή μιας σελίδας δεδομένων, αντίστοιχα.

  Ο κώδικας των δυο μεθόδων αυτών είναι πανομοιότυπος, για αυτό και στο σχετικό σχήμα (4.4) δίνεται μόνο η υλοποίηση της μιας εξ' αυτών.

- ***QueryMetaData***: Χρησιμοποιείται για την ανάκτηση των μεταδεδομένων ενός συγκεκριμένου μοντέλου.

  Η υλοποίηση της μεθόδου αυτής είναι πανομοιότυπη με αυτή της μεθόδου *CheckClientToken*. Η διαφορά των δύο έγκειται μονάχα στο ότι η *QueryMetaData* επιστρέφει τα αυτούσια δεδομένα που ανακτώνται από το κατάστιχο, αντί μια παραγόμενη τιμή (*boolean*) όπως η *CheckClientToken*.

```
func (t *SimpleChaincode) SubmitMeta(ctx contractapi.TransactionContextInterface, token,
    modelID, entryString string) error {

    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return fmt.Errorf("failed to retrieve token %s from world state. %v", token, err)
    }

    if tokenBytes == nil {
        return fmt.Errorf("authorization not granded: token %s is invalid", token)
    }

    var entry MetaData
    err = json.Unmarshal([]byte(entryString), &entry)
    if err != nil {
        return fmt.Errorf("failed to process json data; ensure correct structure")
    }

    entryBytes, err := json.Marshal(entry)
    if err != nil {
        return err
    }

    err = ctx.GetStub().PutState(modelID+"_metadata", entryBytes)
    if err != nil {
        return err
    }

    return nil
}
```

Σχήμα 4.4 : Κώδικας *SubmitMeta*.

- ***QueryMetaDataWithPagination, QueryLatestModelWithPagination*** (Σχήματα 4.5, 4.6): Χρησιμοποιούνται για την ανάκτηση πολλαπλών εγγραφών μεταδεδομένων (διαφόρων μοντέλων), ή πολλαπλών σελίδων δεδομένων (ενός συγκεκριμένου μοντέλου), αντίστοιχα.

```
func (t *SimpleChaincode) QueryMetaDataWithPagination(ctx contractapi.TransactionContextInterface,
    token, queryString string, pageSize int, bookmark string) (*MetaPaginatedQueryResult, error) {

    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return nil, fmt.Errorf("failed to retrieve token %s from world state. %v", token, err)
    }

    if tokenBytes == nil {
        return nil, fmt.Errorf("authorization not granded: token %s is invalid", token)
    }

    return getMetaQueryResultForQueryStringWithPagination(ctx, queryString, int32(pageSize), bookmark)
}
```

Σχήμα 4.5 : Απόσπασμα του κώδικα του *QueryMetaDataWithPagination*.

Η υλοποίηση των μεθόδων αυτών είναι πανομοιότυπη, με την μόνη αλλαγή να παρουσιάζεται στην παράμετρο *queryString* (βλ. Σχήμα 4.5), η οποία αφήνεται ανοικτή (δηλ. λαμβάνεται ως είσοδος) στη περίπτωση ανάκτησης μεταδεδομένων, ενώ συγκεκριμενοποιείται στην ανάκτηση σελίδων δεδομένων (βλ. Σχήμα 4.6). Ειδικότερα, να αναφέρουμε ότι η παράμετρος *queryString* περιλαμβάνει ένα *CouchDB adhoc query string* [33], που εκτελείται πάνω στο *World State Database* (*CouchDB*) ανακτώντας τις εγγραφές/σελίδες που ικανοποιούν το συγκεκριμένο επερώτημα (query).

```
fmt.Sprintf(`{"selector":{"$and":[{"model_id":"%s"},{"page_id": {"$lt":%d}}]}}`,
        modelID, meta.PageNumber)
```

Σχήμα 4.6 : Το *CouchDB adhoc query string* της μεθόδου *QueryLatestModelWithPagination*.

Τέλος, ο αριθμός των εγγραφών/σελίδων που επιστρέφονται δεν ξεπερνά το *pageSize* (βλ. Σχήμα 4.5) που παρέχετε ως είσοδος, ενώ το *bookmark* (βλ. Σχήμα 4.5) χρησιμοποιείται ως σελιδοδείκτης ανάγνωσης αποτελεσμάτων. Για παράδειγμα, με *pageSize*=2 και *bookmark*=“ ” (κενή συμβολοσειρά), η πρώτη κλήση της μεθόδου επιστρέφει τις πρώτες δύο εγγραφές/σελίδες και ένα νέο σελιδοδείκτη αποτελεσμάτων που δύναται να χρησιμοποιηθεί ως είσοδο στην επόμενη κλήση της μεθόδου (για την ανάκτηση των επόμενων δύο αποτελεσμάτων), κλπ.

- ***GetKeys*** (Σχήμα 4.7)*:* Χρησιμοποιείται για εύρεση όλων των εγγραφών/σελίδων του κατάστιχου οι οποίες ικανοποιούν συγκεκριμένα κριτήρια επιλογής τα οποία λαμβάνονται στη μορφή *CouchDB adhoc query string*. Επιστρέφει τα μοναδικά αναγνωριστικά (κλειδιά) των εγγραφών/σελίδων αυτών.

```go
func (t *SimpleChaincode) GetKeys(ctx contractapi.TransactionContextInterface, token, modelID,
    queryString string, pageSize int, bookmark string) ([]string, error) {

    var keys []string

    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return keys, fmt.Errorf("failed to retrieve token %s from world state. %v", token, err)
    }

    if tokenBytes == nil {
        return keys, fmt.Errorf("authorization not granded: token %s is invalid", token)
    }

    //retrieve the model's metadata
    metaBytes, err := ctx.GetStub().GetState(modelID + "_metadata")
    if err != nil {
        return keys, fmt.Errorf("failed to get metadata for model %s: %v", modelID, err)
    }
    if metaBytes == nil {
        return keys, fmt.Errorf("model_id %s does not exist", modelID)
    }

    book := bookmark

    for true {
        resultsIterator, responseMetadata, err :=
            ctx.GetStub().GetQueryResultWithPagination(queryString, int32(pageSize), book)
        if err != nil {
            return keys, err
        }
        defer resultsIterator.Close()

        for resultsIterator.HasNext() {
            queryResult, err := resultsIterator.Next()
            if err != nil {
                return keys, err
            }
            keys = append(keys, queryResult.Key)

        }

        if responseMetadata.FetchedRecordsCount == 0 {
            break
        }

        book = responseMetadata.Bookmark

    }

    return keys, nil
}
```

Σχήμα 4.7 : Κώδικας *GetKeys*.

- ***GetCleanUpKeys*** (Σχήμα 4.8): Χρησιμοποιείται για τον εντοπισμό των trail over (άχρηστων) σελίδων δεδομένων ενός συγκεκριμένου μοντέλου μηχανικής μάθησης. Επιστρέφει τα μοναδικά αναγνωριστικά (κλειδιά) των σελίδων αυτών.

Η μέθοδος *GetCleanUpKeys* είναι πανομοιότυπη με την *GetKeys*, η μόνη διαφορά τους έγκειται στο *queryString* (βλ. Σχήμα 4.7) που χρησιμοποιούν για τον εντοπισμό των δεδομένων με τα οποία πραγματεύεται η κάθε μια.

```
fmt.Sprintf(`{"selector":{"$and":[{"model_id":"%s"},{"page_id": {"$gte":%d}}]}}`,
        modelID, meta.PageNumber)
```

Σχήμα 4.8 : Το *CouchDB adhoc query string* της μεθόδου *GetCleanUpKeys*.

- ***DeleteKeys*** (Σχήμα 4.9)***:*** Χρησιμοποιείται για τη διαγραφή μιας, ή περισσότερων, εγγραφών/σελίδων του κατάστιχου χρησιμοποιώντας το μοναδικό αναγνωριστικό (κλειδί) τους (που λαμβάνεται ως είσοδος).

```
func (t *SimpleChaincode) DeleteKeys(ctx contractapi.TransactionContextInterface,
    token string, keys []string) error {

    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return fmt.Errorf("failed to retrieve token %s from world state. %v", token, err)
    }

    if tokenBytes == nil {
        return fmt.Errorf("authorization not granded: token %s is invalid", token)
    }

    for _, key := range keys {
        err = ctx.GetStub().DelState(key)
        if err != nil {
            return fmt.Errorf("failed to delete ledger entry %s: %v", key, err)
        }
    }

    return nil
}
```

Σχήμα 4.9 : Κώδικας *DeleteKeys*.

Σημείωσεις Υπο-κεφαλάιου:

Σε όλες τις μεθόδους (εκτός της *CreateClientToken*) υπάρχει έλεγχος εξουσιοδότησης, και η εκτέλεση της μεθόδου ματαιώνεται εάν το *token* που παρέχεται ως είσοδος δεν υπάρχει ήδη αποθηκευμένο στο κατάστιχο (δηλ. δεν είναι «έγκυρο»). Επίσης, να αναφερθεί πως ο έλεγχος εξουσιοδότησης γίνετε χωρίς την κλήση της

CheckClientToken λόγω διαφόρων ιδιαιτεροτήτων κωδικοποίησης των *Smart Contracts* στο *Hyperledger Fabric framework*. Η μέθοδος *CheckClientToken* παρέχεται μόνο για λόγους πληρότητας του *API* του *Smart Contract*.

Επιπρόσθετα, η μέθοδος *CreateClientToken* δεν χρησιμοποιείται στα πλαίσια της παρούσας δουλειάς- παραθέτεται, ωστόσο, για λόγους πληρότητας, αλλά και ως βάση μελλοντικών βελτιώσεων της διαδικασίας απόκτησης «έγκυρων» *tokens* (βλ. υποκεφάλαιο 7.2).

Κεφάλαιο 5

Συστατικό Διεπαφής

5.1 Απαιτήσεις Επιπέδου Εφαρμογής 49

5.2 Περιορισμοί 49

5.3 Σχεδιασμός Συστατικού Διεπαφής 50

5.3.1 Αρχιτεκτονική REST 50

5.3.2 Σχήμα Δεδομένων Διεπαφής 51

5.3.3 API Επιπέδου Εφαρμογής 53

5.3.4 Επιπρόσθετες Λειτουργίες Διακομιστή 54

5.4 Υλοποίηση Διακομιστή REST 55

5.4.1 Κύριες Συναρτήσεις 55

5.4.2 Ενεργοποίηση Επιπρόσθετων Λειτουργιών Διακομιστή 60

5.4.3 Παραδείγματα Χρήσης Διεπαφής 61

Συνεχίζοντας με το συστατικό διεπαφής, στο κεφάλαιο αυτό γίνεται ανάλυση των απαιτήσεων και περιορισμών που λήφθηκαν υπόψη κατά τον σχεδιασμό και υλοποίηση του διακομιστή (*server*) του επιπέδου εφαρμογής (*Application Layer*) του συστήματος *Triabase*. Επιπλέον, παρουσιάζονται οι λειτουργίες που υποστηρίζει η παρούσα έκδοση του συστήματος, και επεξηγείται με λεπτομέρεια ο τρόπος αξιοποίησης των μεθόδων που υλοποιεί το συστατικό αποθήκευσης (βλ. Κεφάλαιο 4) για την παροχή αυτών (των ολοκληρωμένων λειτουργιών).

5.1 Απαιτήσεις Επιπέδου Εφαρμογής

Αρχικά, όπως η κάθε διεπαφή που δύναται να χρησιμοποιηθεί και από ανθρώπους, το συστατικό διεπαφής οφείλει να είναι εύκολο στη κατανόηση, εκμάθηση και χρήση του. Επίσης, πρέπει να αποκρύπτει την περίπλοκη επικοινωνία με το *Blockchain* του συστήματος, παρέχοντας ήδη υλοποιημένες εύχρηστες μεθόδους αλληλεπίδρασης με αυτό.

Προχωρώντας σε πιο τεχνικής φύσεως απαιτήσεις, το συστατικό διεπαφής πρέπει να μπορεί να διαχειριστεί τα (πιθανόν μεγάλου μεγέθους και) διαφόρων τύπων αρχεία των σειριοποιημένων μοντέλων μηχανικής μάθησης με τα οποία πραγματεύεται το σύστημα *Triabase* (βλ. υπο-κεφάλαιο 4.1).

Συνεχίζοντας, σημαντικό ακόμα είναι η τεχνολογία υλοποίησης του συστατικού να είναι συμβατή με ένα ευρύ φάσμα άλλων τεχνολογιών τις οποίες δύναται να χρησιμοποιούν οι έξυπνες συσκευές και οι διαχειριστές του *IoT* δικτύου που αξιοποιεί το σύστημα *Triabase*. Επιπλέον, λόγω των (συνήθως) περιορισμένων πόρων των συσκευών στα *IoT* δίκτυα, ιδανικά, η εξασφάλιση επικοινωνίας με το συστατικό διεπαφής πρέπει να απαιτεί ελάχιστες επιπλέον βιβλιοθήκες ή/και εγκαταστάσεις λογισμικών από την πλευρά των *IoT* συσκευών.

Τέλος, το συστατικό διεπαφής καλείται να διασφαλίσει την ακεραιότητα των δεδομένων κατά την μεταφορά τους από το επίπεδο εφαρμογής στο επίπεδο αποθήκευσης, αφού η λεπτότητα των δεδομένων ενός μοντέλου μηχανικής μάθησης είναι αδιαμφισβήτητη. Συγκεκριμένα, (π.χ.) το χάσιμο μερικών *bits* σε ένα *lossy channel* μπορεί να επηρεάσει δραματικά την ακρίβεια πρόβλεψης ενός μοντέλου ή/και να προκαλέσει την πλήρη καταστροφή του (*model corruption*).

5.2 Περιορισμοί

Ο μόνος περιορισμός υλοποίησης του συστατικού διεπαφής έγκειται στην επιλογή της γλώσσας προγραμματισμού των μεθόδων επικοινωνίας με το *Blockchain* του επιπέδου αποθήκευσης.

Ειδικότερα, παρ’ όλο που η εκπαίδευση και διαχείριση μοντέλων μηχανικής μάθησης συνήθως γίνεται στη γλώσσα *Python*, εν τούτοις η υλοποίηση του συστατικού

χρησιμοποιώντας *Python* δεν είναι δυνατή, λόγω περιορισμών του *Hyperledger Fabric framework*. Εμβαθύνοντας, το *Hyperledger Fabric* μέχρι στιγμής παρέχει βιβλιοθήκες (*SDKs*) μόνο για τις γλώσσες *Java* και *JavaScript* (*Node.js*), ενώ υπόσχετε την μελλοντική υποστήριξη και άλλων γλωσσών (*Python* και *Golang*) και τεχνολογιών (*REST*) [34].

Διευκρίνηση:

Υπάρχει υποστήριξη ενός *Python SDK* για το *Hyperledger Fabric framework* [35] (για την έκδοση v1.4.x), αλλά το συγκεκριμένο *SDK* δεν είναι επίσημο ή/και συμβατό με την έκδοση του *Hyperledger Fabric* που αξιοποιεί το σύστημα *Triabase* (v2.x).

5.3 Σχεδιασμός Συστατικού Διεπαφής

Σε αυτό το υπο-κεφάλαιο αναλύονται οι διάφορες τεχνικές και σχήματα που επιλέγηκαν κατά τη φάση σχεδιασμού του συστατικού διεπαφής, λαμβάνοντας υπόψη τις απαιτήσεις και τους περιορισμούς που προαναφέρθηκαν.

5.3.1 Αρχιτεκτονική REST

Ως *REST* (*Representational State Transfer*) αποκαλείται η αρχιτεκτονική Πελάτη-Διακομιστή (*Client-Server*) που χρησιμοποιείται κατά κόρων στις εφαρμογές ιστού σήμερα.

Μεταξύ άλλων, ίσως το κυριότερο χαρακτηριστικό ενός *RESTful API* (δηλ. ενός *API* που ακολουθεί τις προδιαγραφές της αρχιτεκτονικής *REST*) αποτελεί η παροχή υπηρεσιών μέσω *URL-based endpoints* από έναν αφοσιωμένο διακομιστή (*dedicated server*). Πιο συγκεκριμένα, για τη χρήση των υπηρεσιών αυτών, ο Πελάτης αποστέλλει ένα αίτημα στον Διακομιστή χρησιμοποιώντας το πρωτόκολλο *HTTP* (*HyperText Transfer Protocol*). Αφού επεξεργαστεί το αίτημα και παράξει το επιθυμητό αποτέλεσμα, ο Διακομιστής αποστέλλει την απάντηση στο αίτημα του Πελάτη, επίσης χρησιμοποιώντας το *HTTP*.

Στη περίπτωση του συστατικού διεπαφής, η επιλογή της αρχιτεκτονικής *REST* έγινε για λόγους ευκολίας εκμάθησης και χρήσης (απλές, ευρέως γνωστές έννοιες), όπως και για τη μεγιστοποίηση του αριθμού έξυπνων συσκευών που δύναται να αλληλεπιδράσουν με το *API* του επιπέδου εφαρμογής- συμβατότητα (οι πλείστες συσκευές υποστηρίζουν ανταλλαγές μηνυμάτων μέσω *HTTP*).

Επιπρόσθετα, όσο αφορά την τεχνολογία υλοποίησης του *RESTful API* του συστατικού, δεδομένου των περιορισμένων επιλογών (*SDKs*) που αναφέρθηκαν στην προηγούμενη υπο-ενότητα, επιλέγηκε το οικοσύστημα *Node.js*. Η επιλογή αυτή έγινε για σκοπούς επίδοσης [36].

5.3.2 Σχήμα Δεδομένων Διεπαφής

Συνεχίζοντας, σε αυτό το υπο-κεφάλαιο παρουσιάζονται τα σχήματα δεδομένων τα οποία οι πελάτες («αιτητές») του συστήματος *Triabase* καλούνται, είτε να δημιουργήσουν, είτε να ερμηνεύσουν (όταν λάβουν τα αποτελέσματα των αιτημάτων τους).

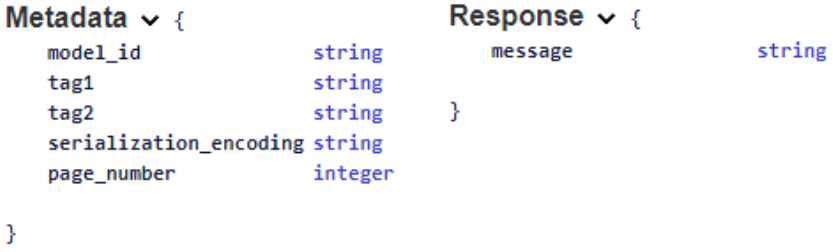
Στο Σχήμα 5.1 παρουσιάζεται η δομή του αντικειμένου *JSON* η οποία χρησιμοποιείται για την αποστολή/παραλαβή των δεδομένων (και μεταδεδομένων) ενός σειριοποιημένου μοντέλου στο/από το συστατικό διεπαφής. Όπως βλέπουμε, τα διάφορα αρχεία που προκύπτουν από τη διαδικασία σειριοποίησης ενός μοντέλου αντιμετωπίζονται ως συμβολοσειρές (*strings*). Αυτό επιφέρει την απαλοιφή του τύπου τους, επιτυγχάνοντας τη γενίκευση του τρόπου επεξεργασίας και διαχείρισης τους (μέσω της χρήσης των *GenericFiles*) από τα επίπεδα εφαρμογής και αποθήκευσης. Συνεχίζοντας, ένα γενικού τύπου αρχείο (*GenericFile*) περιέχει κάποια μεταδεδομένα που αφορούν το αρχικό αρχείο (τον ταυτοποιητή/όνομα και τον τύπο του) και τη συμβολοσειρά με τα σειριοποιημένα δεδομένα του. Τα αρχεία αυτά οργανώνονται σε τέσσερεις εννοιολογικές λίστες (*arrays*) που αποφασίστηκαν μετά από μελέτη του περιεχομένου και τρόπου αξιοποίησης τους. Οι λίστες αυτές ομαδοποιούν τα αρχεία ως εξής:

- *model*: γενικά αρχεία (αρχιτεκτονικής) μοντέλου
- *weights*: αρχεία βαρών
- *initialization*: αρχεία αρχικοποίησης

- *checkpoints*: αρχεία σημαντικών *checkpoints*

Σχήμα 5.1 : Σχήμα δεδομένων (και μεταδεδομένων) μοντέλων.

Τα δύο άλλα κύρια σχήματα δεδομένων που αξιοποιούνται από το συστατικό διεπαφής δίνονται στο Σχήμα 5.2. Στο Σχήμα 5.2 αριστερά παρουσιάζεται η δομή στην οποία επιστρέφονται τα μεταδεδομένα ενός μοντέλου όταν γίνεται χρήση των σχετικών επερωτημάτων (*Metadata*), ενώ δεξιά δίνεται η γενική δομή των μηνυμάτων επιτυχίας/σφάλματος (*Response*) εκτέλεσης των λειτουργιών που δεν επιφέρουν κάποια ανάκτηση δεδομένων.



Σχήμα 5.2 : Σχήματα μεταδεδομένων μοντέλων και μηνυμάτων.

5.3.3 API Επιπέδου Εφαρμογής

Σε αυτή την ενότητα αναλύονται τα endpoints που κρίθηκε αναγκαίο να παρέχονται από τον *REST server* του επίπεδου εφαρμογής. Τα *endpoints* αυτά αποφασίστηκαν μετά από μελέτη παρόμοιων δουλειών (κυρίως *BlockchainDB* [24]) και λαμβάνοντας υπόψη τις ιδιαιτερότητες του περιβάλλοντος των αλυσίδων συστοιχιών που έχουν σημειωθεί σε προηγούμενα κεφάλαια της παρούσας διπλωματικής εργασίας.

Endpoints που αφορούν τη δημιουργία/ενημέρωση/διαγραφή/συντήρηση ενός συγκεκριμένου μοντέλου, δοθέντος του ταυτοποιητή του:

- ***model***: Ανάκτηση μοντέλου.
- ***model/submit***: Ασύγχρονη δημιουργία ή ενημέρωση (update) μοντέλου.
- ***model/submit/check***: Έλεγχος προόδου του τελευταίου αιτήματος δημιουργίας/ ενημέρωσης.
- ***model/delete***: Ασύγχρονη διαγραφή μοντέλου.
- ***model/delete/check***: Έλεγχος προόδου του τελευταίου αιτήματος διαγραφής.
- ***model/cleanup***: Ασύγχρονη συντήρηση μοντέλου. Περιλαμβάνει τη διαγραφή μόνο των trail over σελίδων δεδομένων ενός μοντέλου (βλ. τελευταία παράγραφο υπο-κεφαλαίου 4.3.2).
- ***model/cleanup/check***: Έλεγχος προόδου του τελευταίου αιτήματος συντήρησης.

Endpoints που παρέχονται για σκοπούς ελέγχου κατάστασης/μεταδεδομένων ενός, ή περισσότερων, μοντέλων του συστήματος:

- ***metadata***: Ανάκτηση μεταδεδομένων ενός συγκεκριμένου μοντέλου, βάση του ταυτοποιητή του.
- ***metadata/tags***: Δοθέντος δύο τιμών, ανάκτηση όλων των εγγραφών μεταδεδομένων οι οποίες έχουν συγκεκριμένες περιγραφικές ετικέτες στα πεδία *tag1* και *tag2*, αντίστοιχα.
- ***metadata/tags/tag1***: Δοθέντος μιας τιμής, ανάκτηση όλων των εγγραφών μεταδεδομένων οι οποίες έχουν την τιμή αυτή στο πεδίο *tag1*.
- ***metadata/tags/tag2***: Δοθέντος μιας τιμής, ανάκτηση όλων των εγγραφών μεταδεδομένων οι οποίες έχουν την τιμή αυτή στο πεδίο *tag2*.

Όπως παρατηρούμε, οι λειτουργίες που παρέχει το επίπεδο εφαρμογής τη δεδομένη στιγμή είναι αρκετά βασικές/περιορισμένες. Αυτό, ωστόσο, δεν δημιουργεί ιδιαίτερα προβλήματα στο πεδίο για το οποίο προορίζεται το σύστημα *Triabase*. Στην παρούσα (πρώιμη) έκδοση του, το σύστημα παρέχει επαρκής μηχανισμούς που εξυπηρετούν την διαδικασία δημιουργίας μοντέλων μηχανικής μάθησης με χρήση *Federated Learning* σε *IoT* συστήματα.

5.3.4 Επιπρόσθετες Λειτουργίες Διακομιστή

Κλείνοντας με το υπο-κεφάλαιο σχεδίασης, να αναφέρουμε τους τρόπους με τους οποίους το συστατικό διεπαφής αξιοποιεί τις τεχνικές των *digests* και της συμπίεσης δεδομένων (*data compression*).

Όσο αφορά τη χρήση των *digests*, κατά τη λήψη δεδομένων στο *endpoint* ***model/submit*** ο *server* του συστατικού διεπαφής δημιουργεί τις αντίστοιχες σελίδες (δεδομένων) που προωθούνται για αποθήκευση στο *Blockchain* του συστήματος (βλ. Σχήματα 4.1 και 4.2). Για την κάθε σελίδα, ο *server* υπολογίζει ένα *digest*, βάση των δεδομένων της, και της το επισυνάπτει πριν την αποστείλει για αποθήκευση. Όταν, σε μεταγενέστερο στάδιο, παραληφθεί αίτημα ανάκτησης των δεδομένων αυτών (*endpoint* ***model***), ο *server* ανακτά τις σελίδες του μοντέλου του αιτήματος, βεβαιώνεται για την ακεραιότητα των δεδομένων της κάθε μιας (χρησιμοποιώντας το *digest* της), επαναφέρει τα δεδομένα στη μορφή του Σχήματος 5.1 και τα αποστέλλει στο χρήστη.

Συγκεκριμένα, για την υφιστάμενη έκδοση του συστατικού διεπαφής επιλέγηκε η χρήση της συνάρτησης κατακερματισμού *MD5*, κυρίως για λόγους εξοικονόμησης χώρου (*MD5 digest size* = 128 bits), αλλά και της χαλάρωσης που επιτρέπει το σημείο εφαρμογής της.

Προχωρώντας στην επόμενη τεχνική υπό εξέταση, έχουμε τα *endpoints* ***model/submit*** και ***model*** να υποστηρίζουν την συμπίεση (και αποσυμπίεση, αντίστοιχα) δεδομένων με τη χρήση *Node.js* βιβλιοθηκών (*modules*). Σκοπός της εφαρμογής μηχανισμών συμπίεσης είναι η μείωση του όγκου δεδομένων που διακινείται στο δίκτυο του συστήματος *Blockchain* και ταυτόχρονα η αύξηση της ταχύτητας ολοκλήρωσης των λειτουργιών αποθήκευσης και ανάκτησης μοντέλων. Σημειώνοντας ότι τα αρχεία δεδομένων των μοντέλων μηχανικής μάθησης δύναται να είναι ήδη σε αρκετά

συμπαγή/συμπιεσμένη μορφή, η επιπρόσθετη επιλογή συμπίεσης έχει υλοποιηθεί μόνο για σκοπούς έρευνας (βλ. υπο-ενότητα 6.4.4).

5.4 Υλοποίηση Διακομιστή REST

Συνεχίζοντας, στην παρούσα ενότητα περιγράφονται οι συναρτήσεις των λειτουργιών που παρέχονται μέσω των *endpoints* του συστατικού διεπαφής (βήματα), όπως και ο τρόπος ενεργοποίησης των επιπρόσθετων λειτουργιών που παρουσιάζονται στην ενότητα 5.3.4. Επίσης, δίνονται παραδείγματα χρήσης των κυριότερων *endpoints* του συστήματος μέσω του γραφικού περιβάλλοντος που αναπτύχθηκε για σκοπούς επίδειξης και δοκιμής των δυνατοτήτων του *Triabase*.

5.4.1 Κύριες Συναρτήσεις

Στο παρόν υπο-κεφάλαιο παραθέτονται τα επιμέρους βήματα των λειτουργιών αποθήκευσης, ανάκτησης και διαγραφής/συντήρησης των δεδομένων ενός μοντέλου, όπως και της λειτουργίας ελέγχου προόδου αυτών. Επιπρόσθετα, περιγράφονται οι συναρτήσεις απάντησης επερωτήσεων σε μεταδεδομένα. Σημαντική συνεισφορά της ενότητας αυτής αποτελεί και η επεξήγηση του τρόπου αξιοποίησης των μεθόδων που προσφέρει το *Smart Contract* του συστήματος (βλ. κεφάλαιο 4) στην υλοποίηση των endpoints που παρουσιάζονται στη συνέχεια.

Συνάρτηση ***model*** *endpoint*:

1. Ελέγχεται εάν όλοι οι απαραίτητοι παράμετροι δόθηκαν από το χρήστη ως είσοδο. Εάν όχι, επιστρέφεται μήνυμα λάθους, και η διαδικασία ανάκτησης μοντέλου τερματίζει.
2. Οι σελίδες δεδομένων του μοντέλου ανακτώνται από το επίπεδο αποθήκευσης, μέσω πολλαπλών κλήσεων στη μέθοδο *QueryLatestModelWithPagination*. Σε αυτό το σημείο, εάν το *token* του χρήστη κριθεί ως μη-έγκυρο (εντός της *QueryLatestModelWithPagination*), επιστρέφεται μήνυμα λάθους και η διαδικασία τερματίζει.

3. Ελέγχεται η ακεραιότητα των δεδομένων που ανακτήθηκαν, παράγοντας τα *digest* των ανακτημένων σελίδων και συγκρίνοντάς τα με τα αντίστοιχα *digest* που είχαν αποθηκευτεί μαζί με τα δεδομένα. Εάν έστω και ένα ζεύγος *digests* δεν συμφωνεί, τότε επιστρέφεται μήνυμα σφάλματος και η διαδικασία τερματίζεται.
4. Γίνεται αποσυμπίεση δεδομένων χρησιμοποιώντας το κατάλληλο *module*, (εάν η επιπρόσθετη επιλογή συμπίεσης/αποσυμπίεσης είναι ενεργοποιημένη).
5. Τα δεδομένα επαναφέρονται στη μορφή του Σχήματος 5.1 και επιστρέφονται στον χρήστη.

```
while(true){ //get all the data pages
    try{
        let start_timer2 = new Date().getTime();//for experimental purposes
        let response = await getLastPages(1, bookmark, req.query.token, req.query.id);
        sub += new Date().getTime() - start_timer2;
        if (page_counter == 0 && response.includes('authorization')){
            console.log('retrieve_time', new Date().getTime() - start_timer);
            res.status(401).send({'message':'Unauthorized: authorization not granded: token "'
                                    + req.query.token + '" is invalid'});
            return
        }
        let page = JSON.parse(response);
        model[page.records[0].Key] = page.records[0].Value;
        bookmark = page.bookmark;
        page_counter++;
    }catch{
        break;
    }
}
var data = "";
for (let i = 0; i < page_counter; i++){
    let temp_digest = MD5(model[i].data).toString();
    if (!(temp_digest === model[i].digest)){
        console.log('retrieve_time', new Date().getTime() - start_timer);
        res.status(500).send({'message':'Internal Server Error: Model data are corrupted.'});
        return
    }
    data += model[i].data;
}
```

Σχήμα 5.3 : Απόσπασμα κώδικα ανάκτησης και ελέγχου ακεραιότητας σελίδων μοντέλου.

Συνάρτηση ***model/submit*** *endpoint*:

1. Ελέγχεται εάν όλοι οι απαραίτητοι παράμετροι δόθηκαν από το χρήστη ως είσοδο. Εάν όχι, επιστρέφεται μήνυμα λάθους, και η διαδικασία τερματίζει.

2. Γίνεται συμπίεση των δεδομένων του μοντέλου που λήφθηκε ως είσοδος, χρησιμοποιώντας το κατάλληλο *module*, (εάν η επιπρόσθετη επιλογή συμπίεσης/αποσυμπίεσης είναι ενεργοποιημένη).
3. Τα δεδομένα διαιρούνται σε σελίδες προκαθορισμένου μεγέθους.
4. Υπολογίζεται το *digest* των δεδομένων της κάθε σελίδας και τοποθετείται στη δομή *GenericPage* μαζί με τα αυτούσια δεδομένα (και μεταδεδομένα που χρειάζονται).
5. Γίνονται οι αναγκαίες εκδόσεις δοσοληψιών (*transaction issuing*) για την αποθήκευση των *GenericPages* (και της δομής *MetaData*) που δημιουργήθηκαν στο *Blockchain*. Η διαδικασία αυτή γίνεται ασύγχρονα, χρησιμοποιώντας τις μεθόδους *SubmitMeta* και *SubmitPage* από το *Smart Contract* του δικτύου. Συγκεκριμένα, μόλις εκδοθούν με επιτυχία οι δοσοληψίες, επιστρέφεται στον χρήστη μήνυμα επιτυχίας (εκδόσεων), χωρίς όμως αυτό να εγγυείται και την επιτυχή αποθήκευση των δεδομένων στο *Blockchain*. Για τον έλεγχο επιτυχής αποθήκευσης δίνεται το *endpoint* ***model/submit/check***. (Σημείωση: Στο βήμα αυτό γίνεται και ο έλεγχος εγκυρότητας του *token* του χρήστη.)

```
async function submitPage(tx_data,  token, model_id) {

    let peerName = channel.getChannelPeer(PEER_NAME)
    var tx_id = client.newTransactionID();
    let tx_id_string = tx_id.getTransactionID();

    var request = {
        targets: peerName,
        chaincodeId: CHAINCODE_ID,
        fcn: 'SubmitPage',
        args: [ token, model_id, JSON.stringify(tx_data)],
        chainId: CHANNEL_NAME,
        txId: tx_id
    };

    if (DEBUG === 'true'){
        console.log("#0 Transaction id is: " + tx_id_string)
        console.log("#1 Transaction proposal successfully sent to channel.")
    }
    try{
        let results = await channel.sendTransactionProposal(request);
        var proposalResponses = results[0];
        var proposal = results[1];

        var all_good = true;
        for (var i in proposalResponses) {
            let good = false
            if (proposalResponses && proposalResponses[i].response &&
                proposalResponses[i].response.status === 200) {
                good = true;
                if (DEBUG === 'true'){console.log(`\tChaincode invocation proposal response #${i} was good`);}
            } else {
                console.log(`\tChaincode invocation proposal response #${i} was bad!`);
            }
            all_good = all_good & good
        }
        if (DEBUG === 'true'){console.log("#2 Looped through the proposal responses all_good=", all_good)}
```

```
        await setupTxListener(tx_id_string, 'page', model_id)
        if (DEBUG === 'true'){console.log('#3 Registered the Tx Listener')}

        var orderer_request = {
            txId: tx_id,
            proposalResponses: proposalResponses,
            proposal: proposal
        };

        await channel.sendTransaction(orderer_request);
        if (DEBUG === 'true'){console.log("#4 Transaction has been submitted.")}

    }catch(e){
        console.log(e);
        submit_tx_map_meta.set(model_id, {is_completed: 'true', status: 'UNAUTHORIZED', pending: -1});
        return
    }

}
```

Σχήμα 5.4 : Κώδικας συνάρτησης έκδοσης δοσοληψιών γενικών σελίδων (*GenericPages*).

Συνάρτηση ***model/[delete, cleanup]*** *endpoint*:

1. Ελέγχεται εάν όλοι οι απαραίτητοι παράμετροι δόθηκαν από το χρήστη ως είσοδο. Εάν όχι, επιστρέφεται μήνυμα λάθους, και η διαδικασία τερματίζει.
2. Ανακτώνται τα κλειδιά των σελίδων που θα διαγραφούν. Στην περίπτωση του ολοκληρωτικής διαγραφής (***delete***), ανακτώνται τα κλειδιά όλων των σελίδων του μοντέλο υπο αναφορά, συμπεριλαμβανομένων τυχόν μη-προσβάσιμων σελίδων του (*unreachable data*). Η ανάκτηση αυτών των κλειδιών γίνεται με τη χρήση της μεθόδου *GetKeys* από το *Smart Contract* του δικτύου, με την παράμετρο εισόδου *queryString* να ορίζεται όπως φαίνεται στο Σχήμα 5.5. Στην περίπτωση συντήρησης (***cleanup***), ανακτώνται μόνο τα κλειδιά των μη-προσβάσιμων σελίδων του μοντέλου, χρησιμοποιώντας τη μέθοδο *GetCleanUpKeys* (του *Smart Contract*).
3. Εκδίδεται δοσοληψία που καλεί τη μέθοδο *DeleteKeys* του *Smart Contract* του δικτύου, προωθώντας της τα κλειδιά που ανακτήθηκαν στο βήμα 2, με σκοπό την διαγραφή των σελίδων στις οποίες αντιστοιχούν. Παρομοίως με το ***model/submit*** *endpoint*, με την επιτυχή έκδοση της δοσοληψίας αυτής, επιστρέφεται στον χρήστη μήνυμα επιτυχίας (έκδοσης), χωρίς όμως να δίνεται εγγύηση επιτυχούς ολοκλήρωσης όλων των απαραίτητων διαγραφών. Για σκοπούς ελέγχου επιτυχούς ολοκλήρωσης των διαγραφών παρέχονται τα *endpoints* ***model/delete/check*** και ***model/cleanup/check***. (Σημείωση: Στο βήμα αυτό γίνεται και ο έλεγχος εγκυρότητας του *token* του χρήστη.)

```
`{"selector":{"model_id":"` + model_id + `"}}`
```

Σχήμα 5.5 : *queryString* ανάκτησης των κλειδιών όλων των σελίδων ενός μοντέλου.

Στο σημείο αυτό, συμπληρωματικά, να αναφέρουμε πως, καθ' όλη τη διάρκεια ζωής μιας δοσοληψίας, γίνεται ενημέρωση κάποιων υποβοηθητικών δομών δεδομένων (*HashMaps*). Οι συγκεκριμένες δομές αξιοποιούνται για σκοπούς αποθήκευσης πληροφοριών κατάστασης/προόδου των αιτημάτων ***submit***, ***delete***, και ***cleanup***, έτσι ώστε να μπορούν να απαντήσουν στα αιτήματα ελέγχου επιτυχίας (***check***) των χρηστών του συστήματος. Επιπρόσθετα, στις δομές αυτές δηλώνονται και οι μη-εξουσιοδοτημένες ενέργειες (που αποτράπηκαν). Αποτέλεσμα αυτού, είναι η ανάγκη χρήσης των ***check*** *endpoints* για τον εντοπισμό αποτυχίας αιτήματος λόγω μη-εξουσιοδότησης. Η επιλογή του συγκεκριμένου εσωτερικού τρόπου ελέγχου εξουσιοδότησης (δηλ. ο έλεγχος να γίνεται εντός των επιμέρους μεθόδων του *Smart Contract*) έγινε για λόγους απόδοσης (αποφυγή περεταίρω κλήσεων σε μεθόδους).

Συνάρτηση ***model/[submit, delete, cleanup]/check*** *endpoint*:

1. Ελέγχεται εάν όλοι οι απαραίτητοι παράμετροι δόθηκαν από το χρήστη ως είσοδο. Εάν όχι, επιστρέφεται μήνυμα λάθους, και η διαδικασία τερματίζει.
2. Ελέγχεται εάν το *token* του χρήστη είναι έγκυρο, χρησιμοποιώντας την μέθοδο *CheckClientToken* του *Smart Contract* του επιπέδου αποθήκευσης. Εάν το *token* δεν είναι έγκυρο, η διαδικασία ελέγχου προόδου τερματίζεται, επιστρέφοντας το αντίστοιχο μήνυμα λάθος στον χρήστη.
3. Ανάλογα με το είδος του αιτήματος ελέγχου, γίνεται ανάκτηση πληροφοριών κατάστασης προόδου από τις βοηθητικές δομές δεδομένων του *server*, οι οποίες επιστρέφονται στο χρήστη.

Συνάρτηση ***metadata*** *endpoint*:

1. Ελέγχεται εάν όλοι οι απαραίτητοι παράμετροι δόθηκαν από το χρήστη ως είσοδο. Εάν όχι, επιστρέφεται μήνυμα λάθους, και η διαδικασία τερματίζει.

2. Ελέγχεται εάν το *token* του χρήστη είναι έγκυρο, χρησιμοποιώντας την μέθοδο *CheckClientToken* του *Smart Contract* του επιπέδου αποθήκευσης. Εάν το *token* δεν είναι έγκυρο, η διαδικασία ελέγχου προόδου τερματίζεται, επιστρέφοντας το αντίστοιχο μήνυμα λάθος στον χρήστη.
3. Τα μεταδεδομένα του μοντέλου υπο αναφορά ανακτώνται από το Blockchain, μέσω κλήσης της μεθόδου *QueryMetaData* (του *Smart Contract*), και επιστρέφονται στο χρήστη.

Συνάρτηση ***medata/tags/[<empty>, tag1, tag2]*** *endpoint*:

1. Ελέγχεται εάν όλοι οι απαραίτητοι παράμετροι δόθηκαν από το χρήστη ως είσοδο. Εάν όχι, επιστρέφεται μήνυμα λάθους, και η διαδικασία τερματίζει.
2. Χρησιμοποιώντας τη μέθοδο *QueryMetaDataWithPagination* του Smart Contract και το κατάλληλο *queryString* ως είσοδο, ανακτώνται οι σχετικές εγγραφές μεταδεδομένων από το Blockchain. Το *queryString* που χρησιμοποιείται από το κάθε endpoint επερωτήματος μεταδεδομένων φαίνεται στο Σχήμα 5.6. Στο βήμα αυτό γίνεται και ο έλεγχος εξουσιοδότησης (του token), και επιστρέφεται μήνυμα λάθους στον χρήστη πριν τον τερματισμό της διαδικασίας.

```
metadata/tags :       '{"selector":{"$and":[{"tag1":"' + tag1 + '"},{"tag2":"' + tag2 + '"}]},
                                                  "use_index":["_design/indexTag12", "indexTag12"]}'

metadata/tags/tag1 : '{"selector":{"tag1":"' + tag1 + '"}, "use_index":["_design/indexTag1", "indexTag1"]}'

metadata/tags/tag2 : '{"selector":{"tag2":"' + tag2 + '"}, "use_index":["_design/indexTag2", "indexTag2"]}'
```

Σχήμα 5.6 : Τα *queryStrings* των επερωτημάτων μεταδεδομένων.

5.4.2 Ενεργοποίηση Επιπρόσθετων Λειτουργιών Διακομιστή

Το παρόν υπο-κεφάλαιο επικεντρώνεται στις επιλογές συμπίεσης/αποσυμπίεσης που προσφέρει ο *server* του επιπέδου εφαρμογής.

Συγκεκριμένα, υποστηρίζονται τέσσερεις επιλογές συμπίεσης/αποσυμπίεσης, με την κάθε μια να αντιστοιχεί σε ένα διαφορετικό *Node.js module*. Οι επιλογές αυτές κυρίως επικεντρώνονται στην μείωση των κλειδιών που παρουσιάζονται στο *JSON* που φέρει τα δεδομένα των μοντέλων που λαμβάνει το σύστημα, ή/και την συμπίεση των δεδομένων τους, με σκοπό την μείωση του χώρου που καταλαμβάνουν. Τα *modules* που υποστηρίζονται στην τελευταία έκδοση του συστήματος Triabase είναι τα *compress-json, compressed-json, jsonpack* και *zipson*.

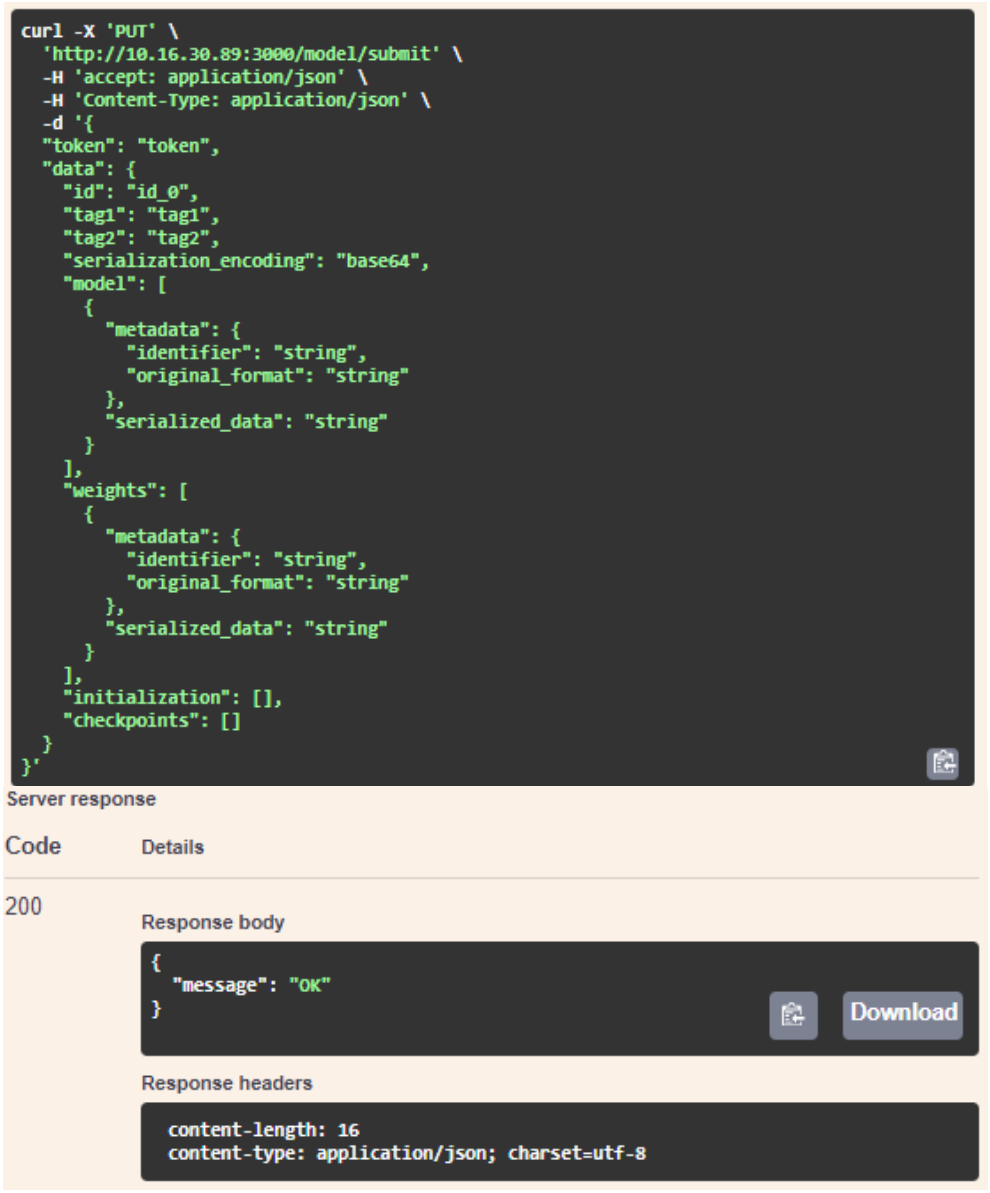
- *compress-json* [37]: Χρησιμοποιεί συνδυασμό των τεχνικών των *modules compressed-json* και *jsonpack*.
- *compressed-json* [38]: Βασίζεται στη συμπίεση συμβολοσειρών. Αντιμετωπίζει, δηλαδή, όλα τα δεδομένα του *JSON* που επεξεργάζεται ως συμβολοσειρές προς συμπίεση.
- *jsonpack* [39]: Μειώνει τον συνολικό αριθμό των κλειδιών που χρησιμοποιεί το *JSON* που επεξεργάζεται, διατηρώντας μόνο ένα σύνολο διακριτών κλειδιών. Επιφέρει αισθητή μείωση του όγκου των δεδομένων κυρίως όταν η δομή *JSON* προς συμπίεση είναι αναδρομική (*recursive*).
- *zipson* [40]: Προσφέρει τις κύριες λειτουργίες του βασικού *module JSON* της *Node.js*/*JavaScript* (*JSON.parse/stringify*) εμπλουτίζοντας τις με την επιλογή συμπίεσης του όγκου των δεδομένων που καλούνται να επεξεργαστούν.

Η επιλογή χρήσης κάποιου, εκ των προαναφερθέντων, *module* συμπίεσης/αποσυμπίεσης γίνεται κατά την ενεργοποίηση του ίδιου του *server* και δεν παρέχεται μηχανισμός επαναρυθμισμού (ή απενεργοποίησης της) εν όσο βρίσκετε σε λειτουργία.

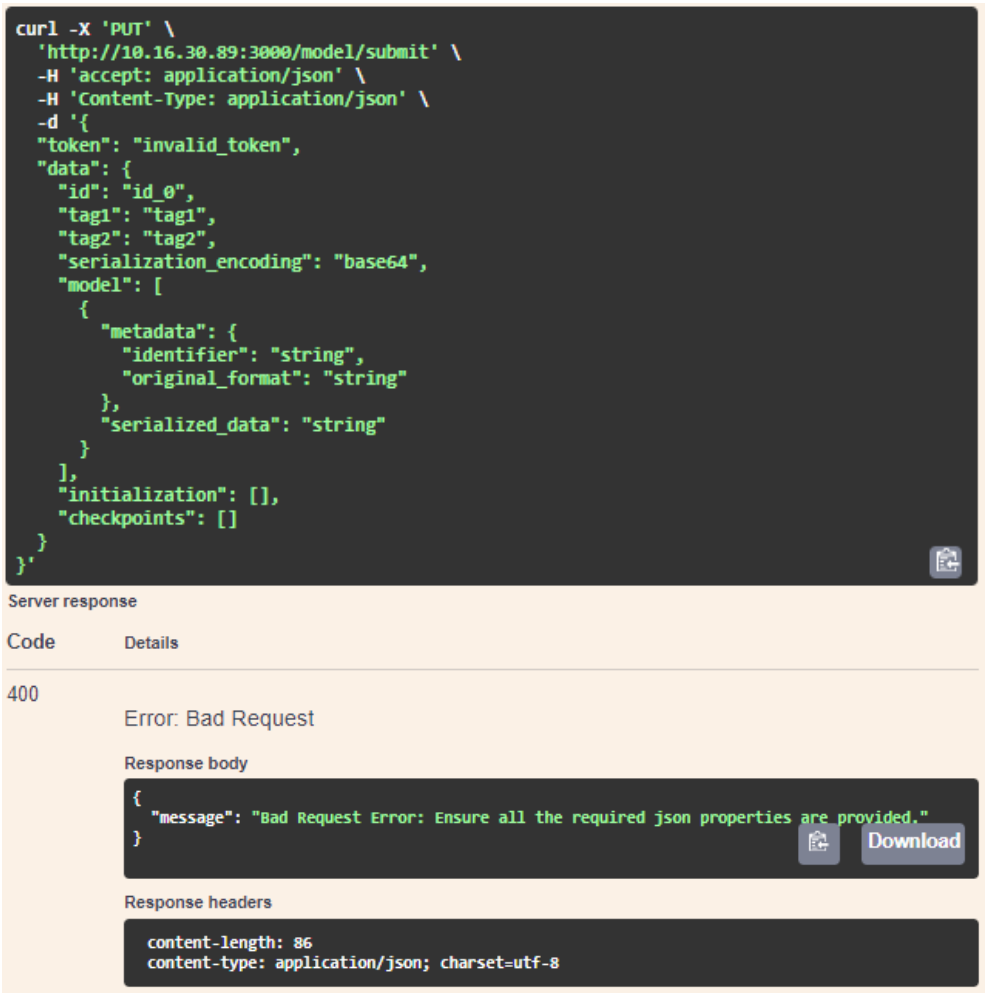
5.4.3 Παραδείγματα Χρήσης Διεπαφής

Κλείνοντας με το συστατικό διεπαφής, στη συνέχεια παραθέτονται *screenshots* που λήφθηκαν από την πιλοτική διεπαφή χρήστη που αναπτύχθηκε μέσω του εργαλείου *Swagger UI* κατά την χρήση των κυριότερων *endpoints* του συστατικού διεπαφής (με *test data*).

Submit Model



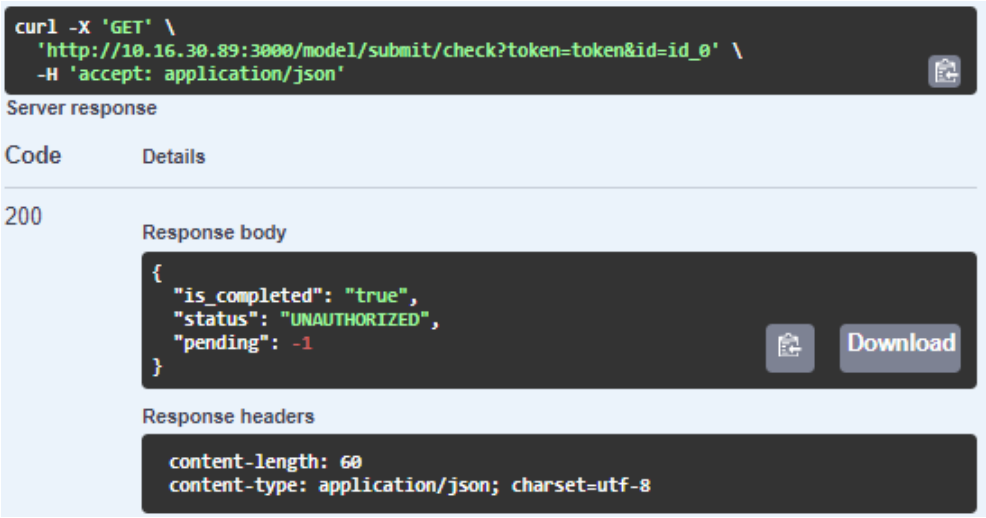
Σχήμα 5.7 : Επιτυχής εκτέλεση αιτήματος στο *model/submit* endpoint.



Σχήμα 5.8 : Αποτυχημένη εκτέλεση αιτήματος στο *model/submit* endpoint (έλλειψη παραμέτρων εισόδου).

Check Model (Submission)

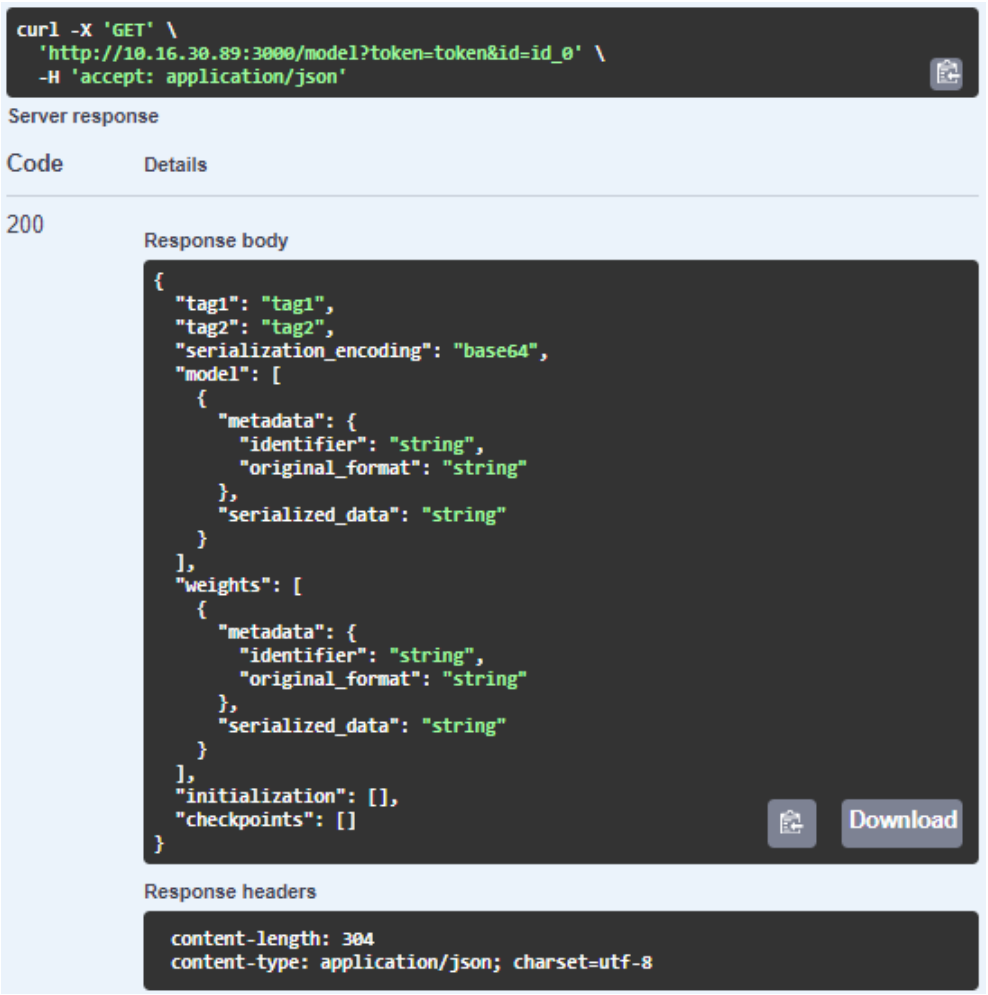
Σχήμα 5.9 : Έλεγχος προόδου αιτήματος submit - αίτημα εν εξέλιξη / PENDING.



Σχήμα 5.10 : Έλεγχος προόδου αιτήματος submit - επιτυχής ολοκλήρωση / VALID.

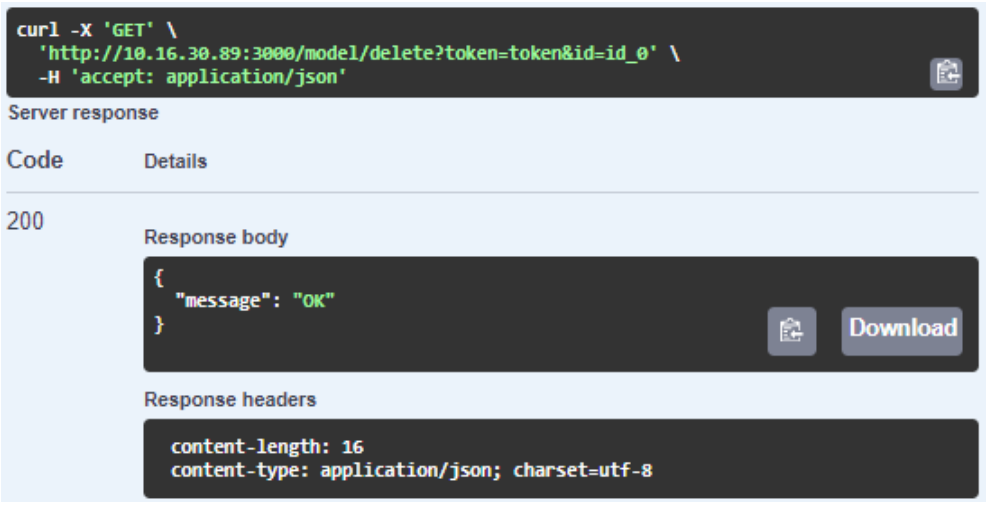
Σχήμα 5.11 : Έλεγχος προόδου αιτήματος submit - σφάλμα έλλειψης εξουσιοδότησης / UNAUTHORIZED.

Retrieve Model



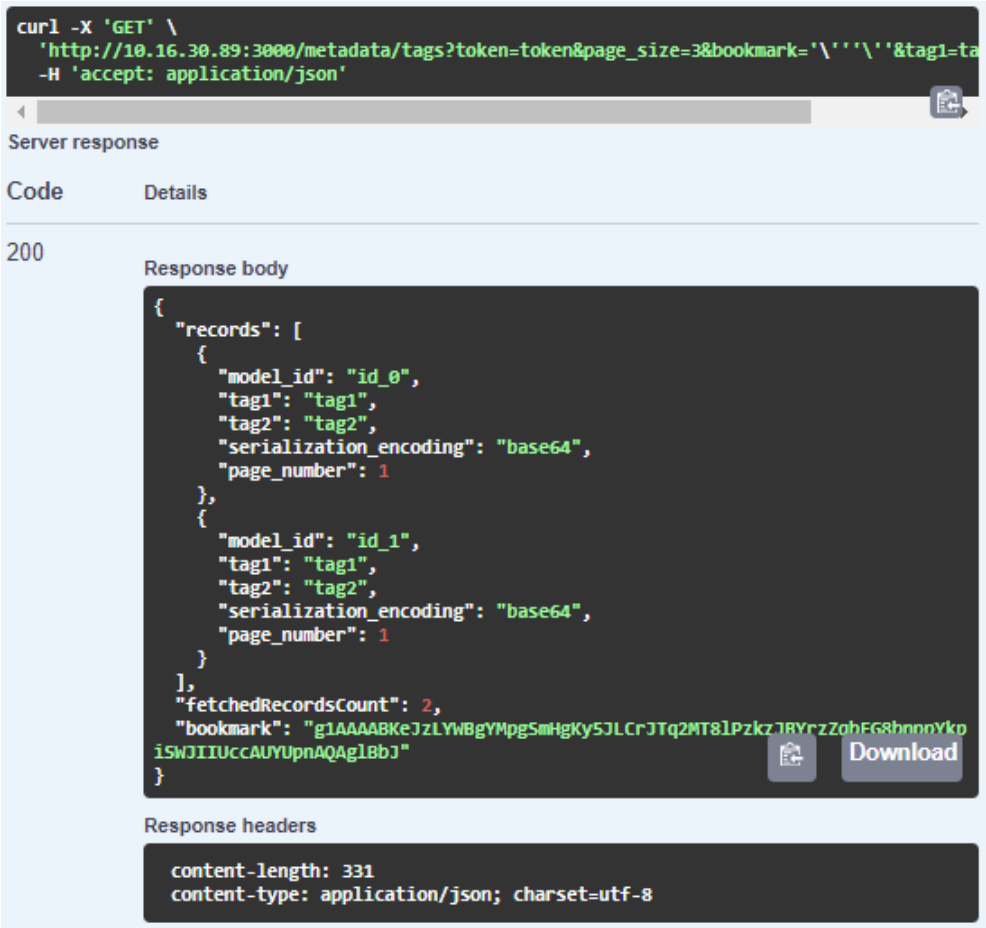
Σχήμα 5.12 : Επιτυχής ανάκτηση μοντέλου.

Delete / Clean-Up Model



Σχήμα 5.13 : Επιτυχής ανάκτηση μοντέλου.

Query Metadata (tag1 & tag2)



Σχήμα 5.14 : Επιτυχής ανάκτηση μεταδεδομένων με tag='tag1' και tag2='tag2'.

Κεφάλαιο 6

Πειραματική Αξιολόγηση

6.1 Πειραματική Διάταξη 68

6.2 Δεδομένα Αξιολόγησης 69

6.3 Διαδικασία Διεξαγωγής Πειραμάτων 70

6.3.1 Εισαγωγή Δεδομένων 71

6.3.2 Ανάκτηση Δεδομένων 71

6.4 Αποτίμηση Αποτελεσμάτων 72

6.4.1 Μέγεθος Σελίδας Δεδομένων 72

6.4.2 Μοντέλα Μεγάλου Μεγέθους 75

6.4.3 Ρυθμός Αιτημάτων Εισόδου 76

6.4.4 Συμπίεση Δεδομένων 77

Σε αυτό το κεφάλαιο περιγράφεται η σειρά πειραμάτων που διενεργήθηκαν με σκοπό την αξιολόγηση της δουλείας που έγινε στα πλαίσια της παρούσας διπλωματικής εργασίας. Όπως αναλύεται και σε μετέπειτα στάδιο, σκοπός των πειραμάτων είναι (κυρίως) η ρύθμιση διαφόρων παραμέτρων που επηρεάζουν την επίδοση του συστήματος *Triabase*, όπως και η μελέτη των δυνατοτήτων/ορίων της υφιστάμενης υλοποίησης.

6.1 Πειραματική Διάταξη

Η υπο-ενότητα αυτή επικεντρώνεται στην περιγραφή του πειραματικού *Blockchain* δικτύου και των πόρων που χρησιμοποιήθηκαν για την διεξαγωγή των διαφόρων πειραμάτων αξιολόγησης.

Αρχίζοντας με το κομμάτι τεχνολογιών *Blockchain*, να αναφέρουμε ότι το πειραματικό δίκτυο που αξιοποιήθηκε έχει στηθεί με τη χρήση του *Hyperledger Fabric framework*. Στο δίκτυο συμμετέχουν δύο *Organizations*, με δύο *Peers* ο καθένας (και με έναν εξ αυτών -των *Peers*- να είναι και *Endorsing Peer*), ενώ το *Ordering Service* αποτελείτε από μόνο έναν *Orderer*. Το συγκεκριμένο δίκτυο τρέχει τοπικά σε μια ιδεατή μηχανή (*virtual machine - VM*) στο *Data Center* του *Data Management System Laboratory* (*DMSL*) στο Πανεπιστήμιο Κύπρου, με τον κάθε κόμβο του δικτύου να εσωκλείεται σε ένα *Docker Container* στη μηχανή αυτή.

Η υλοποίηση του δικτύου επιτυγχάνεται με τη χρήση διαφόρων *shell scripts* τα οποία συγγράφηκαν στα πλαίσια της διπλωματικής εργασίας [4]. Στο Παράρτημα «Α» δίνονται τα βήματα που πρέπει να ακολουθηθούν για το στήσιμο του πειραματικού δικτύου που περιγράφηκε, του *REST server* του επιπέδου εφαρμογής. Στο σημείο αυτό να σημειωθεί πως, σε αντίθεση με τα αποτελέσματα των οδηγιών του παραρτήματος «Α», το δίκτυο *Blockchain* και ο *REST server* που έχουν χρησιμοποιηθεί στα πειράματα αξιολόγησης βρίσκονταν σε δύο διαφορετικά v*irtual machines* στο *DMSL* λόγω διαφόρων προβλημάτων που παρουσιάστηκαν κατά τη διεξαγωγή (των πειραμάτων).

Συνεχίζοντας, να αναφέρουμε πως οι πειραματικές δοκιμές έγιναν στο DMSL VCenter IaaS Datacenter, το οποίο περιλαμβάνει 5 IBM System x3550 M3 και HP Proliant DL 360 G7 servers, οι οποίοι έχουν δυνατότητες για single (8 cores), ή double (16 cores) socket Intel(R) Xeon(R) CPU E5620 @ 2.40GHz, αντίστοιχα. Οι server αυτοί έχουν συγκεντρωτικά 300 GB κύριας μνήμης και 16TB RAID-5 δευτερεύουσας μνήμης. Επιπλέον, η διαχείρηση του datacenter γίνεται μέσω του VMWare vCenter Server 5.1 που συνδέεται με τους αντίστοιχους κεντρικούς υπολογιστές VMWare ESXi 5.0.0.

Τέλος, στους υπολογιστικούς πόρους που χρησιμοποιήθηκαν στην πειραματική μας διάταξη περιλαμβάνονται τα ακόλουθα:

- VM Blockchain δικτύου: 120GB δευτερεύουσας μνήμης, 8 GB RAM, 16 ιδεατοί επεξεργαστές, και λειτουργικό σύστημα Ubuntu 18.04.
- VM REST server: 60GB δευτερεύουσας μνήμης, 16 GB RAM, 6 ιδεατοί επεξεργαστές, και λειτουργικό σύστημα Ubuntu 18.04.

6.2 Δεδομένα Αξιολόγησης

Όπως φαίνεται και στον Πίνακα 6.1, τα δεδομένα που χρησιμοποιήθηκαν στην παρούσα πειραματική αξιολόγηση είναι μοντέλα μηχανικής μάθησης, τα οποία πάρθηκαν έτοιμα (*pre-trained*) από διάφορα *Machine Learning hubs*. Χαρακτηριστικά παραδείγματα τέτοιων hubs αποτελούν το *TensorFlow Hub* και το *Hugging Face*.

Το πιο κάτω *dataset* (Πίνακας 6.1) περιλαμβάνει μοντέλα που αξιοποιούνται στην περιοχή της Υπολογιστικής Όρασης (*Computer Vision*). Συγκεκριμένα, λόγω της πολύτιμης προσφοράς του τομέα στο πεδίο του *IoT*, και δεδομένης της πλέον εκτεταμένης αξιοποίησης τεχνικών μηχανικής μάθησης στον τομέα της Υπολογιστικής Όρασης [41], η συγκεκριμένη περιοχή αποτελεί χαρακτηριστικό παράδειγμα εφαρμογής του συστήματος *Triabase* στο μέλλον. Ως ένα πιθανό σενάριο χρήσης, μπορούμε να θεωρήσουμε ένα *IoT* δίκτυο έξυπνων αυτοκινήτων σε μια πόλη, τα οποία συλλέγουν δεδομένα (σχετικά με το οδήγημα και τους διάφορους δρόμους), μαθαίνουν από αυτά (εκπαιδεύουν το τοπικό τους μοντέλο), και τα μοιράζονται με τα υπόλοιπα οχήματα του δικτύου, μέσω της συμμετοχής τους στη διαδικασία του *Federated Learning*. Επιπλέον, τα δεδομένα του συστήματος (μοντέλα) θα βρίσκονται αποθηκευμένα σε κοινή θέα σε ένα *Blockchain* δίκτυο για σκοπούς ελέγχου των οχημάτων του δικτύου από τις αρμόδιες αρχές.

Συνεχίζοντας, τα μοντέλα του *dataset* έχουν εκπαιδευτεί για αναγνώριση αντικειμένων (*object detection*), χρησιμοποιώντας το γνωστό *COCO dataset* (εικόνων) [42] (όχι απαραίτητα την ίδια έκδοση του). Επιπλέον, να αναφέρουμε και ότι τα μοντέλα υλοποιούν διάφορους/διαφορετικούς αλγόριθμους ή/και αρχιτεκτονικές για να επιτύχουν το *task* τους (*object detection*), όπως το *YOLO* [43], το *SSD Mobilenet v2*

[44] και *EfficientDet* [45]. Αυτό δεν επηρεάζει την ορθότητα των πειραμάτων που διεξάχθηκαν, αφού η χρήση τους αποσκοπεί τη μελέτη απόδοσης του συστήματος όταν καλείται να διαχειριστεί πραγματικά/ρεαλιστικά δεδομένα διαφόρων *frameworks*, μεγεθών και τύπων, αρχιτεκτονικών, αλγορίθμων ή/και τομέων.

Framework	**Model**	
	Disk Size (MB)	**Technology/Algorithm**
Darknet	236	YOLO
PyTorch	14,1	YOLO
Tensorflow v2	31,8	SSD Mobilenet v2
Tensorflow JS	75,4	SSD Mobilenet v2
Tensorflow Lite	4,34	EfficientDet

Πίνακας 6.1 : Πειραματικά δεδομένα – έτοιμα μοντέλα από hubs.

Επιπρόσθετα, για την αξιολόγηση απόδοσης κατά την διαχείριση μεγαλύτερων αρχείων χρησιμοποιήθηκαν κάποια *test data*. Εμβαθύνοντας, δεδομένου ότι τα σειριοποιημένα δεδομένα των μοντέλων λαμβάνονται, και διαχειρίζονται, από το σύστημα ως συμβολοσειρές (βλ. υπο-ενότητα 5.3.2), κατά την διεξαγωγή του πειράματος της ενότητας 6.4.3 εφαρμόστηκε η τεχνική της δυναμικής δημιουργίας *test* «μοντέλων». Ουσιαστικά, τα δεδομένα ενός *test* «μοντέλου» αποτελούνται από τυχαίες συμβολοσειρές συγκεκριμένου μεγέθους, που παράγονται δυναμικά πριν γίνει υποβολή του «μοντέλου» στο σύστημα Triabase.

6.3 Διαδικασία Διεξαγωγής Πειραμάτων

Σε αυτή την ενότητα γίνεται αναφορά στον προσομοιωτή Πελατών/αιτητών που δημιουργήθηκε για την πειραματική αποτίμηση της παρούσας δουλειάς. Ο συγκεκριμένος προσομοιωτής αποτελεί ένα αρχείο (*simulator.js*) γραμμένο σε *Node.js*, το οποίο αποστέλλει αιτήματα αποθήκευσης και ανάκτησης δεδομένων μοντέλων στα αντίστοιχα endpoints του *REST server* του συστήματος *Triabase*.

Ο κώδικας του προσομοιωτή δίνεται στο Παράρτημα «Β».

6.3.1 Εισαγωγή Δεδομένων

Όπως φαίνεται και στο Σχήμα 6.1, με κόκκινα βέλη, η πειραματική αποθήκευση δεδομένων στο σύστημα περιλαμβάνει πέντε βήματα.

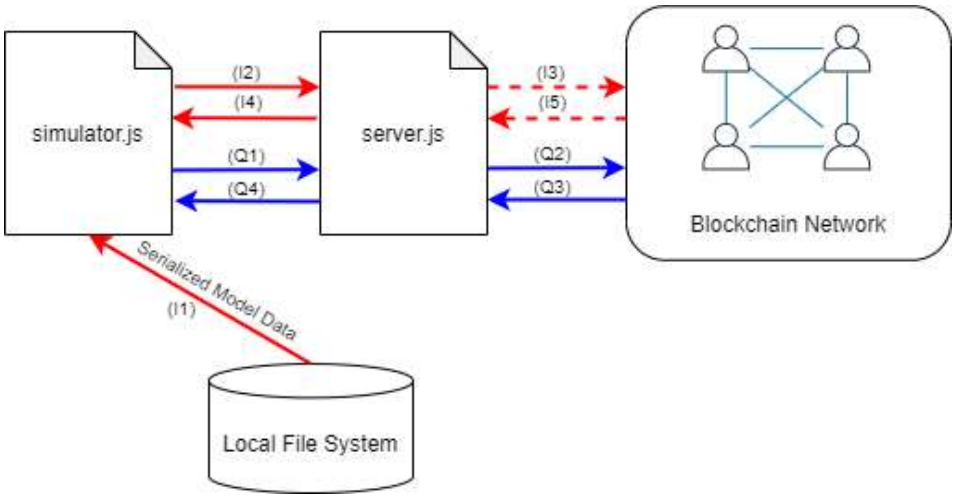
Όταν το αρχείο *simulator.js* τρέξει, πρωτίστως (I1), ετοιμάζει τα δεδομένα προς αποθήκευση. Όπως προαναφέρθηκε στην ενότητα 6.2, υπάρχουν δύο τύποι δεδομένων, τα πραγματικά μοντέλα (του Πίνακα 6.1) και τα *test* «μοντέλα». Εάν επιλεγεί η χρήση των πραγματικών μοντέλων, τα δεδομένα διαβάζονται από το τοπικό σύστημα αρχείων (*Local File System*), αλλιώς τα δεδομένα παράγονται δυναμικά από τον προσομοιωτή με τον τρόπο που ήδη έχει περιγραφεί στο 6.2 (ενότητα).

Στη συνέχεια, δημιουργείται το αίτημα αποθήκευσης και αποστέλλεται στο *endpoint* ***model/submit*** του *server* του συστήματος – *server.js* – (I2), οποίος με τη σειρά του προωθεί ασύγχρονα τα δεδομένα για αποθήκευση (I3) και απαντά στον προσομοιωτή με το κατάλληλο μήνυμα (I4). Σε μεταγενέστερο στάδιο, ο *server* του συστήματος ειδοποιείται για την επιτυχία/αποτυχία της αποθήκευσης των δεδομένων από κάποιο κόμβο του δικτυού *Blockchain* (I5).

6.3.2 Ανάκτηση Δεδομένων

Όσο αφορά την ανάκτηση δεδομένων, σύμφωνα με τα μπλε βέλη του Σχήματος 6.1, η λειτουργία αυτή αποτελείται από τέσσερα βήματα, όλα εκ των οποίων είναι σύγχρονα (*synchronous*).

Αρχικά, ο προσομοιωτής δημιουργεί το αίτημα ανάκτησης και το αποστέλλει στο *endpoint* ***model*** (Q1). Ακολούθως, ο *server* ανακτά τα δεδομένα από το δίκτυο *Blockchain* (Q2, Q3), χρησιμοποιώντας ένα ή περισσότερα επερωτήματα (βλ. υπο-ενότητα 5.4.1). Τέλος, αφού γίνει επαναφορά τους αρχική τους μορφή (Σχήμα 5.1), τα δεδομένα επιστρέφονται στον προσομοιωτή (Q4).



Σχήμα 6.1 : Εισαγωγή (κόκκινα βέλη) και ανάκτηση (μπλε βέλη) πειραματικών δεδομένων.

6.4 Αποτίμηση Αποτελεσμάτων

Για την αποτίμηση της παρούσας δουλειάς πραγματοποιήθηκαν τέσσερα πειράματα. Αυτή η ενότητα αφιερώνεται στην περιγραφή και τον σχολιασμό των αποτελεσμάτων τους.

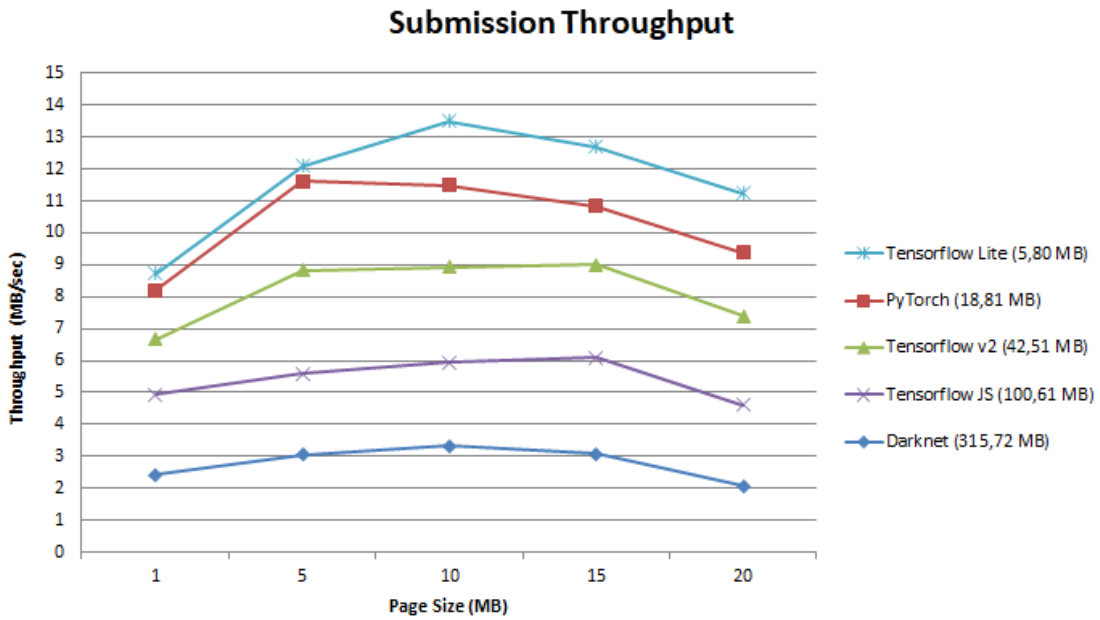
6.4.1 Μέγεθος Σελίδας Δεδομένων

Το πρώτο πείραμα που διεξάχθηκε αποσκοπούσε στον προσδιορισμό του μεγέθους σελίδας δεδομένων (βλ. υπο-κεφάλαιο 4.3.1) που θα χρησιμοποιείται στο σύστημα. Πιο συγκεκριμένα, εξετάστηκε η επίδοση των επιπέδων εφαρμογής και αποθήκευσης με τη χρήση διαφόρων μεγεθών σελίδων. Τα αποτελέσματα του πειράματος συνοψίζονται από τα Σχήματα 6.2 και 6.3, όπου παρουσιάζεται το *throughput* (*MB/sec*) του συστήματος κατά την εισαγωγή/αποθήκευση (*submission*) και ανάκτηση (*retrieval*) των μοντέλων του Πίνακα 6.1 (χρησιμοποιώντας τον προσομοιωτή Πελατών), αντίστοιχα.

Εισαγωγή

Στο σχήμα που ακολουθεί παρατηρούμε πως, αναμενόμενα, τα μεγαλύτερα μοντέλα έχουν μικρότερο (*submission*) *throughput*. Αυτό συμβαίνει γιατί η επεξεργασία τους

χρειάζεται περισσότερο χρόνο, αφού έχουν μεγαλύτερο αριθμό σελίδων (από τα μικρότερα μοντέλα) και, εν συνέπεια, χρειάζονται πιο πολλές δοσοληψίες για να αποθηκευτούν. Επιπλέον, παρατηρούμε τα μικρότερα μεγέθη σελίδων να επιφέρουν υψηλότερο *throughput*, κάτι το οποίο οφείλεται στην μείωση του όγκου των δεδομένων που ανταλλάζονται μεταξύ των *Peers* του δικτύου κατά την εκτέλεση του αλγορίθμου ομοφωνίας τους(, με αποτέλεσμα οι δοσοληψίες να ολοκληρώνονται γρηγορότερα). Σαφώς, όμως, η χρήση πολύ μικρών σελίδων (1 *MB* στο Σχήμα 6.2) επίσης δεν συνίσταται, αφού ο χρόνος επεξεργασίας δοσοληψιών (και όχι ωφέλιμης δουλειάς) – *overhead* – αυξάνεται, μειώνοντας το *throughput*.

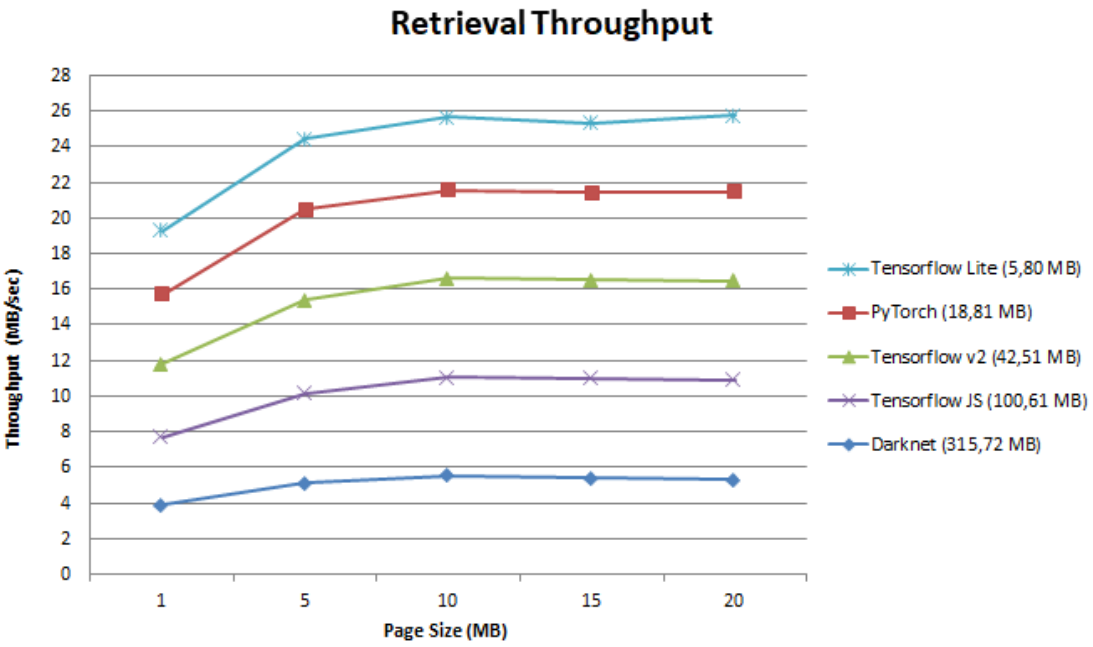


Σχήμα 6.2 : *Submission Throughput* (*Page Size*).

Ανάκτηση

Παρομοίως με την περίπτωση εισαγωγής δεδομένων, στο Σχήμα 6.3 που ακολουθεί παρατηρούμε πως τα μεγαλύτερα μοντέλα έχουν χαμηλότερο (*retrieval*) *throughput* από τα πιο μικρά. Αυτό συμβαίνει, και πάλι, λόγω του μεγαλύτερου αριθμού δοσοληψιών που χρειάζονται για να ανακτηθούν οι πολλαπλές σελίδες τους. Συνεχίζοντας, βλέπουμε πως τα μεγαλύτερα μεγέθη (σελίδων) έχουν τις καλύτερες επιδώσεις, αντιθέτως με πιο πάνω (εισαγωγή), αφού ό αριθμός των σελίδων που

ανακτώνται είναι αισθητά μικρότερος μειώνοντας έτσι το *overhead* επεξεργασίας των δεδομένων (και αυξάνοντας το *throughput*). Αυτό κυρίως οφείλεται στο γεγονός πως κατά την εκτέλεση δοσοληψιών ανάκτησης δεν γίνεται κάποια ανταλλαγή δεδομένων μεταξύ των *Peers* του δικτύου (δεν εκτελείται ο αλγόριθμος ομοφωνίας κατά την ανάγνωση δεδομένων), κάτι το οποίο τονίζεται από το γεγονός πως το *submission throughput* ανάκτησης είναι υψηλότερο από το *retrieval throughput* (σε όλες τις περιπτώσεις/μοντέλα των σχημάτων 6.2 και 6.3).



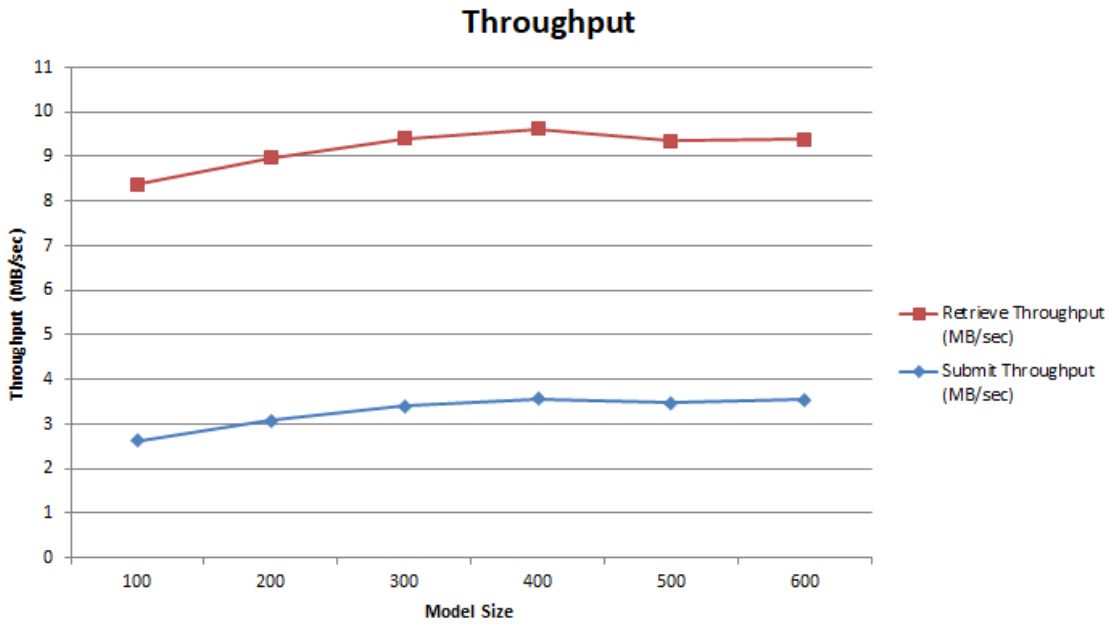
Σχήμα 6.3 : *Retrieval Throughput* (*Page Size*).

Με βάση τα αποτελέσματα που σχολιάστηκαν πιο πάνω, για την διεξαγωγή των υπολοίπων πειραμάτων αξιολόγησης, επιλέχθηκε η χρήση των 10 MB ως το μέγεθος σελίδας δεδομένων, η οποία επιφέρει ικανοποιητικές επιδώσεις τόσο κατά την εισαγωγή, όσο και κατά την ανάκτηση των δεδομένων των μοντέλων.

6.4.2 Μοντέλα Μεγάλου Μεγέθους

Το δεύτερο στη σειρά πείραμα αποσκοπούσε τη μελέτη επίδοσης του συστήματος κατά τη διαχείριση (εισαγωγή και ανάκτηση) μεγάλων μοντέλων. Για την παρούσα αξιολόγηση χρησιμοποιήθηκαν *test* «μοντέλα» (βλ. ενότητα 6.2) και τα 10 MB ως το μέγεθος σελίδας δεδομένων. Η εισαγωγή και ανάκτηση δεδομένων έγινε μέσω του προσομοιωτή Πελάτη, ενώ ως μετρική αξιολόγησης χρησιμοποιήθηκε, ξανά, το *throughput* (*MB/sec*).

Όπως φαίνεται και στο Σχήμα 6.4, τα αποτελέσματα του πειράματος ήταν αρκετά θετικά, αφού βλέπουμε πως το σύστημα μπορεί να χρησιμοποιηθεί και για μεγαλύτερα μεγέθη μοντέλων χωρίς προβλήματα. Επίσης, παρατηρούμε ότι όταν το μέγεθος του μοντέλου ξεπεράσει τα 400 MB η επίδοση (*throughput*) του συστήματος πέφτει τόσο στην εισαγωγή (*submission*), όσο στην ανάκτηση (*retrieval*) δεδομένων. Και στις δύο περιπτώσεις, ωστόσο, το *submission* και *retrieval throughput* σταθεροποιείτε στα 3,5 και 9,4 MB (περίπου), αντίστοιχα.

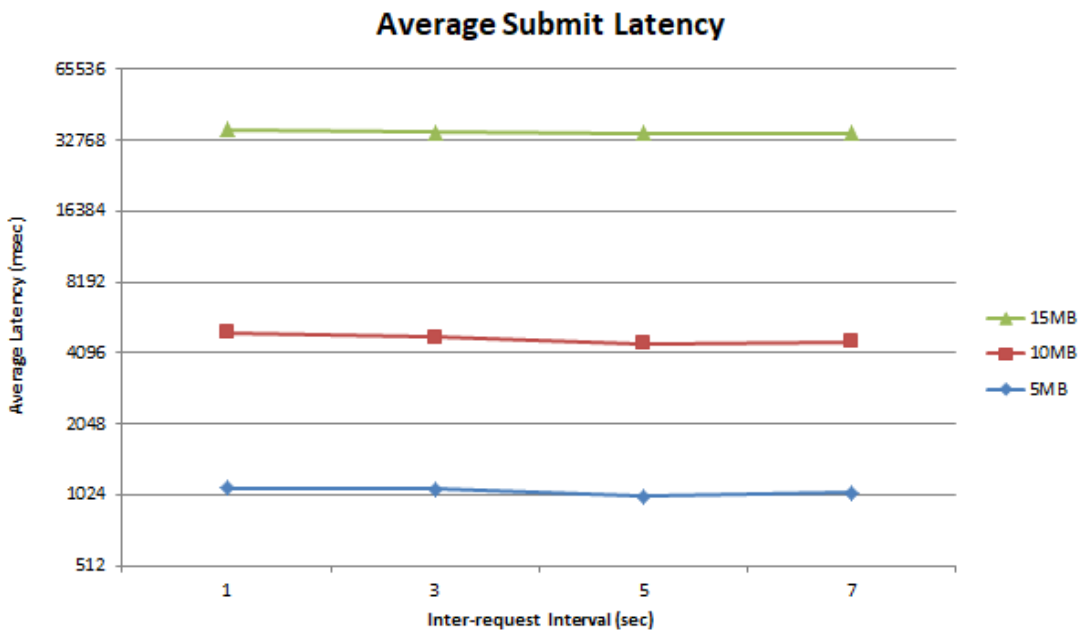


Σχήμα 6.4 : (*Submission and Retrieval*) *Throughput* (*Model Size*).

Τέλος, να αναφέρουμε ότι τα μεγέθη μοντέλων που εξετάστηκαν θεωρούμε πως είναι ικανοποιητικά λόγω του τομέα εφαρμογής του *Triabase* (*IoT, Machine Learning on the Edge*- μικρά μοντέλα λόγω περιορισμού πόρων των έξυπνων συσκευών).

6.4.3 Ρυθμός Αιτημάτων Εισόδου

Το τρίτο πείραμα που διεξάχθηκε αφορά τον ρυθμό άφιξης αιτημάτων εισαγωγής δεδομένων στο σύστημα. Για την παρούσα αξιολόγηση χρησιμοποιήθηκαν *test* «μοντέλα» (βλ. ενότητα 6.2), τα 10 MB ως το μέγεθος σελίδας δεδομένων, και οι εισαγωγές (*submissions*) των μοντέλων έγιναν μέσω του προσομοιωτή Πελάτη. Ως μετρική αξιολόγησης χρησιμοποιήθηκε ο χρόνος ολοκλήρωσης των αιτημάτων – *latency* – (*msec*). Τα αποτελέσματα παρουσιάζονται στο Σχήμα 6.5.



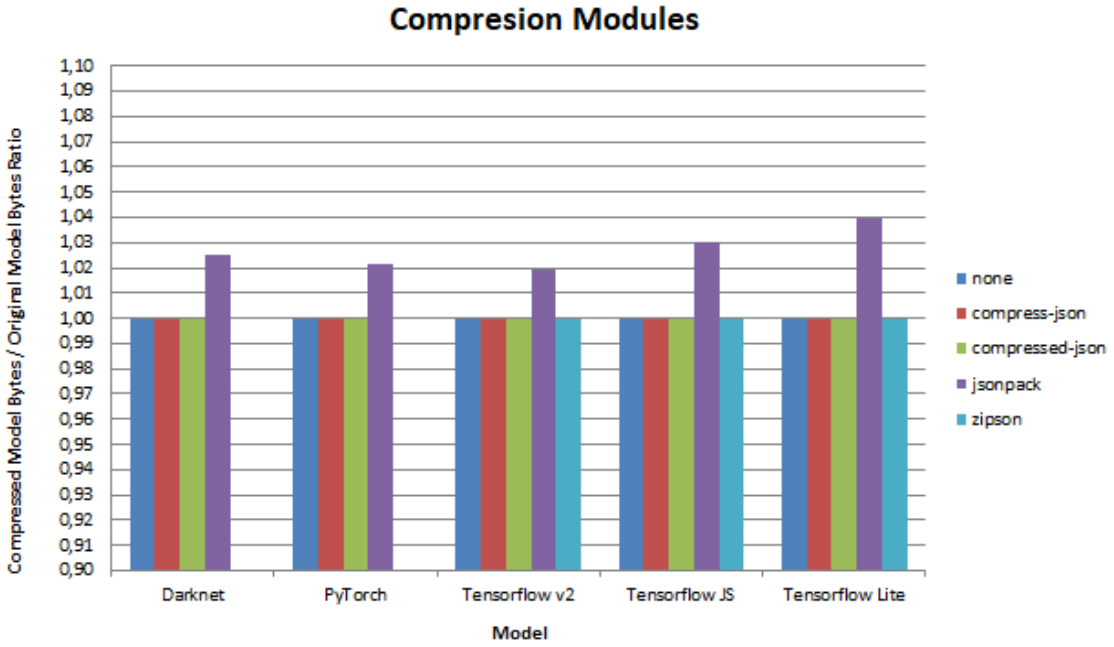
Σχήμα 6.5 : *Average Submit Latency* (*Inter-request Interval*).

Σύμφωνα με το πιο πάνω σχήμα, με πείραμα αυτό αποδεικνύεται πως το σύστημα *Triabase* μπορεί να χρησιμοποιηθεί και σε πιο απαιτητικά περιβάλλοντα, όπου ο χρόνος μεταξύ δύο διαδοχικών εισαγωγών δεδομένων (*Inter-request Interval*) είναι σχετικά μικρός (π.χ., 1 *sec* στο Σχήμα 6.5). Αυτό, ωστόσο, ισχύει για μοντέλα μικρών μεγεθών, αφού η αξιολόγηση μεγαλύτερων μοντέλων στο πείραμα υπό αναφορά δεν ήταν εφικτή. Συγκεκριμένα, η χρήση μοντέλων μεγέθους μεγαλύτερου από 15 MB επέφερε απόρριψη εκτέλεσης των απαραίτητων δοσοληψιών αποθήκευσης δεδομένων από τους *Peers* του δικτύου, ή/και κατάρρευση του δικτύου *Blockchain* (*Docker container failures*). Όπως έχει ήδη αναφερθεί, το σύστημα *Triabase* προορίζεται κυρίως για

μοντέλα μικρών μεγεθών (τάξης μερικών MB), επομένως, θεωρούμε πως η πιο πάνω παρατήρηση δεν αποτελεί σοβαρό πρόβλημα στην παρούσα φάση.

6.4.4 Συμπίεση Δεδομένων

Το τελευταίο πείραμα της σειράς αποσκοπεί στη μελέτη ωφελιμότητας της επιπρόσθετης λειτουργίας συμπίεσης/αποσυμπίεσης που προσφέρει ο *REST server* του επιπέδου εφαρμογής, μέσω αξιολόγησης της επίδοσης των τεσσάρων *Node.js modules* που ενσωματώνονται στο συστατικό διεπαφής. Για το πείραμα αυτό χρησιμοποιήθηκαν τα μοντέλα του Πίνακα 6.1.



Σχήμα 6.6 : *Compression Modules*.

Στο πιο πάνω σχήμα παρουσιάζεται η αναλογία του μεγέθους των συμπιεσμένων μοντέλων ως προς το αρχικό μέγεθος τους (*compressed model bytes / original model bytes*), όπως αυτή προέκυψε κατά τη διαδικασία αξιολόγησης. Όπως βλέπουμε, οι/τα διάφορες βιβλιοθήκες/*modules* συμπίεσης, δεν επιφέρουν μείωση στον όγκο των δεδομένων, αλλά δύναται και να τον αυξήσουν λόγω των μεταδεδομένων που προσθέτουν (όπως στην περίπτωση του *module jsonpack*). Επίσης, να σημειώσουμε ότι

το *module zipson* δεν μπόρεσε να χρησιμοποιηθεί για τη συμπίεση μεγάλων μοντέλων (*Darknet, PyTorch*) λόγω του μεγέθους τους (*compression failure*).

Εν κατακλείδι, μέσω του πειράματος αυτού αποδείχθηκε πως η χρήση της επιπρόσθετης λειτουργίας συμπίεσης/αποσυμπίεσης δεν συνίσταται, αφού τα δεδομένα των μοντέλων ήδη βρίσκονται σε μια αρκετά συμπαγή μορφή και η περεταίρω συμπίεση τους δεν επιφέρει κάποιο όφελος.

Κεφάλαιο 7

Συμπεράσματα & Μελλοντική Δουλειά

7.1 Συμπεράσματα 79

7.2 Μελλοντική δουλειά 82

7.1 Συμπεράσματα

Σκοπός αυτής της διπλωματικής εργασίας ήταν η μελέτη της τεχνολογίας αλυσίδων συστοιχιών (*Blockchain*), και, μετέπειτα, η υλοποίηση μέρους της πρώτης έκδοσης του συστήματος *Triabase*, το οποίο επιδιώκει την αποδοτική και ασφαλή διαχείριση *IoT* δεδομένων μέσω της χρήσης της τεχνολογίας αυτής (*Blockchain*). Αφού, αρχικά, μελετήθηκε ο τρόπος και οι ιδιαιτερότητες λειτουργίας των τεχνολογιών *Blockchain*, ακολούθως ερευνήθηκαν οι υφιστάμενες τεχνικές διαχείρισης των δεδομένων σε *IoT* συστήματα, όπως τη χρήση της *InfluxDB*, αλλά και τα κλασσικά *IoT Pipelines*. Επιπλέον, μελετήθηκαν αλγόριθμοι αποθήκευσης και αποσύνθεσης (μεγάλων) δεδομένων που αξιοποιούν μοντέλα μηχανικής μάθησης *LSTM*, όπως και η τεχνική ομοσπονδιακής μάθησης (*Federated Learning*), η οποία επίσης αξιοποιείτε από το σύστημα *Triabase*.

Οι στόχοι που τέθηκαν επιτεύχθηκαν σε μεγάλο βαθμό, αφού η απόπειρα υλοποίησης μιας πρώτης έκδοσης του συστήματος *Triabase* ήταν επιτυχής. Συγκεκριμένα, η συνεισφορά της παρούσας δουλειάς εντοπίζεται στα επίπεδα αποθήκευσης και εφαρμογής (*Storage* και *Application*), όπως αυτά συνοπτικά περιγράφονται πιο κάτω, με την υλοποίηση των πρωτοτύπων των συστατικών αποθήκευσης και διεπαφής, αντίστοιχα.

Συνεχίζοντας, να αναφέρουμε πως τη βασική δομή του συστήματος (*Triabase*) αποτελούν το επίπεδο άκρων (*Edge Layer*), το επίπεδο εφαρμογής (*Application Layer*) και το επίπεδο αποθήκευσης (*Storage Layer*). Στο *Edge Layer* βρίσκονται οι διάφορες *IoT* συσκευές και ο *Model Aggregator* του συστήματος, οι οποίοι συμμετέχουν στην διαδικασία ομοσπονδιακής μάθησης που λαμβάνει χώρα στο σύστημα *Triabase*. Ειδικότερα, οι συσκευές εκπαιδεύουν τοπικά μοντέλα μηχανικής μάθησης, ενώ ο *Model Aggregator* αναλαμβάνει τη δημιουργία του καθολικού μοντέλου (μέσω συσσωμάτωσης των πολλαπλών τοπικών). Στο *Application Layer* υλοποιούνται μηχανισμοί που επιτρέπουν την υψηλού επιπέδου (*high-level*) επικοινωνία με το *Storage Layer*, αποκρύπτοντας την πολυπλοκότητα του. Με το *Application Layer* μπορούν να επικοινωνήσουν τόσο οι έξυπνες συσκευές του *IoT* δικτύου, όσο και ο *Model Aggregator* και οι διαχειριστές του συστήματος *Triabase*, για την αποθήκευση (και διαχείριση) των δεδομένων (μοντέλων) τους στο *Storage Layer*. Το *Storage Layer* του συστήματος αποτελείτε από ένα *Permissioned Blockchain Network* το οποίο φροντίζει για τη διαφανή, ασφαλή και αποτελεσματική (αποθήκευση και) διαχείριση των δεδομένων του συστήματος στην συστοιχία κόμβων που το αποτελούν. Στο επίπεδο αυτό δύναται να υπάρχουν *Smart Contracts* και ευρετήρια (*indexes*) που υλοποιούν διάφορες βασικές λειτουργίες που προσφέρουν οι παραδοσιακές βάσεις δεδομένων. Η όποια επικοινωνία με το *Storage Layer* γίνεται μέσω του *Application Layer*.

Για την υλοποίηση των τριών επιπέδων του συστήματος *Triabase* χρησιμοποιήθηκαν διάφορες τεχνολογίες και γλώσσες προγραμματισμού. Αρχικά, για τη μηχανική μάθηση που λαμβάνει χώρα στο *Edge Layer* αξιοποιείται η πλατφόρμα *TensorFlow*, στη γλώσσα *Python*. Επιπλέον, η υλοποίηση των απαραίτητων *endpoints* του *RESTful API* που προσφέρει το *Application Layer* έγινε με τη χρήση του *Node.js runtime environment* και τη γλώσσα *JavaScript*. Για το *Blockchain* δίκτυο του *Storage Layer*, έχουν δημιουργηθεί διάφορα *shell scripts* και αρχεία τύπου *Docker Compose* που αυτοματοποιούν τη δημιουργία του, στα πλαίσια της διπλωματικής εργασίας [4]. Όσο αφορά τα *Smart Contracts* που χρησιμοποιούνται, αυτά έχουν γραφτεί στη γλώσσα *Golang*. Τέλος, να αναφέρουμε ότι για το πρώιμο/πιλοτικό γραφικό περιβάλλον αλληλεπίδρασης με το *Storage Layer* που υλοποιήθηκε αξιοποιούνται βιβλιοθήκες του εργαλείου *Swagger UI* και αρχεία τύπου *YAML*.

Τα αποτελέσματα της πειραματικής αξιολόγησης της παρούσας δουλειάς ήταν αρκετά θετικά. Εμβαθύνοντας, κρίνοντας το μέγεθος σελίδας δεδομένων των 10 MB ως το πιο αποτελεσματικό, και χρησιμοποιώντας το στη μελέτη των δυνατοτήτων/ορίων του συστήματος, αποδείχθηκε πως δεν υπάρχει κάποιος περιορισμός στο μέγεθος μοντέλου που μπορεί να διαχειριστεί το *Triabase*. Φάνηκε, επίσης, πως το σύστημα μπορεί να χρησιμοποιηθεί σε περιβάλλοντα υψηλού ρυθμού αιτημάτων εισαγωγής δεδομένων (*high submission request arrival rate*, ή, αντίστοιχα, *low inter-request interval*), δεδομένου ότι τα μοντέλα που θα πραγματεύεται δεν θα ξεπερνούν τα 15 MB σε μέγεθος. Ακόμα, διαπιστώθηκε πως τα δεδομένα των μοντέλων μηχανικής μάθησης είναι ήδη αρκετά συμπαγής, και, ως εκ τούτου, η όποια επιπλέον συμπίεση τους είναι περιττή.

Κλείνοντας, οι μεγαλύτερες προκλήσεις που παρουσιάστηκαν κατά τη διάρκεια εκπόνησης της παρούσας διπλωματικής εργασίας ήταν η υλοποίηση των *endpoints* (*REST server*) που τροποποιούν τα δεδομένα του *Blockchain* δικτύου και η εφαρμογή της τεχνικής της σελιδοποίησης δεδομένων στο σύστημα. Αναλυτικότερα, η σύγχρονη (*synchronous*) επεξεργασία των δεδομένων του κατάστιχου δεν ήταν εφικτή, λόγω των περιορισμένων μεθόδων του *Node.js SDK* που προσφέρει το *Hyperledger Fabric*. Αυτό μας προκάλεσε ιδιαίτερο προβληματισμό, αφού οι Πελάτες του συστήματος *Triabase* δεν μπορούσαν να λάβουν μια ξεκάθαρη απάντηση για την κατάσταση των αιτημάτων τους από τον server του *Application Layer* (π.χ., το *status code*=200 δεν αποτελεί ορθή απάντηση στον χρήστη όταν η σειρά των δοσοληψιών που εκδόθηκαν είναι ασύγχρονη…). Για την επίλυση του προαναφερθέντος ζητήματος υλοποιήθηκαν τα *.../check endpoints* ελέγχου προόδου, όπως αυτά περιγράφηκαν στις υπο-ενότητες 5.3.3 και 5.4. Όσο αφορά την τεχνική σελιδοποίησης, η δυσκολία παρουσιάστηκε στην εύρεση ενός τρόπου αποτελεσματικής ενημέρωσης των σελίδων των δεδομένων ενός μοντέλου, αφού η κάθε σελίδα αποτελεί ξεχωριστή εγγραφή/οντότητα του κατάστιχου του συστήματος. Συγκεκριμένα, δεδομένου των μεγάλων καθυστερήσεων και της μη-ατομικής (*non-atomic*) εκτέλεσης των δοσοληψιών τροποποίησης δεδομένων που χαρακτηρίζει την τεχνολογία Blockchain, η τροποποίηση δεδομένων έπρεπε να γίνεται με τον ελάχιστο δυνατό αριθμό δοσοληψιών για σκοπούς μείωσης του κινδύνου αποτυχίας της. Για την αντιμετώπιση του ζητήματος αυτού, χρησιμοποιήθηκαν δύο προσεγγίσεις- η ομαδοποίηση πολλαπλών τροποποιήσεων σε μια δοσοληψία (εφαρμογή στη διαγραφή/συντήρηση δεδομένων), και η απλή επεγγραφή (*overwrite*)

σελίδων με κατάλληλη ενημέρωση των σχετικών μεταδεδομένων (εφαρμογή στην ενημέρωση δεδομένων). Η χρήση της ενημέρωσης με επεγγραφή ήταν αναγκαία, αφού η επισύναψη ενός μονολιθικού μεγάλου όγκου δεδομένων σε μια δοσοληψία (στην περίπτωση υιοθέτησης της ομαδοποίησης) είναι ακριβώς αυτό που προσπαθούμε να αποφύγουμε με τη σελιδοποίηση (βλ. τελευταία παράγραφο υπο-ενότητας 4.2). Ωστόσο, το πρόβλημα της μη-ατομικότητας ακόμα δεν έχει επιλυθεί.

7.2 Μελλοντική δουλειά

Αδιαμφισβήτητα, με την συνεισφορά και της παρούσας διπλωματικής εργασίας, η πρώτη έκδοση του συστήματος έχει αποδείξει ότι (το *Triabase*) αποτελεί μια πολύ υποσχόμενη πρόταση αξιοποίησης της τεχνολογίας *Blockchain* για την διαχείριση δεδομένων *IoT* εφαρμογών. Εντούτοις, όπως ισχύει για την όποια ερευνητική δουλειά, υπάρχουν αρκετά περιθώρια βελτίωσης– στην ενότητα αυτή παρουσιάζονται μερικά εξ' αυτών.

Πρώτο περιθώριο βελτίωσης αποτελεί η αυτοματοποίηση της λειτουργίας συντήρησης μοντέλων (*Clean-Up*), που μέχρι στιγμής προσφέρεται μόνο για χειροκίνητη (*manual*) χρήση/ενεργοποίηση στο το χρήστη. Για την επίτευξη αυτού υπάρχει η επιλογή υλοποίησης κάποιου βοηθητικού μηχανισμού τύπου *cron job* (όπως στα συστήματα *Unix*). Ο μηχανισμός αυτός θα μπορούσε να υλοποιηθεί σε οποιοδήποτε από τα επίπεδα του συστήματος, αν και η ενσωμάτωση του στο *Smart Contract* του επιπέδου αποθήκευσης θεωρούμε πως θα είναι η αποδοτικότερη (μικρότερες καθυστερήσεις λόγω μη-επικοινωνίας επιπέδων).

Περεταίρω επεξεργασία χρειάζεται, επίσης, η τεχνική της χρήσης των token στο σύστημα. Ειδικότερα, όπως ήδη έχει αναφερθεί (βλ. ενότητα 3.1.2), στην παρούσα έκδοση του συστήματος δεν υπάρχει κάποιος εύκολος τρόπος απόκτησης *tokens* από τους (εξουσιοδοτημένους) χρήστες του συστήματος. Το κενό αυτό σαφώς και πρέπει να καλυφθεί σε μετέπειτα εκδόσεις.

Συνεχίζοντας, κάποιες άλλες λειτουργίες που θα μπορούσαν να ενσωματωθούν στο σύστημα είναι η χρήση *delta updates* των δεδομένων του, η υποστήριξη συνύπαρξης πολλαπλών εκδόσεων ενός μοντέλου στο σύστημα (*model versioning*), όπως και η

ενσωμάτωση της τεχνικής της τυχαίας δειγματοληψίας στα *endpoints* που ανακτούν περισσότερες από μια εγγραφές μεταδεδομένων (έτσι ώστε να μην επιστρέφονται πάντα τα ίδια αποτελέσματα).

Κλείνοντας, να αναφέρουμε πως, έως ότου το *Triabase* αποκτήσει τις δυνατότητες των παραδοσιακών βάσεων δεδομένων, πρέπει να αντιμετωπιστεί το πρόβλημα του μη-ατομικού (*Non-Atomic* / *Non-Transactional*) χαρακτήρα των λειτουργιών που προσφέρει. Μια πρώτη προσέγγιση που θα μπορούσε να διερευνηθεί είναι η χρήση πιο περίπλοκης αλγοριθμικής λογικής. Ως δεύτερη προσέγγιση προς διερεύνηση, έχουμε την αλλαγή του *Blockchain Network development framework* που χρησιμοποιείται στο επίπεδο αποθήκευσης, με σκοπό την εύρεση πιο αρμόζων τρόπων αλληλεπίδρασης με το δίκτυο *Blockchain* του επιπέδου αυτού. Συγκεκριμένα, υπάρχει η επιλογή αντικατάστασης του *Hyperledger Fabric framework* με κάποιο άλλο υπάρχον *framework*, ενώ δυνατή είναι και η δημιουργία ενός ειδικά διαμορφωμένου *Blockchain Network* από το μηδέν. Όπως ξεκάθαρα φαίνεται, την πιο ευνοϊκή προσέγγιση αποτελεί η δημιουργία ενός νέου *Blockchain Network*, αφού η υποσχόμενη ευελιξία του προσφέρει το καταλληλότερο περιβάλλον ανάπτυξης επιπλέον μηχανισμών του τομέα βάσεων δεδομένων στο σύστημα. Στους μηχανισμούς αυτούς εντάσσονται, μεταξύ άλλων, η ευρετηρίαση (*Indexing*) και η συνέπεια (*Consistency*) δεδομένων, όπως και η ανάκαμψη σε περιπτώσεις αποτυχίας (*Data Recovery*).

Βιβλιογραφία

[1] C. Costa and D. Zeinalipour-Yazti, “Telco big data research and open problems,” in *Proceedings of the 35th IEEE International Conference on Data Engineering*, ser. ICDE‘19. 8-12 April 2019, Macau SAR, China: IEEE Computer Society, 2019, conference, pp. 2056–2059.

[2] C. Costa, G. Chatzimilioudis, D. Zeinalipour-Yazti, and M. F. Mokbel, “Efficient Exploration of Telco Big Data with Compression and Decaying,” in *2017 IEEE 33rd International Conference on Data Engineering* (ICDE), Apr. 2017, pp. 1332–1343, 2375-026X.

[3] P. Drakatos, E. Demetriou, S. Koumou, A. Konstantinidis, and D. Zeinalipour-Yazti**,** “Triastore: A Web 3.0 Blockchain Datastore for Massive IoT Workloads”, The 22nd IEEE International Conference on Mobile Data Management (MDM '21), IEEE Press, pp. 6 pages, June 15 - June 18, 2021, Toronto, Canada, 2021.

[4] E. Demetriou, "*Υλοποίηση και Αποτίμηση μιας Blockchain Βάσης για IoT Δεδομένα*," University of Cyprus, 2021.

[5] J. Calusinski, "Build the foundation of an *IoT* app with Node-RED and Raspberry Pi," IBM, [Online]. Available: https://www.ibm.com/*cloud*/architecture/tutorials/raspberry-pi-*IoT*/. [Accessed 2020].

[6] "Node-Red," IBM, [Online]. Available: https://nodered.org/. [Accessed 2020].

[7] "Influxdata – InfluxDB concepts," [Online]. Available: https://docs.influxdata.com/influxdb/v1.8/concepts/. [Accessed 2021].

[8] D. Zeinalipour, "Lecture 10," in *EPL646 – Advanced Topics in Databases*, 2020.

[9] B. McMahan and D. Ramage, "*Federated Learning*: Collaborative *Machine learning* without Centralized Training Data," Google, 6 4 2017. [Online]. Available: https://ai.googleblog.com/2017/04/federated-learning-collaborative.html. [Accessed 2020].

[10] "*Federated Learning*," Google, [Online]. Available: https://federated.withgoogle.com/. [Accessed 2020].

[11] "*Bitcoin* Home Page," [Online]. Available: https://Bitcoin.org/el/. [Accessed 2020].

[12] S. Nakamoto, "*Bitcoin: A Peer-to-Peer Electronic Cash System*," 2008. [Online]. Available: https://Bitcoin.org/Bitcoin.pdf. [Accessed 2020].

[13] Z. Zheng, S. Xie, H. Dai, X. Chen, and H. Wang, "*An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends*," IEEE 6th International Congress on Big Data, 2017, 10.1109/BigDataCongress.2017.85.

[14] "*Consensus* algorithms," [Online]. Available: https://www.geeksforgeeks.org/consensus-algorithms-in-Blockchain/. [Accessed 2020].

[15] "Public and Private *Blockchain*s," [Online]. Available: https://101*Blockchain*s.com/public-vs-private-*Blockchain*/. [Accessed 2020].

[16] C. Mohan, "Tutorial: *Blockchain*s and Databases," *VLDB Endowment,* vol. 10, no. 12, p. 2000–2001, 2017.

[17] D. Massessi, "Public Vs Private *Blockchain* In A Nutshell," 12 12 2018. [Online]. Available: https://medium.com/coinmonks/public-vs-private-*Blockchain*-in-a-nutshell-c9fe284fa39f. [Accessed 2020].

[18] J. Sedlmeir, H. U. Buhl, G. Fridgen and R. Keller, "The Energy Consumption of *Blockchain* Technology: Beyond Myth," *Business & Information Systems Engineering,* vol. 62, no. 6, pp. 599-608, 2020.

[19] "Ethereum," [Online]. Available: https://ethereum.org/en/. [Accessed 2020].

[20] C. Costa, A. Charalambous, A. Konstantinides, D. Zeinalipour-Yazti and M. F. Mokbel, "Decaying *Telco* Big Data with Data Postdiction," in *Proceedings of the 19th IEEE International Conference on Mobile Data Management*, Aalborg, Denmark, 2018.

[21] J. Brownlee, "Machine Learning Mastery – What Is *Time Series Forecasting*?," [Online]. Available: https://machinelearningmastery.com/time-series-forecasting/. [Accessed 2021].

[22] J. Brownlee, "Machine Learning Mastery – How to Develop *LSTM Models for Time Series Forecasting*?," [Online]. Available: https://machinelearningmastery.com/how-to-develop-lstm-models-for-time-series-forecasting/. [Accessed 2021].

[23] M. J. Amiri, D. Agrawal and A. . E. Abbadi, "CAPER: a cross-application permissioned *Blockchain*," *VLDB Endowment,* vol. 12, no. 11, p. 1385–1398, 2019.

[24] M. El-Hindi, C. Binnig, A. Arasu, D. Kossmann, R. Ramamurthy, "BlockchainDB – A Shared Database on Blockchains", in *Proceedings of the VLDB Endowment,* 2019, Los Angeles, California, 2019.

[25] F. Nawab, D. J. Abadi, O. Arden and M. Shadmon, "AnyLog: a Grand Unification of the Internet of Things," in *9th Biennial Conference on Innovative Data Systems Research*, California, USA, 2019.

[26] P. Drakatos, E. Demetriou, S. Koumou, A. Konstantinidis and D. Zeinalipour-Yazti, "Towards a Blockchain Database f or Massive IoT Workloads", in Proceedings of the 3rd Intl. Workshop on Blockchain and Data Management, IEEE Press, April 19, 2021, Chania, Crete, Greece, 2021.

[27] M. Belk, "Ανασκόπηση Διαδικτύου", in EPL425 Topic 2 – Lecture Slides, 2021.

[28] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan and C. Murthy, "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains," in Proceedings of the Thirteenth EuroSys Conference, Porto, Portugal, 2018.

[29] "Hyperledger Fabric - A Blockchain Platform for the Enterprise," [Online]. Available: https://Hyperledger-fabric.readthedocs.io/en/latest/. [Accessed 2021].

[30] "Speechmatics – Machine learning is getting BIG (Part I)," [Online]. Available: https://www.speechmatics.com/blog/machine-learning-is-getting-big-part-i/. [Accessed 2021].

[31] "Hyperledger Topic Lists – Transaction payload size : standard limit for any application," [Online]. Available: https://lists.hyperledger.org/g/fabric/topic/22392464. [Accessed 2021].

[32] "Wikipedia – LevelDB," [Online]. Available: https://en.wikipedia.org/wiki/LevelDB. [Accessed 2021].

[33] "CouchDB – adhoc queries," [Online]. Available: https://docs.couchdb.org/en/latest/api/database/find.html. [Accessed 2021].

[34] "Hyperledger Fabric – Hyperledger Fabric SDKs," [Online]. Available: https://hyperledger-fabric.readthedocs.io/en/release-2.2/fabric-sdks.html. [Accessed 2021].

[35] "Fabric-SDK-Py Documentation," [Online]. Available: https://fabric-sdk-py.readthedocs.io/en/latest/. [Accessed 2021].

[36] "INTEXSOFT – Node.js vs Java: Why Compare?," [Online]. Available: https://www.intexsoft.com/blog/post/java-vs-nodejs.html. [Accessed 2021].

[37] "npm – compress-json," [Online]. Available: https://www.npmjs.com/package/compress-json. [Accessed 2021].

[38] "npm – compressed-json," [Online]. Available: https://www.npmjs.com/package/compressed-json. [Accessed 2021].

[39] "npm – jsonpack," [Online]. Available: https://www.npmjs.com/package/jsonpack. [Accessed 2021].

[40] "npm – zipson," [Online]. Available: https://www.npmjs.com/package/zipson. [Accessed 2021].

[41] "IoT For All – What is Computer Vision? Evolution and Applications in IoT," [Online]. Available: https://www.iotforall.com/computer-vision-iot. [Accessed 2021].

[42] "COCO Dataset," [Online]. Available: https://cocodataset.org/#home. [Accessed 2021].

[43] J. Redmon, S. Divvala, R. Girshick and A. Farhadi, "You Only Look Once: Unified, Real-Time Object Detection," *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2016, Las Vegas, Nevada, USA.

[44] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov and L. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks," *IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018, Salt Lake City, Utah, USA.

[45] T. Mingxing, and L. Quoc V., "EfficientNet: Rethinking Model Scaling for Convolutional Neural Networks," in *Proceedings of the 36th International Conference on Machine Learning Research*, 2019, California, USA.

Παράρτημα Α

Στο παράρτημα αυτό δίνονται τα βήματα εγκατάστασης και έγερσης του δικτύου *Blockchain* και του *REST server* των επιπέδων αποθήκευσης και διεπαφής, αντίστοιχα.

Οι ακόλουθες οδηγίες είναι κατάλληλες για *Linux* μηχανές με λειτουργικό σύστημα *Ubuntu 18.04.04 LTS*.

1- ***Εγκατάσταση προαπαιτούμενων λειτουργίας Hyperledger Fabric***

Εγκατάσταση Git

```
$ sudo apt install git
```

Εγκατάσταση cURL

```
$ sudo apt install curl
```

Εγκατάσταση wget

```
$ sudo apt install wget
```

Εγκατάσταση Docker

```
$ sudo apt install docker.io
$ sudo systemctl start docker
$ sudo systemctl enable docker
```

Εγκατάσταση Docker Compose

```
$ sudo curl -L
"https://github.com/docker/compose/releases/download/1.26.2/dock
er-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-
compose
$ sudo chmod +x /usr/local/bin/docker-compose
```

Εγκατάσταση Golang

```
$ wget https://dl.google.com/go/go1.13.8.linux-amd64.tar.gz
$ sudo tar -xvf go1.13.8.linux-amd64.tar.gz
$ sudo mv go /usr/local.
```

Οι πιο κάτω εντολές πρέπει να προστεθούν στο αντίστοιχο αρχείο *profile* του χρήστη (π.χ., το *.bashrc*).

```
export GOROOT=/usr/local/go
export GOPATH=$HOME/go
export PATH=$GOPATH/bin:$GOROOT/bin:$PATH
```

Μετά τρέχουμε την εντολή `$source` για να ενημερωθεί το *terminal*.

Εγκατάσταση NodeJS

```
$ curl -o- https://raw.githubusercontent.com/nvm-
sh/nvm/v0.35.3/install.sh | bash
$ nvm install 10.23.0
$ nvm install 12.19.0
```

Μετά τρέχουμε την εντολή `$nvm use 10.23.0` για να ενεργοποιήσουμε την συγκεκριμένη έκδοση του nvm.

Εγκατάσταση των *Hyperledger Fabric* binaries και Docker Images

```
$ curl -sSL https://bit.ly/2ysbOFE | bash -s
```

Οι πιο κάτω εντολή πρέπει να προστεθεί στο αντίστοιχο αρχείο *profile* του χρήστη (π.χ., το *.bashrc*).

```
export PATH=<path to download location>/fabric-samples/bin:$PATH
```

Μετά τρέχουμε την εντολή `$source` για να ενημερωθεί το *terminal*.

2- ***Έγερση του Blockchain δικτύου***

Αρχικά, κατεβάζουμε το σχετικό *GitHub repository*, που περιέχει τα s*hell scripts* και *Configuration files* του δικτύου και του *REST server* του συστήματος.

```
$ git clone https://github.com/skoumo01/Model-Meta-Database.git
```

Στη συνέχεια, πλοηγούμαστε στο φάκελο *Model-Meta-Database* που έχει δημιουργηθεί, και προχωρούμε στον φάκελο *./network*. Εκεί εκτελούμε τις ακόλουθες εντολές, οι οποίες δύναται να διαρκέσουν μερικά λεπτά.

```
$ ./network.sh up -c mychannel -db couchdb
$ ./network.sh deployCC -c mychannel -ccn contract_models -ccv 1
-ccp ../chaincode/src/models/ -ccl golang
```

Σε αυτό το σημείο το δίκτυο *Blockchain* του συστήματος *Triabase* είναι έτοιμο για χρήση. Συγκεκριμένα, το σχετικό *Smart Contract* που περιγράφηκε στο κεφάλαιο 4 έχει γίνει εγκατασταθεί στους τέσσερεις *Peers* του *Hyperledger Fabric test Blockchain Network* που έχει στηθεί. Το *World State database* που χρησιμοποιείται είναι η *CouchDB*.

Για να σιγουρευτούμε ότι το δίκτυο έχει εγερθεί με επιτυχία, μπορούμε να τρέξουμε την ακόλουθη εντολή, η οποία εκτυπώνει στην οθόνη τα ενεργά *Docker containers*. Εάν η διαδικασία ήταν επιτυχής, πρέπει να εκτυπώνονται πληροφορίες για συνολικά δεκατρία *Docker containers*.

```
$ docker ps
```

Για να τερματίσουμε την λειτουργία του δικτύου τρέχουμε την εντολή που φαίνεται πιο κάτω.

```
$ network.sh down
```

3- *Έγερση REST server*

Αφού πλοηγηθούμε στον υποφάκελο *./app* του *GitHub repository* που ήδη έχει ανακτηθεί, εκτελούμε τις ακόλουθες εντολές, οι οποίες εγκαθιστούν τις βιβλιοθήκες που χρειάζεται ο *REST server*, του δίνουν εξουσιοδότηση για να επικοινωνεί με το *Blockchain* δίκτυο, και τον ενεργοποιούν/τρέχουν.

```
$ nvm use 12.19.0
$ npm install
$ rm -rf credstore/
$ node cred-store.js org1 Admin
$ node --max-old-space-size=4096 server.js none 10 false true 15
```

Σε αυτό το σημείο ο *REST server* του συστήματος *Triabase* είναι έτοιμος για χρήση. Ειδικότερα, η τελευταία εντολή, που τρέχει το αρχείο του *server* (**`server.js`**), του εξασφαλίζει 4096 *bytes* – 4 GB – *heap space* (**`--max-old-space-size=4096`**), επιλέγει τη βιβλιοθήκη/*module* συμπίεσης δεδομένων (**`none`** / κανένα), θέτει το μέγεθος σελίδας δεδομένων στα 10 MB (**`10`**), και απενεργοποιεί αρκετά *debugging console logs* (**`false`**). Οι τελευταίες δύο παράμετροι που φαίνονται χρησιμοποιήθηκαν στα πλαίσια της πειραματικής πειραματικής αξιολόγησης του συστήματος (βλ. υπο-ενότητα 6.4.3). Στη συγκεκριμένη περίπτωση, η πιο πάνω εντολή ενεργοποιεί (**`true`**) την εκτύπωση του *submission latency* των πρώτων N *submission requests* που αναλαμβάνει ο *server*, όπου N=15 (**`15`**).

Για την απενεργοποίηση και ενεργοποίηση των διαφόρων λειτουργιών που προαναφέρθηκαν χρησιμοποιούμε "*false*" και "*true*", αντίστοιχα. Όσο αφορά την

επιλογή βιβλιοθήκης συμπίεσης, οι έγκυροι παράμετροι επιλογής των αντίστοιχων *Node.js modules* παραθέτονται πιο κάτω:

- **`none`** (δεν χρησιμοποιείται συμπίεση)
- **`compress-json`**
- **`compressed-json`**
- **`jsonpack`**
- **`zipson`**

4- *Swagger UI*

Οδηγίες για την έγερση της διεπιφάνειας χρήστη που φαίνεται στο υπο-κεφάλαιο 5.4.3, υπάρχουν στη σελίδα του *Swagger UI*. Ειδικότερα, τα βήματα που πρέπει να ακολουθηθούν περιγράφονται λεπτομερώς στο URL https://swagger.io/docs/open-source-tools/swagger-ui/development/setting-up/, και το σχετικό αρχείο YAML (*swagger_doc.yaml*) βρίσκεται στον υποφάκελο *./app* του *GitHub repository* που έχει ανακτηθεί.

Παράρτημα Β

Στο παράρτημα αυτό παραθέτεται ο ολοκληρωμένος κώδικας του Smart Contract και του προσομοιωτή Πελάτη που γράφτηκαν στα πλαίσια της παρούσας διπλωματικής εργασίας.

Smart Contract (Συστατικό Αποθήκευσης – models.go)

```
package main

import (
    "encoding/json"
    "fmt"
    "strconv"

    "github.com/hyperledger/fabric-chaincode-go/shim"
    "github.com/hyperledger/fabric-contract-api-go/contractapi"
)

// SimpleChaincode implements the fabric-contract-api-go programming model
type SimpleChaincode struct {
    contractapi.Contract
}

type MetaData struct {
    ModelID               string `json:"model_id"`
    Tag1                  string `json:"tag1"`
    Tag2                  string `json:"tag2"`
    SerializationEncoding string `json:"serialization_encoding"`
    PageNumber            int    `json:"page_number"`
}

type GenericPage struct {
    ModelID   string `json:"model_id"`
    PageID    int    `json:"page_id"`
    PageBytes int    `json:"page_bytes"` //data+digest
    Data      string `json:"data"`
    Digest    string `json:"digest"`
}

type GenericPageWrapper struct {
    Key   string      `json: key`
    Value GenericPage `json: value`
}

// PaginatedQueryResult structure used for returning paginated query results and
// metadata
type PaginatedQueryResult struct {
    Records             []*GenericPageWrapper `json:"records"`
    FetchedRecordsCount int32                 `json:"fetchedRecordsCount"`
    Bookmark            string                `json:"bookmark"`
}

type MetaPaginatedQueryResult struct {
    Records             []*MetaData `json:"records"`
    FetchedRecordsCount int32       `json:"fetchedRecordsCount"`
    Bookmark            string      `json:"bookmark"`
}

////////////////////////////////////////////////////////////////////////////////
```

```go
// Creates a new client token using uuid
func (t *SimpleChaincode) CreateClientToken(ctx contractapi.TransactionContextInterface,
    token string) (string, error) {

    err := ctx.GetStub().PutState(token, []byte{'t', 'o', 'k', 'e', 'n'})
    if err != nil {
        return "", err
    }

    return token, nil
}

// Checks if a client token exists
func (t *SimpleChaincode) CheckClientToken(ctx contractapi.TransactionContextInterface,
    token string) (bool, error) {

    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return false, fmt.Errorf("failed to retrieve token %s from world state. %v",
                                                                        token, err)
    }

    return tokenBytes != nil, nil
}

////////////////////////////////////////////////////////////////////////////////////////

// Submits a metadata entry to the ledger.
// Metadata entries have to do with the model's page metadata.
func (t *SimpleChaincode) SubmitMeta(ctx contractapi.TransactionContextInterface,
    token, modelID, entryString string) error {

    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return fmt.Errorf("failed to retrieve token %s from world state. %v", token, err)
    }

    if tokenBytes == nil {
        return fmt.Errorf("authorization not granded: token %s is invalid", token)
    }

    var entry MetaData
    err = json.Unmarshal([]byte(entryString), &entry)
    if err != nil {
        return fmt.Errorf("failed to process json data; ensure correct structure")
    }

    entryBytes, err := json.Marshal(entry)
    if err != nil {
        return err
    }

    err = ctx.GetStub().PutState(modelID+"_metadata", entryBytes)
    if err != nil {
        return err
    }

    return nil
}

// Submits a page-data entry to the ledger.
// Page-data entries have to do with the model's actual data.
// PpageType can be either of the following: model, weights, initialization, checkpoints
func (t *SimpleChaincode) SubmitPage(ctx contractapi.TransactionContextInterface,
    token, modelID, entryString string) error {

    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return fmt.Errorf("failed to retrieve token %s from world state. %v", token, err)
    }

    if tokenBytes == nil {
        return fmt.Errorf("authorization not granded: token %s is invalid", token)
    }
```

```
128
129     var entry GenericPage
130     err = json.Unmarshal([]byte(entryString), &entry)
131     if err != nil {
132         return fmt.Errorf("failed to process json data; ensure correct structure")
133     }
134
135     entryBytes, err := json.Marshal(entry)
136     if err != nil {
137         return err
138     }
139
140     err = ctx.GetStub().PutState(modelID+"_"+strconv.Itoa(entry.PageID), entryBytes)
141     if err != nil {
142         return err
143     }
144
145     return nil
146 }
147
148 // QueryMetaData retrieves the metadata of a model from the ledger
149 func (t *SimpleChaincode) QueryMetaData(ctx contractapi.TransactionContextInterface,
150     token, modelID string) (*MetaData, error) {
151
152     tokenBytes, err := ctx.GetStub().GetState(token)
153     if err != nil {
154         return nil, fmt.Errorf("failed to retrieve token %s from world state. %v",
155                                                                         token, err)
156     }
157
158     if tokenBytes == nil {
159         return nil, fmt.Errorf("authorization not granded: token %s is invalid", token)
160     }
161
162     //retrieve the model's metadata
163     metaBytes, err := ctx.GetStub().GetState(modelID + "_metadata")
164     if err != nil {
165         return nil, fmt.Errorf("failed to get metadata for model %s: %v", modelID, err)
166     }
167     if metaBytes == nil {
168         return nil, fmt.Errorf("model_id %s does not exist", modelID)
169     }
170
171     var meta MetaData
172     err = json.Unmarshal(metaBytes, &meta)
173     if err != nil {
174         return nil, err
175     }
176
177     return &meta, nil
178 }
179
180 ////////////////////////////////////////////////////////////////////////////////////////////
181 // Performs an adhoc query on MetaData ledger entries
182 func (t *SimpleChaincode) QueryMetaDataWithPagination
183     (ctx contractapi.TransactionContextInterface, token, queryString string,
184     pageSize int, bookmark string) (*MetaPaginatedQueryResult, error) {
185
186     tokenBytes, err := ctx.GetStub().GetState(token)
187     if err != nil {
188         return nil, fmt.Errorf("failed to retrieve token %s from world state. %v",
189                                                                         token, err)
190     }
191
192     if tokenBytes == nil {
193         return nil, fmt.Errorf("authorization not granded: token %s is invalid", token)
194     }
195
196     return getMetaQueryResultForQueryStringWithPagination(ctx, queryString,
197                                                         int32(pageSize), bookmark)
198 }
199
```

```
func getMetaQueryResultForQueryStringWithPagination
    (ctx contractapi.TransactionContextInterface, queryString string,
    pageSize int32, bookmark string) (*MetaPaginatedQueryResult, error) {

    resultsIterator, responseMetadata, err :=
            ctx.GetStub().GetQueryResultWithPagination(queryString, pageSize, bookmark)
    if err != nil {
        return nil, fmt.Errorf("here 3") //err
    }
    defer resultsIterator.Close()

    metas, err := constructMetaQueryResponseFromIterator(resultsIterator)
    if err != nil {
        return nil, err
    }

    if metas == nil {
        return nil, nil
    }

    return &MetaPaginatedQueryResult{
        Records:             metas,
        FetchedRecordsCount: responseMetadata.FetchedRecordsCount,
        Bookmark:            responseMetadata.Bookmark,
    }, nil
}

// constructQueryResponseFromIterator constructs a slice of assets from the
// resultsIterator
func constructMetaQueryResponseFromIterator
    (resultsIterator shim.StateQueryIteratorInterface) ([]*MetaData, error) {

    var metas []*MetaData
    for resultsIterator.HasNext() {
        queryResult, err := resultsIterator.Next()
        if err != nil {
            return nil, fmt.Errorf("here 1") //err
        }

        var meta MetaData
        err = json.Unmarshal(queryResult.Value, &meta)
        if err != nil {
            return nil, fmt.Errorf("here 2") //err
        }

        metas = append(metas, &meta)
    }

    return metas, nil
}

////////////////////////////////////////////////////////////////////////////////////////////

// Returns the ledger entries which correspond to a specific model
func (t *SimpleChaincode) QueryLatestModelWithPagination
    (ctx contractapi.TransactionContextInterface, token, modelID string, pageSize int,
    bookmark string) (*PaginatedQueryResult, error) {

    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return nil, fmt.Errorf("failed to retrieve token %s from world state. %v",
                                                                        token, err)
    }

    if tokenBytes == nil {
        return nil, fmt.Errorf("authorization not granded: token %s is invalid", token)
    }

    //retrieve the model's metadata
    metaBytes, err := ctx.GetStub().GetState(modelID + "_metadata")
    if err != nil {
        return nil, fmt.Errorf("failed to get metadata for model %s: %v", modelID, err)
    }
    if metaBytes == nil {
        return nil, fmt.Errorf("model_id %s does not exist", modelID)
```

```
275     }
276
277     var meta MetaData
278     err = json.Unmarshal(metaBytes, &meta)
279     if err != nil {
280         return nil, err
281     }
282
283     queryString := fmt.Sprintf(`{"selector":{"$and":[{"model_id":"%s"}`
284                             + `,{"page_id": {"$lt":%d}}]}}`, modelID, meta.PageNumber)
285     return getQueryResultForQueryStringWithPagination(ctx, queryString, int32(pageSize),
286                                                                                 bookmark)
287 }
288
289 func getQueryResultForQueryStringWithPagination
290     (ctx contractapi.TransactionContextInterface, queryString string, pageSize int32,
291     bookmark string) (*PaginatedQueryResult, error) {
292
293     resultsIterator, responseMetadata, err :=
294             ctx.GetStub().GetQueryResultWithPagination(queryString, pageSize, bookmark)
295     if err != nil {
296         return nil, err
297     }
298     defer resultsIterator.Close()
299
300     pages, err := constructQueryResponseFromIterator(resultsIterator)
301     if err != nil {
302         return nil, err
303     }
304
305     if pages == nil {
306         return nil, nil
307     }
308
309     return &PaginatedQueryResult{
310         Records:             pages,
311         FetchedRecordsCount: responseMetadata.FetchedRecordsCount,
312         Bookmark:            responseMetadata.Bookmark,
313     }, nil
314 }
315
316 // constructQueryResponseFromIterator constructs a slice of assets from the
317 // resultsIterator
318 func constructQueryResponseFromIterator(resultsIterator shim.StateQueryIteratorInterface)
319     ([]*GenericPageWrapper, error) {
320
321     var pages []*GenericPageWrapper
322     for resultsIterator.HasNext() {
323         queryResult, err := resultsIterator.Next()
324         if err != nil {
325             return nil, err
326         }
327
328         var page GenericPage
329         err = json.Unmarshal(queryResult.Value, &page)
330         if err != nil {
331             return nil, err
332         }
333
334         var wrap = GenericPageWrapper{
335             Key:   strconv.Itoa(page.PageID),
336             Value: page,
337         }
338
339         pages = append(pages, &wrap)
340     }
341
342     return pages, nil
343 }
344
345 ////////////////////////////////////////////////////////////////////////////////////////////
346
```

```
// DeleteKeys deletes the keys in the given array
func (t *SimpleChaincode) DeleteKeys(ctx contractapi.TransactionContextInterface,
    token string, keys []string) error {

    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return fmt.Errorf("failed to retrieve token %s from world state. %v", token, err)
    }

    if tokenBytes == nil {
        return fmt.Errorf("authorization not granded: token %s is invalid", token)
    }

    for _, key := range keys {
        err = ctx.GetStub().DelState(key)
        if err != nil {
            return fmt.Errorf("failed to delete ledger entry %s: %v", key, err)
        }
    }

    return nil
}

// GetKeys returns all the keys that much the provided adhoc
func (t *SimpleChaincode) GetKeys(ctx contractapi.TransactionContextInterface, token,
    modelID, queryString string, pageSize int, bookmark string) ([]string, error) {

    var keys []string

    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return keys, fmt.Errorf("failed to retrieve token %s from world state. %v",
                                                                        token, err)
    }

    if tokenBytes == nil {
        return keys, fmt.Errorf("authorization not granded: token %s is invalid", token)
    }

```

```
386     //retrieve the model's metadata
387     metaBytes, err := ctx.GetStub().GetState(modelID + "_metadata")
388     if err != nil {
389         return keys, fmt.Errorf("failed to get metadata for model %s: %v", modelID, err)
390     }
391     if metaBytes == nil {
392         return keys, fmt.Errorf("model_id %s does not exist", modelID)
393     }
394
395     book := bookmark
396
397     for true {
398         resultsIterator, responseMetadata, err :=
399             ctx.GetStub().GetQueryResultWithPagination(queryString, int32(pageSize), book)
400         if err != nil {
401             return keys, err
402         }
403         defer resultsIterator.Close()
404
405         for resultsIterator.HasNext() {
406             queryResult, err := resultsIterator.Next()
407             if err != nil {
408                 return keys, err
409             }
410
411             keys = append(keys, queryResult.Key)
412
413         }
414
415         if responseMetadata.FetchedRecordsCount == 0 {
416             break
417         }
418
419         book = responseMetadata.Bookmark
420
421     }
422
423     return keys, nil
```

```
}

func (t *SimpleChaincode) GetCleanUpKeys(ctx contractapi.TransactionContextInterface,
    token, modelID string, pageSize int, bookmark string) ([]string, error) {

    var keys []string

    tokenBytes, err := ctx.GetStub().GetState(token)
    if err != nil {
        return keys, fmt.Errorf("failed to retrieve token %s from world state. %v",
                                                                        token, err)
    }

    if tokenBytes == nil {
        return keys, fmt.Errorf("authorization not granded: token %s is invalid", token)
    }

    //retrieve the model's metadata
    metaBytes, err := ctx.GetStub().GetState(modelID + "_metadata")
    if err != nil {
        return keys, fmt.Errorf("failed to get metadata for model %s: %v", modelID, err)
    }
    if metaBytes == nil {
        return keys, fmt.Errorf("model_id %s does not exist", modelID)
    }

    var meta MetaData
    err = json.Unmarshal(metaBytes, &meta)
    if err != nil {
        return nil, err
    }

    queryString := fmt.Sprintf(`{"selector":{"$and":[{"model_id":"%s"}`
                            +`,{"page_id": {"$gte":%d}}]}}`, modelID, meta.PageNumber)

    book := bookmark

    for true {
        resultsIterator, responseMetadata, err :=
            ctx.GetStub().GetQueryResultWithPagination(queryString, int32(pageSize), book)
        if err != nil {
        if err != nil {
            return keys, err
        }
        defer resultsIterator.Close()

        for resultsIterator.HasNext() {
            queryResult, err := resultsIterator.Next()
            if err != nil {
                return keys, err
            }

            keys = append(keys, queryResult.Key)

        }

        if responseMetadata.FetchedRecordsCount == 0 {
            break
        }

        book = responseMetadata.Bookmark

    }

    return keys, nil
}

////////////////////////////////////////////////////////////////////////////////////////

func (t *SimpleChaincode) Init(ctx contractapi.TransactionContextInterface) error {

    err := ctx.GetStub().PutState("token", []byte{'t', 'o', 'k', 'e', 'n'})
    if err != nil {
        return err
    }

```

```
499     return nil
500 }
501
502 func main() {
503
504     sc := new(SimpleChaincode)
505
506     cc, err := contractapi.NewChaincode(sc)
507
508     if err != nil {
509         panic(err.Error())
510     }
511
512     if err := cc.Start(); err != nil {
513         panic(err.Error())
514     }
515
516 }
517
```

Προσομοιωτής Πελατών (Πειραματική Αξιολόγηση – simulator.js)

```
//node simulator.js submit test100 true 100 10 1
const http = require('http')
const fs = require('fs')

var myArgs = process.argv.slice(2);
const MODEL = myArgs[1]; //darknet | pytorch | tfv2 | tfjs | tflite
const RUN = myArgs[0]; // submit | retrieve
const TEST = myArgs[2]; // true | false
const TEST_MODEL_SIZE = parseInt(myArgs[3]) * 1024 * 1024; //MB
const AVG_COUNT = parseInt(myArgs[4]);
const INTERVAL = parseInt(myArgs[5]);

function darknet(){
    let architecture = fs.readFileSync('./data/experiments/darknet/yolov3.cfg',
                                       {encoding: 'base64'});
    let weights = fs.readFileSync('./data/experiments/darknet/yolov3.weights',
                                       {encoding: 'base64'});
    let data = JSON.stringify({
        "token": "token",
        "data": {
            "id": "darknet",
            "tag1":"yolo",
            "tag2":"object_detection",
            "serialization_encoding":"base64",
            "model": [
                {
                    "metadata":
                    {
                        "identifier":"yolov3",
                        "original_format":"cfg"
                    },
                    "serialized_data": architecture
                }
            ],
            "weights":[
                {
                    "metadata":
                    {
                        "identifier":"yolov3",
                        "original_format":"weights"
                    },
                      "serialized_data": weights
                }
             ],
             "initialization":[ ],
             "checkpoints":[ ]
        }
    })

    return data;
}

function pytorch(){
    let model = fs.readFileSync('./data/experiments/pytorch/yolov5s.pt',
                        {encoding: 'base64'});
    let data = JSON.stringify({
        "token": "token",
        "data": {
            "id": "pytorch",
            "tag1":"yolo",
            "tag2":"object_detection",
            "serialization_encoding":"base64",
            "model": [
                {
                    "metadata":
                    {
                        "identifier":"yolov5s",
                        "original_format":"pt"
                    },
                    "serialized_data": model
                }
            ],
            "weights":[ ],
            "initialization":[ ],
            "checkpoints":[ ]
        }
    })

    return data;
}
```

```
81
82  function tfv2(){
83      let variables1 = fs.readFileSync('./data/experiments/tf-v2/variables'
84                                       +'/variables.data-00000-of-00001', {encoding: 'base64'});
85      let variables2 = fs.readFileSync('./data/experiments/tf-v2/variables'
86                                       +'/variables.index', {encoding: 'base64'});
87      let model = fs.readFileSync('./data/experiments/tf-v2/saved_model.pb',
88                                       {encoding: 'base64'});
89      let data = JSON.stringify({
90          "token": "token",
91          "data": {
92              "id": "tfv2",
93              "tag1":"ssd_mobilenet_v2",
94              "tag2":"object_detection",
95              "serialization_encoding":"base64",
96              "model": [
97                  {
98                      "metadata":
99                      {
100                         "identifier":"saved_model",
101                         "original_format":"pb"
102                     },
103                     "serialized_data": model
104                 }
105             ],
106             "weights":[ ],
107             "initialization":[ ],
108             "checkpoints":[
109                 {
110                     "metadata":
111                     {
112                         "identifier":"variables",
113                         "original_format":"data-00000-of-00001"
114                     },
115                     "serialized_data": variables1
116                 },
117                 {
118                     "metadata":
119                     {
120                         "identifier":"variables",
121                         "original_format":"index"
122                     },
123                     "serialized_data": variables2
124                 }
125              ]
126         }
127     })
128
129     return data;
130 }
131
132 function tfjs(){
133     let shard1 = fs.readFileSync('./data/experiments/tfjs/group1-shard1of17',
134                                          {encoding: 'base64'});
135     let shard2 = fs.readFileSync('./data/experiments/tfjs/group1-shard2of17',
136                                          {encoding: 'base64'});
137     let shard3 = fs.readFileSync('./data/experiments/tfjs/group1-shard3of17',
138                                          {encoding: 'base64'});
139     let shard4 = fs.readFileSync('./data/experiments/tfjs/group1-shard4of17',
140                                          {encoding: 'base64'});
141     let shard5 = fs.readFileSync('./data/experiments/tfjs/group1-shard5of17',
142                                          {encoding: 'base64'});
143     let shard6 = fs.readFileSync('./data/experiments/tfjs/group1-shard6of17',
144                                          {encoding: 'base64'});
145     let shard7 = fs.readFileSync('./data/experiments/tfjs/group1-shard7of17',
146                                          {encoding: 'base64'});
147     let shard8 = fs.readFileSync('./data/experiments/tfjs/group1-shard8of17',
148                                          {encoding: 'base64'});
149     let shard9 = fs.readFileSync('./data/experiments/tfjs/group1-shard9of17',
150                                          {encoding: 'base64'});
151     let shard10 = fs.readFileSync('./data/experiments/tfjs/group1-shard10of17',
152                                          {encoding: 'base64'});
153     let shard11 = fs.readFileSync('./data/experiments/tfjs/group1-shard11of17',
154                                          {encoding: 'base64'});
155     let shard12 = fs.readFileSync('./data/experiments/tfjs/group1-shard12of17',
156                                          {encoding: 'base64'});
157     let shard13 = fs.readFileSync('./data/experiments/tfjs/group1-shard13of17',
158                                          {encoding: 'base64'});
159     let shard14 = fs.readFileSync('./data/experiments/tfjs/group1-shard14of17',
```

```
                                            {encoding: 'base64'});
    let shard15 = fs.readFileSync('./data/experiments/tfjs/group1-shard15of17',
                                            {encoding: 'base64'});
    let shard16 = fs.readFileSync('./data/experiments/tfjs/group1-shard16of17',
                                            {encoding: 'base64'});
    let shard17 = fs.readFileSync('./data/experiments/tfjs/group1-shard17of17',
                                            {encoding: 'base64'});
    let model = fs.readFileSync('./data/experiments/tfjs/tensorflowjs_model.pb',
                                            {encoding: 'base64'});
    let architecture = fs.readFileSync('./data/experiments/tfjs/model.json',
                                            {encoding: 'base64'});
    let weights = fs.readFileSync('./data/experiments/tfjs/weights_manifest.json',
                                            {encoding: 'base64'});

    let data = JSON.stringify({
        "token": "token",
        "data": {
            "id": "tfjs",
            "tag1":"ssd_mobilenet_v2",
            "tag2":"object_detection",
            "serialization_encoding":"base64",
            "model": [
                {
                    "metadata":
                    {
                        "identifier":"tensorflowjs_model",
                        "original_format":"pb"
                    },
                    "serialized_data": model
                },
                {
                    "metadata":
                    {
                        "identifier":"model",
                        "original_format":"json"
                    },
                    "serialized_data": architecture
                },
                {
                    "metadata":
                    {
                        "identifier":"shard1of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard1
                },
                {
                    "metadata":
                    {
                        "identifier":"shard2of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard2
                },
                {
                    "metadata":
                    {
                        "identifier":"shard3of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard3
                },
                {
                    "metadata":
                    {
                        "identifier":"shard4of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard4
                },
                {
                    "metadata":
                    {
                        "identifier":"shard5of17",
                        "original_format":"bin"
```

```
                    },
                    "serialized_data": shard5
                },
                {
                    "metadata":
                    {
                        "identifier":"shard6of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard6
                },
                {
                    "metadata":
                    {
                        "identifier":"shard7of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard7
                },
                {
                    "metadata":
                    {
                        "identifier":"shard8of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard8
                },
                {
                    "metadata":
                    {
                        "identifier":"shard9of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard9
                },
                {
                    "metadata":
                    {
                        "identifier":"shard10of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard10
                },
                {
                    "metadata":
                    {
                        "identifier":"shard11of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard11
                },
                {
                    "metadata":
                    {
                        "identifier":"shard12of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard12
                },
                {
                    "metadata":
                    {
                        "identifier":"shard13of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard13
                },
                {
                    "metadata":
                    {
                        "identifier":"shard14of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard14
                },
                {
                    "metadata":
                    {
                        "identifier":"shard15of17",
```

```
                        "identifier":"shard15of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard15
                },
                {
                    "metadata":
                    {
                        "identifier":"shard16of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard16
                },
                {
                    "metadata":
                    {
                        "identifier":"shard17of17",
                        "original_format":"bin"
                    },
                    "serialized_data": shard17
                }
            ],
            "weights":[
                {
                    "metadata":
                    {
                        "identifier":"weights_manifest",
                        "original_format":"json"
                    },
                    "serialized_data": weights
                }
             ],
            "initialization":[ ],
            "checkpoints":[ ]
        }
    })

    return data;
}

function tflite(){
    let model = fs.readFileSync('./data/experiments/tflite/lite-'
        +'model_efficientdet_lite0_detection_default_1.tflite', {encoding: 'base64'});
    let data = JSON.stringify({
        "token": "token",
        "data": {
            "id": "tflite",
            "tag1":"efficientdet",
            "tag2":"object_detection",
            "serialization_encoding":"base64",
            "model": [
                {
                    "metadata":
                    {
                        "identifier":"lite-model_efficientdet_lite0_detection_default_1",
                        "original_format":"tflite"
                    },
                    "serialized_data": model
                }
            ],
            "weights":[ ],
            "initialization":[ ],
            "checkpoints":[ ]
        }
    })

    return data;
}

function test_model(size, name){
    var test_data = Buffer.allocUnsafe(size).fill('s').toString('utf8');


    let data = JSON.stringify({
        "token": "token",
        "data": {
            "id": "test_"+name,
            "tag1":"tag1",
```

```
            "tag2":"tag2",
            "serialization_encoding":"utf8",
            "model": [
                {
                    "metadata":
                    {
                        "identifier":"test",
                        "original_format":"test"
                    },
                    "serialized_data": test_data
                }
            ],
            "weights":[ ],
            "initialization":[ ],
            "checkpoints":[ ]
        }
    })

    return data;
}

function submit(model_id, name){

    if (TEST === 'true'){
        var data = test_model(TEST_MODEL_SIZE, name);
    }else if (model_id === 'darknet'){
        var data = darknet();
    }else if (model_id === 'pytorch'){
        var data = pytorch();
    }else if (model_id === 'tfv2'){
        var data = tfv2();
    }else if (model_id === 'tfjs'){
        var data = tfjs();
    }else if (model_id === 'tflite'){
        var data = tflite();
    }

    const options = {
      hostname: '10.16.30.89',
      port: 3000,
      path: '/model/submit',
      method: 'PUT',
      headers: {
        'Content-Type': 'application/json',
        'Content-Length': data.length
      }
    }

    const req = http.request(options, res => {
      console.log(`statusCode: ${res.statusCode}`)

      res.on('data', d => {
        console.log(d.toString());
      })
    })

    req.on('error', error => {
      console.error(error)
    })

    req.write(data)
    req.end()
}

function retrieve(model_id){

    const options = {
        hostname: '10.16.30.89',
        port: 3000,
        path: '/model?id='+model_id+'&token=token',
        method: 'GET'
      }

      const req = http.request(options, res => {
        console.log(`statusCode: ${res.statusCode}`)

```

```
        res.on('data', d => {
          //process.stdout.write(d)
        })
      })

      req.on('error', error => {
        console.error(error)
      })

      req.end()

}

if (RUN === 'submit'){

    let i = 0;
    let a = new Date();
    while(i < AVG_COUNT){
        if (new Date().getTime() >= a.getTime()){
            submit(MODEL, i);
            a.setSeconds(a.getSeconds()+INTERVAL);
            console.log(a.getSeconds());
            i++;
        }
    }

    //submit(MODEL, '');
}else{
    retrieve(MODEL);
}
```