ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

Αρχιτεκτονικές λογισμικού για παροχή δικτυακής υπηρεσίας επίλυσης υπολογιστικών προβλημάτων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΙΩΑΝΝΗ ΛΙΑΚΑΤΣΙΔΑ

Επιβλέπων: Βασίλειος Βεσκούκης

Καθηγητής ΕΜΠ

Αθήνα, Φεβρουάριος 2024

Εθνικο Μετσοβιο Πολυτεχνειο
Σχολη Ηλεκτρολογων Μηχανικων
και Μηχανικων Υπολογιστων
Τομεας Τεχνολογιας Πληροφορικης και Υπολογιστων

Αρχιτεκτονικές λογισμικού για παροχή δικτυακής υπηρεσίας επίλυσης υπολογιστικών προβλημάτων

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

του

ΙΩΑΝΝΗ ΛΙΑΚΑΤΣΙΔΑ

Επιβλέπων: Βασίλειος Βεσκούκης
Καθηγητής ΕΜΠ

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 15η Φεβρουαρίου 2024.

Βασίλειος Βεσκούκης	Νικόλαος Παπασπύρου	Γεώργιος Γκούμας
Καθηγητής ΕΜΠ	Καθηγητής ΕΜΠ	Αναπληρωτής Καθηγητής ΕΜΠ

Αθήνα, Φεβρουάριος 2024

Ιωάννης Ν. Λιακατσίδας

Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Copyright © Ιωάννης Λιακατσίδας, 2024.
Με επιφύλαξη παντός δικαιώματος. All rights reserved.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

Οι απόψεις και τα συμπεράσματα που περιέχονται σε αυτό το έγγραφο εκφράζουν τον συγγραφέα και δεν πρέπει να ερμηνευθεί ότι αντιπροσωπεύουν τις επίσημες θέσεις του Εθνικού Μετσόβιου Πολυτεχνείου.

Περίληψη

Η ανάπτυξη λογισμικού αποτελεί έναν από τους πιο έντονα αναπτυσσόμενους τεχνολογικούς τομείς του 21ου αιώνα, με την καθιέρωση του Cloud και τη σταδιακή απομάκρυνση από τα παραδοσιακά τοπικά προγράμματα να επιφέρουν ραγδαίες αλλαγές στον επιχειρηματικό κόσμο. Σε αυτό το πλαίσιο, οι μοντέρνες εφαρμογές γίνονται ολοένα και περισσότερο πολύπλοκες και απαιτητικές ως προς την κατανάλωση υπολογιστικών πόρων. Προκειμένου οι διάφορες εταιρείες να μπορέσουν να συμβαδίσουν με τον καταιγιστικό ρυθμό που τα υπάρχοντα δεδομένα μεταβάλλονται και νέα δεδομένα κάνουν την εμφάνισή τους, καθίσταται επιτακτική η ανάγκη αποδοτικής, ασφαλούς και αξιόπιστης διαχείρισης ενός όγκου πληροφοριών σημαντικά μεγαλύτερου συγκριτικά με το παρελθόν. Μία από τις λύσεις που διατίθενται στην αγορά εργασίας για την αντιμετώπιση αυτών των ζητημάτων σε ένα λογισμικό σύστημα είναι η χρήση ενός διαμεσολαβητή μηνυμάτων, ευρέως γνωστό ως message broker, ο οποίος αναλαμβάνει εξ ολοκλήρου την λήψη και δρομολόγηση των δεδομένων μέχρις ότου να γίνει η απαραίτητη διευθέτηση τους από το λογισμικό, οπότε και επιστρέφει το αποτέλεσμα πίσω στον τελικό χρήστη.

Στόχο της παρούσας διπλωματικής εργασίας αποτελεί η κατασκευή και διερεύνηση ενός solver υπολογιστικών προβλημάτων ο οποίος αξιοποιεί τον παγκοσμίως διαδεδομένο message broker RabbitMQ για τη διαχείριση των εισερχόμενων αιτημάτων προς επίλυση και υπακούει στις θεμελιώδεις αρχές του Software-as-a-Service μοντέλου. Χρησιμοποιώντας τη βιβλιοθήκη OR-Tools της Google, επιχειρείται η επίλυση μίας σειράς προβλημάτων που υπάγονται στον κλάδο της Επιχειρησιακής Έρευνας, τόσο με την απευθείας επικοινωνία του πελάτη και του διακομιστή όσο και με τη διαμεσολάβηση της RabbitMQ μεταξύ τους, ενώ ακόμη επιδιώκεται η σύγκριση διαφορετικών αρχιτεκτονικών υλοποίησης μίας τέτοιας υπολογιστικής πλατφόρμας. Επιπλέον, μελετάται η επίδραση της παρουσίας message broker τόσο στην ορθότητα των αποτελεσμάτων όσο και στην ανθεκτικότητα κατά την αντιμετώπιση μεγάλου και ιδιαίτερα απαιτητικού όγκου προβλημάτων, αναδεικνύοντας παράλληλα τα πλεονεκτήματα που προσδίδει η χρήση ενός message broker σε ένα σύνθετο και υπολογιστικά ακριβό software προϊόν. Τέλος, επισημαίνονται κάποιες ιδέες οι οποίες σε θεωρητικό επίπεδο διασφαλίζουν τη βελτίωση της απόδοσης και της κλιμακωσιμότητας της κατασκευασμένης υπηρεσίας προκειμένου να μπορεί να ανταπεξέλθει στις απαιτήσεις του πραγματικού κόσμου.

Λέξεις – Κλειδιά

Message Broker, RabbitMQ, Solver, Αρχιτεκτονική Λογισμικού, API, Μαθηματική Βελτιστοποίηση, Επιχειρησιακή Έρευνα, Google OR-Tools, Οικοσύστημα .NET

Abstract

Software development is one of the most rapidly growing technological sectors of the 21st century, with the advent of Cloud computing and the gradual shift away from traditional on-premise applications causing significant changes in the business world. In this context, modern applications are becoming increasingly complex and resource-intensive. Since companies seek to keep pace with the rate at which existing data changes and new data is created, an urgent need for efficient, secure, and reliable management of computational resources emerges. Current technological developments allow sharing of resources on a pay-per-use basis, which, in turn brings up several new challenges. One of the solutions widely used to achieve this, is a family of services offered by "message brokers", which takes full responsibility for receiving and routing data until they are appropriately handled by the software, returning the result back to the end user.

The aim of this thesis is to construct a software architecture to offer as-a-service a solver for computational problems that utilizes the globally widespread message broker RabbitMQ for managing incoming requests and adheres to the fundamental principles of the Software-as-a-Service model. Using Google's OR-Tools library as an example solver, configured to solve a series of problems falling under the domain of Operations Research, we investigate architectures that implement both direct communication between the client (service consumer) and the server (resources and solver-as-a-service provider) through the mediation of the RabbitMQ messaging service. Additionally, the comparison of different architectural implementations of such a computational platform is pursued. Furthermore, the impact of the presence of a message broker on the quality of service, as well as on the resilience in dealing with a significant load of demanding problems is studied, while also highlighting the advantages that the use of a message broker provides in a complex and resource-intensive software product. Finally, some ideas are outlined at a that ensure the improvement of performance and scalability of the constructed service theoretical level meet the requirements of the real world.

Keywords

Message Broker, RabbitMQ, Solver, Software Architecture, API, Mathematical Optimization, Operational Research, Google OR-Tools, .NET Ecosystem

Ευχαριστίες

Με την παρούσα διπλωματική εργασία, σηματοδοτείται το τέλος της φοίτησής μου στη Σχολή Ηλεκτρολόγων Μηχανικών & Μηχανικών Υπολογιστών του Εθνικού Μετσόβιου Πολυτεχνείου. Θα ήθελα να ευχαριστήσω θερμά τον Καθηγητή Βασίλειο Βεσκούκη για το ενδιαφέρον και την καθοριστική συνεισφορά του στην εκπόνηση του έργου αυτού, προσφέροντάς μου την απαραίτητη καθοδήγηση καθ' όλη τη διάρκειά του. Επίσης, θα ήθελα να ευχαριστήσω τους γονείς μου αλλά και όλα τα κοντινά μου άτομα, καθώς αυτό το όμορφο ταξίδι δε θα μπορούσε να ολοκληρωθεί δίχως την αμέριστη και ανιδιοτελή στήριξή τους.

Περιεχόμενα

Περίληψη ... 1

Abstract ... 3

Ευχαριστίες ... 5

Περιεχόμενα ... 7

1. Εισαγωγή ... 10

1.1. Πλαίσιο – Κίνητρο ... 10

1.2. Πλάνο – Δομή της εργασίας ... 11

2. Τεχνολογικό υπόβαθρο ... 12

2.1. Solvers ... 12

2.1.1. Ορισμός – Ιστορική αναδρομή ... 12

2.1.2. Τυπικοί solvers ... 13

2.2. Application Programming Interface (API) ... 15

2.2.1. Ορισμός ... 15

2.2.2. RESTful APIs ... 16

2.2.3. Non-RESTful APIs ... 18

2.2.4. Stateful vs Stateless API ... 19

2.2.5. Remote Procedure Call (RPC) ... 20

2.2.6. Σύγκριση REST με RPC API ... 22

2.2.7. H gRPC Remote Procedure Call ... 23

2.2.8. Διαφορές REST και gRPC πρωτοκόλλων ... 25

2.3. Message brokers ... 27

2.3.1. Ορισμός - Ιστορική αναδρομή ... 27

2.3.2. Γενικά στοιχεία ενός message broker ... 30

2.3.3. Το λογισμικό RabbitMQ ... 32

2.3.4. Εγκατάσταση σε Ubuntu ... 40

2.4. Οικοσύστημα .NET ... 41

2.4.1. Ιστορία του .NET ... 41

2.4.2. Εργαλεία και εξαρτήματα ... 42

2.4.3. Βιβλιοθήκες και frameworks 45
2.4.4. .NET σε Linux 46
2.4.5. Windows vs. Linux 47
2.5. Εξισορρόπηση φορτίου (Load Balancing) 49
2.5.1. Στατικό load balancing 50
2.5.2. Δυναμικό load balancing 51
2.6. Επιχειρησιακή Έρευνα 52
2.7. Γενικά για το Google OR-Tools 54

3. Μελέτη περίπτωσης 59

3.1. Επεξήγηση του προβλήματος 59
3.2. Δημιουργία σεναρίων μελέτης 60
3.3. Τα δομικά στοιχεία του λογισμικού 62
3.3.1. Γενικά στοιχεία αρχιτεκτονικής 62
3.3.2. Τα δομικά στοιχεία του συστήματος 65
3.3.3. Το service αποστολής των δεδομένων 66
3.3.4. Ο solver controller του συστήματος 67
3.3.5. Ο solver dispatcher του συστήματος 68
3.3.6. Ο πυρήνας και ο solver wrapper του συστήματος 69
3.4. Αρχιτεκτονικές προς διερεύνηση 71
3.4.1. Απλουστευμένη υλοποίηση χωρίς τη χρήση APIs 72
3.4.2. Υλοποίηση με χρήση wrapper γύρω από το solver 73
3.4.3. Πλήρης υλοποίηση με χρήση wrapper και message broker 75
3.5. Σύγκριση των αρχιτεκτονικών 78
3.5.1. Αποτελέσματα δεύτερης αρχιτεκτονικής 78
3.5.2. Αποτελέσματα τρίτης αρχιτεκτονικής 80

4. Συμπεράσματα – Μελλοντική Εργασία 83

4.1. Συμπεράσματα 83
4.2. Μελλοντικές επεκτάσεις 84

Βιβλιογραφία 86

1. Εισαγωγή

1.1. Πλαίσιο – Κίνητρο

Στη σημερινή εποχή, το πλήθος, το μέγεθος και η υπολογιστική πολυπλοκότητα των εφαρμογών αυξάνεται με ραγδαίους ρυθμούς. Παράλληλα, το μέγεθος των κάθε λογής δεδομένων που αναλύονται σε κάθε λογής υπολογιστικά προβλήματα παρουσιάζει αντίστοιχη αυξητική τάση, με αποτέλεσμα να απαιτούνται σημαντικοί και ακριβοί υπολογιστικοί πόροι για την επίλυση τέτοιων προβλημάτων. Με δεδομένες τις σημερινές τεχνολογίες, η επίλυση απαιτητικών προβλημάτων μπορεί η ίδια να παρέχεται ως υπηρεσία και με τον τρόπο αυτό δημιουργείται μια ανταγωνιστική αγορά. Ειδικότερα, για τέτοιες περιπτώσεις εφαρμογών που επικεντρώνονται στην επίλυση πολύπλοκων υπολογιστικών προβλημάτων που απαιτούν σημαντικούς πόρους και συνακόλουθα μεγάλο χρόνο επεξεργασίας, οι διαθέσιμες επιλογές για κάποιον που επιθυμεί να επιλύει σε δικές του υποδομές τέτοια προβλήματα είναι ιδιαίτερα επιβαρυντικές οικονομικά σε άδειες χρήσης ή/και υπολογιστικούς πόρους.

Για την παρούσα διπλωματική εργασία, κίνητρο αποτελεί η διερεύνηση αρχιτεκτονικών λογισμικού για την παροχή μιας υπολογιστικής υπηρεσίας για επίλυση υπολογιστικά ακριβών προβλημάτων, η οποία αξιοποιεί τις αρχές της μεθόδου Software as a Service (SaaS), με την τιμή την οποία καλείται να πληρώσει ο χρήστης να καθορίζεται από το πλήθος εκτελέσεων του προγράμματος σε πόρους (άδειες λογισμικού, CPU, μνήμη κλπ) οι οποίοι παρέχονται ως υπηρεσία. Προκειμένου να επιτευχθεί αυτό, αναζητείται μια αρχιτεκτονική βασισμένη σε διεπαφές προγραμματισμού εφαρμογών (API), η οποία θα διαχειρίζεται έναν επιλύτη (solver), δηλαδή μια απαιτητική υπολογιστική υποδομή (λογισμικό και πόροι εκτέλεσης) που εκτελεί την επίλυση του εκάστοτε απεσταλμένου υπολογιστικού προβλήματος. Τέτοια τυπικά προβλήματα είναι τα προβλήματα επιχειρησιακής έρευνας και βελτιστοποίησης. Σε αυτή τη λογική, κρίνεται πολύ σημαντική η αποσύνδεση (decoupling) του επιλύτη από τον πελάτη-αποστολέα, ώστε η τεχνολογία με την οποία αναπτύσσεται ο solver να μην επηρεάζει τα αιτήματα επίλυσης προβλημάτων. Για το λόγο αυτό, επιλέγεται η διερεύνηση της υλοποίησης μιας αρχιτεκτονικής λογισμικού που περιλαμβάνει έναν "διαμεσολαβητή" που αποθηκεύει τα μηνύματα-αιτήσεις προς τον solver σε ουρές, ευρύτερα γνωστός στον κλάδο της Μηχανικής Λογισμικού ως message broker. Καθώς θεμελιώδη στόχο αποτελεί η μελέτη της συμπεριφοράς ενός τέτοιου συστήματος με σύγχρονες και διαδεδομένες τεχνολογίες, για το ρόλο του ίδιου του message broker επιλέγεται η χρήση ενός έτοιμου λογισμικού που παρέχει αυτή τη λειτουργικότητα. Συγκεκριμένα, για την κατασκευή του διαμεσολαβητή μηνυμάτων, θα αξιοποιηθεί το ευρέως διαδεδομένο λογισμικό RabbitMQ.

Στο πλαίσιο αυτής της διπλωματικής εργασίας, θα διερευνηθεί μια αρχιτεκτονική στην οποία ο τελικός χρήστης μεταφέρει τις απαιτητικές υπολογιστικές εργασίες που επιθυμεί να επιλυθούν στο message broker μέσω ενός API, οι οποίες με τις σειρά τους καταναλώνονται από ένα άλλο μέρος του λογισμικού. Έπειτα, εκτελείται η επίλυσή τους στον πυρήνα του solver και ακολούθως επιστρέφονται στην κατάλληλη μορφή αποτελέσματος στο αρχικό API, ώστε τελικά να φτάσουν επιλυμένες στο χρήστη. Προκειμένου να εξεταστεί στο

απαιτούμενο βάθος οι λειτουργικότητες του μοντέλου, θα διεξαχθεί εκτενής σύγκριση ανάμεσα στη συμπεριφορά της αρχιτεκτονικής αυτής με τη συμπεριφορά μίας αρχιτεκτονικής που επιτελεί το ίδιο έργο, δηλαδή τη δρομολόγηση προβλημάτων προς επίλυση, χωρίς τη χρήση message broker και κατ' επέκταση χωρίς τη χρήση ουρών.

1.2. Πλάνο – Δομή της εργασίας

Στο **πρώτο κεφάλαιο**, ορίζεται το υπό εξέταση πρόβλημα. Επισημαίνονται θεμελιώδεις έννοιες που συμβάλλουν στην κατανόηση του προβλήματος, παρουσιάζονται οι σημαντικότεροι λόγοι που προσέδωσαν το απαραίτητο κίνητρο για τη μελέτη του συγκεκριμένου ζητήματος και αναφέρεται συνοπτικά η δομή της εργασίας και το πλάνο το οποίο θα ακολουθηθεί για την επίτευξη των στόχων που τέθηκαν.

Στο **δεύτερο κεφάλαιο**, αναλύεται το τεχνολογικό υπόβαθρο που απαιτείται προκειμένου να διασφαλιστεί η ορθότητα των πρακτικών που θα εφαρμοστούν. Αρχικά, παρατίθεται η έννοια του solver ως ένα λογισμικό μαθηματικής βελτιστοποίησης. Στη συνέχεια, περιγράφεται ο ρόλος ενός API στην ανταλλαγή πληροφοριών μεταξύ διαφόρων υπολογιστικών προγραμμάτων και παρουσιάζονται οι βασικές κατηγορίες του. Έπειτα, γίνεται αναφορά στους message brokers και παρέχονται πληροφορίες για το διαδεδομένο broker RabbitMQ που θα χρησιμοποιηθεί στην παρούσα εργασία. Τέλος, επισημαίνονται τα σημαντικότερα γνωρίσματα του .NET οικοσυστήματος, το οποίο θα φιλοξενήσει όλον τον πηγαίο κώδικα του λογισμικού και συγκρίνονται συνοπτικά τα Windows με τα Linux.

Στο **τρίτο κεφάλαιο**, πραγματοποιείται η μελέτη περίπτωσης, όπου επιλέγονται δεδομένα για προβλήματα Επιχειρησιακής Έρευνας που θα επιλυθούν μέσω της βιβλιοθήκης Google OR-Tools. Παρουσιάζονται οι παράμετροι που θα εφαρμοστούν, τα εξεταζόμενα σενάρια και τα δομικά στοιχεία του λογισμικού στο σύνολό του. Ακόμη, αναλύονται εκτενώς οι αρχιτεκτονικές που θα διερευνηθούν και συγκρίνονται τα αποτελέσματά τους.

Στο **τέταρτο κεφάλαιο**, επισημαίνονται τα συμπεράσματα που προκύπτουν από την επιτέλεση της διαδικασίας. Παράλληλα, προτείνονται ορισμένες τεχνικές βελτίωσης της αποδοτικότητας του μοντέλου που κατασκευάστηκε, όπως το load balancing και η πλήρης υλοποίηση του message broker σε Cloud υποδομή.

2. Τεχνολογικό υπόβαθρο

2.1. Solvers

2.1.1. Ορισμός – Ιστορική αναδρομή

Ως solver, ή εναλλακτικά επιλύτης, ορίζεται ένα κομμάτι μαθηματικού λογισμικού, κύριος σκοπός του οποίου είναι η επίλυση μαθηματικών προβλημάτων. Ένας solver λαμβάνει περιγραφές των εκάστοτε προβλημάτων σε κάποιου είδους γενική μορφή και πραγματοποιεί τις κατάλληλες μαθηματικές διαδικασίες προκειμένου να υπολογίσει τη λύση τους. Καθώς συχνά βρίσκεται στη μορφή ενός *stand-alone* (αυτόνομο) υπολογιστικού προγράμματος ή μίας *software library*, δηλαδή μίας συλλογής από resources, το σημαντικότερο ζήτημα κατά την κατασκευή του solver είναι η ευκολία με την οποία θα μπορούν να εφαρμοστούν οι μέθοδοι που περιλαμβάνει σε προβλήματα παρόμοιου τύπου. Το σύστημα πρέπει να μην περιορίζεται σε μία αυστηρά καθορισμένη κατηγορία προβλημάτων, αλλά να μπορεί να γενικευτεί σε ικανοποιητικό βαθμό, με τις ελάχιστες δυνατές τροποποιήσεις και επεκτάσεις [1].

Η απαρχή των επιλυτών στην Πληροφορική σηματοδοτήθηκε με την κατασκευή του *General Problem Solver (GPS)* από τους Herbert Simon, J.C. Shaw και Allen Newell, το 1957. Το GPS είναι ένα υπολογιστικό πρόγραμμα το οποίο δημιουργήθηκε με απώτερο σκοπό να λειτουργήσει ως ένας καθολικός επιλύτης προβλημάτων, θεωρητικά ικανός να επιλύσει κάθε πιθανό υπολογιστικό πρόβλημα το οποίο μπορούσε να γραφεί σε ένα *συμβολικό σύστημα*, δοθείσας των απαραίτητων παραμέτρων εισόδου (input). Έτσι, αποτέλεσε το πρώτο υπολογιστικό πρόγραμμα το οποίο διαχώρισε έμπρακτα τη *γνώση* των προβλημάτων στη μορφή κανόνων, από τη *στρατηγική* που εφαρμόζεται για την επίλυση των προβλημάτων, ως μία γενικευμένη μηχανή αναζήτησης.

Σήμερα, οι γενικοί solvers χρησιμοποιούν μία αρχιτεκτονική παρόμοια με εκείνη του GPS προκειμένου να αποσυνδέσουν τον ορισμό ενός προβλήματος από τη στρατηγική επίλυσής του. Το γεγονός αυτό *(decoupling)* διευκολύνει σημαντικά τη διαδικασία, καθώς ο solver δεν εξαρτάται από τις λεπτομέρειες του συγκεκριμένου instance προβλήματος, αλλά βασίζεται σε κάποιον γενικό αλγόριθμο που καλύπτει μία ή περισσότερες κατηγορίες προβλημάτων και έχει ως μοναδικό στόχο την ολοκλήρωση της επίλυσης. Ωστόσο, η γενικευμένη αυτή υλοποίηση επιφέρει εκθετικό υπολογιστικό χρόνο, περιορίζοντας πολύ τη χρησιμότητά τους. Για το λόγο αυτό, οι μοντέρνοι solvers συνήθως ακολουθούν μία πιο εξειδικευμένη προσέγγιση η οποία εκμεταλλεύεται τη δομή των προβλημάτων για την εξαγωγή συμπερασμάτων που οδηγούν στη μείωση του υπολογιστικού χρόνου της επίλυσης.

Ενδεικτικά, παρατίθενται μερικοί από τους πιο διαδεδομένους τύπους προβλημάτων οι οποίοι διαθέτουν υπάρχοντες solvers:

- Γραμμικές και μη-γραμμικές εξισώσεις
- Γραμμικά και μη-γραμμικά προβλήματα βελτιστοποίησης
- Συστήματα συνήθων διαφορικών εξισώσεων

- Συστήματα διαφορικών αλγεβρικών εξισώσεων
- Προβλήματα *λογικής* (logic) και *ικανοποιησιμότητας* (satisfiability)
- Προβλήματα *ικανοποίησης περιορισμών* (constraint satisfaction)
- Προβλήματα *συντομότερου μονοπατιού* (shortest path)
- Προβλήματα *ελαχίστου επικαλυπτόμενου δέντρου* (Minimum Spanning Tree – MST)

Στην τρέχουσα εργασία, θα διερευνηθεί η επίλυση προβλημάτων τα οποία ανήκουν στον κλάδο της Επιχειρησιακής Έρευνας. Έτσι, από τις ανωτέρω κατηγορίες προβλημάτων, θα μελετηθούν κυρίως προβλήματα συντομότερου μονοπατιού, και συγκεκριμένα το ευρέως γνωστό πρόβλημα δρομολόγησης οχημάτων σε ένα οδικό δίκτυο (Vehicle Routing Problem – VRP). Επιπλέον, τα παραδείγματα solvers από τον πραγματικό κόσμο που θα αναφερθούν στη συνέχεια, θα σχετίζονται μόνο με τα προβλήματα του Operational Research, ώστε να μείνουν ευθυγραμμισμένοι με το σκοπό της εργασίας.

2.1.2. Τυπικοί solvers

Ο τομέας της Επιχειρησιακής Έρευνας παρουσιάζει μεγάλη γκάμα προβλημάτων και παραλλαγών τους, καλύπτοντας ένα ευρύ φάσμα πολλών σεναρίων μελέτης τα οποία διαθέτουν ρεαλισμό και κατ' επέκταση ιδιαίτερη σημασία στην αγορά εργασίας. Επομένως, έχουν αναπτυχθεί πολλά υπολογιστικά συστήματα, τα οποία επιτυγχάνουν την αποδοτική επίλυση προβλημάτων βελτιστοποίησης, ικανοποιησιμότητας κ.α., με αποτέλεσμα να έχουν καθιερωθεί ως οι επικρατέστερες επιλογές για ενδιαφερόμενους στην επίλυση προβλημάτων Επιχειρησιακής Έρευνας. Συγκεκριμένα, θα παρουσιαστούν δύο εκ των δημοφιλέστερων computer solver προγραμμάτων: το GAMS και το Gurobi.

1. **Generic Algebraic Modeling System (GAMS)**

   Το GAMS αποτελεί ένα high-level σύστημα μοντελοποίησης για μαθηματική βελτιστοποίηση. Έχοντας εκδοθεί ως εμπορικό προϊόν από το 1987, πρόκειται για το πρώτο λογισμικό σύστημα που συνδυάζει τη γλώσσα της μαθηματικής άλγεβρας με παραδοσιακές έννοιες του υπολογιστικού προγραμματισμού, προκειμένου να περιγράψει και να επιλύσει προβλήματα βελτιστοποίησης με αποδοτικό τρόπο. Ειδικότερα, το σύστημα ταιριάζει ιδιαίτερα σε περίπλοκες εφαρμογές μοντελοποίησης μεγάλης κλίμακας, ενώ παράλληλα επιτρέπει στο χρήστη να κατασκευάσει μεγάλα, επαναχρησιμοποιούμενα μοντέλα τα οποία μπορούν να προσαρμοστούν σε νέες καταστάσεις. Ακόμη, καθώς το GAMS είναι διαθέσιμο σε περισσότερες από 10 υπολογιστικές πλατφόρμες, όλα τα σχεδιαζόμενα μοντέλα μπορούν να μεταφερθούν από μία πλατφόρμα σε μία άλλη. Σε ότι αφορά τις λειτουργικότητές του, επιτρέπει στους χρήστες να υλοποιούν ένα είδος υβριδικού αλγορίθμου που συνδυάζει διαφορετικούς επιλύτες. Διαθέτει περισσότερους από 25 ενσωματωμένους solvers, ενώ ταυτόχρονα περιλαμβάνει παραπάνω από 10 υποστηριζόμενες κλάσεις μοντέλων για την καλύτερη δυνατή αντιμετώπιση όσο το δυνατών περισσότερων προβλημάτων [2]. Τα μοντέλα περιγράφονται με συνοπτικές, ευανάγνωστες αλγεβρικές δηλώσεις, γεγονός που καθιστά το GAMS μία από τις δημοφιλέστερες μορφές εισόδου (input formats) για το NEOS server, μία

εφαρμογή πελάτη-διακομιστή που παρέχει δωρεάν πρόσβαση σε μία βιβλιοθήκη από solvers βελτιστοποίησης. Παρότι αρχικά σχεδιάστηκε για εφαρμογές που σχετίζονται με την οικονομία και τις επιστήμες διαχείρισης, έχει μια κοινότητα χρηστών από διάφορα υπόβαθρα της μηχανικής και της επιστήμης.

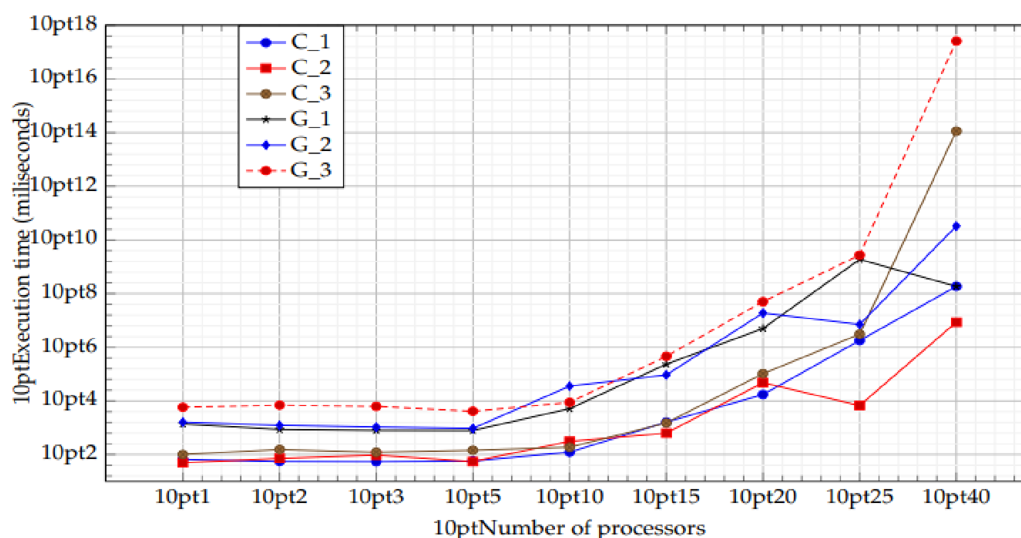
Ο βασικός λόγος που συνετέλεσε στην ανάπτυξη του GAMS ήταν η δυσκολία μοντελοποίησης των προβλημάτων βελτιστοποίησης με τις τότε καθιερωμένες μεθόδους. Οι τεχνικές για την κατασκευή και το χειρισμό τέτοιων μοντέλων απαιτούσε πολλές χειροκίνητες, χρονοβόρες και επιρρεπείς σε λάθη μεταφράσεις σε διαφορετικές αναπαραστάσεις, εξατομικευμένες αποκλειστικά για το συγκεκριμένο πρόβλημα. Έτσι, προέκυψε η ιδέα μίας αλγεβρικής προσέγγισης για την αναπαράσταση και επίλυση μαθηματικών μοντέλων μεγάλης κλίμακας, μεταφέροντας τα σε ένα σταθερό και υπολογιστικά διαχειρίσιμο σύστημα όπου εκτελούνται οι απαραίτητοι αλγόριθμοι, καθώς και διάφορες, ακριβείς μαθηματικές ενέργειες, όπως διαφόριση και πολυδιάστατη αλγεβρική αναπαράσταση. Το GAMS αποτέλεσε την πρώτη *γλώσσα αλγεβρικής μοντελοποίησης* (algebraic modeling language – AML), και το formality της είναι παρόμοιο με τις γλώσσες προγραμματισμού τέταρτης γενιάς, όπως η SQL [3].

2. Gurobi Optimizer

Η Gurobi Optimizer, συχνά αποκαλούμενη απλώς ως Gurobi, αποτελεί μία πλατφόρμα *καθοδηγητικής αναλυτικής (prescriptive analytics)*, δηλαδή συμβάλλει στη λήψη αποφάσεων στον επιχειρηματικό κόσμο. Είναι μεταγενέστερη του GAMS, καθώς ιδρύθηκε μόλις το 2008 από τος διδάκτορες Zonghao Gu, Edward Rothberg και Robert Bixby. Πρόκειται για έναν μοντέρνο solver, ο οποίος και αυτός εφαρμόζει τεχνικές μαθηματικής βελτιστοποίησης για την εύρεση της βέλτιστης δυνατής απόφασης υπό την επίδραση πολλών και πολύπλοκων παραγόντων. Χρησιμοποιείται ευρέως για την επίλυση προβλημάτων γραμμικού προγραμματισμού (linear programming), τετραγωνικού προγραμματισμού (quadratic programming) προγραμματισμού περιορισμών (constraint programming) και προγραμματισμού με μεικτούς ακεραίους (mixed-integer programming) [4].

Ως λογισμικό που επικεντρώνεται σε επιχειρηματικά δεδομένα μεγάλης κλίμακας, το Gurobi δίνει έμφαση στην ταχύτητα των υπολογισμών του. Αξιοποιεί πληθώρα ισχυρών αλγορίθμων και επιτρέπει την προσθήκη πολυπλοκότητας σε ένα υπάρχον μοντέλο ώστε να αναπαραστήσει καλύτερα τον πραγματικό κόσμο, δίχως μεγάλη χρονική επιβάρυνση της επίλυσής του. Συγκεκριμένα, προσφέρει υψηλό scalability, με την αποτελεσματικότητα του μοντέλου να αυξάνεται όσο το μέγεθος του μοντέλου και η δυσκολία στην επίλυσή του μεγαλώνουν. Επιπροσθέτως, το Gurobi είναι ρυθμισμένο ώστε να βελτιστοποιεί την απόδοση του σε ένα ευρύ φάσμα από instances, γεγονός που το καθιστά αξιόπιστο και ταχύ solver για real-case σενάρια λήψης αποφάσεων, όπου συχνά παρατηρούνται πολύπλοκες σχέσεις μεταξύ των οντοτήτων που συμμετέχουν και κάθε δυνατή απόφαση ενδέχεται να επιφέρει μία σειρά από σημαντικές επιπτώσεις [5].

Όπως είναι λογικό, υπάρχουν και άλλοι commercial solvers για προβλήματα Γραμμικού Προγραμματισμού και Βελτιστοποίησης, οι υπηρεσίες των οποίων παρέχονται επί πληρωμή. Ενδεικτικά, ορισμένοι γνωστοί εξ αυτών είναι το CPLEX, ένα λογισμικό βελτιστοποίησης εξειδικευμένο και αυτό σε large-scale προβλήματα που αναπτύχθηκε το 1988 και ανήκει στην IBM από το 2009, το MATLAB, γνωστή γλώσσα προγραμματισμού γενικού σκοπού η οποία διαθέτει optimization toolbox και η AIMMS, μία εταιρεία αντίστοιχη της Gurobi, η οποία πρωτοεμφανίστηκε το 1993 και επικεντρώνεται στα prescriptive analytics. Στη συνέχεια, παρατίθεται ένα διάγραμμα που συγκρίνει τον υπολογιστικό χρόνο επίλυσης ενός προβλήματος Γραμμικού Προγραμματισμού χρησιμοποιώντας πολλαπλούς επεξεργαστές, αξιοποιώντας τους solvers CPLEX και Gurobi.



Εικόνα 1. Σύγκριση απόδοσης των επιλυτών CPLEX και Gurobi. Τα υπολογιζόμενα προβλήματα είναι: Size 1 με CPLEX (C_1), Size 2 με CPLEX(C_2), Size 3 με CPLEX (C_3) και αντίστοιχα Size 1 με Gurobi (G_1), Size 2 με Gurobi (G_2), Size 3 με Gurobi (G_3).

2.2. Application Programming Interface (API)

2.2.1. Ορισμός

Ως διεπαφή προγραμματισμού εφαρμογών, ευρέως γνωστό ως API, ορίζεται ο τρόπος επικοινωνίας μεταξύ δύο ή περισσότερων υπολογιστικών προγραμμάτων. Ένα API καθορίζει πώς ο προγραμματιστής καλείται να αιτηθεί υπηρεσίες από ένα λειτουργικό σύστημα ή μία άλλη εφαρμογή, καθώς και πως να εκθέσει τα δεδομένα ώστε αυτά να συνάδουν με τις αρχές λειτουργίας του. Σε αντίθεση με τη διεπαφή του χρήστη (UI – User Interface), το API δε συνδέει έναν υπολογιστή με ένα πρόσωπο, αλλά αυθεντικοποιεί και παρέχει πρόσβαση στα δεδομένα τα οποία αιτείται ο χρήστης ή μία εφαρμογή, συνδέοντας έτσι διαφορετικούς υπολογιστές ή κομμάτια λογισμικού μεταξύ τους. Για το λόγο αυτό, δεν προορίζεται να έλθει σε επαφή με τον τελικό χρήστη του εκάστοτε υπολογιστικού

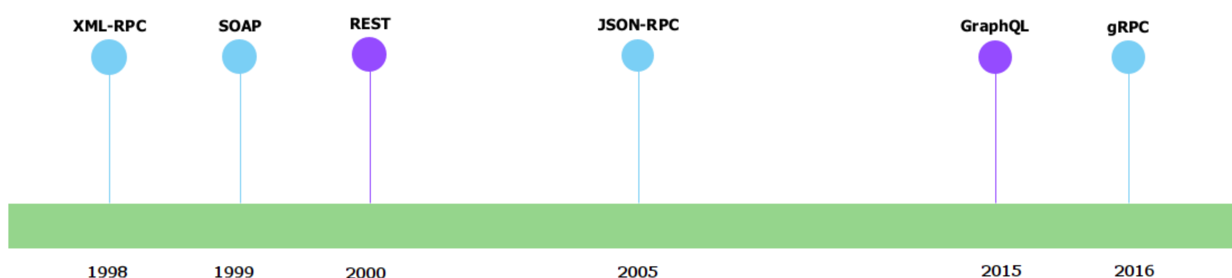
συστήματος, παρά μόνο με τον προγραμματιστή που καλείται να το ενσωματώσει στο λογισμικό [6].

Τα APIs αποτελούνται από δύο, σχετικά μεταξύ τους, χαρακτηριστικά:

- Μια *προδιαγραφή (specification)* που περιγράφει τον τρόπο με τον οποίο η πληροφορία ανταλλάσσεται μεταξύ των προγραμμάτων με τη μορφή ενός αιτήματος προς επεξεργασία και την επιστροφή των απαραίτητων δεδομένων.
- Μια *διεπαφή λογισμικού (software interface)*, γραμμένη βάσει του καθορισμένου specification και κοινοποιημένης, με κάποιον τρόπο, προς χρήση.

Ένα API μπορεί να θεωρηθεί ως «συμβόλαιο υπηρεσιών» ανάμεσά σε δύο εφαρμογές, οι οποίες επικοινωνούν μεταξύ τους χρησιμοποιώντας αιτήματα (requests) και απαντήσεις (responses). Η αρχιτεκτονική των APIs συνήθως αντιπροσωπεύεται με τους όρους client και server: η εφαρμογή που στέλνει το αίτημα αποκαλείται client, ενώ η εφαρμογή που στέλνει την απάντηση αποκαλείται server.

Σήμερα, η ταχεία εξάπλωση του SaaS μοντέλου στο διαδίκτυο, καθιστά επιτακτική την ανάγκη δημιουργίας ολοένα και περισσότερων APIs βασισμένα στη συνδεσιμότητα στο διαδίκτυο (web APIs). Οι αρχιτεκτονικές τις οποίες υλοποιούν οι διεπαφές προγραμματισμού εφαρμογών διακρίνονται σε δύο ευρύτερες κατηγορίες: τη REST και τις υπόλοιπες non-REST αρχιτεκτονικές που ακολουθούν διαφορετικές δομικές αρχές. Στη συνέχεια, θα αναλυθούν τα χαρακτηριστικά και οι σημαντικότερες διαφορές τους.



Εικόνα 2. Το timeline της εμφάνισης των διάφορων μορφών APIs. Παρακάτω θα αναλυθούν οι RPC, gRPC και REST κατηγορίες [7].

2.2.2. RESTful APIs

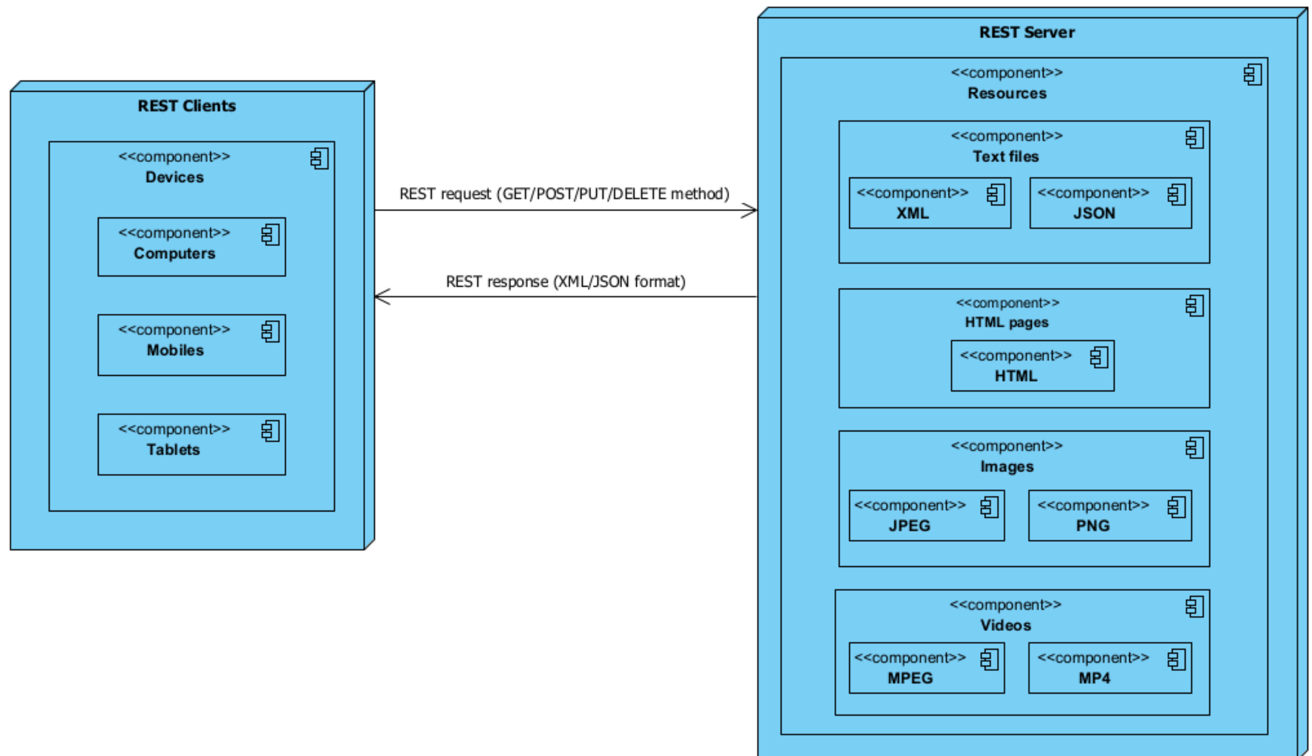
Ο όρος REST αποτελεί ακρωνύμιο για το "Representational State Transfer" και συνιστά αρχιτεκτονικό στυλ για κατανεμημένα συστήματα υπερμέσων (distributed hypermedia systems). Παρουσιάστηκε για πρώτη φορά το 2000 από τον Αμερικανό επιστήμονα υπολογιστών Roy Fielding και σημείωσε σε σύντομο χρονικό διάστημα αξιοσημείωτη επιτυχία, αποτελώντας πλέον την σπουδαιότερη αρχιτεκτονική αρχή για την ανάπτυξη διαδικτυακών υπηρεσιών [8].

Ως RESTful API ορίζεται ένα web API ή service το οποίο συμμορφώνεται με τους κανόνες και περιορισμούς που επιτάσσει η αρχιτεκτονική REST. Οι κυριότεροι εξ αυτών είναι οι ακόλουθοι:

- **Uniform interface:** Εφαρμόζοντας την αρχή της γενικότητας στη διεπαφή των δομικών στοιχείων του λογισμικού, καθίσταται δυνατή η απλοποίηση της συνολικής αρχιτεκτονικής του συστήματος και η βελτίωση των αλληλεπιδράσεων, συμβάλλοντας στη δημιουργία ενός ομοιογενούς REST interface.
- **Client - Server:** Το πρότυπο σχεδίασης client-server επιβάλλει το διαχωρισμό των ευθυνών στη διεπαφή του χρήστη από εκείνους στην αποθήκευση των δεδομένων, ο οποίος στην Μηχανική Λογισμικού αποδίδεται με τον όρο *Separation of Concerns*. Έτσι, ο πελάτης και ο διακομιστής μπορούν να εξελίσσονται και να λειτουργούν ανεξάρτητα, αρκεί το συμβόλαιο μεταξύ τους να μην «αλλοιωθεί». Το γεγονός αυτό επιτρέπει ευκολότερο διαμοιρασμό του user interface σε πολλές πλατφόρμες και βελτιώνει την κλιμακωσιμότητα μέσω της απλοποίησης των συστατικών του server.
- **Stateless:** Με τον όρο απουσία κατάστασης *(statelessness)* σε μία client-server αρχιτεκτονική γίνεται αναφορά στις περιπτώσεις όπου ο client είναι υπεύθυνος για την αποθήκευση και τον χειρισμό όλων των πληροφοριών που σχετίζονται με την κατάσταση του session (session state), όπως τα tokens αυθεντικοποίησης, τα credentials στη δική του πλευρά. Αυτός ο περιορισμός συνεπάγεται πως κάθε HTTP request συμβαίνει σε πλήρη απομόνωση. Ο χρήστης περιλαμβάνει όλες τις απαραίτητες πληροφορίες για την κατανόηση και ολοκλήρωσή του, καθώς ο server δεν μπορεί να εκμεταλλευθεί καμία προηγουμένως αποθηκευμένη πληροφορία σε αυτόν.
- **Cacheable:** Κάθε απόκριση (response) του RESTful API οφείλει να επισημαίνει αν είναι *cacheable* ή *non-cacheable*. Στην πρώτη περίπτωση, ο χρήστης δικαιούται να επαναχρησιμοποιήσει τα δεδομένα της απόκρισης για ισοδύναμα αιτήματα εντός καθορισμένου χρονικού διαστήματος, γεγονός που δεν ισχύει στην δεύτερη περίπτωση.
- **Layered system:** Η αρχιτεκτονική αποτελείται από ιεραρχικά επίπεδα (layers), σε καθένα από τα οποία τα components δεν μπορούν να δουν πέρα από το αμέσως επόμενο επίπεδο με το οποίο αλληλεπιδρούν. Χαρακτηριστικό παράδειγμα layered system συνιστά το MVC (Model – View – Controller) μοντέλο, το οποίο επιτρέπει την ύπαρξη ξεκάθαρου separation of concerns.

Τα RESTful APIs επικοινωνούν μέσω HTTP requests προκειμένου να πραγματοποιήσουν τις standard functions μίας βάσης δεδομένων, όπως η δημιουργία, ανάγνωση, ενημέρωση και διαγραφή εγγραφών μέσα σε ένα resource, δηλαδή εντός της πληροφορίας που παρέχει ο server. Η κατάσταση ενός resource σε μία δεδομένη χρονική στιγμή, γνωστή ως resource representation, μπορεί να μεταφερθεί στον πελάτη σε formats

όπως JSON, HTML, XML ή ακόμη και απλού κειμένου (plain-text) [9].



Εικόνα 3. Σχηματική αναπαράσταση ενός RESTful API σε λειτουργία.

2.2.3. Non-RESTful APIs

Τα APIs τα οποία δεν ακολουθούν τους αρχιτεκτονικούς περιορισμούς του REST, ανήκουν στην γενικότερη κατηγορία των non-RESTful APIs. Η επικρατέστερη διεπαφή προγραμματισμού εφαρμογών που υπάγεται σε αυτήν την κατηγορία είναι το SOAP API, το οποίο παρουσιάστηκε το 1998 και αποτελεί μια από τις απλούστερες μορφές API. Σε αντίθεση με το REST, το SOAP (Simple Object Access Protocol) δε συνιστά αρχιτεκτονικό στυλ αλλά πρωτόκολλο, το οποίο μεταδίδει XML αρχεία, συνήθως μέσω HTTP/HTTPS αιτημάτων, αλλά ακόμη και με SMTP, TCP και UDP. Ουσιαστικά, πρόκειται για έναν απλό μηχανισμό μετάδοσης, καθώς ενσωματώνει αντικείμενα σε ένα XML αρχείο, καθιστώντας τη δημιουργία του εύκολη. Ωστόσο, η απλότητά του έχει ως συνέπεια την δυσκολία ανεύρεσης πιθανών λαθών. Σήμερα, τα SOAP συστήματα χρησιμοποιούνται κυρίως από τα ηλεκτρονικά ταχυδρομεία [10].

Άλλες μορφές APIs που δεν ακολουθούν τη REST αρχιτεκτονική, είναι το πρωτόκολλο επικοινωνίας WebSocket, το οποίο επιτρέπει την ασύγχρονη αποστολή και λήψη δεδομένων δίχως την ανάγκη καθιέρωσης νέας σύνδεσης και η Remote Procedure Call (RPC), η οποία θα αναλυθεί εκτενώς στο επόμενο υποκεφάλαιο, καθώς η χρήση της επεκτείνεται πέραν των APIs.

2.2.4. Stateful vs Stateless API

Μέχρι πριν μερικές δεκαετίες οι πελάτες και οι τελικοί χρήστες ήταν «στενοί», δηλαδή η κατείχαν περιορισμένο όγκο hardware και software, με συνέπεια το μεγαλύτερο μέρος του φόρτου εργασίας να διενεργείται από πολύ ισχυρούς servers, μεταβιβάζοντας έτσι την ανάληψη της ευθύνης των πολύπλοκων και υπολογιστικά ακριβών διαδικασιών στην πλευρά του διακομιστή (server-side). Ωστόσο, η ταχεία διεύρυνση του Παγκόσμιου Ιστού και η επακόλουθη ανάπτυξη του HTTP σηματοδότησε την ανάγκη εκτέλεσης πολλών services από τους τελικούς χρήστες, οι οποίοι πλέον έγιναν «ευρείς». Ως απόρροια της ιδιαίτερα αυξημένης επικοινωνίας των χρηστών με εφαρμογές μέσω του Διαδικτύου, προέκυψε η ανάγκη αποθήκευσης των αλληλεπιδράσεων τους ώστε να εξάγονται συμπεράσματα για την εκάστοτε κατάσταση τους (state). Η ομαδοποίηση των αλληλεπιδράσεων αυτών συνήθως αποδίδεται με τον όρο «session», ενώ το σύνολο των κανόνων οι οποίοι καθορίζουν τον τρόπο απόκτησης και διαχείρισης των πληροφοριών από τα υπολογιστικά εργαλεία ονομάζονται «δικτυακά πρωτόκολλα» (network protocols) [11].

Τα πρωτόκολλα δικτύου διακρίνονται, ανάλογα με το πόσα δεδομένα των χρηστών και των αλληλεπιδράσεων τους αποθηκεύονται στο server, σε δύο κατηγορίες: με κατάσταση (stateful) και χωρίς κατάσταση (stateless). Κατ' αντιστοιχία με τα network protocols, τα APIs κατηγοριοποιούνται επίσης σε stateless και stateful, ανάλογα με το είδος των πρωτοκόλλων που αξιοποιούν ή γενικότερα τον τρόπο που ορίζουν την επικοινωνία ανάμεσα στον πελάτη και το διακομιστή. Παρακάτω παρουσιάζονται αναλυτικά οι κύριες διαφορές των δύο μεθόδων:

- Stateful APIs: Όπως υποδηλώνει το όνομά τους, τα stateful APIs συντηρούν μία εγγραφή της τρέχουσας κατάστασης των αλληλεπιδράσεων του πελάτη. Συγκεκριμένα, κατά την πρώτη αυθεντικοποίηση και σύνδεση του χρήστη με την εφαρμογή, ο server αποθηκεύει τα credentials του και τα διατηρεί καθ' όλη τη διάρκεια του session που έχει εγκαθιδρύσει ο χρήστης. Έτσι, σε μελλοντικά requests από την πλευρά του client, ο server γνωρίζει εκ των προτέρων τόσο την ταυτότητά του όσο και το state του, οπότε ο χρήστης δε χρειάζεται να αποστείλει εκ νέου καμία πληροφορία πέραν των δεδομένων του αιτήματος. Αυτή η persistent σύνδεση ανάμεσα σε πελάτη και διακομιστή διευκολύνει την ομαλή ανταλλαγή πληροφοριών, καθιστώντας το stateful API ιδανικό για εφαρμογές που απαιτούν συνεχή επικοινωνία με τον κάθε χρήστη [12].

- Stateless APIs: Σε αντίθεση με τη stateful προοπτική, τα stateless APIs λειτουργούν δίχως να επιβαρύνονται με την απομνημόνευση προηγούμενων αλληλεπιδράσεων. Κάθε request από τον πελάτη προς το διακομιστή είναι ανεξάρτητο και αυτάρκες. Αντίστοιχα, ο server αντιμετωπίζει κάθε request ως μία καινούρια, απομονωμένη συναλλαγή (transaction), στερούμενος οποιαδήποτε γνώση για τις πρότερες αλληλεπιδράσεις. Τα stateless APIs αποτελούν, υπό μία έννοια, «προσωρινούς αγγελιαφόρους», καθώς μεταφέρουν ένα μήνυμα, δηλαδή το αίτημα, ο διακομιστής το επεξεργάζεται κατάλληλα, και η επικοινωνία ολοκληρώνεται με την επιστροφή του αποτελέσματος σε αυτούς. Αυτό το design principle απλοποιεί τις αρμοδιότητες του server, υποστηρίζοντας παράλληλα το scalability της εφαρμογής σε περιπτώσεις μεγάλου φόρτου εργασίας [12].

Όπως γίνεται φανερό, η σπουδαιότερη διαφορά ανάμεσα στους δύο τρόπους δόμησης ενός API έγκειται στον αποθηκευτικό χώρο στην πλευρά του server: τα stateful πρωτόκολλα εξαρτώνται από το server-side storage για να διατηρούν το state του χρήστη, ενώ τα stateless πρωτόκολλα δεν έχουν καμία ανάγκη αποθήκευσης τέτοιου είδους πληροφοριών στο διακομιστή. Ένα από τα σημαντικότερα πλεονεκτήματα των stateless APIs που συνέβαλε σημαντικά στην καθιέρωσή τους, είναι η δυνατότητα κλιμακωσιμότητας. Ειδικότερα, το γεγονός πως κάθε request είναι ανεξάρτητο επιτρέπει στους servers να επεξεργαστούν αιτήματα σε παραλληλία χωρίς πρόβλημα, ακόμα και αν ο αριθμός των clients είναι ιδιαίτερα υψηλός. Αντίθετα, τα stateful APIs είναι λιγότερο scalable διότι σε ένα σενάριο παράλληλης επεξεργασίας ο διακομιστής πρέπει να διαχειριστεί και να συγχρονίσει το state σε πολλαπλά instances, με αποτέλεσμα αφενός την αύξηση της πολυπλοκότητας και αφετέρου την καθυστέρηση στην απόκριση του server. Η ανεξαρτησία του κάθε αιτήματος στα stateless APIs οδηγεί επίσης στην εγγενή υπεροχή τους έναντι των stateful APIs στην αντιμετώπιση πιθανών σφαλμάτων. Αν ένας server που ακολουθεί το stateless πρωτόκολλο αποτύχει, μπορεί ανεμπόδιστα να αναλάβει την επεξεργασία των αιτημάτων κάποιος άλλος, επιτρέποντας έτσι στους πελάτες να συνεχίσουν απρόσκοπτα την αποστολή requests. Η ελευθερία διανομής των εργασιών στους servers είναι ανέφικτη στα stateful APIs, με συνέπεια να απαιτούνται πρόσθετα μέτρα για την αντιμετώπιση σφαλμάτων, όπως η κατασκευή *μηχανισμών ανακατεύθυνσης* (failover mechanisms). Επιπλέον, η ανάγκη αποθήκευσης δεδομένων σχετικά με το session κάθε client καθιστά τη stateful αρχιτεκτονική υπολογιστικά ακριβότερη από την stateless αρχιτεκτονική [12].

2.2.5. Remote Procedure Call (RPC)

Ως κλήση απομακρυσμένης διαδικασίας (remote procedure call), ευρέως διαδεδομένη ως RPC, ορίζεται ένα πρωτόκολλο επικοινωνίας λογισμικού, το οποίο επιτρέπει σε ένα πρόγραμμα να αιτηθεί μία λειτουργία σε ένα απομακρυσμένο δίκτυο, δίχως να απαιτείται να γνωρίζει λεπτομέρειες του δικτύου αυτού. Έτσι, επιτυγχάνεται η κλήση διαδικασιών σε remote συστήματα όπως θα γίνονταν σε ένα τοπικό σύστημα [13].

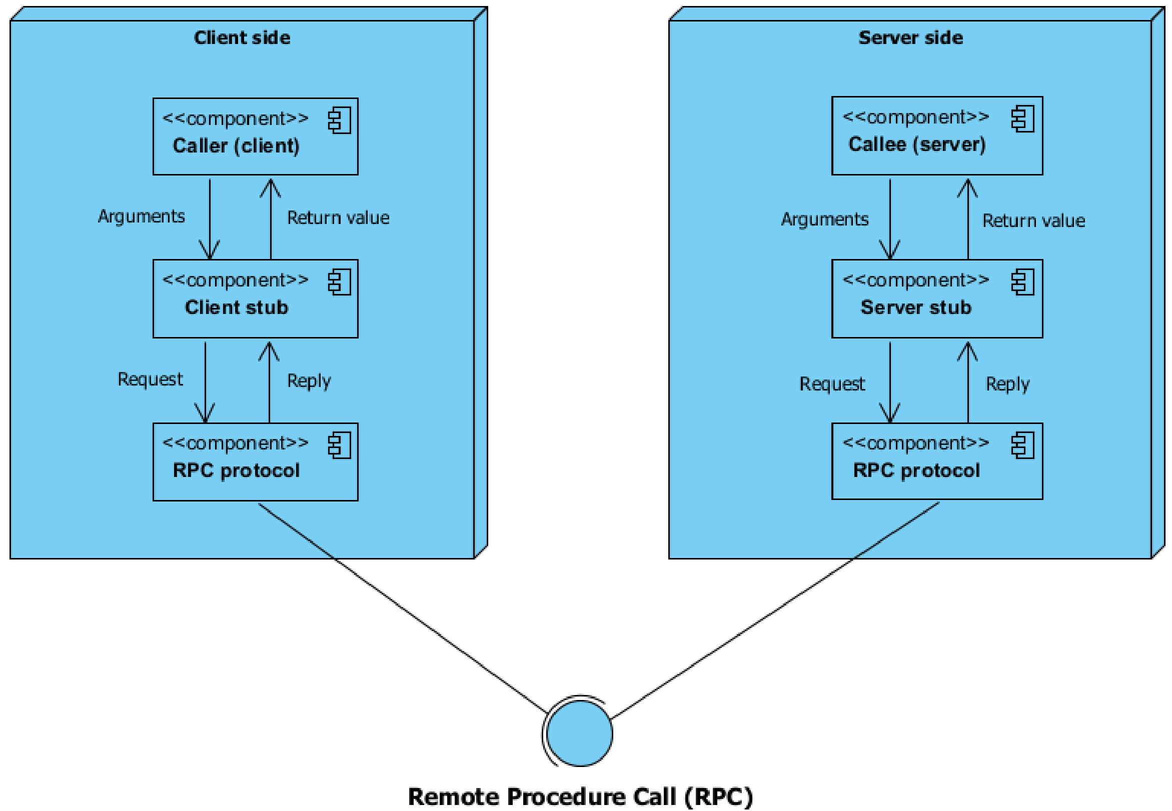
Τα πρωτόκολλα αίτησης – απόκρισης (request – response protocols) εμφανίζονται ήδη στα τέλη της δεκαετίας του 1960, ενώ στην αρχή της δεκαετίας του 1970 αναπτύσσονται οι ιδέες που σταδιακά θα δομήσουν τη βάση για τη δημιουργία των RPCs. Ο όρος εισάγεται και ξεκινά να χρησιμοποιείται τη δεκαετία του 1980, όταν και σχεδιάζονται τα πρώτα πρωτόκολλα κλήσεων σε απομακρυσμένα μηχανήματα ειδικά για RPC, όπως το Xerox Network Systems RPC και το Network File System (NFS). Αργότερα στη δεκαετία του 1990, όπου η δημοφιλία των αντικειμενοστρεφών γλωσσών προγραμματισμού παρουσίασε μεγάλη άνοδο, οι απομακρυσμένες κλήσεις διαδικασίας αρχίζουν να χρησιμοποιούνται ευρέως σε enterprise εφαρμογές, ενώ από το 2000 και έπειτα αξιοποιήθηκαν οι πρώτες RPC τεχνολογίες βασισμένες στο διαδίκτυο, εκμεταλλευόμενες συνήθως το πρωτοεμφανιζόμενο Πρωτόκολλο Μεταφοράς Υπερκειμένου (HTTP) ως το επίπεδο μεταφοράς δεδομένων. Χαρακτηριστικό παράδειγμα αποτελεί το πρωτόκολλο SOAP, το οποίο τεχνικά αποτελεί παράδειγμα Απομακρυσμένης Κλήσης Διαδικασίας, καθώς επιτρέπει την επικοινωνία διαφορετικών υπηρεσιών πάνω στο HTTP μέσω της χρήσης XML format. Μέχρι και σήμερα, παρά την γενικότερη κυριαρχία των RESTful APIs στο πεδίο του web development, οι αλληλεπιδράσεις τύπου RPC εξακολουθούν να

είναι συχνές, ιδίως σε εφαρμογές όπου η απόδοση ανάγεται σε ζήτημα κριτικής σημασίας και σε αρχιτεκτονικές microservices [14].

Η τεχνική του RPC αξιοποιεί το client-server μοντέλο. Το πρόγραμμα που δημιουργεί το request είναι ο client, ενώ το πρόγραμμα που παρέχει την υπηρεσία και το αποτέλεσμα αυτής, αποτελεί τον server. Προκειμένου να αποκτηθεί πρόσβαση τόσο από τον πελάτη προς το διακομιστή όσο και αντιστρόφως, χρησιμοποιούνται δύο stubs, ένα για το κάθε μέρος. Το stub αποτελεί ένα πρόγραμμα το οποίο λειτουργεί ως προσωρινή αντικατάσταση για μία απομακρυσμένη υπηρεσία ή αντικείμενο, κρύβοντας τις λεπτομέρειες της υποβόσκουσας δικτυακής επικοινωνίας. Τα βήματα που απαρτίζουν μια απομακρυσμένη κλήση διαδικασίας, φαίνονται συνοπτικά παρακάτω:

1. Ο client καλεί το client stub. Η κλήση είναι ένα τοπικό procedure call με τις παραμέτρους να προωθούνται στο stack, όπως συμβαίνει εξ ορισμού.
2. Το client stub ενσωματώνει τις παραμέτρους τις διαδικασίας σε ένα μήνυμα και πραγματοποιεί μία κλήση συστήματος για να το αποστείλει. Η διαδικασία «πακεταρίσματος» των παραμέτρων διαδικασίας ονομάζεται *marshalling*.
3. Το τοπικό λειτουργικό σύστημα του πελάτη στέλνει το μήνυμα από το δικό του μηχάνημα σε εκείνο του απομακρυσμένου διακομιστή.
4. Το λειτουργικό σύστημα του server προωθεί τα εισερχόμενα πακέτα στο server stub.
5. Ο server stub «ξεπακετάρει» τις παραμέτρους από το μήνυμα *(unmarshalling)*.
6. Όταν η διαδικασία του server ολοκληρωθεί, επιστρέφει στο server stub, το οποίο πραγματοποιεί marshalling για τις τιμές επιστροφής, δημιουργώντας ένα μήνυμα. Έπειτα, μεταφέρει το μήνυμα στο επίπεδο μεταφοράς (transport layer).
7. Το transport layer στέλνει το μήνυμα απάντησης πίσω στο αντίστοιχο layer του πελάτη, που με τη σειρά του το μεταφέρει στο client stub.
8. Το client stub διενεργεί unmarshalling των παραμέτρων επιστροφής, και η εκτέλεση επιστρέφει στον caller, δηλαδή τον πελάτη.

Σε αντιστοιχία με μία τοπική procedure call, το RPC συνιστά μία σύγχρονη (synchronous) διαδικασία, προϋποθέτοντας τη διακοπή των διεργασιών του client έως ότου επιστραφούν σε αυτόν τα αποτελέσματα της απομακρυσμένης κλήσης. Ωστόσο, η δυνατότητα εφαρμογής multi-threading τεχνικών, δίνει τη δυνατότητα ενεργοποίησης πολλαπλών RPCs, τα οποία λειτουργούν ταυτόχρονα και παράλληλα.



Εικόνα 4. Γενική λειτουργία μιας απομακρυσμένης κλήσης διαδικασίας [15].

Σε κάθε περίπτωση, τόσο το Remote Procedure Call όσο και το REST συνιστούν τις δύο κύριες αρχιτεκτονικές επιλογές για τη σχεδίαση ενός API, με τα χαρακτηριστικά στοιχεία του καθενός να το καθιστούν κατάλληλο σε συγκεκριμένες περιπτώσεις. Παρακάτω, παρουσιάζονται οι σημαντικότερες διαφορές των δύο αυτών μεθόδων δόμησης μίας διεπαφής προγραμματισμού εφαρμογών.

2.2.6. Σύγκριση REST με RPC API

Αν και τόσο η REST αρχιτεκτονική, όσο και η Remote Procedure Call εξυπηρετούν τον κοινό σκοπό διευκόλυνσης της επικοινωνίας μεταξύ δύο διαφορετικών εφαρμογών και υπηρεσιών, χρησιμοποιούν αρκετά διαφορετική προσέγγιση προκειμένου να τον επιτύχουν. Συγκεκριμένα, το REST αποτελεί μία stateless αρχιτεκτονική που βασίζεται στη θεμελίωση μιας σχέσης πελάτη με διακομιστή, όπου τα server-side δεδομένα γίνονται διαθέσιμα μέσω αναπαραστάσεών τους σε απλά formats, όπως είναι τα JSON και XML. Αντίθετα, όταν λαμβάνει χώρα μία Απομακρυσμένη Κλήση Διαδικασίας, το περιβάλλον κλήσης (calling environment) διακόπτεται προσωρινά, οι παράμετροι της διαδικασίας μεταφέρονται μέσω του δικτύου στο περιβάλλον όπου πρόκειται να εκτελεστεί, και κατόπιν η διαδικασία εκτελείται στο περιβάλλον αυτό. Όταν ολοκληρωθεί, τα αποτελέσματα επιστρέφονται πίσω στο calling environment, στο οποίο πλέον συνεχίζεται κανονικά η εκτέλεση των προγραμμάτων του, όπως θα συνέβαινε αν είχε επιστραφεί μία τοπική procedure call.

Ο ξεχωριστός τρόπος λειτουργίας των δύο αρχιτεκτονικών υποδεικνύει έμμεσα τη θεμελιώδη τους διαφορά: το REST επικεντρώνεται στους πόρους (resource-centric), ενώ το RPC είναι σχεδιασμένο για την τέλεση ενεργειών (actions). Έτσι, τα REST-based APIs είναι ιδανικά για τη μοντελοποίηση του domain, δηλαδή των οντοτήτων και των σχέσεων του προβλήματος, καθώς καθιστούν διαθέσιμες όλες τις CRUD εντολές και είναι ικανά να διαχειριστούν μεγάλες ποσότητες δεδομένων. Από την άλλη πλευρά, τα RPC-based APIs κρίνονται προτιμότερα για την εκτέλεση διαδικασιών και εντολών σε ένα απομακρυσμένο σύστημα. Παρατηρώντας λοιπόν πως η κάθε προσέγγιση λειτουργεί διαφορετικά και μάλιστα συχνά αποσκοπεί και στην εξυπηρέτηση διαφορετικών αναγκών, γίνεται εμφανής πως δεν είναι απαραίτητη η επιλογή αυστηρά μίας αρχιτεκτονικής, καθώς μπορούν να συνυπάρξουν στο ίδιο υπολογιστικό πρόγραμμα. Για παράδειγμα, μία εφαρμογή μπορεί να διαθέτει ένα ή περισσότερα APIs, αλλά και να εκτελεί κάποιου είδους υπηρεσίες οι οποίες θεωρούνται πως απαρτίζουν εξ ορισμού APIs. Στην περίπτωση αυτή, η οποία θα μελετηθεί και στην υλοποίηση του μοντέλου της παρούσας εργασίας, συχνά είναι προτιμότερο να γίνει συνδυασμός των δύο αρχιτεκτονικών, έχοντας ένα REST API και μερικές RPC υπηρεσίες, αντίστοιχα [16].

2.2.7. Η gRPC Remote Procedure Call

Ένας από τους λόγους που η χρήση του RPC αναζωπυρώθηκε τα τελευταία χρόνια είναι η εμφάνιση και καθιέρωση του gRPC, ενός σύγχρονου, open-source framework υψηλής απόδοσης που μπορεί να λειτουργήσει σε κάθε υπολογιστικό περιβάλλον. Δημιουργήθηκε από τη Google, από την οποία πήρε και το γράμμα "g" στο ακρωνύμιό του, αποτελώντας την επόμενη έκδοση της Stubby, μίας closed-source RPC υποδομής γενικής χρήσης. Το Stubby ξεκίνησε να δημιουργείται από το 2001, προοριζόμενο να συνδέσει το μεγάλο αριθμό από microservices μέσα και μεταξύ των κέντρων δεδομένων της. Το Μάρτιο του 2015, η Google αποφάσισε να εξελίξει το Stubby, μετατρέποντας το σε πλατφόρμα ανοιχτού κώδικα και μετονομάζοντάς το σε gRPC [17].

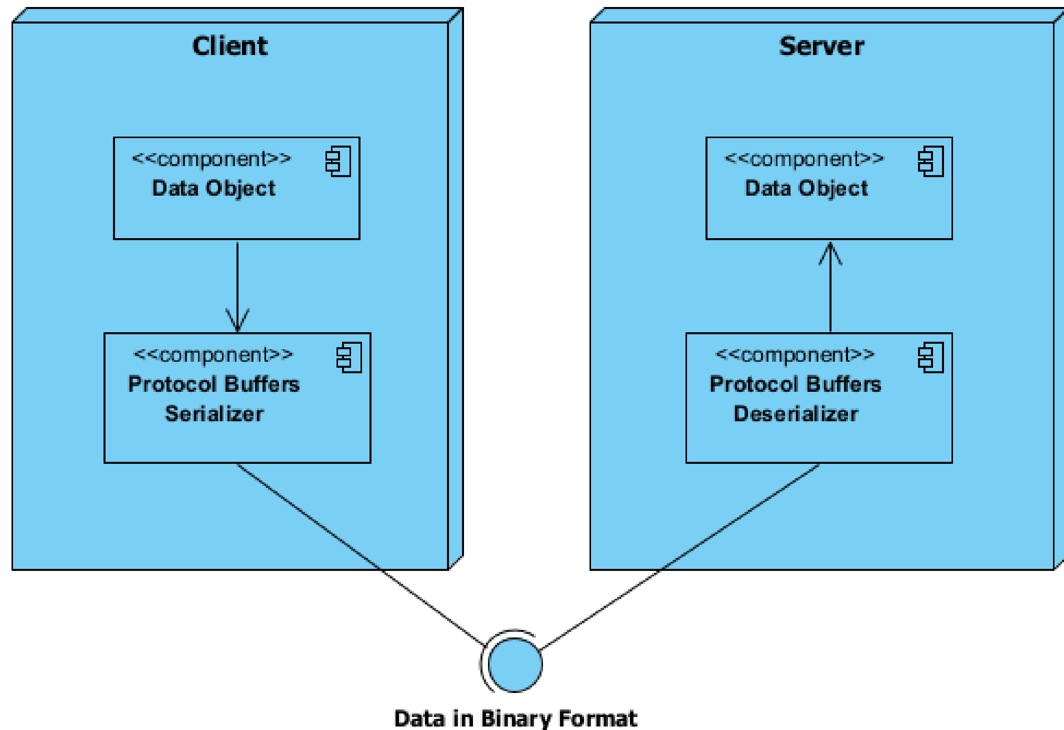
Ουσιαστικά, το gRPC αποτελεί μία διασκευή των παραδοσιακών RPC frameworks. Η σημαντικότερη του διαφοροποίηση έγκειται στο γεγονός πως χρησιμοποιεί protocol buffers ως την γλώσσα ορισμού του interface για τη σειριοποίηση και επικοινωνία, εν αντιθέσει με τις συνήθεις RPC αρχιτεκτονικές που ακολουθούν JSON και XML formats. Οι buffers πρωτοκόλλου είναι *language-agnostic*, δηλαδή μπορούν να χρησιμοποιηθούν σε οποιαδήποτε προγραμματιστική γλώσσα υποστηρίζει το αντίστοιχο πρωτόκολλο buffer [18]. Για το λόγο αυτό, το gRPC προτιμάται έντονα για web εφαρμογές που έχουν υλοποιηθεί με διαφορετικές τεχνολογίες.

Το βασικό πλεονέκτημα τους είναι η ικανότητα να περιγράψουν τη δομή των δεδομένων και ο κώδικας μπορεί να παραχθεί από την περιγραφή αυτή. Συγκεκριμένα, όταν τα διάφορα services στέλνουν μηνύματα, το gRPC χρησιμοποιεί τους protocol buffers για να κωδικοποιήσει τα δεδομένα σε δυαδική μορφή. Μέσα σε αυτή την πληροφορία από bits και bytes, περιγράφεται η δομή των πραγματικών δεδομένων, ώστε στη συνέχεια να είναι εφικτό να ανακτηθούν. Γενικά, η μέθοδος των protocol buffers περιλαμβάνει τα εξής δύο συστατικά στοιχεία [19]:

- Μία *Interface Description Language (IDL)*, δηλαδή μια γλώσσα η οποία επιτρέπει σε ένα πρόγραμμα γραμμένο σε κάποια συγκεκριμένη γλώσσα να επικοινωνήσει με ένα

άλλο πρόγραμμα του οποίου η γλώσσα είναι άγνωστη. Η IDL αξιοποιείται για την περιγραφή της δομής ορισμένων δεδομένων.

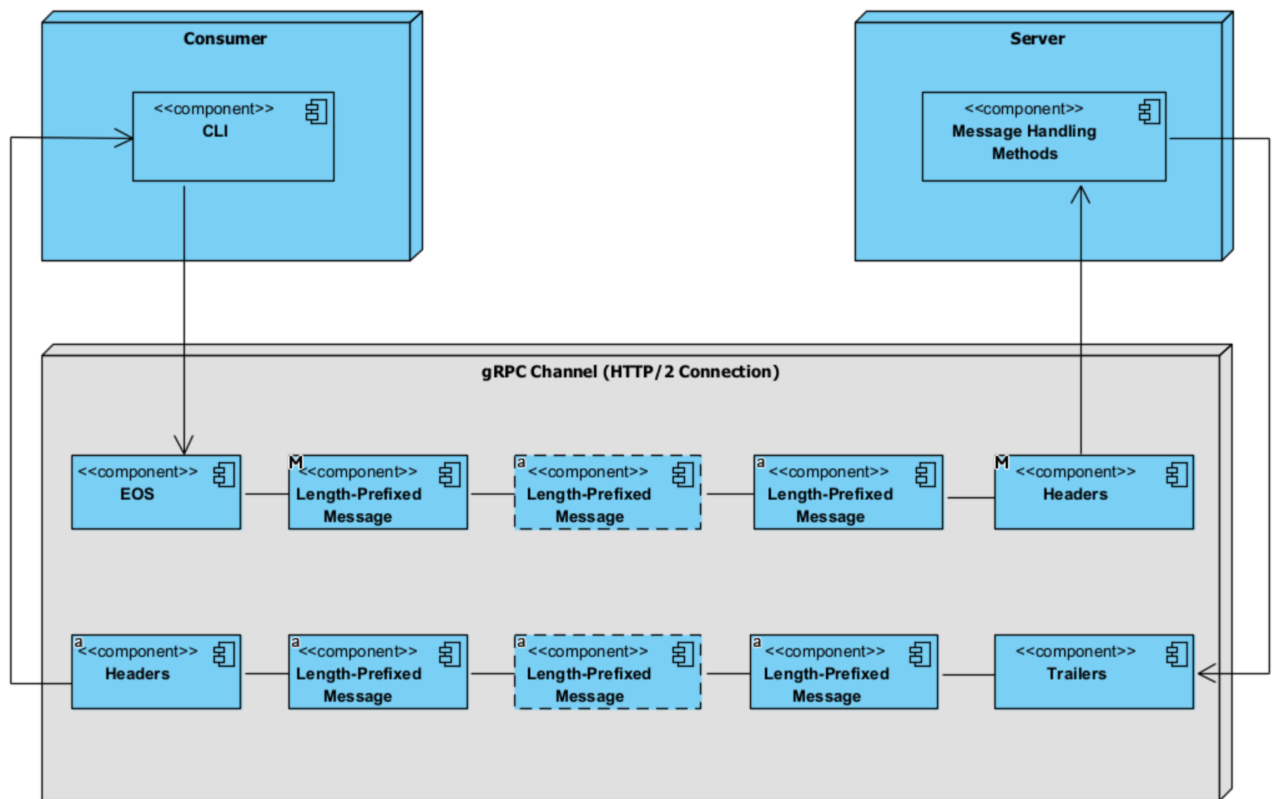
- Ένα πρόγραμμα το οποίο παράγει πηγαίο κώδικα από την περιγραφή αυτή, ώστε να παραχθεί ή να διασχιστεί *(parsing)* μία σειρά από bytes η οποία αναπαριστά τα structured data.



Εικόνα 5. Η διαδικασία κωδικοποίησης και αποκωδικοποίησης των δεδομένων προς μετάδοση μεταξύ πελάτη και διακομιστή [20].

Το δυαδικό (binary) format των δεδομένων καθιστά την επικοινωνία «ελαφρύτερη» από τις παραδοσιακές remote procedure calls, όπου συνήθως τα δεδομένα αποστέλλονται σε μορφή κειμένου, το οποίο μάλιστα ενδέχεται να περιέχει περιττούς χαρακτήρες όπως αλλαγή γραμμής. Γενικά, η επεξεργασία των text δεδομένων συνιστά υπολογιστικά ακριβή διαδικασία.

Επιπλέον, το gRPC είναι χτισμένο πάνω στο HTTP/2, το οποίο παρουσιάστηκε το 2015 και αποτελεί μία σημαντική ενημέρωση του HTTP διαδικτυακού πρωτοκόλλου, αναφερόμενο συνήθως ως HTTP/1.1. Ειδικότερα, το HTTP/1.1 είναι ένα text-based πρωτόκολλο, ενώ το HTTP/2 χρησιμοποιεί ένα δυαδικό στυλ που συμπιέζει τα δεδομένα για τη μετάδοσή τους. Ακόμη, υποστηρίζει την επικοινωνία διπλής κατεύθυνσης μαζί με την παραδοσιακή σχέση αιτήματος και απόκρισης, επιτρέποντας έτσι μία χαλαρή σύνδεση μεταξύ server και client (loose coupling). Στην πράξη, ο πελάτης εγκαθιδρύει μια μακροχρόνια σύνδεση με το gRPC server, ενώ για κάθε RPC κλήση θα ανοίξει ένα νέο HTTP/2 stream, δηλαδή ένα ανεξάρτητο, δυναμικό και διμερές σύνολο δεδομένων που ανταλλάσσεται μεταξύ server και client στην κοινή HTTP/2 σύνδεση. Κάθε stream συνιστά και ξεχωριστό κανάλι επικοινωνίας εντός της μακροχρόνιας σύνδεσης.



Εικόνα 6. Πλήρης αναπαράσταση των σταδίων μιας gRPC κλήσης αμφίδρομης ροής [21].

2.2.8. Διαφορές REST και gRPC πρωτοκόλλων

Καθώς το gRPC αποτελεί μετεξέλιξη και βελτίωση της παραδοσιακής Remote Procedure Call, υπάρχουν ορισμένες θεμελιώδεις διαφορές στον τρόπο λειτουργίας τους, γεγονός που αποδείχθηκε και κατά την πρότερη σύγκριση ανάμεσα στα RESTful και τα RPC APIs. Ειδικότερα, το gRPC είναι ένα πρωτόκολλο υψηλής απόδοσης, δυαδικό και με ισχυρή τυποποίηση (strongly-typed) που χρησιμοποιεί HTTP/2, ενώ το REST είναι ένα πιο απλό, text-based και stateless πρωτόκολλο που χρησιμοποιεί HTTP και μορφή κειμένου JSON ή XML. Η κύρια διαφορά τους συνοψίζεται στο γεγονός πως το gRPC συνιστά ένα υψηλής απόδοσης, open-source framework σχεδιασμένο για αποδοτική και ταχεία επικοινωνία μεταξύ υπηρεσιών, ενώ το REST αποτελεί μια πιο καθιερωμένη προσέγγιση, η οποία εκμεταλλεύεται τις τυπικές μεθόδους HTTP για την κατασκευή web APIs. Επιπλέον, το gRPC πρωτόκολλο υποστηρίζει την αμφίδρομη ροή πληροφορίας (bidirectional streaming), ενώ το REST περιορίζεται στα συνηθισμένα μοτίβα επικοινωνίας που βασίζονται στη σχέση αιτήματος – απόκρισης (request – response). Παρακάτω, απαριθμούνται οι διαφορές που παρατηρούνται στη λειτουργικότητα και την υλοποίηση των δύο πρωτοκόλλων [22]:

- **Format δεδομένων:** Παρά την ευελιξία στην επικοινωνία μεταξύ ανόμοιων γλωσσών ή/και πλατφορμών που παρουσιάζει το JSON format, το οποίο συνήθως πρόκειται για την επικρατέστερη επιλογή στα RESTful APIs, μπορεί να αποδειχθεί επιρρεπές σε σφάλματα και ιδιαίτερα αργό κατά τη χρήση του. Το gRPC ξεπερνά τα εμπόδια της ταχύτητας και του υπολογιστικού κόστους χρησιμοποιώντας τους

προαναφερθέντες protocol buffers, επιτυγχάνοντας με αυτόν τον τρόπο υψηλότερη αποδοτικότητα στη μετάδοση των μηνυμάτων. Το Protobuf υπολείπεται αισθητά σε ανθρώπινη αναγνωσιμότητα συγκριτικά με το JSON, γεγονός που του επιτρέπει να αφαιρεί πολλές από τις αρμοδιότητες του τελευταίου ώστε να επικεντρώνεται αυστηρά στη σειριοποίηση και αποσειριοποίηση των δεδομένων. Έτσι, μειώνεται σημαντικά το μέγεθος των μηνυμάτων και η μετάδοση των δεδομένων αυξάνεται ανάλογα.

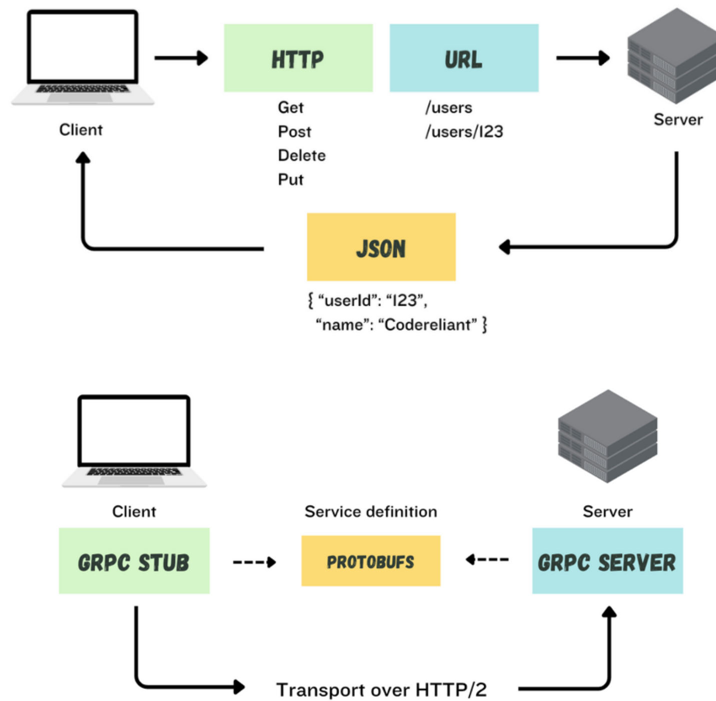
- **Πρωτόκολλο επικοινωνίας:** Ένας άλλος τρόπος με τον οποίον το gRPC ενισχύει την αποδοτικότητά του είναι η χρήση του HTTP/2 έναντι του HTTP/1.1 πρωτοκόλλου. Συγκεκριμένα, το HTTP/1.1 εκδόθηκε το 1997 και θεωρείται ως η βάση για της επικοινωνίες στον Παγκόσμιο Ιστό. Ουσιαστικά, μεταφέρει πληροφορίες από τον πελάτη προς το διακομιστή και αντιστρόφως: ο τελικός χρήστης στέλνει ένα text-based αίτημα σε JSON ή XML format και ο web server του επιστρέφει ένα resource, το οποίο συνήθως είναι μία σελίδα HTML ή ένα PDF έγγραφο. Αντίθετα το HTTP/2, το οποίο εκδόθηκε το 2015 και χρησιμοποιείται από τους περισσότερους μοντέρνους browsers, λειτουργεί διαφορετικά. Αντί να συντηρεί όλη την πληροφορία σε format καθαρού κειμένου, εμπεριέχει τα δεδομένα σε δυαδική μορφή (binary format encapsulation). Έτσι, δίνεται στο browser η δυνατότητα να αποστέλλει πολλαπλά αιτήματα ταυτόχρονα στο ίδιο connection και να λάβει τις αποκρίσεις πίσω σε οποιαδήποτε σειρά (multiplexing), καθιστώντας το HTTP/2 γρηγορότερο, αποτελεσματικότερο και με μικρότερη καθυστέρηση δικτύου από το HTTP/1.1.

- **Δημιουργία κώδικα:** Τα gRPC APIs χρησιμοποιούν τον δικό τους compiler, γνωστό ως «Protoc Compiler», ο οποίος επιτρέπει τη δημιουργία κώδικα και μπορεί να λειτουργήσει σε διάφορες γλώσσες προγραμματισμού και κατ' επέκταση πολυγλωσσικά περιβάλλοντα. Αντίθετα, τα REST APIs στερούνται την native γένεση κώδικα και χρειάζονται ένα εξωτερικό εργαλείο τρίτου μέρους (third-party tool), όπως το Swagger. Το γεγονός αυτό αποτελεί μειονέκτημα για πολλούς Επιστήμονες Πληροφορικής, καθώς θεωρείται πως επιδεινώνει τόσο την ταχύτητα όσο και την ευκολία υλοποίησης της διεπαφής.

- **Ταχύτητα μετάδοσης μηνυμάτων:** Τα gRPC APIs είναι πολύ γρηγορότερα από τα REST APIs, καθώς μεταφέρουν δεδομένα περίπου 7 έως 10 φορές ταχύτερα, λόγω της υψηλής συμπίεσης των δεδομένων που επιτρέπει το Protobuf και της χρήσης του πρωτοκόλλου HTTP/2.

- **Ταχύτητα Υλοποίησης:** Παρά τα οφέλη στην ταχύτητα μετάδοσης μηνυμάτων, η υλοποίηση των gRPC APIs είναι πολύ πιο αργή σε σχέση με την υλοποίηση των REST APIs. Θεωρείται πως η υλοποίηση ενός απλού gRPC service απαιτεί, σε μία συνηθισμένη περίπτωση, περίπου 45 λεπτά, ενώ για ένα Web ή REST API χρειάζονται μόλις 10 λεπτά. Αυτός ο παράγοντας είναι σημαντικός κατά την αξιολόγηση της καλύτερης επιλογής για το εκάστοτε υπολογιστικό σύστημα. Η καθυστέρηση οφείλεται στο γεγονός ότι δεν υπάρχει ακόμα αφενός πολλή υποστήριξη για αυτήν την δομή API σε ότι αφορά τα third-party εργαλεία, αφετέρου τόσο εκτεταμένο documentation όσο στην περίπτωση του REST. Καθώς

το gRPC συνιστά ακόμα καινούριο εργαλείο, στερείται της απαραίτητης τεχνικής υποστήριξης και αντί να ενσωματώνεται εύκολα και αλάνθαστα, ολόκληρη η διαδικασία συχνά καθυστερεί.



Εικόνα 7. Σύγκριση του τρόπου λειτουργίας ενός REST API με ένα gRPC API [23].

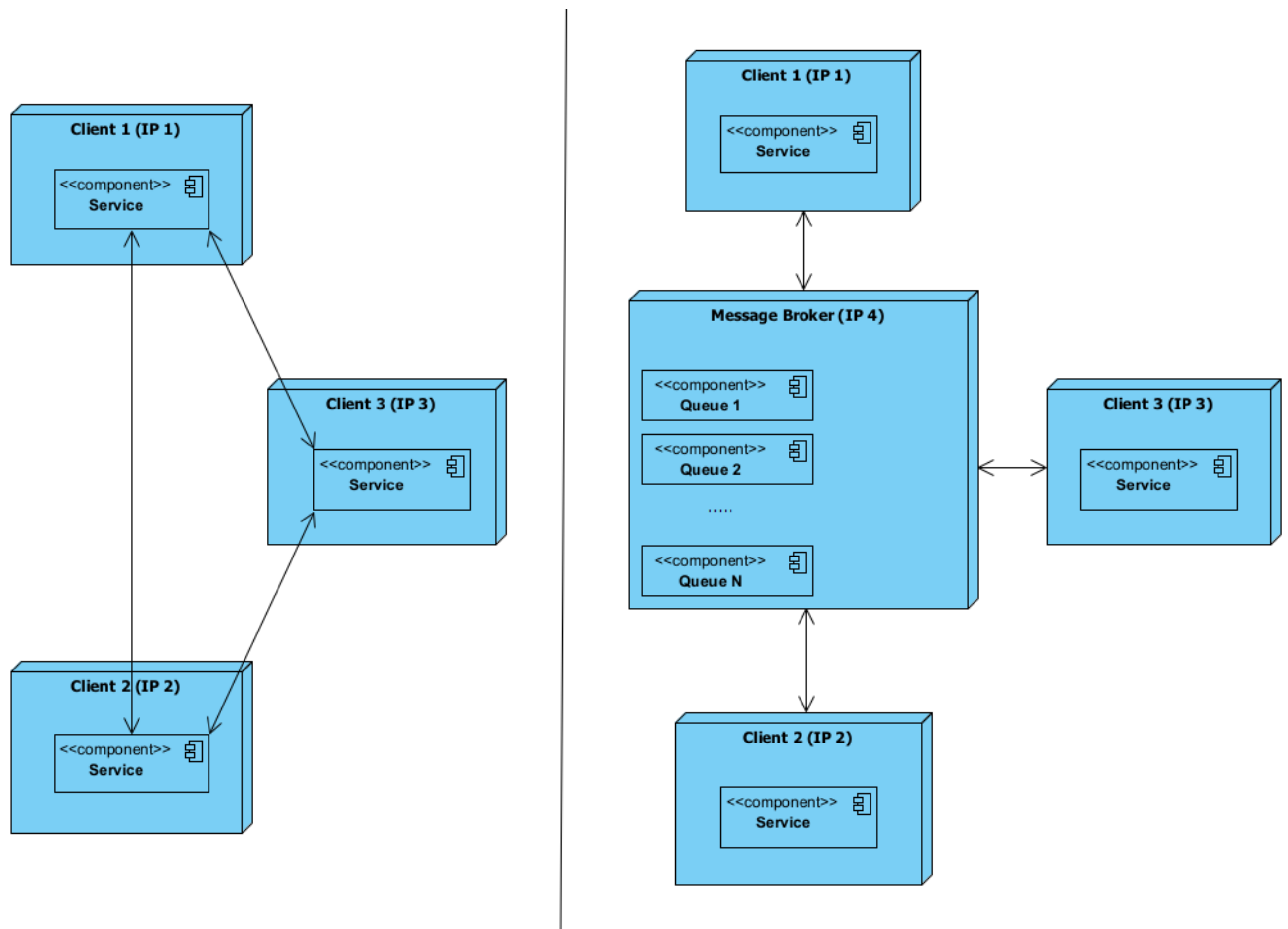
2.3. Message brokers

Στο κεφάλαιο αυτό, θα αναλυθεί σε βάθος το σημαντικότερο λογισμικό στοιχείο της εργασίας: οι διαμεσολαβητές μηνυμάτων (message brokers). Θα παρουσιαστούν οι αιτίες που οδήγησαν στην εμφάνισή τους, τα βασικά μοντέλα επικοινωνίας που χρησιμοποιούν για τη διασφάλιση της σωστής μετάδοσης των μηνυμάτων, ενώ ακόμη θα γίνει εκτενής αναφορά στο message broker που αξιοποιήθηκε για την πρακτική υλοποίηση των απαραίτητων υπηρεσιών, το RabbitMQ.

2.3.1. Ορισμός - Ιστορική αναδρομή

Ως message broker, ή εναλλακτικά integration broker/integration engine, ορίζεται ένα ενδιάμεσο, μεσολαβητικό πρόγραμμα το οποίο εφαρμογές και υπηρεσίες (services) αξιοποιούν για τη μεταξύ τους επικοινωνία και ανταλλαγή πληροφοριών. Ειδικότερα, οι message brokers μπορούν να χρησιμοποιηθούν στην επικύρωση, αποθήκευση, δρομολόγηση και εν τέλει, μεταφορά μηνυμάτων σε καθορισμένους προορισμούς. Το γεγονός αυτό επιτυγχάνεται μέσω της μετάφρασης μηνυμάτων ανάμεσα σε επίσημα πρωτόκολλα επικοινωνίας, επιτρέποντας έτσι σε αλληλοεξαρτώμενες υπηρεσίες να μιλούν άμεσα η μία με την άλλη, ακόμη και αν έχουν συνταχθεί σε διαφορετικές γλώσσες προγραμματισμού η βρίσκονται σε διαφορετικές πλατφόρμες.

Στον 21ο αιώνα, η ραγδαία αύξηση του συνολικού πλήθους των εφαρμογών, σε συνδυασμό με την καθιέρωση των APIs και της αξιοποίησης των HTTP requests για τη διαδικτυακή αλληλεπίδραση διαφορετικών προγραμμάτων, είχε ως αποτέλεσμα την ανάγκη διαχείρισης των μηνυμάτων σε συστήματα ολοένα και μεγαλύτερης πολυπλοκότητας. Οι υψηλές απαιτήσεις των οργανισμών για αποτελεσματική, αξιόπιστη και ευέλικτη επικοινωνία μεταξύ διαφορετικών συστημάτων και υπηρεσιών, δεν καλύπτονταν σε όλες τις περιπτώσεις από τις δυνατότητες που παρείχαν τα RESTful APIs, παρά τη γενικότερη επιτυχία τους. Για το λόγο αυτό, οι επιστήμονες της πληροφορικής έστρεψαν την προσοχή τους στην ανεύρεση καινούριων τεχνικών δρομολόγησης μηνυμάτων μεταξύ διαφορετικών software services. Έτσι, προέκυψε η δημιουργία των message brokers, οι οποίοι σε αντίθεση με τα APIs, δεν αποτελούν ένα σύνολο κανόνων για την επικοινωνία συστημάτων, αλλά μία ενδιάμεση πλατφόρμα ελέγχου ροής των μηνυμάτων μέσω της ένταξής τους σε ουρές (queues). Με τον τρόπο αυτό, παρουσιάζουν ένα πολύ σημαντικό προτέρημα έναντι των APIs: Ενεργοποιούν τη δυνατότητα ασύγχρονης επικοινωνίας μεταξύ των υπηρεσιών. Το γεγονός αυτό σημαίνει πως η αποστέλλουσα υπηρεσία, δηλαδή ο πελάτης στην client-server αρχιτεκτονική, δε χρειάζεται να περιμένει την απόκριση της υπηρεσίας που λαμβάνει το μήνυμα. Με τον τρόπο αυτό, επιτυγχάνεται η αποσύνδεση (decoupling) ανάμεσα στον αποστολέα και τον παραλήπτη του μηνύματος, επιτρέποντας την μεταξύ τους επικοινωνία δίχως να απαιτείται ο ένας να γνωρίζει την ύπαρξη του άλλου. Επιπλέον, οι message brokers διαθέτουν τόσο αυξημένη αξιοπιστία, εγγυώμενοι πως οι εκάστοτε παραλήπτες θα λάβουν το αποσταλέν μήνυμα ακόμα και σε ενδεχόμενη πτώση του διακομιστή είτε αν οι ίδιοι δεν είναι «ενεργοί» τη στιγμή της αποστολής του μηνύματος. Παράλληλα, διαθέτουν και αξιόλογη δυνατότητα κλιμάκωσης (scalability) μέσω εξισορρόπησης φόρτου εργασίας, στοιχεία τα οποία δεν μπορούσαν να καλυφθούν από τα RESTful APIs.



Εικόνα 8. Σύγκριση της λειτουργίας αρχιτεκτονικής χωρίς message brokers με broker-based αρχιτεκτονική [24].

Η άνοδος της χρήσης του Cloud και κατ' επέκταση των SaaS εφαρμογών έχει οδηγήσει στην καθιέρωση των διαμεσολαβητών μηνυμάτων ως βασικό εργαλείο πολλών υπολογιστικών υπηρεσιών για την ανταλλαγή πληροφοριών. Σήμερα, οι τρεις δημοφιλέστεροι message brokers είναι ο RabbitMQ, ο Kafka και ο ActiveMQ, αποτελώντας όλοι τους αξιόπιστα open-source λογισμικά που παρουσιάζουν ευρεία χρήση σε παγκόσμιο επίπεδο. Λόγω των διαφορετικών δομών και χαρακτηριστικών του κάθε λογισμικού, το καθένα ενδείκνυται για διαφορετικές περιπτώσεις χρήσης, οι οποίες συνοπτικά παρατίθενται παρακάτω:

Message broker	Πλεονεκτήματα χρήσης
RabbitMQ	Ανταλλαγή μηνυμάτων χαμηλής καθυστέρησης (low-latency) και ευέλικτη δρομολόγηση
Kafka	Υψηλή ρυθμαπόδοση, κλιμακωσιμότητα και επεξεργασία σε πραγματικό χρόνο
ActiveMQ	Υποστήριξη πολλών πρωτοκόλλων επικοινωνίας και δυνατότητα αξιοποίησης προηγμένων επιχειρησιακών λειτουργικοτήτων

Στην παρούσα διπλωματική εργασία, πρωταρχικός στόχος είναι η υλοποίηση ενός διαμεσολαβητή μηνυμάτων που να διαθέτει υψηλή ταχύτητα και scalability προκειμένου να μπορεί να διαχειριστεί μεγάλο πλήθος αιτημάτων σχετικά γρήγορα και αποδοτικά. Το γεγονός αυτό, σε συνδυασμό με το εκτεταμένο documentation και την μεγάλη γκάμα γλωσσών προγραμματισμού που εξυπηρετεί, καθώς επίσης και το ότι ο ίδιος ο message broker δεν αποτελεί αντικείμενο διερεύνησης στην παρούσα εργασία, κατέστησαν καλύτερη επιλογή για την δρομολόγηση και ανταλλαγή μηνυμάτων το RabbitMQ, οπότε και χρησιμοποιήθηκε, όπως θα αναλυθεί εκτενώς στη συνέχεια.

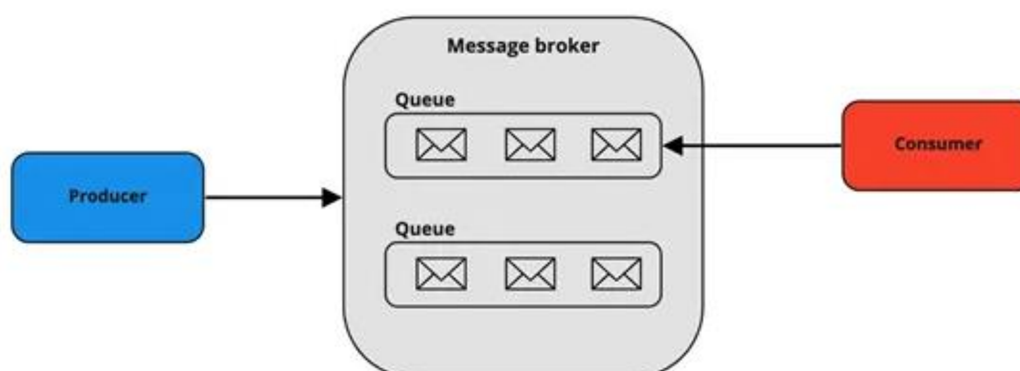
2.3.2. Γενικά στοιχεία ενός message broker

Ένας message broker αποτελείται από τα 3 παρακάτω βασικά δομικά στοιχεία (components):

- **Producer:** Το κομμάτι που είναι υπεύθυνο για την αποστολή των μηνυμάτων στον broker. Εκδίδουν τα μηνύματα στον προορισμό που υπαγορεύει το εκάστοτε πρωτόκολλο επικοινωνίας (protocol).
- **Consumer:** Το κομμάτι που καταναλώνει τα μηνύματα που περιέχονται στο message broker.
- **Queue:** Η δομή δεδομένων στην οποία αποθηκεύονται τα μηνύματα. Ουσιαστικά, πρόκειται για ένα μεγάλο, ευρύχωρο buffer μηνυμάτων, εξαρτώμενο αποκλειστικά από τη μνήμη του host και τους περιορισμούς του δίσκου.

Για την αποστολή και λήψη των μηνυμάτων από τον producer και τον consumer, αντίστοιχα, υπάρχουν δύο επικρατέστερα μοτίβα ανταλλαγής μηνυμάτων (messaging patterns).

1. **Point-to-Point**



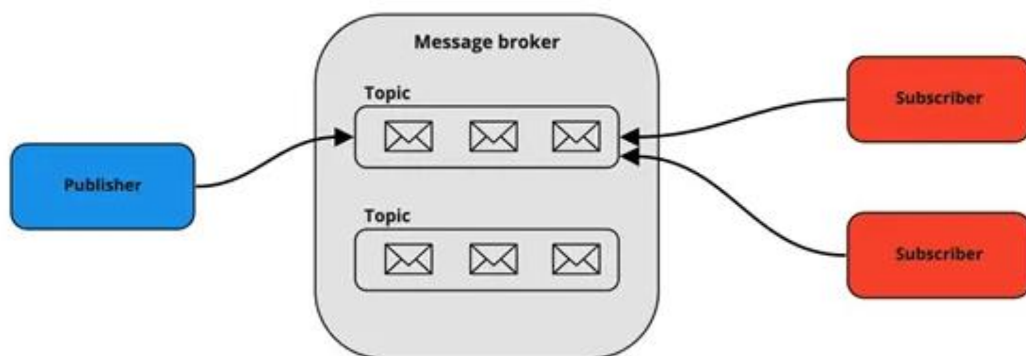
Σε αυτό το μοτίβο, οι ουρές του broker έχουν μία-προς-μία σχέση (one-to-one) με τον αποστολέα και τον παραλήπτη του μηνύματος, οι οποίοι συνήθως αναφέρονται ως producer και consumer. Κάθε μήνυμα στην ουρά μπορεί να αποσταλεί μόνο σε έναν αποδέκτη, ακόμα και αν υπάρχουν πολλαπλοί consumers που «ακούν» στη συγκεκριμένη ουρά. Ο producer στέλνει το μήνυμα στο message broker όπου εντάσσεται σε μία ουρά, και ο consumer ανακτά τα μηνύματα από την

ουρά, επιστρέφοντας παράλληλα ένα *acknowledgement* που υποδηλώνει την επιτυχή λήψη του εκάστοτε μηνύματος. Κάθε queue που δημιουργείται μέσα στο message broker μπορεί να ρυθμιστεί ώστε να είναι *αποκλειστική* (exclusive). Αν μία ουρά είναι αποκλειστική, τότε όλα τα μηνύματά της μπορούν να ληφθούν μόνο από τον πρώτο consumer που έχει οριστεί για τη συγκεκριμένη ουρά. Αντίθετα, αν η ουρά δεν είναι αποκλειστική, οποιοσδήποτε αριθμός από αποδέκτες μπορεί να ανακτήσει μηνύματα από αυτήν. Έτσι, οι μη-αποκλειστικές ουρές χρησιμεύουν στην εξισορρόπηση του φορτίου των εισερχόμενων μηνυμάτων, κατανέμοντας το σε πολλαπλούς αποδέκτες [25]. Ωστόσο, ανεξάρτητα από τον τύπο της εκάστοτε ουράς, τονίζεται πως μόνον ένας consumer μπορεί να καταναλώσει κάθε μήνυμα που αυτή περιέχει. Επιπλέον, ο τελικός αποδέκτης μπορεί να λάβει το μήνυμα μόνο τη στιγμή της μετάδοσης και ποτέ ξανά στο μέλλον, καθιστώντας το point-to-point μοντέλο κατάλληλο για περιπτώσεις όπου το μήνυμα έχει νόημα να ασκήσει την αναμενόμενη «επίδραση» μόνο μία φορά. Για αυτό το λόγο, η point-to-point αρχιτεκτονική χρησιμοποιείται ως επί το πλείστον σε επεξεργασία οικονομικών συναλλαγών, εφόσον η εκάστοτε μεμονωμένη πληρωμή υποχρεούται να εξοφληθεί εφάπαξ. Το μοτίβο αυτό διακρίνεται σε δύο κατηγορίες: την fire-and-forget και το request/reply messaging μοντέλο.

- **Fire-and-forget:** Ο αποστολέας δεν περιμένει για κάποια απάντηση από την ουρά δίχως να ενδιαφέρεται για το αν ο αποστολέας έλαβε το μήνυμα ή όχι. Σε αυτό το μοντέλο, ο πομπός και ο δέκτης δεν έχουν καμία επικοινωνία.

- **Request/reply:** Σε αντίθεση με την προηγούμενη πολιτική, ο αποστολέας στέλνει το μήνυμα σε μία ουρά και στη συνέχεια περιμένει για την απάντηση του παραλήπτη, προκειμένου να γνωρίζει το status του μηνύματος που προώθησε προς επεξεργασία.

2. Publish-Subscribe



Το μοντέλο αυτό, επονομαζόμενο και ως pub/sub, αποτελεί ένα πρότυπο ασύγχρονης επικοινωνίας όπου οι αποστολείς, γνωστοί ως publishers και οι αποδέκτες, γνωστοί ως subscribers, ανταλλάσσουν μηνύματα δίχως να καθιερώνουν άμεση επαφή, δηλαδή παραμένοντας χαλαρά συνδεδεμένοι (loosely coupled). Τα

μηνύματα που παράγουν οι publishers ονομάζονται events. Συγκεκριμένα, οι publishers δεν στέλνουν τα event απευθείας στους subscribers, αλλά χρησιμοποιούν ένα δίκτυο με συνδεδεμένους brokers το οποίο είναι υπεύθυνο για τη μεταφορά των events στους ενδιαφερόμενους subscribers. Ουσιαστικά, οι publishers δεν γνωρίζουν ποιος λαμβάνει τα events και οι subscribers δεν είναι ενήμεροι σχετικά με την πηγή της πληροφορίας, αντίστοιχα. Προκειμένου να δεχθούν events, οι subscribers πρέπει να καταγράψουν το ενδιαφέρον τους σε αυτά που επιθυμούν. Όταν ένα νέο event παραχθεί από τους publishers, αυτό πρέπει να προωθηθεί προωθούν σε όλους τους subscribers οι οποίοι διαθέτουν ένα *φίλτρο* ταιριαστό με το συγκεκριμένο event. Αυτό επιτυγχάνεται μέσω της χρήσης ενός διαμεσολαβητή, δηλαδή ενός pub/sub message broker ο οποίος ομαδοποιεί τα δεδομένα σε οντότητες που ονομάζονται κανάλια ή topics και ουσιαστικά αποτελούν κοινούς, ομαδικούς χώρους συγκέντρωσης μηνυμάτων. Έτσι, ο publisher στέλνει τα μηνύματα σε ένα topic και στη συνέχεια αυτά κατανέμονται σε όλους τους subscribers οι οποίοι έχουν «εγγραφεί» στο συγκεκριμένο κανάλι. Επομένως, το pub/sub μοντέλο υποστηρίζει σχέσεις ένα-προς-πολλά (one-to-many) ανάμεσα στον αποστολέα και τον παραλήπτη, με αποτέλεσμα να αποτελεί χρήσιμη μέθοδο για συστήματα στα οποία τα δεδομένα πρέπει να διανέμονται σταθερά σε περισσότερα του ενός συμβαλλόμενα μέρη (parties).

Γενικότερα, η pub/sub αρχιτεκτονική δημιουργήθηκε λόγω της ανάγκης για βελτίωση της κλιμακωσιμότητας στα πληροφοριακά συστήματα. Ειδικότερα, η στατική κλιμάκωση που επικρατούσε στα περισσότερα συστήματα στην pre-Internet εποχή αδυνατούσε να συμβαδίσει με την ταχεία διάδοση του Internet και των web-based εφαρμογών, γεγονός που σε συνδυασμό με τη μαζική υιοθέτηση των κινητών συσκευών οδήγησε στην ανάγκη δυναμικής κλιμάκωσης.. Το pub/sub πρότυπο επικοινωνίας προτάσσει το πλεονέκτημα ότι επιτρέπει την πλήρη αποσύνδεση (decoupling) των μερών που επικοινωνούν, γεγονός που καθιστά εφικτή τη δυναμική, ευέλικτη ανταλλαγή πληροφοριών ανάμεσα σε μεγάλο αριθμό από οντότητες, δίχως την ανάγκη της μεταξύ τους γνώσης. Επιπλέον, αν ενεργοποιηθεί η *ανθεκτική συνδρομή* (durable subscription), δεν απαιτείται η ταυτόχρονη συμμετοχή των publishers και των subscribers σε μία αλληλεπίδραση˙ αν ένας subscriber είναι εκτός σύνδεσης όταν ο publisher δημιουργεί το event, ο broker θα αποθηκεύσει το γεγονός μέχρι ο subscriber να συνδεθεί ξανά και το event να μπορέσει να παραδοθεί [26].

2.3.3. Το λογισμικό RabbitMQ

Η RabbitMQ αποτελεί την πλέον δημοφιλή περίπτωση message broker ανοικτού κώδικα, γνωρίζοντας ευρεία χρήση παγκοσμίως σε projects κάθε κλίμακας. Δεκάδες χιλιάδες χρήστες αξιοποιούν την RabbitMQ παγκοσμίως, τόσο σε μικρές startups όσο και σε μεγάλες, εδραιωμένες επιχειρήσεις. Αναπτύχθηκε στην πρώτη της μορφή από την Rabbit Technologies Ltd. το 2007 και εξαγοράστηκε από την SpringSource, η οποία αποτελεί ένα τμήμα της διεθνούς αναγνωρίσιμης VMWare, τον Απρίλιο του 2010. Στη συνέχεια, το project έγινε μέρος του Pivotal Software το Μάιο του 2013, ώσπου αποκτήθηκε εκ νέου από την VMWare το Δεκέμβριο του 2019.

Αρχικά, υλοποιούσε μόνο το Advanced Message Queueing Protocol (AMQP). Έκτοτε, έχει επεκταθεί με plug-ins τα οποία υποστηρίζουν και άλλα πρωτόκολλα, όπως το Streaming Text Oriented Messaging Protocol (STOMP) και το MQ Telemetry Transport (MQTT). Ο RabbitMQ Server, γραμμένος σε Erlang, είναι «χτισμένος» στο Open Telecom Platform (OTP) framework, μια συλλογή από χρήσιμα middleware, βιβλιοθήκες και εργαλεία γραμμένα σε Erlang που αποτελούν ένα αναπόσπαστο κομμάτι της open-source διανομής της συγκεκριμένης γλώσσας προγραμματισμού. Ο πηγαίος κώδικας έχει εκδοθεί υπό την Δημόσια Άδεια της Mozilla (Mozilla Public License – MPL), μία δωρεάν άδεια ανοιχτού κώδικα για τα περισσότερα λογισμικά του οργανισμού Mozilla [27].

Συνολικά, το project αποτελείται από τα ακόλουθα στοιχεία:

- Το RabbitMQ exchange server, ο οποίος συνιστά και τη θεμελιώδη δομή για την επιτέλεση της λειτουργίας του project, τη δρομολόγηση μηνυμάτων με ουρές
- Πύλες δικτύου (gateways) για την υλοποίηση των AMQP, HTTP, STOMP και MQTT πρωτοκόλλων
- Τις βιβλιοθήκες πελατών για το AMQP πρωτόκολλο για διάφορα προγραμματιστικά περιβάλλοντα, όπως Java, .NET, Erlang και Python
- Μία πλατφόρμα που επιδέχεται plug-ins για τη διασφάλιση της επεκτασιμότητας αναφορικά με τις λειτουργίες που παρέχει το project.

Το RabbitMQ πρόκειται για ένα lightweight εργαλείο το οποίο μπορεί να γίνει deployed τόσο ως λογισμικό στην τοπικό IT περιβάλλον του χρήστη (on-premises), όσο και στο Cloud. Η κύρια λειτουργικότητα του είναι η υποστήριξη ασύγχρονης επικοινωνίας μέσω μηνυμάτων, συμβάλλοντας στην αποσύνδεση των εφαρμογών μέσω του διαχωρισμού της διαδικασίας αποστολής και λήψης δεδομένων. Παρακάτω παρατίθενται συνοπτικά τα σημαντικότερα χαρακτηριστικά του RabbitMQ [28]:

- **Αξιοπιστία:** Ενέχει τη δυνατότητα «ανταλλαγής» απόδοσης με αξιοπιστία, με μεθόδους όπως η αναγνώριση παράδοσης (delivery acknowledgement) και η υψηλή διαθεσιμότητα (availability).

- **Ευέλικτη δρομολόγηση:** Τα μηνύματα, προτού φτάσουν στις ουρές, στέλνονται σε μεσολαβητές δρομολόγησης γνωστούς ως exchanges. Αυτοί διαθέτουν πολλούς built-in τύπους αντιστοίχισής τους με το μήνυμα, επιτρέποντας στο χρήστη να επιλέξει τον καταλληλότερο για την εκάστοτε περίπτωση. Ο τρόπος λειτουργίας των exchanges περιγράφεται αναλυτικά στη συνέχεια.

- **Ομαδοποίηση (clustering):** Πολλοί RabbitMQ servers μπορούν να ομαδοποιηθούν σε ένα cluster, σχηματίζοντας έτσι έναν λογικό broker.

- **Πολλαπλά πρωτόκολλα:** Όπως αναφέρθηκε και παραπάνω, το RabbitMQ υποστηρίζει τη χρήση πολλαπλών messaging protocols, δίνοντας στον χρήστη την ευελιξία να διαλέξει το πρωτόκολλο που εξυπηρετεί καλύτερα τις ανάγκες του. Ενδεικτικά παρατίθενται οι προορισμοί του publishing για τα συχνότερα χρησιμοποιούμενα πρωτόκολλα:

Πρωτόκολλο επικοινωνίας	Προορισμός
AMQP 0-9-1	Exchange
AMQP 1.0	Σύνδεσμος (link)
MQTT	Topic
STOMP	Topics, queues, AMQP 0-9-1 exchanges

Στην παρούσα εργασία, χρησιμοποιείται το AMQP 0-9-1 protocol, το οποίο είναι και το μακροβιότερο που υποστηρίζεται από το RabbitMQ, οπότε θα παρακαμφθούν τα υπόλοιπα πρωτόκολλα και θα αναλυθεί μόνο το συγκεκριμένο.

2.3.3.1. Connections

Οι εφαρμογές αλληλεπιδρούν με τη RabbitMQ χρησιμοποιώντας τις βιβλιοθήκες των πελατών (client libraries). Υπάρχουν τέτοιες βιβλιοθήκες διαθέσιμες για πολλές προγραμματιστικές γλώσσες και πλατφόρμες, οι περισσότερες εκ των οποίων είναι ανοιχτού κώδικα. Το κάθε πρωτόκολλο έχει το δικό του σύνολο από client libraries. Σε ότι αφορά το AMQP πρωτόκολλο, το οποίο αξιοποιείται στην παρούσα εργασία, αυτό συνιστά ένα πρωτόκολλο επιπέδου εφαρμογής (application-level protocol), γεγονός που σημαίνει ότι το connection αυτό συμβαίνει on top από μία σύνδεση στο επίπεδο μεταφοράς δεδομένων (transport-level). Για την εξυπηρέτηση αυτού του σκοπού, η RabbitMQ υποστηρίζει πρωτόκολλα επικοινωνίας τα οποία βασίζονται στο Transmission Control Protocol (TCP), το οποίο αποτελεί μια ασφαλή, αξιόπιστη και με έλεγχο σφαλμάτων διαδικασία μεταφοράς σειρών από bytes ανάμεσα σε εφαρμογές των οποίων οι hosts επικοινωνούν μέσω ενός δικτύου IP. Πρόκειται για μία τεχνική επονομαζόμενη ως «connection-oriented», καθώς απαιτεί την καθιέρωση μιας σύνδεσης ανάμεσα στον client και τον server πριν την αποστολή δεδομένων. Για να επιτευχθεί αυτό, ο πελάτης οφείλει να εισάγει έγκυρα credentials στο server ώστε να ελεγχθεί η ταυτότητά του. Μόλις ολοκληρωθεί επιτυχώς η σύνδεση και αυθεντικοποίηση του χρήστη, επιτρέπεται η αποστολή και λήψη μηνυμάτων μέσω του message broker. Πέραν των ζητημάτων ασφαλείας, η διαδικασία καθιέρωσης του connection είναι επίσης υπεύθυνη για τη διαχείριση κάποιων πλευρών του AMQP πρωτοκόλλου. Σε αυτό το σημείο, αν ο client ή/και ο server δεν μπορούν να «συμφωνήσουν» στο version του πρωτοκόλλου ή σε κάποια ρυθμιστική, για τη σύνδεση, τιμή παραμέτρου, το connection είναι αδύνατο να τεθεί σε ισχύ, με συνέπεια το κλείσιμο της TCP σύνδεσης του επιπέδου μεταφοράς δεδομένων [29].

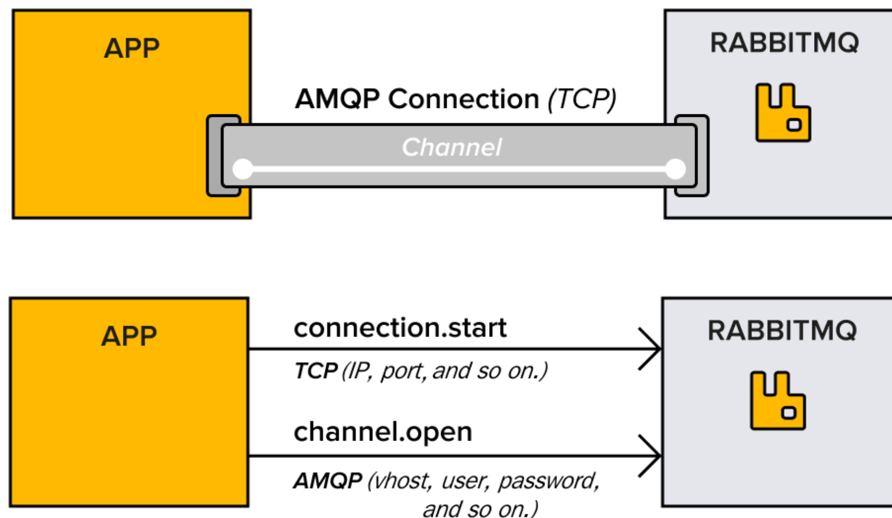
Για την εξασφάλιση καλύτερης απόδοσης, η RabbitMQ προτείνει την δημιουργία μακρόβιων συνδέσεων (long-lived connections), γεγονός που συνεπάγεται πως κάθε πελάτης της RabbitMQ βιβλιοθήκης εγκαθιδρύει ένα connection. Επιπλέον, οι clients συνήθως καταναλώνουν μηνύματα εγγράφοντας εξαρχής ένα subscription, προκειμένου το εκάστοτε μήνυμα να τους παραδίδεται απευθείας (push) και να μην απαιτείται ο συνεχής έλεγχος της κατάστασης στην οποία βρίσκεται η συσκευή του client προγράμματος (polling).

Όταν μια σύνδεση δεν είναι πλέον απαραίτητη, οι εφαρμογές πρέπει να την κλείνουν για να εξοικονομήσουν πόρους. Αν αποτύχουν να το πράξουν, διατρέχουν τον κίνδυνο να

εξαντλήσουν εν τέλει τους πόρους στον server node όπου λειτουργούν. Ακόμη, τα λειτουργικά συστήματα έχουν ένα όριο αναφορικά με τον αριθμό των συνδέσεων TCP (sockets) τις οποίες μια μεμονωμένη διεργασία μπορεί να έχει ανοιχτές ταυτόχρονα. Το όριο αυτό συνήθως επαρκεί για την ανάπτυξη του λογισμικού και μερικά περιβάλλοντα ελέγχου ποιότητας. Τα περιβάλλοντα παραγωγής, ωστόσο, συχνά καλούνται να υποστηρίξουν έναν μεγαλύτερο αριθμό από ταυτόχρονα client connections. Στην περίπτωση αυτή, απαιτείται κατάλληλη ρύθμιση των συγκεκριμένων environments προκειμένου να χρησιμοποιούν υψηλότερο όριο διεργασιών.

2.3.3.2. Channels

Σε ορισμένες περιπτώσεις, όπως η παράλληλη επεξεργασία εργασιών και η ανάγκη συνύπαρξης μηνυμάτων με διαφορετικές προδιαγραφές, οι εφαρμογές χρειάζονται πολλαπλές λογικές συνδέσεις στο διαμεσολαβητή. Ωστόσο, η διατήρηση πολλών ανοιχτών TCP συνδέσεων την ίδια χρονική στιγμή, θα σήμαινε υψηλή κατανάλωση συστημικών πόρων, καθιστώντας ταυτόχρονα την παραμετροποίηση των firewalls δυσκολότερη. Εξαιτίας αυτού, οι AMQP 0-9-1 συνδέσεις είναι εφοδιασμένες με κανάλια (channels), τα οποία στην ουσία αποτελούν εικονικές, lightweight συνδέσεις που μοιράζονται μία κοινή TCP πραγματική σύνδεση. Έτσι, κάθε λειτουργία του πρωτοκόλλου επικοινωνίας πραγματοποιείται σε ένα από τα κανάλια της σύνδεσης που έχει εγκαθιδρύσει ο client. Κάθε κανάλι επικοινωνίας λειτουργεί πλήρως απομονωμένο από τα υπόλοιπα κανάλια, εξασφαλίζοντας πως οι ενέργειες που πραγματοποιούνται σε αυτό δεν θα επηρεάσουν την κατάσταση ή συμπεριφορά των άλλων καναλιών. Ειδικότερα, κάθε κανάλι διαθέτει έναν ακέραιο αριθμό ως αναγνωριστικό (channel id), τον οποίον χρησιμοποιούν τόσο οι clients όσο και ο message broker προκειμένου να προσδιορίσουν σε ποιο κανάλι υπάγεται μία μέθοδος πρωτοκόλλου, όπως η αποστολή και λήψη μηνυμάτων. Καθώς ένα κανάλι υπάρχει μόνο στο πλαίσιο μίας σύνδεσης, το κλείσιμο του εκάστοτε connection έχει ως αποτέλεσμα και το κλείσιμο όλων των καναλιών που υπάγονται σε αυτό [30].



Εικόνα 9. Η αναπαράσταση της ενσωμάτωσης ενός καναλιού εντός μιας σύνδεσης (αριστερά) και η απεικόνιση των πληροφοριών που λαμβάνουν από την εφαρμογή (δεξιά) [31].

Στη συνέχεια, θα αναλυθούν τα δύο σημαντικότερα συστατικά στοιχεία ενός καναλιού: οι ανταλλαγές (exchanges) και οι ουρές (queues).

2.3.3.3. Exchanges

Ο producer δεν αποστέλλει το μήνυμα απευθείας στην ουρά, αλλά σε μία δομή που ονομάζεται «exchange», η οποία λειτουργεί ως διαμεσολαβητής δρομολόγησης (routing mediator). Συγκεκριμένα, ένα exchange καθορίζει τον τρόπο που θα δρομολογηθεί ένα μήνυμα χρησιμοποιώντας παραμέτρους και binding setups, δηλαδή μηχανισμούς «δεσίματος» του exchange με ουρές, ή ακόμα και με άλλα exchanges. Ανάλογα την περίπτωση, το μήνυμα θα μεταβεί σε μία ουρά, σε πολλαπλές ουρές, ή τελικά θα απορριφθεί (discard). Οι clients μπορούν είτε να δημιουργήσουν δικά τους exchanges είτε να αξιοποιήσουν τα προκαθορισμένα default exchanges που παρέχονται. Στη RabbitMQ, υπάρχουν οι ακόλουθοι τέσσερις τύποι exchanges [32]:

- **Direct:** Το μήνυμα προωθείται στην ουρά η οποία διαθέτει ένα συγκεκριμένο κλειδί δρομολόγησης (routing key). Αυτός ο τρόπος επιτρέπει τον εύκολο διαχωρισμό μηνυμάτων που δημοσιεύονται στην ίδια exchange και είναι κατάλληλος για περιπτώσεις point-to-point επικοινωνίας, όπου το μήνυμα μεταβαίνει απευθείας σε μία ουρά που το αναμένει.

- **Topic:** Αυτός ο τύπος exchange παρουσιάζει αρκετές ομοιότητες με την direct exchange, αλλά η δρομολόγηση συμβαίνει σύμφωνα με το αντίστοιχο μοτίβο (routing pattern). Αντί να βασίζεται σε σταθερό routing key, χρησιμοποιεί wildcards. Το μήνυμα προωθείται σε όλες οι ουρές που είναι «δεμένες» στο exchange και επιπλέον επιτυγχάνουν ταίριασμα προτύπου (pattern matching) μεταξύ ενός η περισσότερων διαθέσιμων λεκτικών του συγκεκριμένου exchange, τα οποία οριοθετούνται μεταξύ τους με ένα σημείο στίξης, την τελεία (.). Προορίζεται κυρίως για pub/sub επικοινωνία, όπου τα μηνύματα στέλνονται σε ομάδες παραληπτών βάσει των θεμάτων στα οποία έχουν εγγραφεί (subscribe).

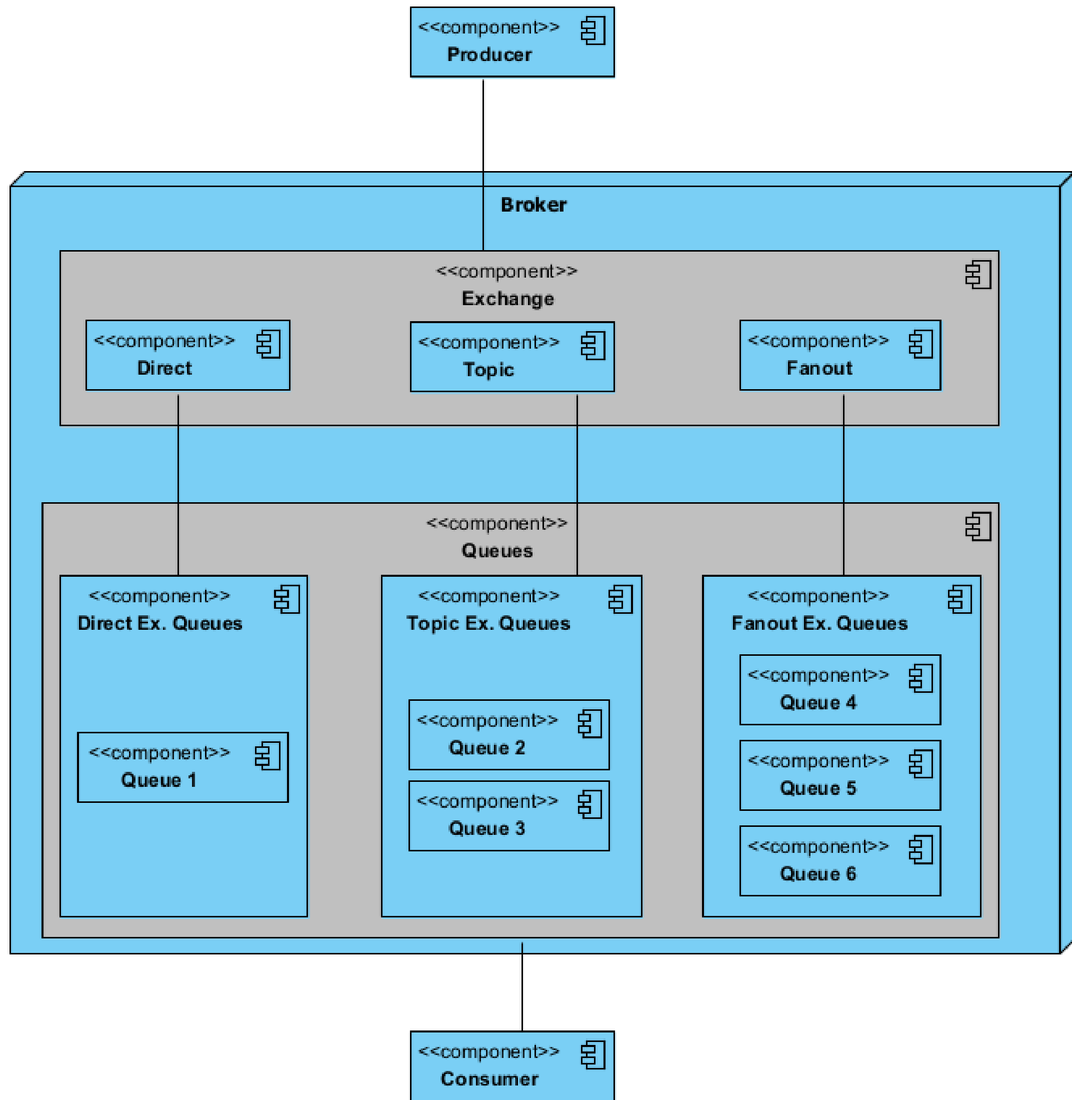
- **Fanout:** Το μήνυμα αποστέλλεται σε όλες τις ουρές οι οποίες είναι «δεμένες» (bound) με το συγκεκριμένο exchange, ανεξαρτήτως από τα routing keys και τα patterns. Τα κλειδιά που ενδεχομένως περιέχονται στις ουρές, απλώς αγνοούνται. Αυτός ο τύπος exchange μπορεί να καταστεί χρήσιμος σε περιπτώσεις όπου το ίδιο μήνυμα πρέπει να αποσταλεί σε μία η περισσότερες ουρές, ώστε ο consumer κάθε ουράς να έχει τη δυνατότητα να το επεξεργαστεί με διαφορετικό τρόπο από τους υπόλοιπους.

- **Headers:** Το exchange δρομολογεί τα μηνύματα βάσει ορισμάτων που περιέχουν επικεφαλίδες (headers) και προαιρετικά, τιμές. Αξιοποιεί τα header attributes του μηνύματος αντί για κάποιο routing key προκειμένου να προωθήσει το εκάστοτε μήνυμα στις ουρές που οφείλει, ενώ παράλληλα υπάρχει ένα ειδικό όρισμα που ονομάζεται «x-match», το οποίο διαθέτει δύο πιθανές τιμές, την «all» και την «any», οι οποίες καθορίζουν αν πρέπει να ταιριάζουν όλες οι επικεφαλίδες ή αν αρκεί να ταιριάζει μόνο μία, αντίστοιχα. Η default τιμή του συγκεκριμένου argument είναι το «all».

Στις περισσότερες υλοποιήσεις ενός message broker με τη RabbitMQ, επιλέγεται μία εκ των τριών πρώτων exchange δομών, δηλαδή direct, topic ή fanout δρομολόγηση. Πέραν των κατηγοριών που αναλύθηκαν παραπάνω, υπάρχουν άλλοι δύο exchange τύποι που αξίζει να σημειωθούν: Η default exchange και η dead letter exchange [33].

- **Default Exchange:** Πρόκειται για μια προκαθορισμένη direct exchange η οποία δεν έχει όνομα. Συνήθως, η αναφορά στο exchange name της επιτυγχάνεται μέσω ενός κενού αλφαριθμητικού. Αν χρησιμοποιηθεί η default exchange, το μήνυμα μεταφέρεται στην ουρά της οποίας το όνομα ισοδυναμεί με το κλειδί δρομολόγησης του μηνύματος. Κάθε ουρά είναι αυτόματα «δεμένη» στην default exchange μέσω ενός routing key το οποίο είναι ίδιο με το όνομα της ουράς.

- **Dead Letter Exchange:** Αν δεν υπάρχει ταιριαστή ουρά για ένα μήνυμα, τότε αυτό εγκαταλείπεται. Η RabbitMQ προσφέρει μία AMQP επέκταση γνωστή ως «Dead Letter Exchange», η οποία παρέχει τη λειτουργικότητα για την αιχμαλώτιση μηνυμάτων τα οποία δεν μπορούν να μεταφερθούν. Οι dead letter exchanges (DLXs) αποτελούν κανονικές exchanges, των οποίων ο τύπος μπορεί να είναι οποιοσδήποτε από τους προαναφερθέντες και επιτρέπουν την επαναποστολή μηνυμάτων τα οποία είτε απορρίφθηκαν από έναν consumer, είτε έληξαν βάσει του μέγιστου χρόνου επιβίωσης ενός μηνύματος, είτε επειδή ξεπεράστηκε το μέγιστο δυνατό μήκος της ουράς στην οποία ανήκαν.



Εικόνα 10. Τυπική διαδρομή ενός μηνύματος στη RabbitMQ για τους τρεις συνηθέστερους exchange μηχανισμούς [34].

2.3.3.4. Queues

Μετά την δρομολόγηση που επιτελεί το exchange, το μήνυμα καταλήγει σε έναν ή περισσότερους buffers, οι οποίοι ονομάζονται ουρές και αποθηκεύουν προσωρινά τα ληφθέντα μηνύματα μέχρι να γίνει η επεξεργασία τους, Στην επιστήμη υπολογιστών, με τον όρο «ουρά» (queue) προσδιορίζεται μία ακολουθιακή δομή δεδομένων, καθώς τα αντικείμενα που περιέχει έχουν συγκεκριμένη θέση στη σειρά του συνόλου. Ουσιαστικά, πρόκειται για μία διαδικασία η οποία προσομοιώνει τον τρόπο λειτουργίας των παραδοσιακών «ουρών» στην καθημερινότητα μας, όπως η γραμμή αναμονής για εξυπηρέτηση σε ένα ταμείο και οι άνθρωποι που επιχειρούν να επιβιβαστούν σε έναν ανελκυστήρα. Σε μία τυπική ουρά, οι κύριες λειτουργίες της είναι δύο: ένα αντικείμενο μπορεί να προστεθεί στην ουρά ως το τελευταίο στοιχείο της (enqueue) και να αφαιρεθεί από την ουρά εφόσον συνιστά το πρώτο στοιχείο στην απαρίθμηση της (dequeue). Έτσι, γίνεται αντιληπτό πως στην περίπτωση ανταλλαγής πληροφοριών, το πρώτο στοιχείο που θα

εισαχθεί στο queue θα καταναλωθεί και ως πρώτο από τον consumer, ενώ το αντικείμενο της ουράς που προστέθηκε ως τελευταίο πρέπει να περιμένει το τέλος όλων των προηγούμενων ώστε να ξεκινήσει η δική του επεξεργασία. Αυτή η μέθοδος οργάνωσης των στοιχείων σε μια ακολουθιακή δομή δεδομένων, είναι γνωστή με το ακρωνύμιο FIFO (First In, First Out) και αποτελεί τον πλέον διαδεδομένο τύπο ουράς στις πρακτικές εφαρμογές τους, γεγονός που ισχύει και στην περίπτωση της RabbitMQ. Το σημαντικότερο μέλημα υπηρεσιών διαμεσολάβησης μηνυμάτων είναι, στις περισσότερες περιπτώσεις, η εξυπηρέτηση του πλήθους πελατών σύμφωνα με τη χρονική σειρά λήψης των αιτημάτων τους, με αποτέλεσμα η FIFO προσέγγιση να συνιστά και τη «default» κατηγορία ουράς στην RabbitMQ.

Πέραν των καθιερωμένων queues, η RabbitMQ παρέχει και άλλα είδη ουρών, με χαρακτηριστικό παράδειγμα τις ουρές προτεραιότητας (*priority queues*). Συγκεκριμένα, επιτρέπεται η προσθήκη της ιδιότητας «προτεραιότητας» στις κλασικές ουρές, μέσω της ανάθεσης ενός ακέραιου αριθμού εντός του διαστήματος [1, 255]. Ωστόσο, προτείνεται η ανάθεση τιμών μεταξύ του 1 και του 5, διότι όσο υψηλότερες τιμές προτεραιότητας υπάρχουν, τόσο περισσότερη υπολογιστική ισχύς και πόροι μνήμης απαιτούνται. Το γεγονός αυτό οφείλεται στην ανάγκη εσωτερικής συντήρησης μιας «υπο-ουράς» (sub-queue) από τη RabbitMQ για κάθε αρίθμηση προτεραιότητας από το 1 μέχρι τη μέγιστη επιλεγμένη τιμή, ώστε να έχει πρόσβαση ανά πάσα χρονική στιγμή και με ομαδοποιημένο τρόπο στην προτεραιότητα κάθε στοιχείου. Επιπλέον, εφόσον χρησιμοποιηθούν priority queues, το enqueueing και dequeueing των μηνυμάτων με FIFO διαδικασία δεν είναι εγγυημένη, καθώς ένα αίτημα πολύ υψηλής προτεραιότητας ενδέχεται να επεξεργαστεί νωρίτερα από ένα αίτημα που, έφτασε πρωτύτερα στην ουρά μεν, αλλά είναι ελάσσονος σημασίας δε.

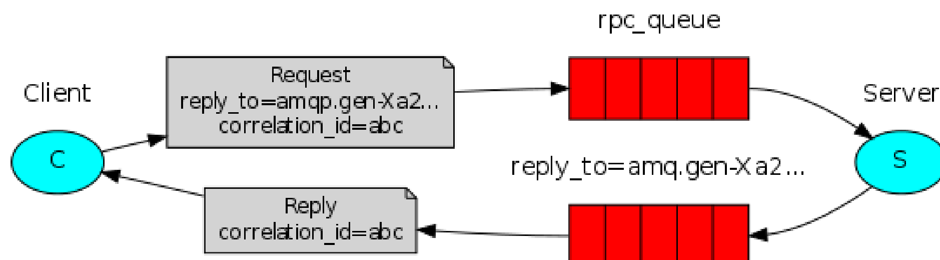
Στη RabbitMQ, οι ουρές μπορούν να είναι ανθεκτικές (*durable*) ή εφήμερες (*transient*). Η βασική τους διαφορά έγκειται στο ότι τα μεταδεδομένα (metadata) μίας durable queue αποθηκεύονται στο δίσκο, ενώ τα μεταδεδομένα μίας transient queues αποθηκεύονται στη μνήμη, όταν είναι εφικτό. Συνεπώς, οι ανθεκτικές ουρές μπορούν να επιβιώσουν σε περίπτωση επανεκκίνησης του διακομιστή, συμπεριλαμβανόμενων και των μηνυμάτων που έχουν δημοσιευθεί σε αυτές ως «persistent». Αντίθετα, οι εφήμερες ουρές αδυνατούν εκ σχεδιασμού να επιβιώσουν ένα server restart, ενώ και όλα τα μηνύματα τους θα απορριφθούν.

2.3.3.5. RPC στη RabbitMQ

Στο πρόβλημα το οποίο πραγματεύεται η τρέχουσα εργασία, γίνεται εύκολα αντιληπτό πως τα εμπλεκόμενα μέρη δύνανται να ανήκουν σε διαφορετικά δίκτυα, γεγονός που αποτελεί το επικρατέστερο σενάριο στον πραγματικό κόσμο. Συγκεκριμένα, στο κεντρικό σενάριο της παρούσας εργασίας ο χρήστης επιθυμεί την επίλυση μιας σειράς προβλημάτων συνδυαστικής βελτιστοποίησης έχοντας συνδεθεί σε ένα δίκτυο Α. Προκειμένου να το καταφέρει αυτό, αποστέλλει τα δεδομένα στον message broker (RabbitMQ), ο οποίος βρίσκεται σε ένα δίκτυο Β. Στη συνέχεια, αναλαμβάνει εκείνος να προωθήσει τα δεδομένα του προβλήματος στη βιβλιοθήκη επίλυσης αυτής της κατηγορίας προβλημάτων, η οποία με τη σειρά της μπορεί να λειτουργεί σε ένα ξεχωριστό δίκτυο Γ. Επομένως, πρόκειται για ένα πρόβλημα όπου χρειάζεται να εκτελεστεί μια λειτουργία σε έναν απομακρυσμένο server, με το τελικό αποτέλεσμα μαζί με όσα μεταδεδομένα κριθούν απαραίτητα να επιστρέφεται πίσω στο χρήστη.

Για την επίλυση τέτοιων ζητημάτων, η RabbitMQ παρέχει τη δυνατότητα κατασκευής ενός RPC συστήματος, αποτελούμενο από έναν client και έναν scalable RPC server. Προκειμένου να επιτευχθεί αυτό, εισάγονται δύο έννοιες οι οποίες επιτελούν καθοριστικό ρόλο:

- Callback queue: Αξιοποιώντας το πακέτο RabbitMQ.Client, μπορεί να οριστεί η διεύθυνση της ουράς στην οποία, μετά την επίλυση του απεσταλμένου προβλήματος, θα επιστραφεί το τελικό αποτέλεσμα. Συγκεκριμένα, στο μήνυμα που θα λάβει ο consumer του message broker, θα υπάρχει ενσωματωμένη η ουρά επιστροφής (callback queue).
- Correlation ID: Η στείρα χρήση μιας ουράς επιστροφής προϋποθέτει τη δημιουργία μίας ουράς ανά request, γεγονός που συνεπάγεται αφενός μεγάλο υπολογιστικό κόστος και αφετέρου πιθανές χρονικές καθυστερήσεις. Προκειμένου να αντιμετωπιστεί αυτό, η RabbitMQ επιτρέπει την ανάθεση μίας μοναδικής τιμής για κάθε request, μέσω ενός άλλου property, γνωστού ως αναγνωριστικό συσχέτισης (correlation id). Ουσιαστικά, αποτελεί ένα μοναδικό αναγνωριστικό, το οποίο δείχνει τη συσχέτιση μεταξύ του προβλήματος της κληθείσας ουράς και του αποτελέσματος που δέχεται η ουρά επιστροφής. Η callback queue, μαζί με τα δεδομένα του προβλήματος στέλνει ως property το ανωτέρω αναγνωριστικό, το οποίο κατ' αντιστοιχία ορίζεται με την ίδια τιμή στα properties της κληθείσας ουράς, ώστε η callback queue να αναγνωρίσει πως το μήνυμα προορίζεται για αυτήν. Με τον τρόπο αυτό, καθίσταται εφικτή η δημιουργία μονάχα μίας callback queue ανά client.



Εικόνα 11. Σχηματική αναπαράσταση του RPC μοντέλου στη RabbitMQ [35].

2.3.4. Εγκατάσταση σε Ubuntu

Προκειμένου να δημιουργηθούν οι κατάλληλες συνθήκες για την αναπαράσταση ενός ρεαλιστικού προβλήματος, απαιτείται, όπως προαναφέρθηκε, η εγκατάσταση του RabbitMQ Server σε δίκτυο διαφορετικό από το τοπικό δίκτυο εργασίας. Για το σκοπό αυτό, πραγματοποιήθηκε σύνδεση μέσω VPN σε διαθέσιμο server του Εθνικού Μετσόβιου Πολυτεχνείου. Κατόπιν, κατασκευάστηκε, μέσω της πλατφόρμας VMWare, ένα εικονικό μηχάνημα Linux, το οποίο διαθέτει λειτουργικό σύστημα Ubuntu 22.04 με 16 GB RAM, 4 πυρήνες επεξεργασίας και 40GB αποθηκευτικό χώρο (SSD).

Προκειμένου να εγκατασταθεί ο RabbitMQ Server στο παραπάνω περιβάλλον, εκτελέστηκαν τα ακόλουθα βήματα:

1. **Εγκατάσταση της γλώσσας Erlang:** Αποτελεί απαραίτητη προϋπόθεση για τη λειτουργία του server της RabbitMQ.
2. **Προσθήκη του RabbitMQ repository στο Ubuntu:** Η ομάδα της RabbitMQ συντηρεί ένα advanced package tool (APT) repository στο PackageCloud. Πρόκειται για ένα directory που περιέχει όλα τα απαραίτητα πακέτα λογισμικού, τα οποία ενημερώνονται διαρκώς με τις τελευταίες εκδόσεις του προϊόντος. Έτσι, λήφθηκαν (εντολή: *apt-get*) και αναβαθμίστηκαν (εντολές: *apt-update* & *apt-upgrade*) τα απαιτούμενα advanced package tools στο εικονικό μηχάνημα.
3. **Εγκατάσταση και εκκίνηση του RabbitMQ Server:** Αφότου ολοκληρωθεί η προσθήκη του repository, καθίσταται πλέον εφικτή η εγκατάσταση του RabbitMQ Server package. Στη συνέχεια, ο διακομιστής του message broker μπορεί να τεθεί σε λειτουργία μέσω της συστημικής εντολής *systemctl start rabbitmq-server.*

2.4. Οικοσύστημα .NET

Η ανάπτυξη του κώδικα στην πλευρά του client που αποστέλλει τα δεδομένα προς επίλυση, των web APIs που απαιτούνται αλλά και τα services που υλοποιούν τις λειτουργικότητες των publishers και των consumers στη RabbitMQ πραγματοποιήθηκε στην πλατφόρμα ανάπτυξης εφαρμογών .NET. Στη συνέχεια, παρουσιάζεται τόσο μία σύντομη ανασκόπηση της ιστορίας του .NET developer platform, όσο και οι λόγοι για τους οποίους επιλέχθηκε το συγκεκριμένο περιβάλλον για την συγγραφή των υπηρεσιών που απαρτίζουν αυτήν την πτυχιακή εργασία.

2.4.1. Ιστορία του .NET

Στα τέλη της δεκαετίας του 1990, η Microsoft ξεκίνησε τη δημιουργία ενός νέου runtime, δηλαδή τη συγκρότηση λογισμικού για την εκτέλεση των απαραίτητων εντολών κατά τη διάρκεια της εκτέλεση ενός προγράμματος, οι οποίες διασφαλίζουν την ομαλή και ορθή λειτουργία του. Συγκεκριμένα, το runtime αυτό ήταν διαχειρίσιμου κώδικα (managed code), δηλαδή απαιτεί προκειμένου να εκτελεστεί την παρουσία μίας Common Language Infrastracture (CLI). Με τον όρο αυτό, ο οποίος επινοήθηκε από την ίδια τη Microsoft, γίνεται αναφορά σε ένα ενιαίο σύνολο προδιαγραφών που περιγράφουν τον εκτελέσιμο κώδικα και ένα περιβάλλον εκτέλεσης που επιτρέπει τη χρήση πολλών high-level γλωσσών προγραμματισμού σε διάφορες υπολογιστικές πλατφόρμες, χωρίς να χρειαστεί να τροποποιούνται κάθε φορά. Παράλληλα, αναπτύχθηκε μία νέα γλώσσα προγραμματισμού επονομαζόμενη ως C#, η οποία ανακοινώθηκε για πρώτη φορά το 2000, αποτελώντας ουσιαστικά τη συνέχεια της Visual Basic. Ο συνδυασμός του runtime με την C# αποτελεί αναπόσπαστο κομμάτι της .NET πλατφόρμας, στο επίκεντρο της οποίας βρίσκεται το .NET Framework, το περιβάλλον πολυγλωσσικής (multi-language) ανάπτυξης και εκτέλεσης κώδικα για την κατασκευή και τρέξιμο των επιθυμητών web services. Η υλοποίηση αυτή συνέχισε την ανάπτυξη της για αρκετά χρόνια, συνιστώντας ένα ιδιόκτητο λογισμικού κλειστού κώδικα (proprietary closed-source software).

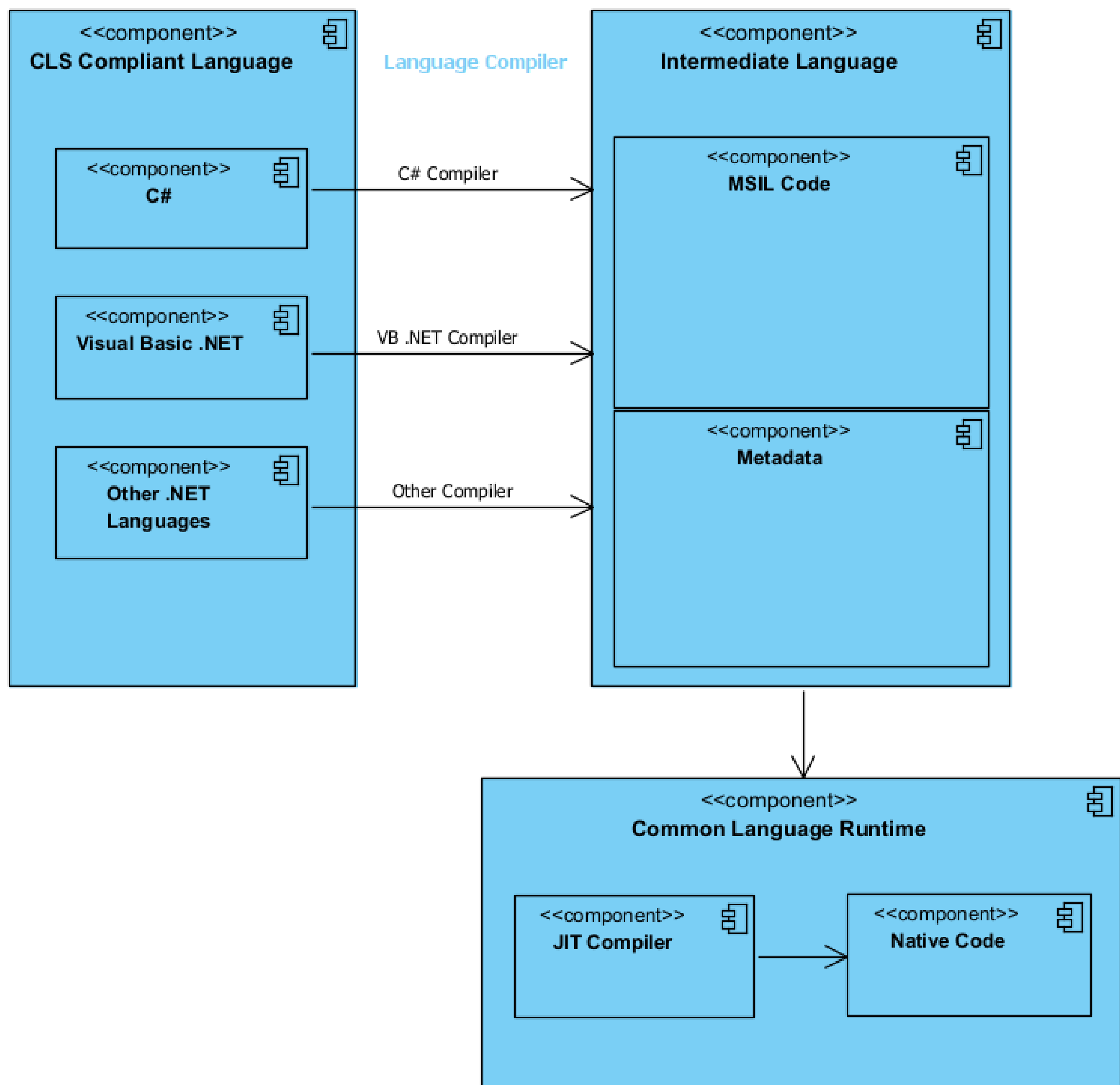
Το Νοέμβριο του 2014, η Microsoft εισήγαγε στην αγορά το .NET Core ως διάδοχο του .NET Framework. Σε αντίθεση με τον προκάτοχό του, το .NET Core αποτελεί

λογισμικό ανοιχτού κώδικα, το οποίο μάλιστα δύναται να λειτουργήσει σε πολλές υπολογιστικές πλατφόρμες (open-source, cross-platform software). Θεωρήθηκε ως μια ανασχεδιασμένη έκδοση του .NET, βασισμένη στην απλουστευμένη μορφή των class libraries και έκτοτε τέθηκε ως βάση για όλες τις μελλοντικές πλατφόρμες, οι οποίες πρακτικά πρόκειται για στοχευμένες βελτιώσεις και εμβαθύνσεις του .NET Core. Από το 2021, όταν και κυκλοφόρησε το .NET 6, το .NET Core και το .NET Framework ενοποιήθηκαν προκειμένου να παρέχεται μία ενιαία πλατφόρμα για τις εφαρμογές όλων των ειδών. Σήμερα, η τελευταία έκδοση μακράς υποστήριξης (LTS -Long-Term Support) του .NET, η οποία χρησιμοποιήθηκε για τη συγγραφή του κώδικα στην παρούσα εργασία, είναι το .NET 8, το οποίο κυκλοφόρησε το Νοέμβριο του 2023 [36].

2.4.2. Εργαλεία και εξαρτήματα

Πέραν του .NET Framework και του .NET Core, που συναποτελούν το θεμελιώδες δομικό συστατικό του .NET οικοσυστήματος, υπάρχει και μία πληθώρα άλλων στοιχείων τα οποία είναι υπεύθυνα για χρήσιμες λειτουργικότητες της ανάπτυξης λογισμικού. Τα πιο διαδεδομένα και σημαντικά εξ αυτών, είναι τα ακόλουθα:

- **Common Language Runtime:** Στον πυρήνα του .NET οικοσυστήματος βρίσκεται ο CLR (Common Language Runtime). Ο CLR είναι υπεύθυνος για τη διαχείριση της εκτέλεσης κώδικα που έχει γραφτεί σε διάφορες γλώσσες του .NET. Παρέχει υπηρεσίες όπως διαχείριση μνήμης, συλλογή αχρησιμοποίητων αντικειμένων (garbage collection) και ασφάλεια. Ο κώδικας που είναι γραμμένος σε C#, F#, Visual Basic ή άλλες γλώσσες του .NET, μεταγλωττίζεται σε μια ενδιάμεση γλώσσα (Intermediate Language - IL) που ο CLR μπορεί να εκτελέσει.



Εικόνα 12. Διαδικασία εκτέλεσης μίας .NET εφαρμογής. Ο κώδικας μεταφράζεται στην ενδιάμεση γλώσσα (MSIL) μέσω του compiler, ώστε τελικά να καταλήξει σε native κώδικα [37].

- **ASP.NET:** Πρόκειται για ένα framework μέσα στο οικοσύστημα .NET για τη δημιουργία δυναμικών και διαδραστικών web εφαρμογών και web APIs. Περιλαμβάνει τεχνολογίες όπως το ASP.NET MVC, το οποίο ακολουθεί τις αρχές της δομής Model-View-Controller και το ASP.NET Core για την επίτευξη σύγχρονου web development.

- **Entity Framework:** Αποτελεί ένα πλαίσιο αντιστοίχισης αντικειμένων με σχέσεις (Object-Relational Mapping - ORM) για την πλατφόρμα .NET. Απλοποιεί τις αλληλεπιδράσεις με βάσεις δεδομένων, επιτρέποντας στους προγραμματιστές να εργαστούν με βάσεις δεδομένων χρησιμοποιώντας αντικειμενοστραφείς έννοιες. Το Entity Framework Core είναι η σύγχρονη, cross-platform έκδοση του Entity Framework.

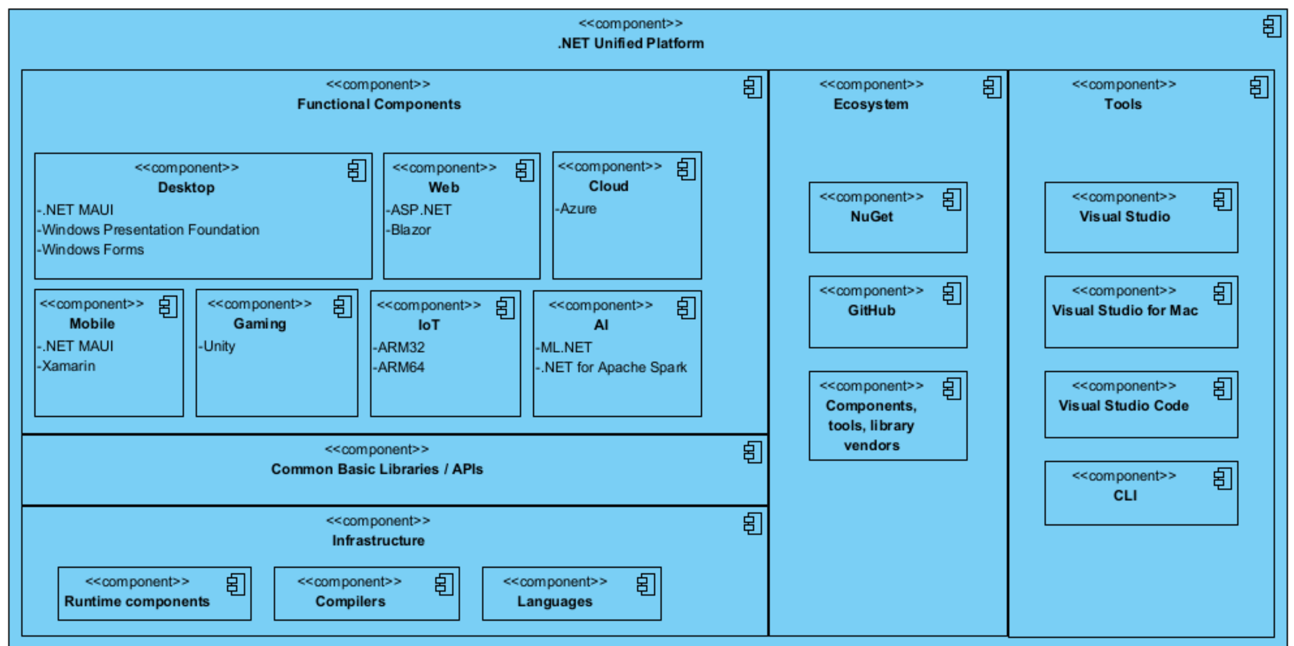
- **Xamarin:** Συνιστά ένα σύνολο εργαλείων και βιβλιοθηκών για τη δημιουργία cross-platform εφαρμογών για κινητές συσκευές με λειτουργικό σύστημα iOS, Android και macOS. Αξιοποιεί γλώσσες του περιβάλλοντος .NET, όπως η C#, και επιτρέπει το διαμοιρασμό κώδικα (code sharing) μεταξύ πλατφορμών, μειώνοντας έτσι τον χρόνο και την προσπάθεια ανάπτυξης των εφαρμογών.

- **Blazor:** Αποτελεί ένα web framework εντός του περιβάλλοντος .NET που επιτρέπει στους προγραμματιστές να δημιουργούν διαδραστικές ιστοσελίδες χρησιμοποιώντας τη γλώσσα C# και το .NET Core αντί για τη γλώσσα JavaScript. Προσφέρει δύο μοντέλα: το Blazor WebAssembly για client-side web εφαρμογές που και το Blazor Server για server-side web εφαρμογές.

- **Visual Studio:** Πρόκειται για το κύριο ενσωματωμένο περιβάλλον ανάπτυξης (integrated development environment - IDE) για την ανάπτυξη εφαρμογών στο περιβάλλον .NET. Προσφέρει λειτουργίες για τη συγγραφή και αποσφαλμάτωση κώδικα (coding – debugging), το testing και το deployment .NET εφαρμογών. Το Visual Studio Code είναι ένα εναλλακτικό, ελαφρύτερο περιβάλλον ανάπτυξης που χρησιμοποιείται επίσης για την ανάπτυξη στο .NET περιβάλλον.

- **NuGet:** Το NuGet είναι ένας διαχειριστής πακέτων (package manager) για το .NET το οποίο επιτρέπει την εύκολη και γρήγορη προσθήκη βιβλιοθηκών, components και εξαρτήσεων (dependencies) σε ένα project [38]. Έτσι, η διαδικασία ενσωμάτωσης κώδικα και βιβλιοθηκών από τρίτους (third-party code) στις εφαρμογές καθίσταται απλή και σαφής.

Κατά τη δημιουργία των απαραίτητων υποδομών για την υλοποίηση του σκοπού της εργασίας, χρησιμοποιήθηκε εκτενώς το .NET Core. Συγκεκριμένα, αξιοποιήθηκε το ASP.NET Core για την κατασκευή δύο web APIs έχοντας ως βάσει το template ASP.NET Core Web API που παρέχεται στο .NET οικοσύστημα, ενώ όλος ο κώδικας γράφτηκε σε γλώσσα C#. Ως περιβάλλον ανάπτυξης χρησιμοποιήθηκε η τελευταία έκδοση του Visual Studio IDE, το Visual Studio 2022.



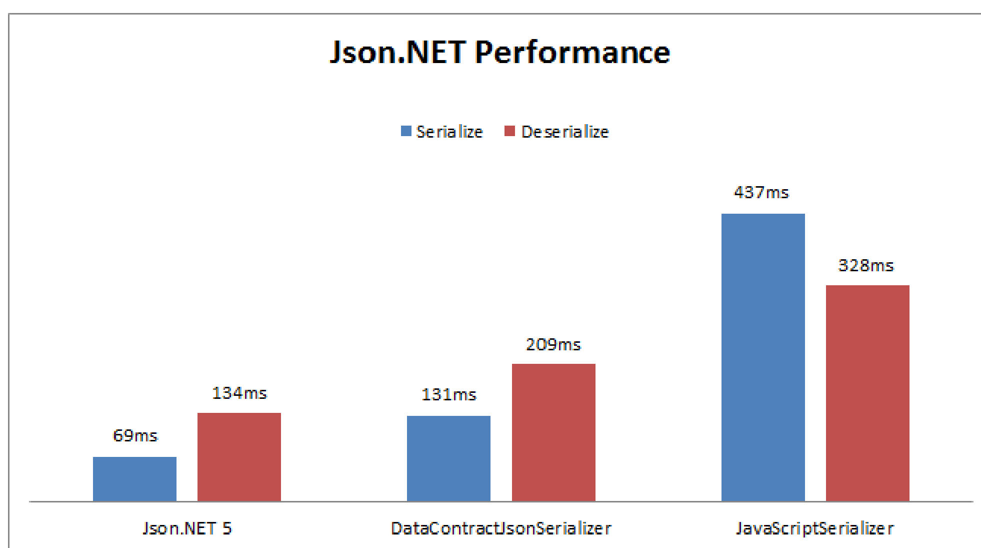
Εικόνα 13. Τα εξαρτήματα του .NET ως μία ενοποιημένη πλατφόρμα, σχεδιασμένα στο Visual Paradigm βάσει της επίσημης παρουσίασης του .NET 6 [39].

2.4.3. Βιβλιοθήκες και frameworks

Για τη συγγραφή του κώδικα ο οποίος να εξυπηρετεί τις απαιτήσεις του προβλήματος προς μελέτη, καθοριστικό ρόλο επιτέλεσε η πληθώρα χρήσιμων βιβλιοθηκών που παρέχει η .NET και μπορεί να εγκατασταθεί εύκολα μέσω του NuGet package manager, όπως αναφέρθηκε παραπάνω. Ειδικότερα, οι βιβλιοθήκες των οποίων η αναφορά κρίνεται απαραίτητη για την πληρέστερη κατανόηση του υπολογιστικού μοντέλου είναι οι ακόλουθες:

- RabbitMQ .NET Client: Αποτελεί την υλοποίηση μίας open-source, client βιβλιοθήκης για τη C# και άλλες γλώσσες του .NET περιβάλλοντος, η οποία χρησιμοποιεί το πρωτόκολλο επικοινωνίας AMQP 0-9-1. Συντηρείται από την ίδια τη RabbitMQ και διαθέτει όλες τις λειτουργικότητες που υποστηρίζει ο message broker της, όπως τα queues, τα exchanges και η RPC υλοποίηση στο pub/sub μοντέλο επικοινωνίας. Προκειμένου να εγκατασταθεί, πρέπει να προστεθεί το namespace RabbitMQ.Client στο αρχείο του κώδικα και να δημιουργηθεί ένα *ConnectionFactory*, το οποίο, κατά την εκτέλεση του προγράμματος, εγκαθιδρύει μία σύνδεση με το διακομιστή της RabbitMQ [40].
- Google.OrTools: Όπως θα αναφερθεί παρακάτω, το σενάριο το οποίο μελετήθηκε αφορά στη διαχείριση μηνυμάτων μεγάλου πλήθους και όγκου για προβλήματα Επιχειρησιακής Έρευνας (Operations Research), όπως για παράδειγμα είναι η δρομολόγηση οχημάτων. Έτσι, για τη διευκόλυνση δημιουργίας ενός επιλύτη προβλημάτων τέτοιου είδους, αξιοποιήθηκε η σχετική βιβλιοθήκη της Google, με όνομα OR-Tools, η οποία παρέχεται σε μορφή «τυλίγματος» (wrapper) για τη γλώσσα .NET, διαθέτοντας solvers για πολλές κατηγορίες προβλημάτων, όπως το Vehicle Routing Problem, το Knapsack Problem αρά και διάφορες παραλλαγές του Constraint Programming.

- Newtonsoft.Json: Τα αποσταλμένα αρχεία από τον εκάστοτε client, αλλά και τα responses μεταξύ των API και η τελική απάντηση που θα επιστρέψει στον πελάτη, επιλέχθηκε να βρίσκονται σε .JSON format. Επομένως, σημαντικό ρόλο επιτελούσε η χρήση αποδοτικών, τόσο υπολογιστικά όσο και χρονικά, συναρτήσεων για τη σειριοποίηση και αποσειριοποίηση (serialization – deserialization) των json αρχείων κατά τη λειτουργία του μοντέλου. Για το σκοπό αυτό, αξιοποιήθηκε το Json.NET, το οποίο παρέχεται από τη Newtonsoft και συνιστά ένα framework υψηλής απόδοσης για το .NET, όντας ταχύτερο από τους ενσωματωμένους JSON serializers του .NET Core. Επιπλέον, επιτρέπει την εύκολη τροποποίηση των εσοχών του παραγόμενου JSON, γεγονός που καθιστά εφικτή την αναπαραγωγή ενός human-readable μηνύματος κονσόλας ή αρχείου ως απάντηση στο αίτημα του πελάτη.



Εικόνα 14. Σύγκριση της απόδοσης του Json.NET με δύο άλλα διαδεδομένα JSON frameworks: το DataContractJsonSerializer και το JavaScript Serializer [41].

2.4.4. .NET σε Linux

Η αναβάθμιση του .NET Framework στο .NET Core η οποία εμπεριείχε την καθιέρωση του .NET οικοσυστήματος ως open-source, κατέστησε εφικτή την εγκατάσταση και χρήση του .NET Framework και σε άλλα λειτουργικά συστήματα πέραν των Microsoft Windows, όπως τα unix-like Operating Systems και το MacOS. Επιπλέον, η Microsoft έδωσε τη δυνατότητα εκτέλεσης μιας εφαρμογής που έχει δημιουργηθεί σε ένα υπολογιστικό σύστημα, σε οποιοδήποτε άλλο σύστημα ανεξαρτήτως του Λειτουργικού του.

Στην περίπτωση του Linux, η οποία και αξιοποιήθηκε στην παρούσα διπλωματική εργασία, ο χρήστης καλείται να επιλέξει μεταξύ δύο προοπτικών: την εγκατάσταση του πακέτου ανάπτυξης του .NET (.NET SDK – Standard Development Kit) ή την εγκατάσταση του ASP.NET Core Runtime, με τη θεμελιώδης διαφορά τους να σχετίζεται με τις δυνατότητες που παρέχουν. Ειδικότερα, το .NET SDK επιτρέπει την ανάπτυξη εφαρμογών του .NET περιβάλλον σε Linux, καθιστώντας το προτιμητέα επιλογή αν ο χρήστης επιθυμεί να υλοποιήσει την εφαρμογή από την αρχή μέχρι το τέλος σε αυτό το υπολογιστικό σύστημα. Όσον αφορά το ASP.NET Core Runtime, αυτό αποτελεί ένα πολύ

ελαφρύτερο πακέτο, το οποίο επιτρέπει το «τρέξιμο» εφαρμογών που δημιουργήθηκαν μέσω του .NET και δεν περιέχουν το runtime, δηλαδή τις βασικές, απολύτως απαραίτητες υπηρεσίες για την εκτέλεση του κώδικα. Στην παρούσα εργασία, η ανάπτυξη όλου του λογισμικού πραγματοποιήθηκε στο Visual Studio 2022 σε Microsoft Windows περιβάλλον, οπότε χρειάστηκε η εγκατάσταση στο Linux μόνον του ASP.NET Core Runtime, το οποίο ουσιαστικά αποτελεί μία «εικονική μηχανή» η οποία τρέχει μία εφαρμογή αφαιρώντας όλη την αλληλεπίδραση με το βασικό λειτουργικό σύστημα, καθιστώντας το αδιάφορο ως προς την εκτέλεσή της.

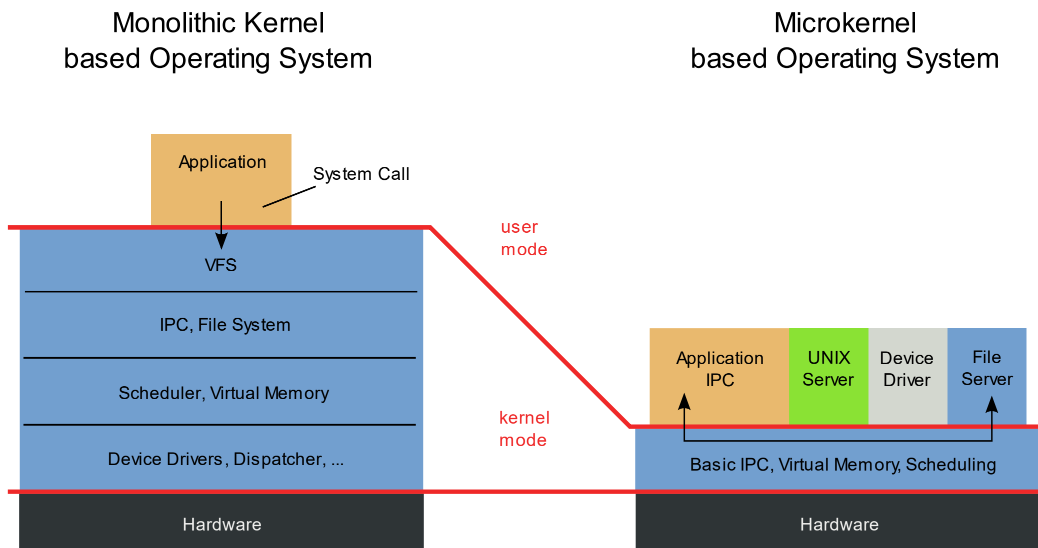
2.4.5. Windows vs. Linux

Όπως είναι λογικό, η κατασκευή ενός λειτουργικού λογισμικού για την επίλυση υπολογιστικών προβλημάτων απαιτεί σημαντικές αρχιτεκτονικές αποφάσεις, με μία εκ των οποίων να είναι τα λειτουργικά συστήματα τα οποία θα χρησιμοποιηθούν. Δεδομένου ότι το λογισμικό αναπτύσσεται με γνώμονα να επιτρέπει στον χρήστη να το καλεί τόσο από Windows, όσο και από Linux, η επιλογή του λειτουργικού συστήματος στο οποίο θα φιλοξενηθούν οι διακομιστές που απαιτούνται επιτελεί καθοριστικό ρόλο στην ομαλή διεξαγωγή ολόκληρης της διαδικασίας: από την αποστολή των request, τη διαχείριση τους εντός του message broker, την επίλυση τους και τελικά της επιστροφής του αποτελέσματος στον client.

Η ανάγκη απόφασης ανάμεσα στα Windows και τα Linux αποτελεί ένα από τα μακροβιότερα και ισχυρότερα διλήμματα στην Ανάπτυξη Εφαρμογών. Πρόκειται για δύο λύσεις υψηλού επιπέδου, οι οποίες έχουν εδραιωθεί για περισσότερες από τρεις δεκαετίες ως τα πλέον διαδεδομένα και επιτυχημένα Operation Systems σε όλον τον κόσμο. Παρουσιάζουν, ωστόσο, αρκετές διαφορές στις υλοποιήσεις τους, οι οποίες παρουσιάζονται συνοπτικά παρακάτω [42]:

- **Αρχιτεκτονική:** Οι περισσότερες παραλλαγές του Linux, όπως το Ubuntu και το Debian, είναι χτισμένες γύρω από τον Linux Kernel, ο οποίος είναι μονολιθικός, δηλαδή ολόκληρο το OS λειτουργεί στον χώρο του πυρήνα (kernel space). Αντίθετα, τα Windows χρησιμοποιούν έναν άλλο τύπο πυρήνα, τον micro-kernel, ο οποίος περιέχει λιγότερο πηγαίο κώδικα, καταλαμβάνοντας έτσι λιγότερο χώρο από

τον monolithic kernel αλλά ταυτόχρονα μειώνοντας την αποτελεσματικότητα του συστήματος.



Εικόνα 15. Διαφορά των συστατικών στοιχείων και της δομής του πυρήνα στις περιπτώσεις των monolithic kernel OS (αριστερά) και micro-kernel OS (δεξιά) [43].

- **Ασφάλεια:** Τα Windows συνιστούν ένα πολύ εκτεταμένο χρησιμοποιούμενο OS, το οποίο επικεντρώθηκε εξαρχής στην ευκολία χρήσης σε έναν υπολογιστή ενός χρήστη. Επιπλέον, πρόκειται για ένα εμπορικό λειτουργικό σύστημα κλειστού κώδικα (closed-source) επί πληρωμή. Τα δύο παραπάνω χαρακτηριστικά κατατάσσουν τα Windows ως ευάλωτα σε θέματα ασφάλειας και σταθερότητας. Από την άλλη πλευρά, τα Linux βασίζονται σε αρχιτεκτονική πολλών χρηστών (multi-user architecture), γεγονός που τα καθιστά πολύ πιο σταθερά από ένα single-user OS όπως τα Windows. Παράλληλα, τα Linux παρουσιάζουν εξαιρετική προστασία απέναντι σε κακόβουλα λογισμικά, διότι το σύστημα τους διαθέτει πολλά built-in features για την αύξηση της ασφάλειάς τους, όπως οι default χαμηλές άδειες χρήστη. Ακόμη, ο πηγαίος κώδικας τους είναι ανοικτός και προσβάσιμος από όλη την κοινότητα χρηστών τους, επιτρέποντας έτσι την άμεση παρατήρηση, διερεύνηση και επίλυση κάθε προβλήματος που προκύπτει.

- **Πόροι συστήματος:** Η σημαντικότερη διαφορά ανάμεσα στα Windows και τα Linux αναφορικά με τα resources που καταναλώνουν είναι η γραφική διεπαφή χρήστη (Graphical User Interface – GUI). Συγκεκριμένα, τα Windows περιλαμβάνουν πάντοτε GUI για την εκτέλεση εντολών και διεργασιών, ενώ τα Linux περιορίζονται στη χρήση Γραμμής Εντολών (Command Line) για την πραγματοποίηση των αντίστοιχων διαδικασιών. Η υποχρεωτική παρουσία του GUI καθιστά τα Windows πιο απαιτητικά σε πόρους συστήματος και επιδεινώνει την απόδοσή τους.

- **Κλιμακωσιμότητα:** Τα Linux συστήματα διαχειρίζονται με αποτελεσματικό τρόπο μεγάλες ποσότητες κίνησης και δεδομένων, με αποτέλεσμα να θεωρούνται ιδανικά

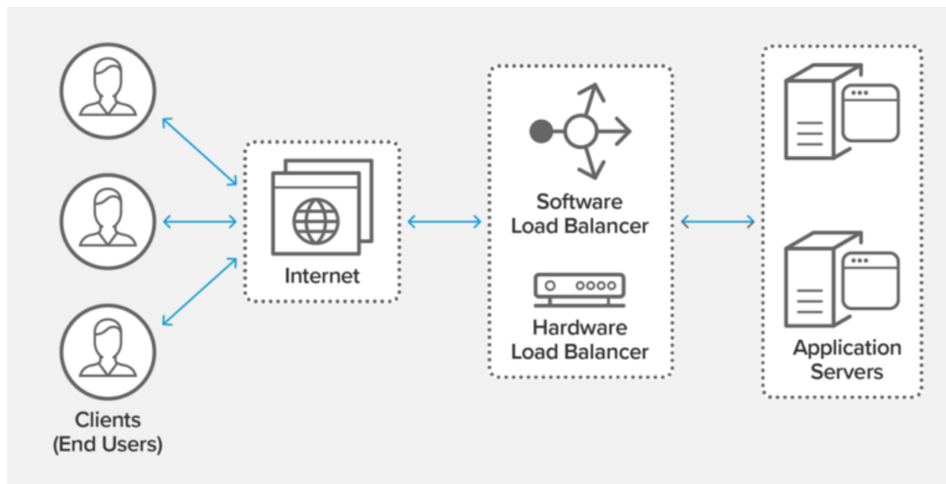
για υπολογισμούς υψηλής απόδοσης και άλλες εφαρμογές μεγάλης κλίμακας. Επίσης, προσφέρουν μια ποικιλία εργαλείων διαχείρισης για την επίτευξη ομαλού scaling των εφαρμογών τους.

Συνολικά, γίνεται εμφανές πως τόσο τα Windows, όσο και τα Linux διαθέτουν πλεονεκτήματα και αποτελούν λογικές επιλογές για να φιλοξενήσουν servers. Στην παρούσα εργασία, διερευνάται η κατανάλωση πόρων, η ταχύτητα απόκρισης και γενικά το performance ενός Software-as-a-Service μοντέλου επίλυσης υπολογιστικά ακριβών προβλημάτων, το οποίο αξιοποιεί τη χρήση ενός message broker για την αποστολή και λήψη μηνυμάτων προς τον ξεχωριστό server όπου και επιλύονται τα προβλήματα. Τα σημαντικότερα γνωρίσματα που πρέπει να περιλαμβάνουν οι servers που χρησιμοποιούνται είναι η αποδοτική διαχείριση των πόρων, η υψηλή ασφάλεια των δεδομένων, η δυνατότητα αποτελεσματικής κλιμάκωσης του μοντέλου και η μείωση των καθυστερήσεων στη μετάδοση των δεδομένων. Συνεπώς, η αναμφισβήτητη υπεροχή που παρουσιάζουν οι Linux Servers στους τομείς αυτούς, τους καθιστούν ως την καταλληλότερη επιλογή για την πλαισίωση του λογισμικού της τρέχουσας εργασίας.

2.5. Εξισορρόπηση φορτίου (Load Balancing)

Στη σημερινή εποχή, η ασύλληπτη αύξηση των χρηστών και οι διαφορετικές απαιτήσεις του καθενός ως προς την υπηρεσία που τους παρέχεται, η επικερδής και αποδοτική χρήση των υπολογιστικών πόρων καθίσταται επιτακτική ανάγκη. Μία από τις συνηθέστερες μεθόδους διαχείρισης της κίνησης σε ένα δίκτυο αποτελεί η *εξισορρόπηση φορτίου λογισμικού* (software load balancing), η οποία αξιολογεί τα αιτήματα των πελατών εξετάζοντας τα χαρακτηριστικά του επιπέδου εφαρμογής, όπως η διεύθυνση IP, η HTTP επικεφαλίδα και τα περιεχόμενα του request. Στη συνέχεια, ο load balancer ελέγχει τους servers και επιλέγει σε ποιον θα προωθήσει το αίτημα, ανάλογα με τα κριτήρια τα οποία είναι προγραμματισμένος να λαμβάνει υπόψιν του. Ουσιαστικά, ο load balancer λειτουργεί ως ένα reverse proxy το οποίο δίνει στον πελάτη μία virtual IP address η οποία αναπαριστά την εφαρμογή. Ο client συνδέεται σε αυτή την εικονική διεύθυνση, και ο load balancer αποφασίζει αν το connection του συγκεκριμένου πελάτη πρέπει να προωθηθεί σε έναν server, βάσει του αλγορίθμου που υλοποιεί.

Η διανομή του φόρτου εργασίας σε πολλαπλούς διακομιστές μέσω του load balancing μπορεί να κάνει ένα δίκτυο πιο αποτελεσματικό και αξιόπιστο. Η χωρητικότητα του δικτύου αυξάνεται, διότι χρησιμοποιούνται όλοι οι διαθέσιμοι servers με τον αποδοτικότερο δυνατό τρόπο. Έτσι, δεν δημιουργούνται καταστάσεις όπου τα workloads βρίσκονται κολλημένα σε υπερφορτωμένους διακομιστές ενώ ταυτόχρονα άλλοι servers παραμένουν αχρησιμοποίητοι, με αποτέλεσμα το δίκτυο να γίνεται ταχύτερο. Επιπλέον, το load balancing διασφαλίζει την αδιάλειπτη λειτουργία του συστήματος όταν ένας διακομιστής αποτύχει, καθοδηγώντας τα φορτία που προορίζονταν για τον προβληματικό server σε άλλους, λειτουργικούς servers [44]. Συνεπώς, η εφαρμογή τεχνικών εξισορρόπησης φορτίου συμβάλλει στην αύξηση τόσο της *διαθεσιμότητας* (availability) όσο και της *ανθεκτικότητας* σε σφάλματα (fault tolerance) του συστήματος.



Εικόνα 16. Ενδεικτικό διάγραμμα για έναν τυπικό load balancer σε μία υπηρεσία [45].

Για την υλοποίηση του load balancing, υπάρχει πληθώρα διαθέσιμων αλγορίθμων οι οποίοι διακρίνονται σε δύο ευρείες, γενικότερες κατηγορίες: τους *στατικούς* (static) και τους *δυναμικούς* (dynamic) αλγορίθμους. Παρακάτω, αναλύονται οι κατηγορίες αυτές, παρατίθενται οι σημαντικότεροι αλγόριθμοι τους και αναλύεται η λογική γύρω από την οποία έχει αναπτυχθεί ο καθένας τους.

2.5.1. Στατικό load balancing

Ως static load balancing ορίζεται η μέθοδος του καταμερισμού του εισερχόμενου φορτίου σε έναν διακομιστή χρησιμοποιώντας αλγορίθμους που διαθέτουν πληροφορία *εκ των προτέρων* (a priori) για τους υπάρχοντες servers στο κατανεμημένο σύστημα. Τα μοντέλα στατικού load balancing έχουν ένα προκαθορισμένο load schedule, περιλαμβάνοντας ένα προκαθορισμένο, σταθερό ποσό φορτίου που μπορεί να διανεμηθεί σε άλλα συστήματα. Το γεγονός αυτό, συνεπάγεται πως δεν απαιτείται επικοινωνία με τους servers σε πραγματικό χρόνο. Έτσι, το static load balancing ενδείκνυται για περιπτώσεις συστημάτων με σχετικά σταθερό όγκο φορτίου, δηλαδή χαμηλή διακύμανση στο πλήθος των εισερχόμενων requests. Σε αυτόν τον τύπο εξισορρόπησης, η κίνηση (traffic) διαιρείται ίσα στους διακομιστές, ενώ ένα φορτίο που έχει ήδη κατανεμηθεί σε κάποιον server δε μπορεί να μεταφερθεί ξανά στη διάρκεια του runtime [46].

Οι βασικοί αλγόριθμοι που απαρτίζουν το στατικό load balancing, είναι οι ακόλουθοι [47]:

- **Round robin:** Κατανέμει την κίνηση σε ένα σύνολο από servers με κυκλική σειρά, χρησιμοποιώντας το Domain Name System (DNS), δηλαδή τη διαδικασία μετάφρασης του ονόματος ενός domain στην IP διεύθυνση που του αντιστοιχεί.

- **Weighted round robin:** Όπως και ο round robin αλγόριθμος, διαιρεί την κίνηση στου servers κυκλικά, επιτρέποντας ωστόσο ταυτόχρονο στον διαχειριστή του συστήματος να αναθέσει διαφορετικά βάρη (weights) σε κάθε server. Έτσι, ένας server με μεγαλύτερο συντελεστή βάρους σημαίνει πως μπορεί να διαχειριστεί υψηλότερη κίνηση και αναμένεται να λάβει περισσότερο φορτίο, ενώ ένας server με

χαμηλότερο βάρος πρόκειται να δεχθεί μικρότερο αριθμό αιτημάτων και πληροφορίας.

- **IP Hash:** Συνδυάζει την πηγή (source) του εισερχόμενου φορτίου και τις IP διευθύνσεις των πιθανών προορισμών και τα μετατρέπει σε μία άλλη χαρακτηριστική τιμή *(hash)* μέσω μίας μαθηματικής συνάρτησης. Βάσει αυτής της τιμής, το connection ανατίθεται σε έναν συγκεκριμένο server.

2.5.2. Δυναμικό load balancing

Οι dynamic load balancers αξιοποιούν πληροφορίες για το performance του κάθε κόμβου του συστήματος προκειμένου να πάρουν αποφάσεις για την εξισορρόπηση του φορτίου. Αποτελεί μία πιο ευέλικτη μέθοδο load balancing η οποία μπορεί να αναγνωρίσει δυναμικά τόσο το ποσό του φορτίου που χρειάζεται να διανεμηθεί κατά τη διάρκεια του runtime, όσο και ποιο σύστημα πρέπει να επωμιστεί το φορτίο αυτό. Προκειμένου να επιτευχθεί η δυναμική κατανομή του φόρτου εργασίας, απαιτείται ενεργή επικοινωνία σε πραγματικό χρόνο με τους διακομιστές, ενώ η εκ των προτέρων γνώση δεδομένων για το σύστημα δεν κρίνεται απαραίτητη. Σε αντίθεση με την στατική εξισορρόπηση, το δυναμικό load balancing επιτρέπει την *αναμεταφορά* (retransferring) του κατανεμημένου φορτίου σε άλλους servers, ώστε να μην υπάρχουν resources που παραμένουν ανεκμετάλλευτα. Επομένως, το δυναμικό load balancing προορίζεται για συστήματα με υψηλή διακύμανση είτε ως προς τον όγκο ή/και ως προς την συχνότητα του εισερχόμενου φορτίου [46].

Παρακάτω, απαριθμούνται οι σημαντικότεροι αλγόριθμοι που περιλαμβάνει το δυναμικό load balancing [47]:

- **Least connection:** Ελέγχει ποια instances έχουν τις λιγότερες ανοιχτές συνδέσεις τη δεδομένη χρονική στιγμή και στέλνει την κίνηση σε αυτούς. Ωστόσο, προϋποθέτει την παραδοχή πως όλα τα connections απαιτούν περίπου ίση υπολογιστική ισχύ.

- **Weighted least connection:** Κατ' αντιστοιχία με τη weighted εκδοχή του round robin αλγορίθμου στο στατικό load balancing, ο weighted least connection επεκτείνει τον αλγόριθμο Ελάχιστων Συνδέσεων δίνοντας στον διαχειριστή τη δυνατότητα να αναθέσει διαφορετικά βάρη σε κάθε instance, τα οποία αντιπροσωπεύουν το βαθμό στον οποίον το καθένα είναι ικανό να χειριστεί αποτελεσματικά πολλές συνδέσεις.

- **Weighted response time:** Υπολογίζει τη μέση τιμή του χρόνου απόκρισης που έχει το κάθε instance και στη συνέχεια το συνδυάζει με τον αριθμό των συνδέσεων που έχει αναλάβει το αντίστοιχο instance, επιτυγχάνοντας την «κανονικοποίηση» του μέσου χρόνου απόκρισης. Με αυτόν τον τρόπο, στέλνει την εισερχόμενη κίνηση στους κόμβους οι οποίοι παρουσιάζουν τον ταχύτερο μέσο χρόνο απόκρισης και διασφαλίζει την ταχύτερη εξυπηρέτηση των χρηστών.

2.6. Επιχειρησιακή Έρευνα

Όπως αναφέρθηκε προηγουμένως, για τον έλεγχο της λειτουργίας και της απόδοσης του υπολογιστικού μοντέλου που κατασκευάστηκε αξιοποιήθηκαν προβλήματα που ανήκουν στην ευρύτερη κατηγορία της Επιχειρησιακής Έρευνας (Operations Research). Με τον όρο αυτό περιγράφεται η κατασκευή και χρήση μαθηματικών μοντέλων με σκοπό τη βελτίωση λήψης αποφάσεων (decision-taking). Στη σημερινή εποχή, θεμελιώδεις στόχοι των διαφόρων εταιρειών και οργανισμών συναποτελούν η μεγιστοποίηση του κέρδους και της παραγωγής, η ελαχιστοποίηση του κόστους και η βέλτιστη διαχείριση του μεγάλου αριθμού ανθρώπων που συνιστούν τους πόρους της παραγωγικής διαδικασίας. Έτσι, έχει υλοποιηθεί πληθώρα μεθόδων, αλγορίθμων και προγραμμάτων, τα οποία αποσκοπούν στην ορθή και αποδοτική, χρονικά και ποσοτικά, επίλυση των προαναφερθέντων προβλημάτων.

Παρακάτω παρατίθενται τα σημαντικότερα προβλήματα λήψης αποφάσεων που επιδιώκει να λύσει ο κλάδος της Επιχειρησιακής Έρευνας [48]:

- **Ανάθεση (Assignment Problems)**
  Στα προβλήματα ανάθεσης, πρέπει να ληφθεί απόφαση για τον τρόπο ανάθεσης εργασιών σε πόρους. Χαρακτηριστικά παραδείγματα αποτελούν η επιτέλεση των ανατιθέμενων καθηκόντων από εργαζόμενους και η εξυπηρέτηση πελατών. Σε αυτόν τον τομέα, οι παράγοντες που λαμβάνονται υπόψιν περιλαμβάνουν το κόστος, τις διαθέσιμες δυνατότητες και τις προτεραιότητες των εργασιών.

- **Δρομολόγηση Οχημάτων (VRP – Vehicle Routing Problems)**
  Ο βασικός στόχος αυτής της κλάσης προβλημάτων είναι η βελτιστοποίηση των διαδρομών σε οχήματα που πραγματοποιούν, κατά κανόνα, παραδόσεις (deliveries) ή συλλογές προϊόντων (pickups). Επιδιώκει αφενός την αποφυγή υπερβολικών καθυστερήσεων και αφετέρου την ελαχιστοποίηση των αποστάσεων που διανύει το εκάστοτε όχημα σε συνδυασμό με τη μεγιστοποίηση του αριθμού παραλαβών ή παραδόσεων.

- **Αποθεματοποίηση (Inventory Problems)**
  Στα προβλήματα αποθεματοποίησης, επιδιώκεται η λήψη αποφάσεων για την κατάλληλη χρονική στιγμή και την ποσότητα των προϊόντων που πρέπει να παραγγελθούν ή να παραχωρηθούν, ώστε το κόστος αποθήκευσης και οι ελλείψεις να ελαχιστοποιηθούν.

- **Προγραμματισμός Παραγωγής (Production Planning Problems)**
  Τα προβλήματα αυτά επικεντρώνονται στο σχεδιασμό και διαχείριση της παραγωγικής διαδικασίας με σκοπό τόσο την αύξηση της απόδοσης όσο και τη μείωση του συνολικού κόστους

- **Αγορά και Διανομή (Supply Chain Problems)**
  Πρόκειται για προβλήματα βελτιστοποίησης της εφοδιαστικής αλυσίδας σε όλη τη διαδρομή της, από την αγορά των πρώτων υλών έως και τη διανομή των προϊόντων στους πελάτες.

- **Διαχείριση Εργασιών (Job Scheduling)**
  Το ζήτημα αυτό περιλαμβάνει ένα σύνολο εργασιών, οι οποίες πρέπει να εκτελεστούν από μια ή περισσότερες μηχανές. Κάθε εργασία έχει συγκεκριμένη διάρκεια εκτέλεσης και προτεραιότητα. Στόχος είναι η εύρεση ενός χρονικού προγράμματος εκτέλεσης που ελαχιστοποιεί είτε τον συνολικό χρόνο εκτέλεσης, ή άλλα κριτήρια όπως η μέγιστη καθυστέρηση μιας εργασίας.

Ο κλάδος του Operations Research αξιοποιεί ποικίλα μαθηματικά μοντέλα προκειμένου να εξάγει με αποδοτικό τρόπο επιστημονικώς ορθά συμπεράσματα στο εκάστοτε πρόβλημα που του ανατίθεται. Παρακάτω, παρουσιάζονται συνοπτικά ορισμένες από τις σημαντικότερες μεθόδους αντιμετώπισης προβλημάτων Επιχειρησιακής Έρευνας παρατίθενται παρακάτω:

- **Γραμμικός Προγραμματισμός (Linear Programming – LP)**
  Αποτελεί μία δημοφιλή τεχνική μαθηματικού προγραμματισμού που αποσκοπεί στη βελτιστοποίηση μιας γραμμικής συνάρτησης μεγίστου ή ελαχίστου, η οποία ονομάζεται αντικειμενική συνάρτηση (objective function) και υπόκειται σε ορισμένους γραμμικούς περιορισμούς. Πολλά προβλήματα της επιχειρησιακής έρευνας, όπως η διαχείριση αποθεμάτων και ο προγραμματισμός παραγωγής, μπορούν να μοντελοποιηθούν ως προβλήματα γραμμικού προγραμματισμού.

- **Δυναμικός Προγραμματισμός (Dynamic Programming – DP)**
  Η τεχνική αυτή χρησιμοποιείται για την επίλυση προβλημάτων όπου η βέλτιστη λύση εξαρτάται από τις βέλτιστες λύσεις μικρότερων υποπροβλημάτων. Βρίσκει εφαρμογή σε προβλήματα όπου η βέλτιστη διαδρομή για ένα υποσύνολο κόμβων μπορεί να αξιοποιηθεί για την εύρεση της βέλτιστης διαδρομής στο σύνολο των κόμβων. Χαρακτηριστικά παραδείγματα αυτού αποτελούν το Πρόβλημα του Πλανόδιου Πωλητή (Travelling Salesman Problem – TSP) και το Πρόβλημα Δρομολόγησης Οχημάτων (VRP), το οποίο συνιστά γενίκευση του TSP.

- **Προγραμματισμός Ακεραίων (Integer Programming – IP)**
  Στη μέθοδο αυτή, οι μεταβλητές που καλούνται να βελτιστοποιηθούν περιορίζονται σε ακέραιες τιμές. Χρησιμοποιείται κυρίως σε Προβλήματα Αναθέσεων, όπου οι εργασίες πρέπει να ανατεθούν σε ολόκληρους πόρους.

- **Προγραμματισμός Περιορισμών (Constraint Programming – CP)**
  Συνιστά μία τεχνική που αξιοποιείται στην επίλυση προβλημάτων όπου οι σχέσεις μεταξύ των μεταβλητών αναπαρίστανται μέσω περιορισμών. Σε αντίθεση με την παραδοσιακή βελτιστοποίηση, το CP επικεντρώνεται στην εύρεση εφικτών και όχι βέλτιστων λύσεων, διαθέτοντας συνήθως ένα ιδιαίτερα ευρύ σύνολο υποψήφιων λύσεων. Παρουσιάζει ιδιαίτερη χρησιμότητα σε περιπτώσεις προβλημάτων με πολλούς περιορισμούς ή μεταβλητών που διαθέτουν ακέραιους περιορισμούς. Συνεπώς, εφαρμόζεται συχνά σε προβλήματα Διαχείρισης Εργασιών και Κατανομής Πόρων, όπου πολλοί περιορισμοί πρέπει να τηρηθούν.

2.7. Γενικά για το Google OR-Tools

Το εργαλείο που αξιοποιήθηκε για την επίλυση των ζητούμενων προβλημάτων Επιχειρησιακής Έρευνας είναι το OR-Tools, λογισμικό ανοιχτού κώδικα της Google το οποίο και αποτελεί ένα από τα σημαντικότερα μέρη της τρέχουσας εργασίας. Στόχος των διάφορων λειτουργιών που παρέχει το λογισμικό, είναι η αναζήτηση της βέλτιστης λύσης σε προβλήματα που διαθέτουν ένα πολύ μεγάλο σύνολο πιθανών λύσεων. Το γεγονός αυτό επιτυγχάνεται μέσω μιας διαδικασίας που ονομάζεται *συνδυαστική βελτιστοποίηση*, η οποία σημαίνει την εύρεση ενός βέλτιστου αντικειμένου από ένα πεπερασμένο σύνολο αντικειμένων, όπου το σύνολο των εφικτών λύσεων είναι διακριτό ή μπορεί να μειωθεί σε ένα διακριτό σύνολο. Ακόμη, προκειμένου να είναι εφικτό από τον υπολογιστή να εντοπίσει μια βέλτιστη ή σχεδόν βέλτιστη λύση ανάμεσα στον πολύ μεγάλο όγκο πιθανών λύσεων, αξιοποιούνται σύγχρονοι, state-of-the-art αλγόριθμοι οι οποίοι συρρικνώνουν αποτελεσματικά το σύνολο αναζήτησης.

Το OR-Tools είναι γραμμένο σε C++, αλλά πέραν αυτής της γλώσσας προγραμματισμού, υποστηρίζει τη χρήση του επιπλέον στην .NET πλατφόρμα, τη Java και την Python. Στην παρούσα εργασία, για την ανάπτυξη του λογισμικού χρησιμοποιήθηκε η .NET πλατφόρμα, οπότε εγκαταστάθηκε και το πακέτο του OR-Tools στο αντίστοιχο περιβάλλον του .NET Framework.

Ως πλατφόρμα, το Google OR-Tools προσφέρει έτοιμα πακέτα επίλυσης για διάφορες κατηγορίες προβλημάτων συνδυαστικής βελτιστοποίησης. Τα σημαντικότερα εξ αυτών είναι τα ακόλουθα [49]:

- **Δρομολόγηση οχημάτων (VRP)**
  Αποτελεί ένα από τα πιο απαιτητικά και περίπλοκα ζητήματα που καλούνται να αντιμετωπίσουν οι εταιρείες στη σημερινή εποχή, λόγω της πληθώρας φυσικών, χρονικών και κάθε είδους περιορισμών που προκύπτουν. Έτσι, η Google παρέχει πολλές υποπαραλλαγές της απλούστερης περίπτωσης δρομολόγησης, προκειμένου να καλυφθούν οι περισσότερες ιδιαιτερότητες της αγοράς. Ενδεικτικά, παρατίθενται οι πιο συχνές περιπτώσεις παραλλαγών τις οποίες καλύπτει το OR-Tools:
  - Χωρικοί περιορισμοί
  - Παραλαβές και παραδόσεις
  - Περιορισμοί χρονικών παραθύρων
  - Περιορισμοί πόρων

- **Βελτιστοποίηση περιορισμών (Constraint Optimization)**
  Συνιστά ένα πρόβλημα υψηλής προτεραιότητας ιδίως για εταιρείες που οφείλουν να ενημερώνουν κάποιες παραμέτρους τους, όπως π.χ. το πρόγραμμα των εργαζόμενων, ανά τακτικά χρονικά διαστήματα. Για το λόγο αυτό, η Google προσφέρει το σύγχρονο, ισχυρό CP-SAT solver, το οποίο επεκτείνει τις δυνατότητες ενός παραδοσιακού CP solver αξιοποιώντας μεθόδους *ικανοποιησιμότητας (satisfiability)* για την εξαγωγή εφικτών λύσεων. Παράλληλα, η τέτοια προβλήματα μπορούν να επιλυθούν και μέσω του MPSolver Interface, το οποίο πρόκειται για ένα wrapper σύστημα που συμβάλλει στην επίλυση προβλημάτων Γραμμικού Προγραμματισμού και Μικτού Ακεραίου

Προγραμματισμού.

- **Πακετάρισμα αντικειμένων (Packing)**
  Άλλο ένα πρόβλημα που συναντάται καθημερινά στον επιχειρηματικό κόσμο, με χαρακτηριστικό παράδειγμα την ανάγκη φόρτωσης προϊόντων σε delivery οχήματα με αποδοτικό τρόπο. Οι βασικές κατηγορίες των packing προβλημάτων είναι το *Πρόβλημα του Σακιδίου (Knapsack Problem)* και του *Πακεταρίσματος σε Κάδους (Bin Packing)*. Για την πρώτη περίπτωση, διατίθεται πληθώρα αλγορίθμων στη βιβλιοθήκη του OR-Tools, ενώ ακόμη υπάρχουν και γενικότερες εκδόσεις του προβλήματος, όπως η περίπτωση πολλαπλών σακιδίων *(Multiple Knapsacks)*, η οποία επιλύνεται είτε μέσω του MPSolver είτε μέσω του CP-SAT Solver. Αντίστοιχα, το Bin Packing Problem, το οποίο επιδιώκει την εύρεση του ελάχιστου αριθμού κάδων που θα χωρέσει όλα τα διαθέσιμα προϊόντα, μπορεί να λυθεί χρησιμοποιώντας τον MPSolver wrapper.

Φυσικά, το Google OR-Tools δεν περιορίζεται στην αντιμετώπιση των παραπάνω τύπων προβλημάτων, αλλά αποτελεί μία γενικευμένη σουίτα λογισμικού ικανή να αντιμετωπίσει πρακτικά όλα τα είδη προβλημάτων Επιχειρησιακής Έρευνας. Ενδεικτικά, επισημαίνεται πως παρέχει λύσεις για προβλήματα *Μέγιστης – Ελάχιστης Ροής (Maximum – Minimum Flows)*, καθώς και προβλήματα συνδυαστικής βελτιστοποίησης όπως το *Πρόβλημα Ανάθεσης (Assignment Problem)* το οποίο επικεντρώνεται στη μελέτη εκτέλεσης ενός όγκου tasks από ένα πλήθος workers.

Για τους σκοπούς της παρούσας διπλωματικής εργασίας, αξιοποιήθηκε σε βάθος το Πρόβλημα Δρομολόγησης Οχημάτων (VRP), συνιστώντας σε κάθε αρχιτεκτονική του υπολογιστικού μοντέλου που κατασκευάστηκε το πρόβλημα προς μελέτη. Κρίνεται λοιπόν ωφέλιμη μία συνοπτική παρουσίαση της μεθόδου επίλυσής του σύμφωνα με το documentation που παρέχει η Google. Αρχικά, πρέπει να καθοριστούν οι απαραίτητες παράμετροι για την επίλυση της εκάστοτε βελτιστοποίησης στη δρομολόγηση, οι οποίες είναι οι ακόλουθες:

- Ένας *πίνακας αποστάσεων* (distance matrix) ο οποίος περιλαμβάνει τις αποστάσεις μεταξύ όλων των τοποθεσιών σε μέτρα.
- Το *πλήθος των τοποθεσιών*
- Ο *αριθμός των οχημάτων* του στόλου
- Ο *δείκτης της εναρκτήριας τοποθεσίας* (depot), θεωρώντας πως όλα τα οχήματα έχουν μία, κοινή αφετηρία.

Στο σημείο αυτό, αξίζει να σημειωθεί πως οι συντεταγμένες των τοποθεσιών δεν είναι απαραίτητες για την επίλυση του προβλήματος. Μοναδική προϋπόθεση είναι να διατίθεται ο πίνακας αποστάσεων που προαναφέρθηκε, ο οποίος σε ένα πρόβλημα με n τοποθεσίες προς επίσκεψη θα έχει διαστάσεις $n \cdot n = n^2$ στοιχεία, καθώς ουσιαστικά προκύπτει από τον υπολογισμό της απόστασης που παρουσιάζει κάθε τοποθεσία με όλες τις υπόλοιπες τοποθεσίες που στοιχειοθετούν το πρόβλημα. Παρακάτω, παρατίθεται ένα παράδειγμα δεδομένων εισόδου του προβλήματος με τη μορφή τοποθεσιών, καθώς και ο πίνακας αποστάσεων που προκύπτει βάσει των συγκεκριμένων συντεταγμένων με τη μορφή μίας κλάσης.

```
[(456, 320), # τοποθεσία 0 – αφετηρία των οχημάτων
(228, 0),    # τοποθεσία 1
(912, 0),    # τοποθεσία 2
(0, 80),     # τοποθεσία 3
(114, 80),   # τοποθεσία 4
(570, 160),  # τοποθεσία 5
(798, 160),  # τοποθεσία 6
(342, 240),  # τοποθεσία 7
(684, 240),  # τοποθεσία 8
(570, 400),  # τοποθεσία 9
(912, 400),  # τοποθεσία 10
(114, 480),  # τοποθεσία 11
(228, 480),  # τοποθεσία 12
(342, 560),  # τοποθεσία 13
(684, 560),  # τοποθεσία 14
(0, 640),    # τοποθεσία 15
(798, 640)]  # τοποθεσία 16
```

```
class DataModel
{
  // Πίνακας αποστάσεων του προβλήματος (n · n = n² = 16² = 256 στοιχεία)
  public long[,] DistanceMatrix = {
    { 0, 548, 776, 696, 582, 274, 502, 194, 308, 194, 536, 502, 388, 354, 468, 776, 662 },
    { 548, 0, 684, 308, 194, 502, 730, 354, 696, 742, 1084, 594, 480, 674, 1016, 868, 1210 },
    { 776, 684, 0, 992, 878, 502, 274, 810, 468, 742, 400, 1278, 1164, 1130, 788, 1552, 754 },
    { 696, 308, 992, 0, 114, 650, 878, 502, 844, 890, 1232, 514, 628, 822, 1164, 560, 1358 },
    { 582, 194, 878, 114, 0, 536, 764, 388, 730, 776, 1118, 400, 514, 708, 1050, 674, 1244 },
    { 274, 502, 502, 650, 536, 0, 228, 308, 194, 240, 582, 776, 662, 628, 514, 1050, 708 },
    { 502, 730, 274, 878, 764, 228, 0, 536, 194, 468, 354, 1004, 890, 856, 514, 1278, 480 },
    { 194, 354, 810, 502, 388, 308, 536, 0, 342, 388, 730, 468, 354, 320, 662, 742, 856 },
    { 308, 696, 468, 844, 730, 194, 194, 342, 0, 274, 388, 810, 696, 662, 320, 1084, 514 },
    { 194, 742, 742, 890, 776, 240, 468, 388, 274, 0, 342, 536, 422, 388, 274, 810, 468 },
    { 536, 1084, 400, 1232, 1118, 582, 354, 730, 388, 342, 0, 878, 764, 730, 388, 1152, 354 },
    { 502, 594, 1278, 514, 400, 776, 1004, 468, 810, 536, 878, 0, 114, 308, 650, 274, 844 },
    { 388, 480, 1164, 628, 514, 662, 890, 354, 696, 422, 764, 114, 0, 194, 536, 388, 730 },
    { 354, 674, 1130, 822, 708, 628, 856, 320, 662, 388, 730, 308, 194, 0, 342, 422, 536 },
    { 468, 1016, 788, 1164, 1050, 514, 514, 662, 320, 274, 388, 650, 536, 342, 0, 764, 194 },
    { 776, 868, 1552, 560, 674, 1050, 1278, 742, 1084, 810, 1152, 274, 388, 422, 764, 0, 798 },
    { 662, 1210, 754, 1358, 1244, 708, 480, 856, 514, 468, 354, 844, 730, 536, 194, 798, 0 }
  };

  // Πλήθος οχημάτων του προβλήματος
  public int VehicleNumber = 4;

  // Αφετηρία ως δείκτης του συνόλου συντεταγμένων (0 = 1η τοποθεσία)
  public int Depot = 0;
};
```

Θεωρώντας πως ο πίνακας αποστάσεων έχει δημιουργηθεί, η επίλυση πλέον μπορεί να ξεκινήσει ακολουθώντας μία σειρά προκαθορισμένων βημάτων, στα οποία καλούνται συναρτήσεις της βιβλιοθήκης OR-Tools. Ενδεικτικά, παρατίθεται η Main μέθοδος για ένα Vehicle Routing πρόβλημα στη C# όπως εντοπίζεται στο documentation του OR-Tools. Σε κάθε καλούμενη συνάρτηση έχουν προστεθεί σχόλια για την αποσαφήνιση του σκοπού και της λειτουργικότητάς της.

```
public static void Main(String[] args)
{
    // Φόρτωση του ανωτέρω μοντέλου δεδομένων με τον πίνακα αποστάσεων, το πλήθος των οχημάτων
    και τη θέση αφετηρίας
    DataModel data = new DataModel();

    // Δημιουργία του Routing Index Manager, το οποίο αναθέτει και χρησιμοποιεί εσωτερικά δείκτες για τα
    στοιχεία του πίνακα
    RoutingIndexManager manager = new RoutingIndexManager(data.DistanceMatrix.GetLength(0),
    data.VehicleNumber, data.Depot);

   // Δημιουργία του μοντέλου δρομολόγησης.
   RoutingModel routing = new RoutingModel(manager);

    // Δημιουργία και εγγραφή ενός transit callback, δηλαδή μίας συνάρτησης που υπολογίζει δυναμικά τις
    μεταβατικές αποστάσεις που προκύπτουν μεταξύ των σημείων κατά την επίλυση.
    int transitCallbackIndex = routing.RegisterTransitCallback((long fromIndex, long toIndex) =>
    {
        // Μετατροπή των δεικτών σε κόμβους για την επίλυση του προβλήματος
        var fromNode = manager.IndexToNode(fromIndex);
        var toNode = manager.IndexToNode(toIndex);
        return data.DistanceMatrix[fromNode, toNode];
    });

    // Καθορισμός του κόστους κάθε ακμής (διαδρομής)
    routing.SetArcCostEvaluatorOfAllVehicles(transitCallbackIndex);

    // Προσθήκη του περιορισμού απόστασης που επιτρέπεται να διανύσει κάθε όχημα (3 km).
    routing.AddDimension(transitCallbackIndex, 0, 3000, true, "Distance");
    RoutingDimension distanceDimension =  routing.GetMutableDimension("Distance");
    distanceDimension.SetGlobalSpanCostCoefficient(100);

    //Καθορισμός της πρώτης ευριστικής της λύσης.
    RoutingSearchParameters searchParameters =
    operations_research_constraint_solver.DefaultRoutingSearchParameters();
    searchParameters.FirstSolutionStrategy = FirstSolutionStrategy.Types.Value.PathCheapestArc;

    //Κλήση της συνάρτησης επίλυσης βάσει των παραμέτρων.
    Assignment solution = routing.SolveWithParameters(searchParameters);

    // Εκτύπωση της λύσης στην κονσόλα.
    PrintSolution(data, routing, manager, solution);
}
```

Η εκτέλεση του ανωτέρω προβλήματος θα εκτυπώσει στην κονσόλα τα κάτωθι αποτελέσματα, τα οποία παράλληλα μπορούν να αναπαρασταθούν με το γράφημα που ακολουθεί.

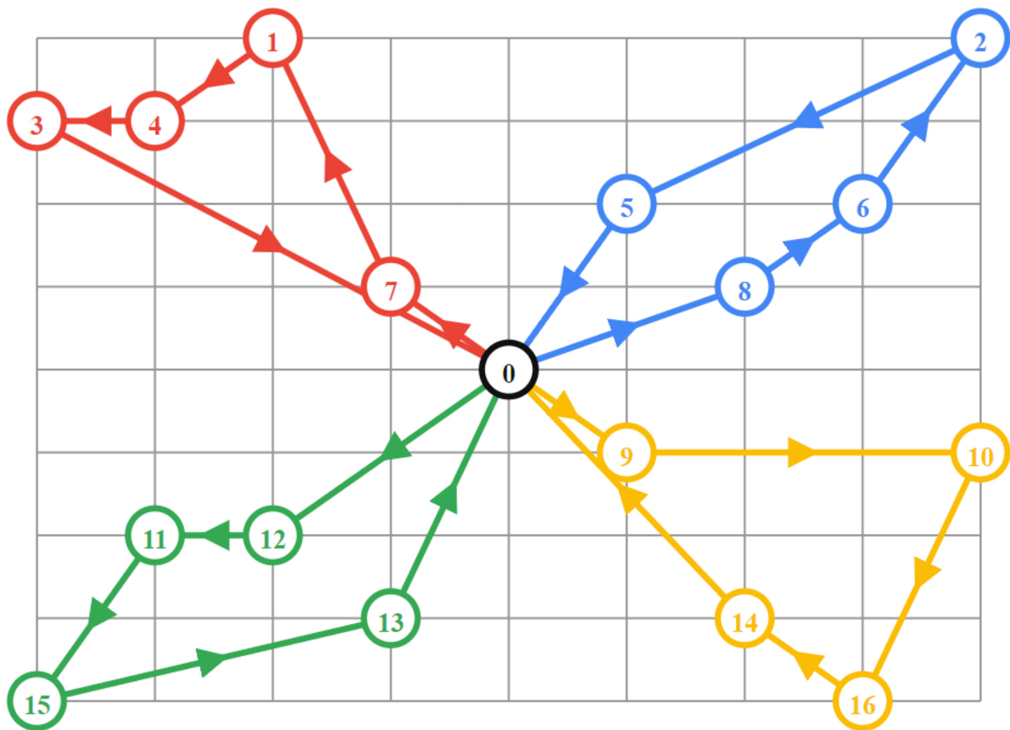
```
Route for vehicle 0:
 0 -> 8 -> 6 -> 2 -> 5 -> 0
Distance of route: 1552m

Route for vehicle 1:
 0 -> 7 -> 1 -> 4 -> 3 -> 0
Distance of route: 1552m

Route for vehicle 2:
 0 -> 9 -> 10 -> 16 -> 14 -> 0
Distance of route: 1552m

Route for vehicle 3:
 0 -> 12 -> 11 -> 15 -> 13 -> 0
Distance of route: 1552m

Total distance of all routes: 6208m
```



3. Μελέτη περίπτωσης

3.1. Επεξήγηση του προβλήματος

Στην περιγραφή του κινήτρου που αποτέλεσε εφαλτήριο για την παρούσα διπλωματική εργασία, έγινε αναφορά στην κατασκευή ενός υπολογιστικού προβλήματος και την επίλυσή του μέσω της σύνθεσης μιας υπολογιστικής υπηρεσίας, η οποία βασίζεται στις αρχές του Software as a Service και αξιοποιεί τεχνικές δρομολόγησης μηνυμάτων μεταξύ συστημάτων και δικτύων για την αποστολή δεδομένων και τη λήψη αποκρίσεων. Συγκεκριμένα, επιλέχθηκε η μελέτη προβλημάτων που υπάγονται στην γενικότερη κατηγορία της Επιχειρησιακής Έρευνας (Operations Research), καθώς αφενός υπάρχει γενικότερη έλλειψη SaaS υπηρεσιών και εφαρμογών για τέτοιου είδους προβλήματα στην αγορά και αφετέρου το πακέτο OR-Tools της Google διατίθεται δωρεάν για το .NET οικοσύστημα, επιτρέποντας τη χρήση του μέσω της πλήρους, εύχρηστης βιβλιοθήκης Google.OrTools. Σημειώνεται ότι η επιλογή αυτή δεν είναι δεσμευτική και ότι στη θέση του OR-Tools θα μπορούσε να είναι οποιοδήποτε απαιτητικό λογισμικό επίλυσης μαθηματικών προβλημάτων, υπολογιστικής προσομοίωσης, κλπ.

Έτσι, δόθηκε η δυνατότητα ανάπτυξης μίας solving υπηρεσίας η οποία να δέχεται ως είσοδο ένα JSON αρχείο στο οποίο περιέχονται τα δεδομένα του προβλήματος με το όνομα «JobData», τα μεταδεδομένα που σχετίζονται με το συγκεκριμένο πρόβλημα όπως η χρονική στιγμή αποστολής του προς επίλυση («Metadata»), καθώς και το είδος του προβλήματος Επιχειρησιακής Έρευνας στην οποία αντιστοιχεί («ProblemType»). Όσον αφορά το τελευταίο στοιχείο του JSON αρχείου, υλοποιήθηκε ως ένα public enum διαθέσιμο από κάθε κλάση του solver, με κάθε τιμή του να αντιστοιχεί σε ένα OR πρόβλημα. Για παράδειγμα, αν ένας χρήστης καλέσει την υπηρεσία προκειμένου να επιλύσει ένα Πρόβλημα Δρομολόγησης Οχημάτων, οφείλει να αποστείλει τις συντεταγμένες των θέσεων τις οποίες επιθυμεί να επισκεφθεί, τον αριθμό των διαθέσιμων οχημάτων για την εκτέλεση αυτών των εργασιών καθώς και να ορίσει πως πρόκειται για VRP πρόβλημα, το οποίο θα αντιστοιχιστεί στην ακέραια τιμή του σχετικού enum και μέσα στο JSON θα λάβει την τιμή 0.

Το πρόβλημα το οποίο μελετήθηκε και αξιοποιήθηκε είναι το Vehicle Routing Problem. Αποτελώντας ένα από τα πιο διαδεδομένα και χρήσιμα προβλήματα για κάθε είδους εταιρεία που εμπλέκεται με προβλήματα διανομής, ο έλεγχος της ταχύτητας και ορθότητας της επίλυσής του παρουσιάζει αυξημένη πρακτική σημασία από άλλα πιο θεωρητικά προβλήματα της Επιχειρησιακής Έρευνας, όπως είναι λ.χ. το Πρόβλημα του Πλανόδιου Πωλητή. Επιπλέον, το VRP μπορεί εύκολα να κλιμακωθεί ως προς τον όγκο και την πολυπλοκότητα των δεδομένων, απλώς αυξάνοντας σημαντικά τις τοποθεσίες που πρέπει να επισκεφθούν ή εισάγοντας περισσότερους περιορισμούς, όπως χρονικά παράθυρα εξυπηρέτησης (time windows) και λιγότερο αριθμό οδηγών. Για τους λόγους αυτούς, θεωρήθηκε κατάλληλο για τους σκοπούς της εργασίας, ως μια περίπτωση προβλήματος που επιτρέπει την εμβάθυνση στην τεχνολογία του message broker που υλοποιήθηκε, προκειμένου να βρεθούν οι υποπαραλλαγές αιτημάτων τις οποίες εξυπηρετεί αποδοτικότερα

και εκείνες στις οποίες χωλαίνει, αντίστοιχα. Πάντως, στη θέση του VRP θα μπορούσε να βρίσκεται οποιοδήποτε απαιτητικό υπολογιστικό πρόβλημα.

Για τα δεδομένα του προβλήματος, χρησιμοποιήθηκε ένα υποθετικό οδικό δίκτυο, προκειμένου να δημιουργηθεί ένας πίνακας συντεταγμένων οι οποίες να βρίσκονται ενός συγκεκριμένου χωρικού εύρους, το οποίο να δύναται να αντιπροσωπεύσει σε ικανοποιητικό βαθμό τις πραγματικές απαιτήσεις δρομολόγησης μιας επιχείρησης. Συγκεκριμένα, εφόσον ο θεμελιώδης στόχος ήταν η διερεύνηση και σύγκριση ορισμένων αρχιτεκτονικών δόμησης ενός υπολογιστικού μοντέλου και όχι η αποτελεσματικότητα του solver, πως οι n αυτές τυχαίες συντεταγμένες μπορούν να επιτελέσουν το ρόλο των τοποθεσιών προς επίσκεψη και οι n^2 μεταξύ τους αποστάσεις, αντίστοιχα, το ρόλο του πίνακα αποστάσεων όπως αναλύθηκε στο προηγούμενο κεφάλαιο. Η σημαντικότερη και μοναδική υποχρεωτική παράμετρος είναι το πλήθος των οχημάτων που διατίθενται για την εκτέλεση των δρομολογίων, ενώ το πρόβλημα μπορεί να επεκταθεί μέσω της εισαγωγής νέων παραμέτρων, όπως περιορισμοί χωρητικότητας των φορτηγών (capacity constraints), χρονικοί περιορισμοί (time windows constraints) και εφαρμογή ποινών σε περίπτωση καθυστέρησης της διανομής όταν αυτή υπερβαίνει μία τιμή (penalties).

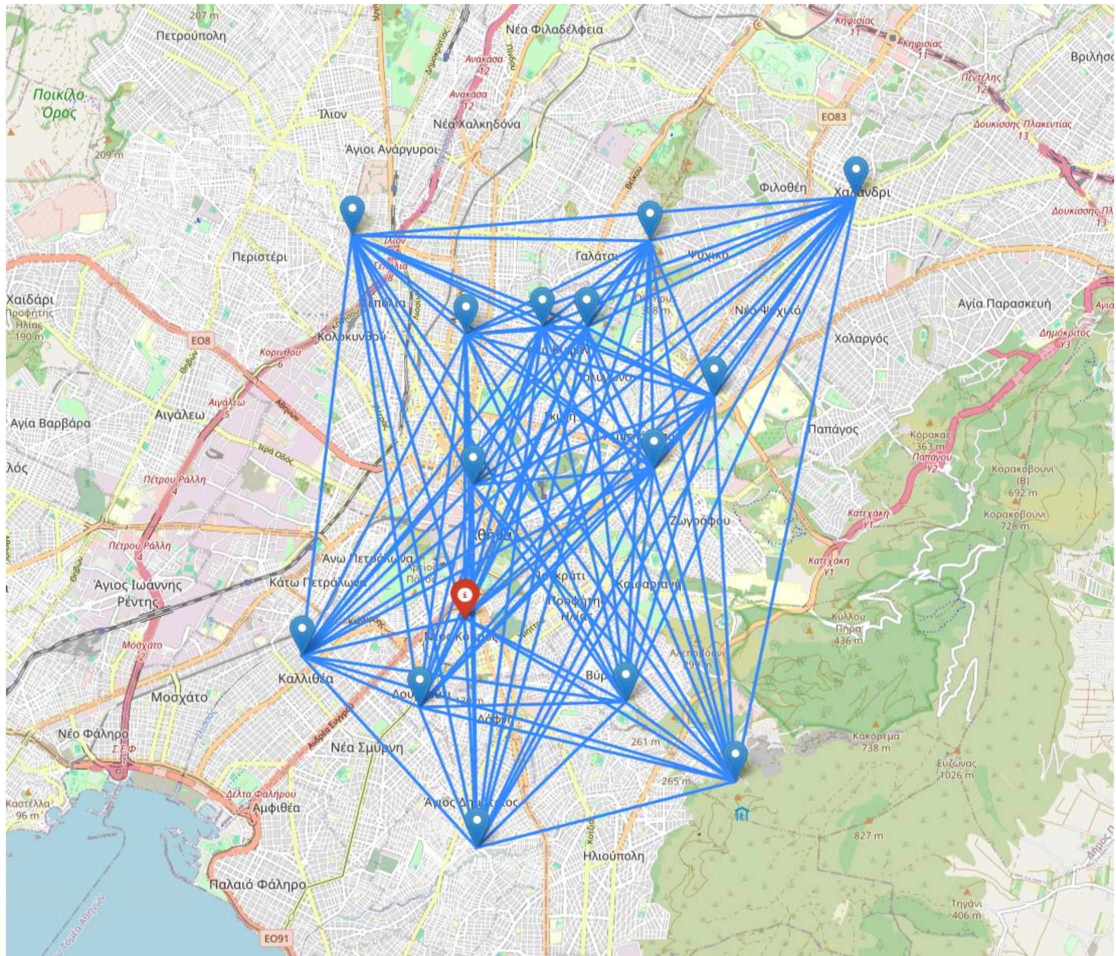
Όπως γίνεται εύκολα αντιληπτό, η απαίτηση σε πόρους και συνακόλουθα η χρονική διάρκεια επίλυσης ενός τέτοιου είδους προβλήματος παρουσιάζει υψηλή μεταβλητότητα, καθώς η αυξομείωση της απόστασης των τοποθεσιών προς επίσκεψη και του πλήθους των οχημάτων μπορεί να επηρεάσει σημαντικά το χρόνο που απαιτείται για να βρεθεί μία σχεδόν βέλτιστη λύση. Σε ένα τυπικό υπολογιστή, η διάρκεια επίλυσης κυμαίνεται από λίγα δευτερόλεπτα για απλά προβλήματα, μέχρι και αρκετές ώρες για περιπτώσεις με πολλούς σύνθετους περιορισμούς. Προκειμένου να υπάρξει υψηλή κατανάλωση των υπολογιστικών πόρων και να μπορέσει να μελετηθεί το σενάριο αποστολής μεγάλου αριθμού μηνυμάτων, τα οποία θα συνυπάρχουν ταυτόχρονα στις ουρές, επιλέχθηκε η δόμηση προβλημάτων τα οποία δεν επιλύονται σχεδόν ακαριαία, αλλά απαιτούν διάστημα μεταξύ 30 δευτερολέπτων και μερικών λεπτών.

3.2. Δημιουργία σεναρίων μελέτης

Για τη δημιουργία των διάφορων προβλημάτων, αξιοποιήθηκαν στο μέγιστο βαθμό τόσο το documentation της Google OR-Tools πλατφόρμας, όσο και η δυνατότητα αυτοματοποίησης της γένεσης δεδομένων με χρήση της ψευδοτυχαιότητας. Ειδικότερα, κατασκευάστηκαν services στο .NET περιβάλλον, τα οποία κατά την εκτέλεσή τους παράγουν δεδομένα στη μορφή που πρέπει να αποσταλούν από το χρήστη προς το διακομιστή για επίλυση, ώστε να μην απαιτείται περεταίρω τροποποίηση μετά τη δημιουργία τους. Προκειμένου να παραχθεί μεγάλος και έγκυρος όγκος δεδομένων σε σύντομο χρονικό διάστημα, αξιοποιήθηκε το πακέτο Random του .NET Framework, το οποίο περιλαμβάνει μεθόδους οι οποίες ακολουθούν το πρωτόκολλο της ψευδοτυχαιότητας, με χαρακτηριστικό παράδειγμα τη συνάρτηση Next(lower_bound, upper_bound), η οποία επιλέγει με τρόπο που προσομοιώνει την στατιστική τυχαιότητα έναν αριθμό στο εύρος του διαστήματος [lower_bound, upper_bound], όπως αυτό καθορίζεται από τις δύο αυτές παραμέτρους.

Προκειμένου τα δεδομένα να ανταποκρίνονται επαρκώς στην πραγματικότητα, τέθηκαν κατάλληλοι περιορισμοί κατά τη διαδικασία παραγωγής τους. Ειδικότερα, σε ότι αφορά το Πρόβλημα Δρομολόγησης Οχημάτων, κατά τη γένεση τυχαίων συντεταγμένων, επιλέχθηκαν τοποθεσίες των οποίων η απόσταση δεν υπερβαίνει τα μερικά δεκάδες χιλιόμετρα, γεγονός που συμβάλλει στη δημιουργία ρεαλιστικών σεναρίων για εκτέλεση δρομολογίων μιας υπηρεσίας διανομής ή γενικότερα μιας επιχείρησης, εντός μίας πόλης. Παράλληλα, εισήχθησαν και άλλες παρόμοιες συνθήκες οι οποίες ενισχύουν το ρεαλισμό των προβλημάτων, επιτρέποντας μία πιο διαισθητική προσέγγιση των αποτελεσμάτων που παράγονται. Ενδεικτικά, επισημαίνονται η δυνατότητα παραλαβής για κάθε πελάτη εντός ενός διαστήματος που να μην υπερβαίνει τις 12 ώρες για την υποπαραλλαγή του προβλήματος με χρονικά παράθυρα, καθώς και η δυνατότητα αποθήκευσης από 50 ως 200 προϊόντα σε κάθε διαθέσιμο όχημα στην περίπτωση όπου υπάρχουν περιορισμοί χωρητικότητας των οχημάτων.

Όπως έχει αναφερθεί νωρίτερα, η ακριβής αναπαράσταση του πραγματικού οδικού δικτύου δεν αποτέλεσε στόχο της παρούσας εργασίας Επομένως, για λόγους απλότητας και ευκολίας στην κατασκευή του απαραίτητου distance matrix, χρησιμοποιήθηκε η haversine formula, η οποία υπολογίζει, δοθέντων του γεωμετρικού πλάτους και μήκους δύο σημείων μιας σφαίρας, την ελάχιστη δυνατή απόστασή τους στην επιφάνεια της σφαίρας. Έτσι, αν σε ένα πρόβλημα που περιλαμβάνει n σημεία επαναληφθεί η διαδικασία υπολογισμού της haversine formula για όλα τα ζεύγη κόμβων που περιέχονται, κατασκευάζονται απευθείας οι n^2 ελάχιστες αποστάσεις μεταξύ όλων των πιθανών σημείων, δηλαδή ο πίνακας αποστάσεων του προβλήματος. Παρακάμπτεται λοιπόν η ανάγκη για επικοινωνία με κάποιο API για την λήψη ακριβών τοποθεσιών και δρόμων του οδικού δικτύου, και επιτυγχάνεται η δημιουργία ενός υποτυπώδους οδικού δικτύου όπου υπάρχουν n^2 οδοί, καθεμία από τις οποίες αναπαριστά και μία βέλτιστη απόσταση μεταξύ δύο σημείων. Στη συνέχεια, παρατίθεται ένα απλό παράδειγμα για την πληρέστερη κατανόηση των σεναρίων μελέτης που δημιουργήθηκαν. Πρόκειται για την περίπτωση δρομολόγησης προϊόντων από μία κοινή αφετηρία σε 19 άλλα σημεία, τα οποία έχουν γεωγραφικό πλάτος εντός του εύρους [37.92, 38.02] και γεωγραφικό μήκος εντός του εύρους [23.70, 23.80] αντίστοιχα, δηλαδή βρίσκονται εντός της Αττικής και ειδικότερα, πλησίον του κέντρου της Αθήνας. Όλες οι τοποθεσίες που πρέπει να επισκεφθούν τα οχήματα απεικονίζονται με μπλε χρώμα, ενώ η κοινή αφετηρία συνιστά το μοναδικό σημείο που διαθέτει κόκκινο χρώμα. Ακόμη, οι 380 αποστάσεις που απαρτίζουν τον distance matrix και κατ' επέκταση το οδικό δίκτυο του προβλήματος, αναπαρίστανται ως ακμές με ευθείες γραμμές μπλε χρώματος.



Εικόνα 17. Αναπαράσταση του οδικού δικτύου που χρησιμοποιήθηκε για τα προβλήματα Δρομολόγησης Οχημάτων.

Με πανομοιότυπο τρόπο, κατασκευάστηκαν τα απαραίτητα δεδομένα και στις δύο άλλες κατηγορίες προβλημάτων για τις οποίες συνθέσαμε γεννήτριες: το Πρόβλημα του Σακιδίου (Knapsack Problem) καθώς και τον Προγραμματισμό Περιορισμών για ακέραιους αριθμούς (Integer Constraint Programming).

3.3. Τα δομικά στοιχεία του λογισμικού

3.3.1. Γενικά στοιχεία αρχιτεκτονικής

Στην τρέχουσα ενότητα, θα παρουσιαστούν ορισμένες από τις θεμελιώδεις αρχιτεκτονικές αρχές που ακολουθήθηκαν προκειμένου να κατασκευαστεί ένα λογισμικό που να διαθέτει όλα τα απαραίτητα συστατικά για τη διασφάλιση τόσο υψηλής απόδοσης, όσο και αξιοπιστίας αναφορικά με την επιτέλεση των διεργασιών που του ανατίθενται.

3.3.1.1. Το SOLID Principle

Για την οργάνωση των διαφόρων components που απαρτίζουν το σύνολο του υπολογιστικού μοντέλου σε συνδυασμό με το διαμεσολαβητή μηνυμάτων της RabbitMQ, κρίθηκε απαραίτητη η συγγραφή πληθώρας services, τα οποία να υπακούν τις βασικές αρχές Ανάπτυξης Λογισμικού, όπως αυτές έχουν καθιερωθεί στις περισσότερες αρχιτεκτονικές των σύγχρονων εφαρμογών. Σχετικά με το .NET οικοσύστημα και τη γλώσσα προγραμματισμού C#, οι αρχές αυτές συνοψίζονται στα λεγόμενα «SOLID Principles», όπου κάθε γράμμα του ακρωνυμίου αναφέρεται σε μία σχεδιαστική μεθοδολογία, ως εξής:

- **S → Single Responsibility Principle (SRP):** Κάθε δομικό στοιχείο (module) ενός λογισμικού πρέπει να συνδέεται μόνο με μία αιτία αλλαγής του. Εναλλακτικά, η αρχή αυτή υπαγορεύει πως κάθε κλάση ή παρόμοια δομή στον κώδικα πρέπει να επιτελεί ένα και μόνο σκοπό σε αυτόν, ώστε να αποφευχθούν περιπτώσεις στις οποίες αν ένα αντικείμενο της κλάσης πρέπει να διαφοροποιηθεί, να απαιτούνται αλλαγές σε ολόκληρη την έκτασή της. Με τον τρόπο αυτό επιτυγχάνεται τόσο η ελαχιστοποίηση των αλλαγών στον κώδικα σε μελλοντικές ενημερώσεις, όσο και η υψηλότερη επαναχρησιμοποίησή του.

- **O → Open/Closed Principle (OCP):** Ένα module ή μία κλάση του λογισμικού πρέπει να είναι ανοιχτή προς επέκταση (extension) και κλειστή προς τροποποίηση (modification). Η σχεδίαση οφείλει να επιτρέπει την προσθήκη μίας λειτουργικότητας μόνον όταν προκύπτουν καινούριες απαιτήσεις, ενώ αντίστοιχα οι τροποποιήσεις πρέπει να αποφεύγονται σε όλες τις περιπτώσεις πλην της εύρεσης δυσλειτουργιών (bugs).

- **L → Liskov Substitution Principle (LSP):** Ουσιαστικά πρόκειται για μία επέκταση του Open/Closed Principle, η οποία αναφέρει πως πρέπει να είναι εφικτή η χρήση οποιασδήποτε derived κλάσης έναντι της γονικής κλάσης και να συμπεριφέρεται με τον ίδιο τρόπο δίχως την ανάγκη τροποποίησης.

- **I → Interface Segregation Principle (ISP):** Οι clients δεν πρέπει να υλοποιούν διεπαφές τις οποίες δεν χρησιμοποιούν. Αντί για ένα μεγάλο, ενιαίο interface, προτιμάται η χρήση πολλών μικρών interfaces, το καθένα από τα ακολουθεί το Single Responsibility Principle, ακριβώς όπως συμβαίνει και με τις κλάσεις.

- **D → Dependency Injection Principle (DIP):** Τα modules και οι κλάσεις υψηλού επιπέδου δεν πρέπει να εξαρτώνται από modules και κλάσεις χαμηλού επιπέδου. Αρχικά, τόσο τα high-level όσο και τα low-level components πρέπει να βασίζονται σε abstractions. Έπειτα, τα abstractions πρέπει να μην εξαρτώνται από λεπτομέρειες (details), αλλά να συμβαίνει το αντίθετο˙ οι λεπτομέρειες να εξαρτώνται από τα abstractions. Συνοπτικά, αυτό σημαίνει ότι τα υψηλού επιπέδου μέρη του κώδικα, όπως τα services, δεν πρέπει να παρουσιάζουν άμεση εξάρτηση από τα χαμηλού επιπέδου μέρη, όπως είναι συγκεκριμένες υλοποιήσεις, αλλά και τα δύο πρέπει να εξαρτώνται από κοινά interfaces ή abstractions.

Για τη δημιουργία λογισμικού με σωστά θεμέλια, οι υπηρεσίες και γενικότερα τα συστατικά στοιχεία της εφαρμογής σχεδιάστηκαν με γνώμονα να υπακούν, όποτε είναι

εφικτό, στις παραπάνω αρχές. Συγκεκριμένα, ακολουθήθηκε το Single Responsibility Principle σε όλα τα services και τα API Controllers, ενώ παράλληλα αξιοποιήθηκε και το Open/Closed Principle, ιδίως κατά την σύνθεση των βασικών στοιχείων του λογισμικού, όπως η γενική κλάση που συνδέεται με το RabbitMQ Server και αποστέλλει σε αυτόν RPC αιτήματα. Έτσι, επιτεύχθηκε η υλοποίηση εύκολα επεκτάσιμων και μη αλληλεξαρτώμενων services, το καθένα από τα οποία επικεντρώνεται στην εκτέλεση μιας συγκεκριμένης εργασίας. Το άθροισμα των λειτουργικοτήτων τις οποίες παρέχει κάθε υπηρεσία, οδηγεί στην ομαλή ροή και επιτέλεση ολόκληρης της διαδικασίας.

3.3.1.2. Συγχρονισμός των δεδομένων

Σε πολλές επιμέρους λειτουργίες του λογισμικού της παρούσας διπλωματικής εργασίας γίνεται χρήση των δυνατοτήτων ασύγχρονης επικοινωνίας που παρέχει το .NET οικοσύστημα. Το γεγονός αυτό προσφέρει μεν αρκετές σημαντικές επιδράσεις στην εφαρμογή, αλλά παράλληλα συνιστά ένα πιο σύνθετο τρόπο οργάνωσης κώδικα συγκριτικά με την παραδοσιακή σύγχρονη προσέγγιση, απαιτώντας προσεκτικούς χειρισμούς. Ειδικότερα, ένα από τα πιο συχνά προβλήματα σχετικά με τη χρήση ασύγχρονων μεθόδων αποτελούν οι αποτυχίες στα αποκαλούμενα κρίσιμα τμήματα (critical sections). Παρακάτω, γίνεται συνοπτική αναφορά στα δύο επικρατέστερα προβλήματα τέτοιου είδους: τα επονομαζόμενα «race conditions» και τα «deadlocks».

- **Race condition:** Ο όρος αυτός εισήχθη από τον Επιστήμονα της Πληροφορικής David A. Huffman το 1954 και περιγράφει τη συνθήκη όπου η ουσιώδης συμπεριφορά ενός συστήματος εξαρτάται από την αλληλουχία ή το χρονισμό άλλων, ανεξέλεγκτων γεγονότων. Στην ανάπτυξη λογισμικού, μία συνήθης περίπτωση ενός race condition αποτελεί η ταυτόχρονη εκτέλεση πολλών σημείων κώδικα, ιδιαίτερα όταν χρησιμοποιούν και τροποποιούν τις ίδιες μεταβλητές. Έτσι, αν αυτά τελειώσουν σε διαφορετική χρονική σειρά από την προβλεπόμενη, πιθανώς θα υπάρξει κάποια μεταβλητή της οποίας η τιμή θα μεταβληθεί με μη αναμενόμενο τρόπο, οδηγώντας τελικά σε κάποια δυσλειτουργία [50].

- **Deadlock:** Αποτελεί ένα από τα πιο ήπια ελαττώματα που μπορούν να προκύψουν στον multithreaded προγραμματισμό. Πρακτικά, αναφέρεται στην κατάσταση όπου πολλαπλά tasks ή threads δεν μπορούν να εκτελεστούν, επειδή το κάθε task αναμένει την απελευθέρωση ενός μηχανισμού κλειδώματος (lock) ο οποίος είναι δεσμευμένος από κάποιο άλλο task του οποίου η εκτέλεση έχει επίσης «μπλοκαριστεί» [51].

Σε πολλές γλώσσες προγραμματισμού, μεταξύ των οποίων και η C#, εντοπίζονται δύο, σε πρώτη όψη παρόμοιοι, μα με αρκετές εις βάθος διαφορές μηχανισμοί συγχρονισμού μεταβλητών και ευρύτερα τμημάτων κώδικα: το αντικείμενο κλειδώματος (lock) και ο σημαφόρος (semaphore). Στην προκειμένη εργασία όπου οι ανάγκες, όπως θα φανεί παρακάτω, δεν είναι ιδιαίτερα περίπλοκες, επαρκεί μία απλουστευμένη, ελαφρύτερη υπολογιστικά εκδοχή του σημαφόρου την οποία παρέχει η .NET με το όνομα SemaphoreSlim.

Το lock αποτελεί ένα synchronization primitive το οποίο εμποδίζει την πρόσβαση ή τροποποίηση του state μίας μεταβλητής από πολλαπλά εκτελεστικά threads την ίδια χρονική στιγμή. Ένα lock αντικείμενο διαθέτει τις εξής δύο λειτουργίες [52]:

- ***Acquire***, η οποία επιτρέπει σε ένα thread να πάρει προσωρινά την «ιδιοκτησία» του αντικειμένου. Αν το thread προσπαθήσει να αποκτήσει ένα lock που ήδη στην κατοχή άλλου thread, τότε μπλοκάρει μέχρι το άλλο thread να απελευθερώσει το αντικείμενο.
- ***Release***, με το οποίο ένα thread παραιτείται από την «ιδιοκτησία» του lock αντικειμένου, δίνοντας τη δυνατότητα να μεταλαμπαδευτεί σε κάποιο άλλο thread.

Τα περισσότερα lock objects ανήκουν στην κατηγορία των «advisory locks», δηλαδή το κάθε thread συνεργάζεται κάνοντας acquire το lock προτού αποκτήσει πρόσβαση στα data που αυτό περικλείει.

Γενικά, η χρήση κλειδωμάτων επιφέρει επιβάρυνση (overhead) για κάθε πρόσβαση σε ένα πόρο, ακόμα και όταν οι πιθανότητες για race conditions είναι πολύ μικρές. Επιπλέον, τα locks καθιστούν εφικτή την εμφάνιση των deadlocks που αναφέρθηκαν παραπάνω, ιδίως σε περιπτώσεις όπου ένα thread μπλοκάρει ή πεθάνει τη στιγμή που κατέχει ένα lock. Για τους δύο αυτούς λόγους, σε περιπτώσεις χρήσης ασύγχρονων μεθόδων και κλήσεων, είναι προτιμότερη η εφαρμογή άλλων μηχανισμών συγχρονισμού που μιμούνται τη λογική του κλειδώματος δίχως να μπλοκάρουν κάποιο thread [53]. Στη C#, η καλύτερη εναλλακτική είναι το SemaphoreSlim με αρχική ανάθεση ενός thread και μέγιστη ανάθεση επίσης ενός thread, προκειμένου να διασφαλιστεί ότι πρόσβαση σε αυτό θα έχει το πολύ ένα νήμα ανά χρονική στιγμή. Η κύρια διαφορά μεταξύ το lock και του SemaphoreSlim στη C#, έγκειται στο ότι ο τελευταίος μηχανισμός επιτρέπει την ασύγχρονη αναμονή για είσοδο στο section που περικλείει. Το γεγονός αυτό, όπως θα αποδειχθεί και εν συνεχεία, καθιστά το σημαφόρο ιδανική επιλογή για περιπτώσεις όπου πρέπει μόνο ένα thread ανά κάθε χρονική στιγμή να μπορεί να πραγματοποιήσει ασύγχρονες κλήσεις προς άλλα services ή APIs.

3.3.2. Τα δομικά στοιχεία του συστήματος

Στη συνέχεια, θα αναλυθούν εκτενώς τα επιμέρους components και οι αρχιτεκτονικές που υλοποιήθηκαν. Προκειμένου η διερεύνησή τους να επιτελεστεί με την απαραίτητη σαφήνεια, απαιτείται μία παρουσίαση της μεγαλύτερης εικόνας του συστήματος η οποία θα συμβάλει στην κατανόηση του ρόλου και των λειτουργικοτήτων όλων των επιμέρους components.

Θεωρώντας πως τα αιτήματα προκύπτουν από έναν request generator το οποίο προσομοιώνει μία πλειάδα χρηστών, αποστέλλονται μέσω ενός service στο server controller, ένα web API που διαχειρίζεται την εισαγωγή τους στο message broker. Έπειτα, προωθούνται στην ουρά εισόδου (job input queue) της RabbitMQ, από την οποία τα καταναλώνει ένα άλλο component που ονομάζεται solver dispatcher, όπου και λειτουργούν οι consumers του message broker. Αξίζει να σημειωθεί πως ο solver controller και ο solver dispatcher συνυπάρχουν στον ίδιο server. Κατόπιν της κατανάλωσης του κάθε μηνύματος, ο solver dispatcher το προωθεί σε ένα web API (OR-Tools Controller), το οποίο ουσιαστικά «τυλίγει» τον πυρήνα του επιλύτη. Έτσι, όταν ένα μήνυμα φτάσει στον OR-Tools Controller, εκτελείται η επίλυσή του ώστε τελικά να επιστρέψει πίσω στον solver controller μέσω μίας ουράς εξόδου (job output queue) και από εκεί στο αρχικό service αποστολής δεδομένων.

Service αποστολής δεδομένων

- Δημιουργία προβλήματος / εισαγωγή δεδομένων
- Αποστολή αιτημάτων στο solver controller

Solver controller

- Web API στον ίδιο server με τη RabbitMQ (IP address #1)
- Προώθηση αιτημάτων στον message broker
- Δημιουργία και συντήρηση logger csv αρχείου για όλα τα στάδια επίλυσης

Solver dispatcher

- Service - Κατανάλωση μηνυμάτων από τις ουρές της RabbitMQ
- Αποστολή μηνυμάτων που λαμβάνει στον solver wrapper ένα-προς-ένα

Solver wrapper

- Web API σε ξεχωριστό server από τη RabbitMQ (IP address #2)
- Τύλιγμα του πυρήνα επίλυσης για διαχείριση φόρτου εργασίας
- Δυαδική αντιμετώπιση μηνυμάτων: Αποδοχή και προώθηση στο solver - απόρριψη

Solver core

- Επίλυση προβλημάτων βάσει της βιβλιοθήκης Google OR Tools
- Επιστροφή αποτελεσμάτων στον τελικό χρήστη

Εικόνα 18. Το block diagram με όλα τα components των εξεταζόμενων αρχιτεκτονικών.

3.3.3. Το service αποστολής των δεδομένων

Πρώτο βήμα για την επίλυση του εκάστοτε υπολογιστικού προβλήματος αποτελεί η αποστολή των δεδομένων από τον πελάτη προς τον αρμόδιο διακομιστή. Προκειμένου να πραγματοποιηθεί αυτό, υλοποιήθηκε ένα service το οποίο δέχεται ως ορίσματα το path του αρχείου με τα απαραίτητα δεδομένα στον τοπικό υπολογιστή του client, τον τύπο του προβλήματος το οποίο επιθυμεί να αποστείλει προς επίλυση και τις παραμέτρους που απαιτούνται για την ορθή αντιμετώπισή του από το solver. Για παράδειγμα, στο πρόβλημα Δρομολόγησης Οχημάτων ο χρήστης χρειάζεται να γράψει στο Command-Line Interface μία εντολή της μορφής dotnet run <file_path> <problem_type> <num_of_vehicles> <max_distance>, όπου <file_path> σημαίνει την τοποθεσία του αρχείου με τα απαραίτητα δεδομένα, <problem_type> τον τύπο του προβλήματος (VRP), <num_of_vehicles> το πλήθος των οχημάτων και <max_distance> τη μέγιστη απόσταση που επιτρέπεται να διανύσει το κάθε όχημα. Εν συνεχεία, το πρόγραμμα ελέγχει αν το πρώτο όρισμα συνιστά έγκυρο JSON αρχείο και, εφόσον ικανοποιείται δη συνθήκη αυτή, το τροποποιεί κατάλληλα, προσθέτοντας σε αυτό τα απαραίτητα μεταδεδομένα και την ακέραια τιμή που αντιστοιχεί στον τύπο του προβλήματος που όρισε ο χρήστης. Έπειτα, καλεί ασύγχρονα ένα REST API το οποίο χειρίζεται τη δρομολόγηση των μηνυμάτων στις ουρές, πραγματοποιώντας ένα POST request στο οποίο περιλαμβάνει το τροποποιημένο JSON αρχείο. Μετά από αυτό το στάδιο, το συγκεκριμένο αίτημα βρίσκεται σε κατάσταση αναμονής μέχρι να επιλυθεί από τα κατάλληλα πακέτα της OR-Tools βιβλιοθήκης, αλλά η ασύγχρονη κλήση προς το server του message broker καθιστά τις υπόλοιπες διεργασίες του πελάτη λειτουργικές, δίχως να απαιτείται καμία αναστολή τους στη διάρκεια της διαδικασίας.

Αξίζει να σημειωθεί πως, για τη δημιουργία ενός αναγνωριστικού που να χαρακτηρίζει κάθε αίτημα των πελατών μοναδικά, χρησιμοποιήθηκε ένας αύξων αριθμός, ο οποίος μπορεί να τροποποιηθεί μόνο εντός ενός lock αντικειμένου, προκειμένου να διασφαλιστεί ο συγχρονισμός του μεταξύ των clients και τελικά αποθηκεύεται σε απλό αρχείο κειμένου, από το οποίο θα αναγνωστεί η ενημερωμένη τιμή του στο επόμενο αίτημα. Αν ο σκοπός της παρούσας εργασίας δεν ήταν η διερεύνηση των αρχιτεκτονικών αλλά η παρουσίαση μίας πλήρους επιχειρηματικής λύσης, πιθανότατα η προσέγγιση με τον αύξοντα αριθμό θα ήταν ανεπαρκής και θα απαιτούνταν μία πιο ανθεκτική και ασφαλής λύση, όπως το globally unique identifier (GUID), δηλαδή ένα text string από 128 bits (16 bytes) το οποίο αναπαριστά μία αναγνώριση.

3.3.4. Ο solver controller του συστήματος

Όταν το αίτημα του χρήστη φύγει από το τοπικό μηχάνημα του χρήστη, έχει ως προορισμό τη διεύθυνση στην οποία στεγάζεται ο server του διαμεσολαβητή μηνυμάτων. Συγκεκριμένα, δημιουργήθηκε ένα web API μέσω της υποδομής που προσφέρει το ASP.NET Core, το οποίο λειτουργεί σε καθορισμένο port εντός της IP address του RabbitMQ server. Στην παρούσα διπλωματική εργασία, το component αυτό αναφέρεται ως server controller, καθώς η κύρια λειτουργικότητα του controller στο συγκεκριμένο API είναι η προώθηση του μηνύματος που λαμβάνεται από το χρήστη, στο exchange της RabbitMQ και, εν συνεχεία, στην κατάλληλη ουρά. Ειδικότερα, κάθε request γίνεται POST μέσω της αντίστοιχης HTTP κλήσης στο κατάλληλο endpoint όπως αυτό ορίζεται από τον controller και γίνεται έλεγχος των δεδομένων του αιτήματος. Εφόσον διαπιστωθεί πως το αίτημα δεν είναι empty, ξεκινά η διαδικασία προώθησης του request, η οποία υλοποιείται μέσω μιας βοηθητικής κλάσης για την αποστολή και λήψη RPC αιτημάτων. Η κλάση αυτή αναλαμβάνει να εγκαθιδρύσει μία σύνδεση με το server της RabbitMQ εντός της ίδιας διεύθυνσης και να επιλέξει το κατάλληλο κλειδί ανάλογα το είδος του υπολογιστικού προβλήματος. Για καλύτερο έλεγχο του φόρτου εργασίας στο RabbitMQ server, επιλέχθηκε η δημιουργία n ουρών, δηλαδή μία για κάθε πιθανή κατηγορία προβλημάτων, ώστε κάθε request να αποστέλλεται στην ουρά η οποία αφορά σε αιτήματα της συγκεκριμένης κατηγορίας που το προσδιορίζει.

Πέραν του κύριου endpoint, το οποίο για τους σκοπούς της εργασίας ονομάστηκε «runjob», κατασκευάστηκε και ένα δεύτερο endpoint προκειμένου να επιτελέσει ένα ξεχωριστό task: τη διαχείριση και συνεχή ενημέρωση ενός logger, ο οποίος θα αποθηκεύει κάθε ολοκληρωμένο βήμα κατά τη ροή της εκτέλεσης, για κάθε αίτημα που αποστέλλουν οι clients. Συγκεκριμένα, επιλέχθηκε η δημιουργία και συντήρηση ενός csv αρχείου, το οποίο καλείται να συγκρατεί τις θεμελιώδεις ενέργειες που πραγματοποιούνται κατά μήκος της διαδρομής που ακολουθεί ένα αίτημα, από το τερματικό του χρήστη μέχρι να επιλυθεί από τις βιβλιοθήκες του OR-Tools και να επιστρέψει πίσω στον client με τη μορφή αποτελέσματος. Επομένως, γίνεται εμφανές πως όλα τα επιμέρους components του συστήματος αποστέλλουν συνεχώς HTTP-POST requests προς το endpoint, τα δεδομένα των οποίων αποτελούν ουσιαστικά μία μεμονωμένη γραμμή που θα προστεθεί στο csv αρχείο ώστε να αναπαραστήσει την πιο πρόσφατα ολοκληρωμένη ενέργεια της διαδικασίας. Το path για αυτά τα POST αιτήματα είναι, μέχρι το endpoint, πανομοιότυπο με τα POST requests του χρήστη που περιλαμβάνουν τα δεδομένα του προβλήματος. Η μοναδική διαφορά βρίσκεται στο endpoint, όπου αντί για «runjob», ονομάζεται «logging».

Προκειμένου να κατασκευαστεί το web API, αξιοποιήθηκε το component για τα web APIs που διατίθεται στο ASP.NET Core, το οποίο βασίζεται στην ενσωμάτωση των επιθυμητών μεθόδων, όπως είναι οι CRUD λειτουργίες, μέσα στους Controllers του API. Ειδικότερα, για το web API του .NET, ως controllers ορίζονται κλάσεις που επεκτείνουν την abstract κλάση ControllerBase, που αποτελεί τη βασική κλάση για την υλοποίηση ενός MVC controller, ο οποίος παρέχει την θεμελιώδη λειτουργικότητα για το χειρισμό HTTP αιτημάτων και αποκρίσεων, δίχως να περιλαμβάνει χαρακτηριστικά που αφορούν στην όψη της εφαρμογής. Καθώς το model controller συνιστά web API, προτιμήθηκε να προστεθεί στον κώδικα το attribute [ApiController], το οποίο όταν εφαρμόζεται σε μία κλάση ενός Controller, ενεργοποιούνται κάποιες χαρακτηριστικές συμπεριφορές, αντιπροσωπευτικές της πλειονότητας των APIs. Για παράδειγμα, απαιτείται η προσθήκη ενός attribute για το endpoint στο οποίο θα συναντάται η διεπαφή και τίθεται σε ισχύ η αυτοματοποίηση των αποκρίσεων σε περίπτωση που ληφθεί λανθασμένο status με κωδικό 400 και μήνυμα σφάλματος «BadRequest» [54].

3.3.5. O solver dispatcher του συστήματος

Εφόσον η σύνδεση με τον message broker ολοκληρωθεί επιτυχώς και το μήνυμα μπορέσει να δρομολογηθεί στην ουρά που του αντιστοιχεί, λαμβάνει τη «σκυτάλη» ένα επόμενο θεμελιώδες component του λογισμικού, ο solver dispatcher. Εν γένει, πρόκειται για ένα service στο οποίο καταναλώνονται σειριακά τα μηνύματα που εισάγονται στις ουρές, όπως ορίζει το πρωτόκολλο FIFO που τηρείται στη RabbitMQ. Για τη σωστή λειτουργία αυτής της μονάδας, απαιτείται η εγκαθίδρυση άλλου ενός connection με το server του διαμεσολαβητή μηνυμάτων και κατ' επέκταση η χρήση ενός καινούριο καναλιού επικοινωνίας στο οποίο επιτυγχάνεται η AMQP virtual connection. Κατά την έναρξη του service, δημιουργούνται και αρχικοποιούνται οι κατάλληλες αντιστοιχίσεις των ουρών με τους τύπους προβλημάτων. Με τον τρόπο αυτό δε χρειάζεται η εξαρχής δημιουργία όλων των πιθανών ουρών, διότι κατά την αποσειριοποίηση κάθε JSON μηνύματος μπορεί να ληφθεί ο τύπος του ώστε να δημιουργηθεί μόνο η ουρά που αντιστοιχεί σε αυτόν, αν αυτό δεν έχει συμβεί ήδη για προγενέστερο μήνυμα ίδιου τύπου. Έτσι, αν υπάρχουν n τύποι προβλημάτων και ένας client στέλνει μαζικά requests για μόνο 1 από αυτούς, επιτυγχάνεται εξοικονόμηση πόρων, αφού n – 1 queues δεν ανατίθενται σε μεταβλητές και δεν καταναλώνουν χώρο στη μνήμη.

Όπως ορίζει το framework της RabbitMQ για το .NET περιβάλλον, κάθε consumer που δημιουργείται αντιστοιχεί σε ένα συγκεκριμένο AMQP channel και διαθέτει ασύγχρονη λειτουργικότητα. Ο solver dispatcher διαθέτει μία θεμελιώδη αρμοδιότητα: τη διασφάλιση πως όλα τα μηνύματα τα οποία θα αποσταλούν από τους πελάτες της υπηρεσίας, θα αντιμετωπιστούν και το αποτέλεσμα της επεξεργασίας τους θα επιστραφεί πίσω στους clients. Προκειμένου να πραγματοποιηθεί αυτό, απαιτείται τόσο η διαχείριση των εισερχόμενων requests σε πραγματικό χρόνο, όσο και η αντιμετώπιση αιτημάτων τα οποία είχαν τοποθετηθεί νωρίτερα στην ουρά αλλά για κάποιο λόγο, όπως π.χ. κάποιο δικτυακό πρόβλημα, δεν καταναλώθηκαν από τον consumer και παρέμειναν στην ουρά. Μάλιστα, τα requests τα οποία προϋπάρχουν αυτών που στέλνονται σε πραγματικό χρόνο θα βρίσκονται πάντα στις πρώτες θέσεις της εκάστοτε ουράς λόγω του πρωτοκόλλου FIFO το οποίο ακολουθείται. Έτσι, σε κάθε «τρέξιμο» του solver dispatcher, αξιοποιείται πρώτα η μέθοδος «BasicGet()» του RabbitMQ.Client, η οποία ανευρίσκει ένα μεμονωμένο μήνυμα

από τον server, αν αυτός περιλαμβάνει τουλάχιστον ένα. Επαναλαμβάνοντας την κλήση της συνάρτησης τόσες φορές όσες και τα μηνύματα που προϋπάρχουν στην ουρά, επιτυγχάνεται η κατανάλωση όλων των μηνυμάτων που είχαν παραμείνει στην ουρά σε προηγούμενα sessions. Μόλις ολοκληρωθεί αυτή η διαδικασία, ο solver dispatcher επικεντρώνεται στην λήψη και κατ' επέκταση κατανάλωση των μηνυμάτων τα οποία αποστέλλονται σε πραγματικό χρόνο. Συγκεκριμένα, χρησιμοποιείται ο EventHandler «Received», δηλαδή μία μέθοδος η οποία εκτελείται κάθε φορά που ένα καινούριο μήνυμα παραδίδεται στον consumer. Έτσι, η διαδικασία αποστολής του εκάστοτε μηνύματος στον solver του υπολογιστικού μοντέλου πυροδοτείται ασύγχρονα, γεγονός που καθιστά απαραίτητη μία δομή ασύγχρονου συγχρονισμού των requests. Για το σκοπό αυτό, υλοποιήθηκε μία SemaphoreSlim δομή όπως αναλύθηκε παραπάνω, ώστε κάθε thread να περιμένει για όσο χρόνο χρειαστεί μέχρι να ελευθερωθεί το section που περικλείει ο σημαφόρος. Όταν εισέλθει στο σημαφόρο, αποστέλλει σε ένα API τα δεδομένα του προβλήματος μέσα από ένα HTTP request. Όταν το αίτημα φτάσει στον επιλύτη, καλεστούν οι απαραίτητες μέθοδοι ανάλογα με τον τύπο του προβλήματος και τελικά το αποτέλεσμά του επιστραφεί στο solver dispatcher, ο σημαφόρος απελευθερώνεται προκειμένου τα υπόλοιπα threads που αναμένουν να αποστείλουν request προς το API να μπορούν να το χρησιμοποιήσουν. Με τον τρόπο αυτό, εξασφαλίζεται πως τα προβλήματα θα φτάσουν τελικά στο solver μηχάνημα ένα προς ένα. Το γεγονός αυτό συμβάλλει αφενός στην αύξηση της αξιοπιστίας της υπηρεσίας λόγω της βέβαιης λήψης αποτελέσματος και αφετέρου στην εξάλειψη του κινδύνου υπερφόρτωσης του API και κατ' επέκταση του πυρήνα επίλυσης με δεδομένα προβλημάτων.

3.3.6. Ο πυρήνας και ο solver wrapper του συστήματος

Το αίτημα που δημιουργείται στο προηγούμενο στάδιο αποστέλλεται μέσω του HTTP Request στο δεύτερο web API της υπηρεσίας, το οποίο ονομάζεται OR-Tools Controller και κύρια αρμοδιότητά του είναι η επίλυση του εκάστοτε προβλήματος που δέχεται αξιοποιώντας τις βιβλιοθήκες του OR-Tools. Παρά το γεγονός πως διαθέτει μόνο μία POST μέθοδο, όπως ακριβώς συμβαίνει και στην περίπτωση του Solver Controller, το συγκεκριμένο API έχει ως host μία ξεχωριστή IP address, επιλογή που πράχθηκε για δύο λόγους:

- την επιθυμία ρεαλιστικής απεικόνισης της ροής των πληροφοριών στην υπηρεσία, διότι ένα πραγματικό τέτοιο σύστημα στην αγορά εργασίας αναμφισβήτητα θα τοποθετούσε το κάθε API διαφορετικούς server για τη βιωσιμότητα της εφαρμογής

- την ευθυγράμμιση με το Single Responsibility Principle, δηλαδή με την κατεύθυνση πως κάθε δομικό στοιχείο του συστήματος πρέπει να διαθέτει έναν και μόνο βασικό σκοπό να εξυπηρετήσει. Αν χρησιμοποιούνταν ένα ενιαίο APIs με δύο controllers όπου ο καθένας επικεντρώνεται σε μία εντελώς διαφορετική λειτουργικότητα του λογισμικού, η αρχή αυτή δε θα τηρούνταν.

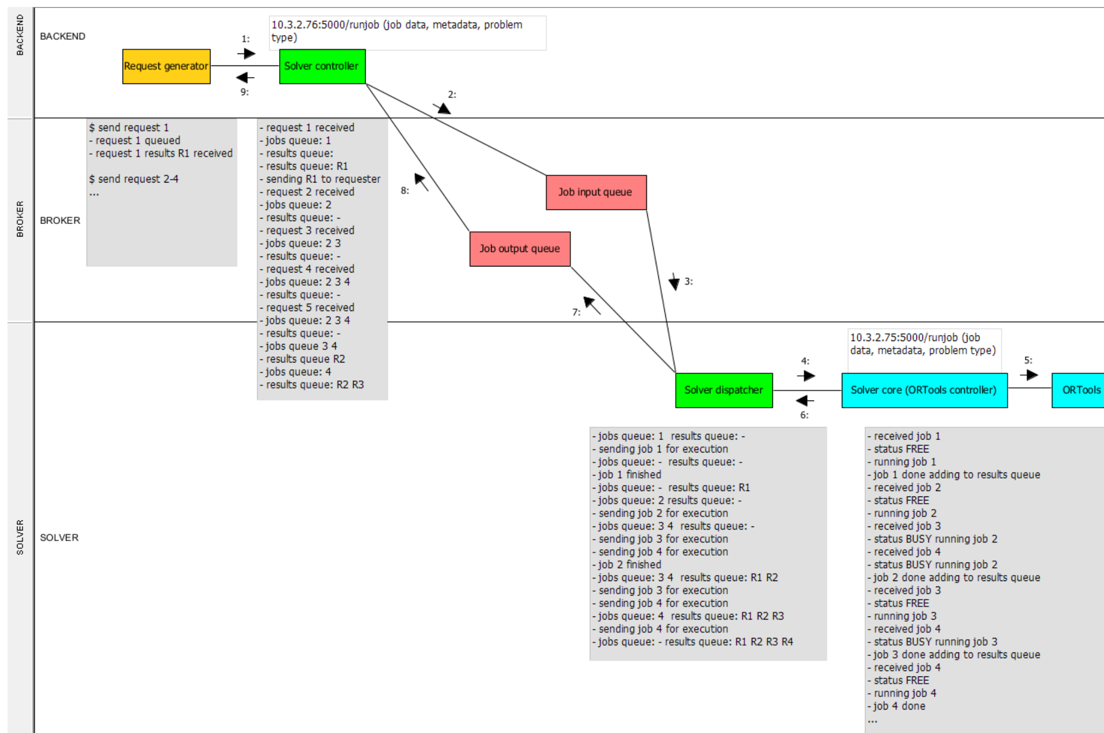
Μόλις το μήνυμα φτάσει στον OR-Tools controller, ελέγχεται ξανά η εγκυρότητα του ως προς το JSON format και, εφόσον προκύψει ως έγκυρο, αναλαμβάνει να καλέσει το core κομμάτι του solver στο λογισμικό, δηλαδή μία κατάλληλη συνάρτηση προβλημάτων Επιχειρησιακής Έρευνας η οποία κατασκευάστηκε αξιοποιώντας τις βιβλιοθήκες του OR-Tools πακέτου που περιλαμβάνει ο .NET NuGet Package Manager, σύμφωνα πάντα με τον

τύπο του εκάστοτε προβλήματος. Συγκεκριμένα, το thread που εκτελείται στον controller ελέγχει αν ο solver core χρησιμοποιείται τη δεδομένη χρονική στιγμή από κάποιο άλλο thread και αποφασίζει αναλόγως: Εφόσον ο επιλύτης δεν είναι κατειλημμένος από άλλη διεργασία, τον δεσμεύει και στέλνει τα δικά του δεδομένα, ενώ στο αντίθετο ενδεχόμενο δεν συνεχίζει στην επίλυση του προβλήματος, αλλά επιστρέφει ένα μήνυμα λάθους με κωδικό 503, ο οποίος υποδεικνύει πως η υπηρεσία είναι προσωρινά μη διαθέσιμη. Ουσιαστικά, ο solver προσομοιώνει τη λειτουργία ενός σύγχρονου lock αντικειμένου, αλλά την επιτυγχάνει με χρήση του SemaphoreSlim. Ο λόγος για την προσέγγιση αυτή είναι πως η συγκεκριμένη διαδικασία θα επαναλαμβάνεται για κάθε request που καταφτάνει στο API και ο σημαφόρος έχει σημαντικά μικρότερο υπολογιστικό κόστος από το κλείδωμα. Συνεπώς, υπάρχει μεγάλη διαφορά στο άθροισμα των πόρων που καταναλώνει η κάθε μέθοδος για το σύνολο των αιτημάτων που καλούνται να αντιμετωπιστούν.

Όπως γίνεται εμφανές, ο solver wrapper δεν εξοπλίζει τα εισερχόμενα requests με κάποιο χρονικό περιθώριο αναμονής της απελευθέρωσης του. Απεναντίας, επιβάλλει δυαδική αντιμετώπιση «αποδοχής» ή «απόρριψης», ανάλογα με την κατάσταση στην οποία βρίσκεται ο πυρήνας της επίλυσης τη στιγμή που το αίτημα φτάνει σε αυτόν. Με τον τρόπο αυτό, δίνεται η δυνατότητα παράλειψης όλου του ενδιάμεσου μηχανισμού όπου τα μηνύματα εντάσσονται σε ουρές, επιτρέποντας την υλοποίηση της υποδομής του solver ως μία stand-alone υπηρεσία, διατηρώντας από το υπάρχον μοντέλο μόνο το service αποστολής των δεδομένων. Έτσι, καθίσταται εφικτή η σύγκριση της αποτελεσματικότητας μεταξύ της αρχιτεκτονικής με broker και χωρίς broker, αντίστοιχα, διαδικασία απαραίτητη για την εξαγωγή των συμπερασμάτων που θα παρουσιαστούν σε επόμενη ενότητα της εργασίας.

Στο σημείο αυτό, αξίζει να αναφερθεί πως στην περίπτωση της αρχιτεκτονικής με χρήση των ουρών και της RabbitMQ, η χρήση του σημαφόρου θα μπορούσε τεχνικά να παραλειφθεί. Η διαδικασία αποστολής του κάθε μηνύματος από το solver dispatcher προς τον OR-Tools controller περικλείεται μέσα σε μία αντίστοιχη δομή σημαφόρου, διασφαλίζοντας πως για όσο χρόνο διαρκεί η επίλυση του request κανένα άλλο thread δεν μπορεί να αποστείλει αίτημα στον solver wrapper και κατ' επέκταση να χρησιμοποιήσει τις μεθόδους του solver core. Ωστόσο, στην περίπτωση της stand-alone υπηρεσίας που αναλύθηκε παραπάνω, ο solver dispatcher και ο solver controller δεν αξιοποιούνται, οπότε η χρήση ενός μηχανισμού συγχρονισμού των αιτημάτων στο τελευταίο επίπεδο επεξεργασίας τους κρίνεται απαραίτητη. Πέραν αυτού, η χρήση σημαφόρου τόσο στο solver dispatcher όσο και στον solver controller συνεισφέρει σε μία πολιτική «διπλού ελέγχου» (double-checking policy), επιβεβαιώνοντας πως στο πρώτο στάδιο ο συγχρονισμός υλοποιήθηκε σωστά και επιδιορθώνοντας οποιαδήποτε περίπτωση αστοχίας του.

Παρακάτω, παρατίθεται ένα communication diagram το οποίο απεικονίζει τη ροή ενός μηνύματος κατά μήκος όλου του συστήματος, όπως δηλαδή ξεκινάει από την πλευρά του client η οποία δημιουργεί το request μέχρι να διασχίσει όλα τα στάδια ώστε να καταλήξει πάλι στον client με το αποτέλεσμα του προβλήματος που ανέθεσε στην υπηρεσία. Το διάγραμμα σχεδιάστηκε με το Visual Paradigm 17.1, όπως συνέβη και με τα διαγράμματα που παρουσιάστηκαν στις προηγούμενες ενότητες του τρέχοντος κεφαλαίου.



Εικόνα 19. Το communication diagram που περικλείει όλα τα στάδια της διαδικασίας επίλυσης ενός προβλήματος OR-Tools.

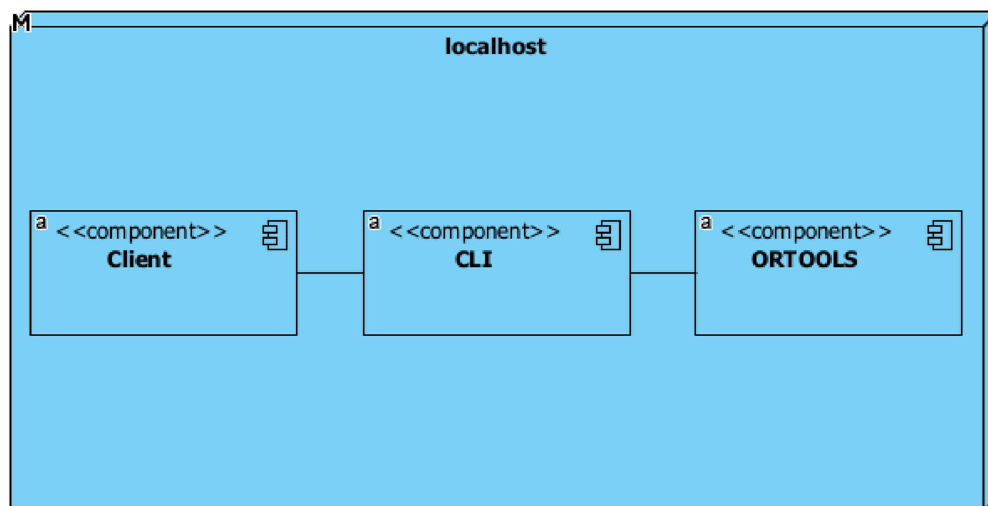
3.4. Αρχιτεκτονικές προς διερεύνηση

Στη συνέχεια, θα εξεταστούν διεξοδικά οι τρεις θεμελιώδεις αρχιτεκτονικές ως προς τη γενική περίπτωση επίλυσης ενός υπολογιστικού προβλήματος με χρήση των Google OR-Tools. Σε πρώτη φάση, θα ελεγχθεί η απλούστερη δυνατή περίπτωση, η οποία περιλαμβάνει την απευθείας επίλυση του προβλήματος δίχως τη διαμεσολάβηση των APIs και του message broker. Έπειτα, θα διερευνηθεί η αρχιτεκτονική με το επόμενο στάδιο πολυπλοκότητας, η οποία προϋποθέτει την αποστολή αιτήματος σε API για την επίλυση του προβλήματος Επιχειρησιακής Έρευνας, ξανά χωρίς την αξιοποίηση κάποιου message broker. Τέλος, θα παρουσιαστεί η πληρέστερη αρχιτεκτονική εκ των τριών, στην οποία υπάρχουν δύο ξεχωριστά APIs και η δρομολόγηση των μηνυμάτων επιτελείται από το RabbitMQ, χρησιμοποιώντας έτσι όλα τα components που αναφέρθηκαν στο προηγούμενο κεφάλαιο. Για καθεμία από τις προαναφερθείσες αρχιτεκτονικές, παρατίθενται δύο σχετικά UML διαγράμματα για την πληρέστερη κατανόηση τους: ένα component diagram, το οποίο απεικονίζει τον τρόπο με τον οποίο τα δομικά στοιχεία συνδέονται μεταξύ τους ώστε να σχηματίσουν μεγαλύτερα components ή software συστήματα, και ένα deployment diagram, το οποίο μοντελοποιεί τη φυσική διάταξη των components και των οντοτήτων στους κόμβους του ευρύτερου συστήματος. Τα διαγράμματα υλοποιήθηκαν με το λογισμικό Visual Paradigm 17.1, το οποίο αποτελεί ενιαία σουίτα για τη σχεδίαση λογισμικού, προσφέροντας μεγάλη γκάμα διαγραμμάτων όπως activity, component, deployment και sequence.

3.4.1. Απλουστευμένη υλοποίηση χωρίς τη χρήση APIs

Σε αυτήν την αρχιτεκτονική, ο client καλεί απευθείας από το Command-Line Interface του (CLI) τις απαιτούμενες μεθόδους, οι οποίες διαχειρίζονται κατάλληλα το υπολογιστικό πρόβλημα που τους δίνεται ως παράμετρος. Όταν ολοκληρώσουν την επίλυσή του, επιστρέφουν πίσω στον χρήστη το αποτέλεσμα μαζί με όποιες άλλες πληροφορίες κρίνονται απαραίτητες, όπως π.χ. η διάρκεια επίλυσης του προβλήματος και η ημερομηνία και ώρα κατά την οποία επετεύχθη η επίλυσή του. Παρακάτω, παρατίθεται το component και το deployment diagram για τη συγκεκριμένη υλοποίηση.

Εικόνα 20. Το component diagram της απλούστερης δυνατής αρχιτεκτονικής



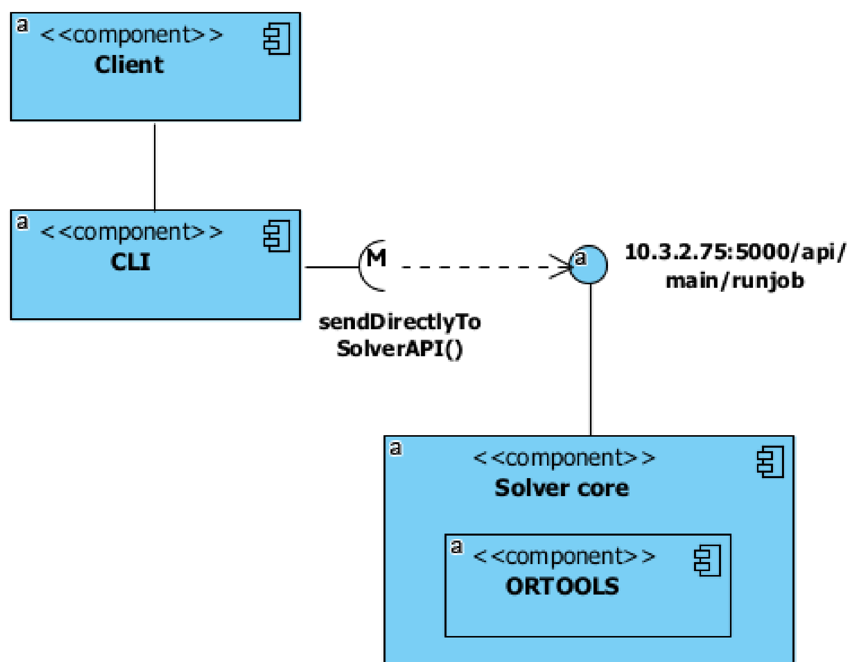
Εικόνα 21. Το deployment diagram της πρώτης αρχιτεκτονικής

Όπως είναι προφανές, η αρχιτεκτονική αυτή δε μπορεί να αποτελέσει μία SaaS υπηρεσία στον πραγματικό κόσμο. Παρόλο που φέρει εις πέρας το πρακτικό κομμάτι της επίλυσης του προβλήματος που της ανατίθεται, αδυνατεί να διαχειριστεί πολλαπλά αιτήματα αποδοτικά και γρήγορα. Ειδικότερα, κάθε request του χρήστη έχει ως αποτέλεσμα την εκτέλεση ενός νέου instance του service που αποστέλλει τα δεδομένα προς επίλυση. Επομένως, αν ένας χρήστης επιδιώξει να αποστείλει μαζικά μεγάλο αριθμό προβλημάτων, το καθένα από τα οποία εμπεριέχει αξιοσημείωτο όγκο δεδομένων, η εκτέλεση δεδομένα θα τερματιστεί ανεπιτυχώς, είτε λόγω υπέρβασης του ορίου μνήμης ή λόγω της έλλειψης threads για την αντιμετώπιση όλων των ανοιχτών αιτημάτων *(thread pool starvation)*. Η υλοποίηση μίας πλατφόρμας επίλυσης σύνθετων, χρονοβόρων υπολογιστικών προβλημάτων, τα οποία μάλιστα σε ρεαλιστικές επιχειρησιακές συνθήκες πιθανότατα θα στέλνονται και με υψηλή συχνότητα, καθίσταται ανέφικτη με μία τόσο απλοϊκή προσέγγιση, καθώς δεν

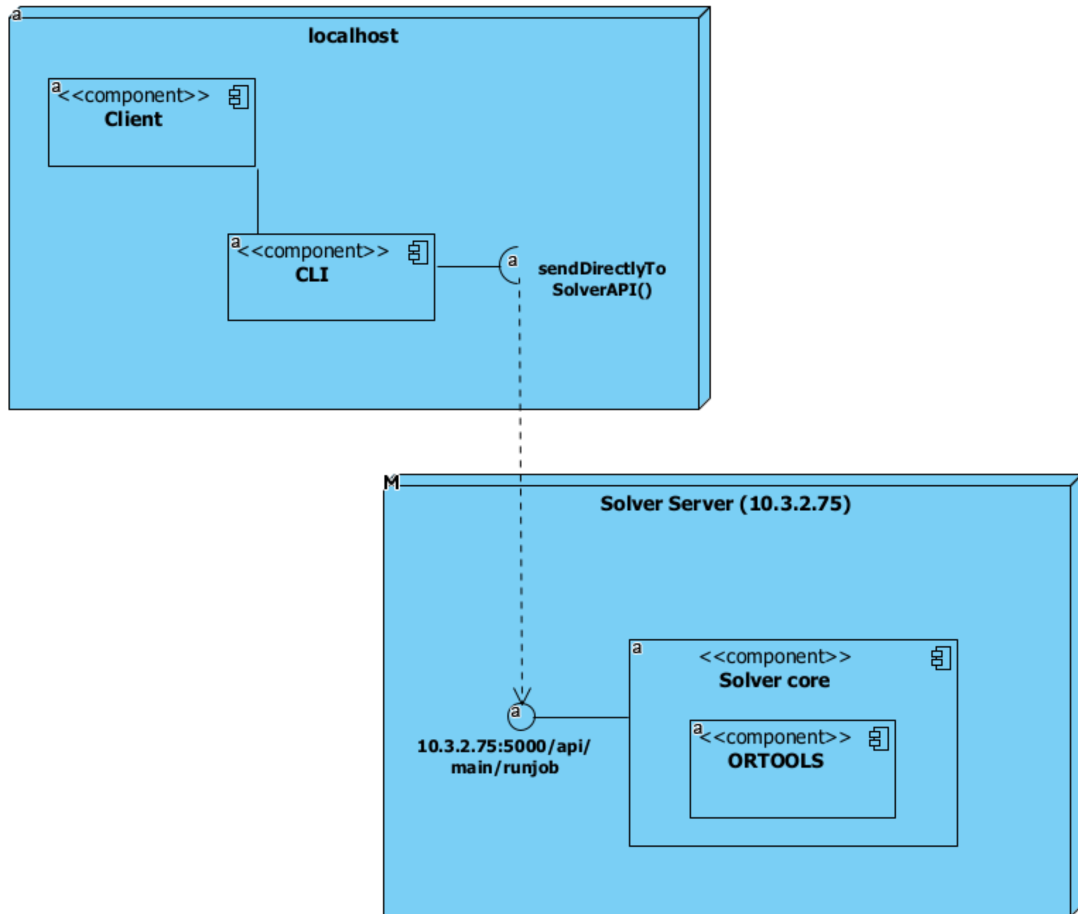
διασφαλίζεται σε καμία περίπτωση η επιτυχής εκτέλεση όλων των tasks και η επιστροφή των αποτελεσμάτων τους στο χρήστη.

3.4.2. Υλοποίηση με χρήση wrapper γύρω από το solver

Ένα πρώτο βήμα για τη βελτίωση τόσο της απόδοσης όσο και της αξιοπιστίας της υπηρεσίας αποτελεί η κατασκευή ενός API το οποίο να ενθυλακώνει το solver των υπολογιστικών προβλημάτων. Ουσιαστικά, πρόκειται για ένα wrapper, ο οποίος «τυλίγει» όλο το κομμάτι που ασχολείται με την επίλυση των αιτημάτων. Έτσι, δεν απαιτείται η απευθείας επικοινωνία του τελικού χρήστη με τις βιβλιοθήκες του OR-Tools, καθώς το API αναλαμβάνει τη λήψη και προώθηση τόσο των requests από τον client προς το solver, όσο και των αποτελεσμάτων που προκύπτουν από το solver προς τον client, αντίστοιχα. Συγκεκριμένα, η χρήση ενός τέτοιου είδους wrapper επιτρέπει την αξιοποίηση τεχνικών συγχρονισμού των δεδομένων, όπως ένα lock αντικείμενο ή ένας σημαφόρος. Στην παρούσα εργασία χρησιμοποιήθηκε η δομή SemaphoreSlim της C#, η οποία ,όπως προαναφέρθηκε προηγουμένως, επιτρέπει τη συγγραφή ασύγχρονων συναρτήσεων στο εσωτερικό της και κατ' επέκταση τον παράλληλο προγραμματισμό. Όταν σταλεί ένα αίτημα από τον πελάτη και φτάσει στο API, τότε ελέγχεται αν εκείνη τη χρονική στιγμή χρησιμοποιεί κάποια άλλη διεργασία το κριτικό τμήμα του wrapper, δηλαδή το σημείο όπου τα δεδομένα γίνονται deserialized από το αρχικό JSON και εν συνεχεία αποστέλλονται προς επίλυση από τις κατάλληλες βιβλιοθήκες του OR-Tools. Αν δεν υπάρχει άλλη διεργασία, το πρόβλημα εισέρχεται στο critical section και αποστέλλει τα δεδομένα προς επίλυση. Αντίθετα, αν άλλη διεργασία επιλύεται ήδη όταν το request φτάσει στο API, τότε επιστρέφεται στο χρήστη μήνυμα λάθους με κωδικό σφάλματος 503, το οποίο χρησιμοποιείται στο HTTP για να σημάνει πως μία υπηρεσία δεν είναι διαθέσιμη (Service Unavailable), οπότε και αδυνατεί να διαχειριστεί το αίτημα που επιχειρείται να της αποσταλεί.



Εικόνα 22. Το component diagram της δεύτερης αρχιτεκτονικής



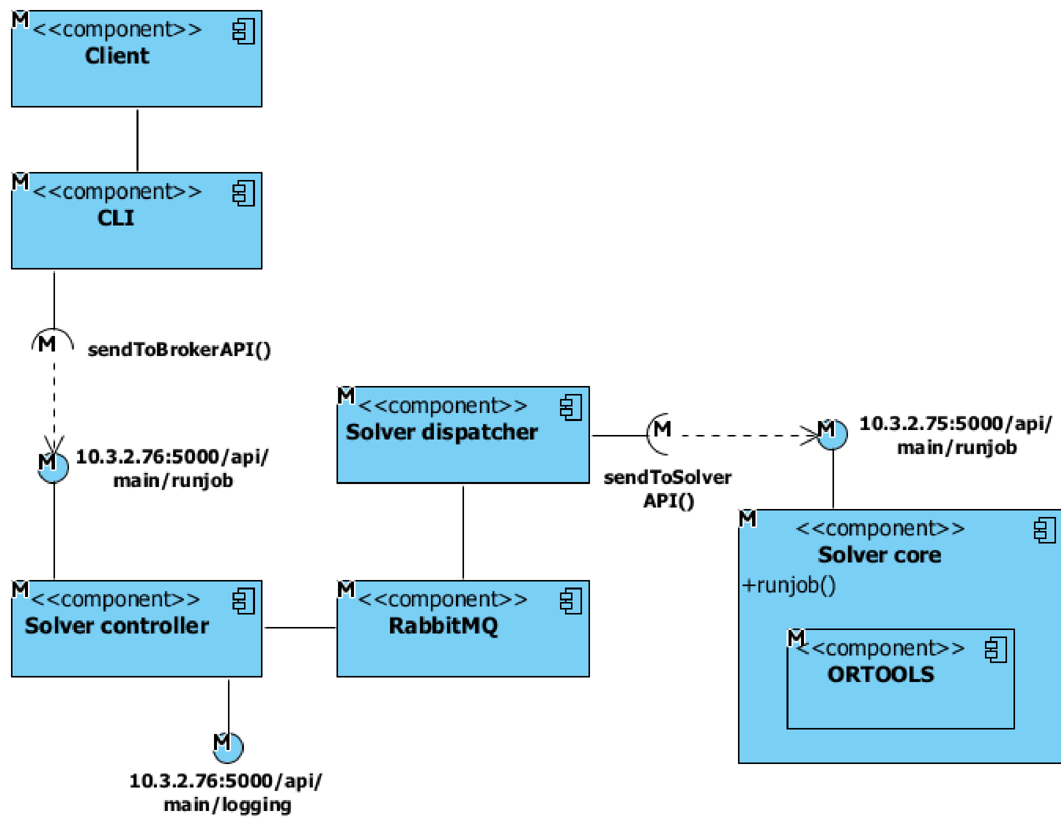
Εικόνα 23. Το deployment diagram της δεύτερης αρχιτεκτονικής

Η αρχιτεκτονική αυτή αποτελεί με βεβαιότητα καλύτερη λύση για μία Software-as-a-Service πλατφόρμα συγκριτικά με την προηγούμενη, καθώς η λειτουργία του solver wrapper ως διαμεσολαβητή και διαχειριστή των αιτημάτων, επιτρέπει εν γένει τη μαζική αποστολή πολλών προβλημάτων δίχως να προκύψουν προβλήματα. Ωστόσο, το γεγονός πως αν ένα αίτημα σταλεί κατά τη διάρκεια επίλυσης κάποιου άλλου αιτήματος, τότε το τελευταίο χρονικά δε θα επιλυθεί και θα επιστραφεί μήνυμα λάθους, δημιουργεί σημαντικά προβλήματα στην αποτελεσματικότητα της αρχιτεκτονικής. Ειδικότερα, γίνεται εμφανές πως το πλήθος των αιτημάτων που στέλνονται μαζικά (ή γενικώς με μικρή χρονική απόκλιση μεταξύ τους) με το ποσοστό επίλυσής τους είναι ποσά αντιστρόφως ανάλογα˙ όσο περισσότερο αυξάνονται τα αιτήματα που στέλνονται ταυτόχρονα, τόσο περισσότερα θα φτάσουν στο API τη στιγμή που κάποιο προηγούμενο πρόβλημα επιλύεται και επομένως τόσο μικρότερο θα είναι το ποσοστό επιτυχίας (success rate) της υπηρεσίας. Επίσης, αξίζει να αναφερθεί πως παρόλο που η αρχιτεκτονική δεν επιτρέπει την ταυτόχρονη επίλυση δύο ή περισσότερων προβλημάτων, τα ζητήματα υπερχείλισης της μνήμης και «λιμοκτονίας» του thread pool είναι εφικτό να εμφανιστούν. Όταν τα αιτήματα στέλνονται με πολύ υψηλό ρυθμό, ακόμη και η είσοδος τους στο API και η άμεση απομάκρυνσή τους λόγω μη διαθεσιμότητας του service, καταναλώνει ένα μικρό μέρος των διαθέσιμων υπολογιστικών πόρων. Έτσι, όταν η διαδικασία επαναλαμβάνεται συνεχόμενα για μαζικά HTTP requests, ενώ παράλληλα επιλύει ένα εξ αυτών, νομοτελειακά θα εμφανιστεί μετά από έναν όγκο αιτημάτων κάποιο από τα δύο προβλήματα που προαναφέρθηκαν. Συνεπώς, παρά τις

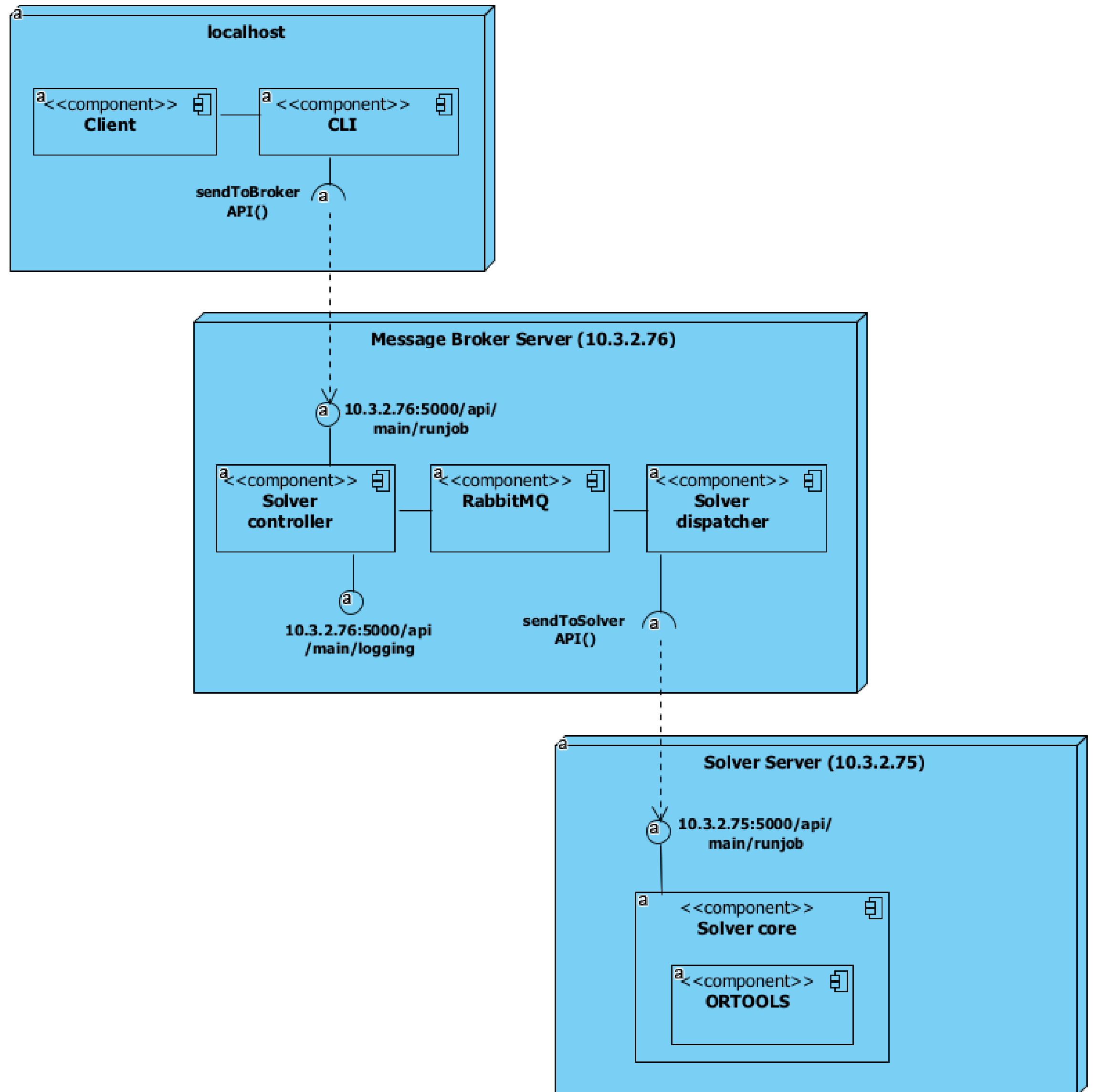
εμφανείς βελτιώσεις που παρουσιάζει συγκριτικά με την πρώτη, απλοϊκή αρχιτεκτονική, ούτε αυτή η υλοποίηση συνιστά πραγματική επιχειρηματική λύση για την κατασκευή μίας αξιόπιστης SaaS πλατφόρμας.

3.4.3. Πλήρης υλοποίηση με χρήση wrapper και message broker

Αξιοποιώντας ως βάση την αρχιτεκτονική που αναπτύχθηκε παραπάνω, καθίσταται εφικτή η κατασκευή του πληρέστερου υπολογιστικού μοντέλου που αναλύεται στην παρούσα εργασία. Συγκεκριμένα, το κύριο πρόβλημα που παρουσιάζει η αρχιτεκτονική του κεφαλαίου (3.4.2) είναι η αδυναμία διασφάλισης της επίλυσης του κάθε αιτήματος, ανεξαρτήτως από τον όγκο του φορτίου που αποστέλλεται στο solver wrapper. Το μειονέκτημα αυτό μπορεί να εξαλειφθεί αξιοποιώντας τη δομή που αποτελεί και το επίκεντρο μελέτης της εργασίας, δηλαδή ένα διαμεσολαβητή μηνυμάτων. Δημιουργώντας ένα νέο API το οποίο λειτουργεί σε διαφορετικό server και άρα σε διαφορετική IP διεύθυνση από το API που «τυλίγει» το solver, επιτυγχάνεται η φιλοξενία (hosting) του RabbitMQ server, ο οποίος αποτελεί απαραίτητη προϋπόθεση τόσο για τη χρήση ουρών, όσο και για την κατασκευή producers και consumers αρμόδιους για την παραγωγή και κατανάλωση, αντίστοιχα, μηνυμάτων από τις ουρές. Έτσι, μόλις ο τελικός χρήστης στείλει ένα αίτημα, αυτό πρώτα λαμβάνεται από το web API του message broker. Έπειτα, το API δημιουργεί έναν client υπεύθυνο για την εκτέλεση remote procedure calls εντός της RabbitMQ, ώστε να αξιοποιήσει την RPC διαδικασία του message broker που αναλύθηκε νωρίτερα. Στο σημείο αυτό, τα requests στέλνονται στην ουρά που αντιστοιχεί στον τύπο του υπολογιστικού προβλήματος, ενώ παράλληλα καθορίζεται η callback queue, η οποία θα αναλάβει τη δρομολόγηση της απάντησης, δηλαδή του αποτελέσματος του προβλήματος, πίσω στο API του message broker και τελικά στον πελάτη. Στο τελικό στάδιο αυτής της αρχιτεκτονικής βρίσκεται ο solver wrapper, ο οποίος παραμένει αμετάβλητος ως προς τη λειτουργικότητα που αναλύθηκε στην προηγούμενη αρχιτεκτονική, με μοναδική διαφοροποίηση ότι τα HTTP requests με τα προβλήματα προς επίλυση πραγματοποιούνται από τον consumer ο οποίος τα καταναλώνει από τις ουρές στις οποίες εντάσσονται, και όχι απευθείας από τον χρήστη της πλατφόρμας. Ο consumer, έχοντας τη δυνατότητα να επιλέξει αν και πότε θα αποστείλει ένα μήνυμα στο server του επιλύτη, προωθεί το εκάστοτε μήνυμα που καταναλώνει μόνον αφότου ολοκληρωθεί η επίλυση του προηγούμενου προβλήματος και επιστρέψει στον ίδιο το αποτέλεσμά του. Το γεγονός αυτό συμβάλλει καθοριστικά στην αποφόρτιση του solver από τη διαχείριση μαζικών requests, καθώς φτάνουν σε αυτόν μηνύματα μόνο μόλις ελευθερωθεί από το μέχρι τότε αίτημα προς επίλυση.



Εικόνα 24. Το component diagram της πλήρους αρχιτεκτονικής



Εικόνα 25. Το deployment diagram της πλήρους αρχιτεκτονικής

Η αρχιτεκτονική αυτή διαθέτει τις κατάλληλες προδιαγραφές για τη δόμηση ενός λογισμικού ικανού να λειτουργήσει as-a-service και να προσφέρει μία αξιόπιστη εναλλακτική λύση σε προγράμματα επίλυσης υπολογιστικών προβλημάτων τα οποία απαιτούν την a priori εξαγορά της εφαρμογής προκειμένου να παράσχουν τις υπηρεσίες στον πελάτη. Ο message broker αναλαμβάνει εξ ολοκλήρου τη διαχείρισή των αιτημάτων σε έναν δικό του server, γεγονός που επιτρέπει την εκμετάλλευση όλων των δυνατοτήτων δρομολόγησης και οργάνωσης μηνυμάτων που εμπεριέχει ως λογισμικό, με σημαντικότερο εξ αυτών την εγγύηση επιστροφής αποτελέσματος στον client και μάλιστα με σχετικά αποδοτικό τρόπο. Ως αποτέλεσμα, ένας ενδιαφερόμενος χρήστης πλέον μπορεί να αιτηθεί την αγορά του software ως υπηρεσία προκειμένου να το χρησιμοποιήσει είτε για ένα μικρό χρονικό διάστημα (π.χ. μία ώρα ή μία ημέρα), είτε για έναν συγκεκριμένο, πεπερασμένο αριθμό αιτήσεων (π.χ. 500). Αυτή η δυνατότητα του λογισμικού το ανάγει σε ιδιαίτερα ελκυστική επιλογή τόσο για πελάτες οι οποίοι γνωρίζουν περίπου πόσα προβλήματα θα χρειαστεί να επιλύσουν, όσο και για πελάτες που πρέπει να λύσουν μαζικά τα υπολογιστικά προβλήματα

που τους απασχολούν και στη συνέχεια αναμένεται να μην το επαναχρησιμοποιήσουν για ένα εύλογο χρονικό διάστημα.

3.5. Σύγκριση των αρχιτεκτονικών

Επόμενο βήμα για τη μελέτη της δρομολόγησης των μηνυμάτων, το οποίο συνιστά και το θεμελιώδη σκοπό της παρούσας διπλωματικής εργασίας, είναι η σύγκριση των αρχιτεκτονικών που αναλύθηκαν παραπάνω. Συγκεκριμένα, διενεργήθηκε η μαζική αποστολή αιτημάτων προς επίλυση σε καθεμία από τις τρεις αρχιτεκτονικές με βαθμιαίο τρόπο, δηλαδή για ολοένα και περισσότερο αυξημένο αριθμό από requests. Με αυτόν τον τρόπο, επιτυγχάνεται η εξαγωγή ασφαλών συμπερασμάτων ως προς την αποτελεσματικότητα που εμφανίζει η κάθε αρχιτεκτονική, καθώς και για την κλιμακωσιμότητά της, εφόσον ο συνολικός φόρτος εργασίας που καλείται να διαχειριστεί γίνεται ολοένα και πιο απαιτητικός. Το πρόβλημα το οποίο χρησιμοποιήθηκε σε όλες τις περιπτώσεις είναι ένα Πρόβλημα Δρομολόγησης Οχημάτων (VRP), το οποίο παράχθηκε μέσω της διαδικασίας που αναλύθηκε στο κεφάλαιο (3.2). Περιλαμβάνει 1000 τοποθεσίες από το λεκανοπέδιο της Αττικής με τη μορφή γεωγραφικών συντεταγμένων, 10 οχήματα, ενώ ως μέγιστη απόσταση την οποία επιτρέπεται να διανύσει ένα όχημα ορίζονται τα 30 χιλιόμετρα. Έτσι, αναζητείται ο βέλτιστος τρόπος ώστε τα 10 οχήματα του στόλου να επισκεφθούν και τις 1000 τοποθεσίες καλύπτοντας όσο το δυνατόν λιγότερα χιλιόμετρα.

3.5.1. Αποτελέσματα δεύτερης αρχιτεκτονικής

Σε ότι αφορά τη δεύτερη αρχιτεκτονική, η οποία περιλαμβάνει το API που επενεργεί ως wrapper γύρω από το solver core αλλά δεν υλοποιεί τη λειτουργικότητα των message brokers, παρατηρήθηκε πολύ χαμηλό efficiency στην επίλυση αιτημάτων. Ειδικότερα, η μαζική αποστολή μεγάλου αριθμού requests είχε ως αποτέλεσμα να αποστέλλεται το πρώτο, κατά σειρά, αίτημα σε διαδικασία επίλυσης, και όλα τα επόμενα requests τα οποία έφταναν στον solver wrapper μέχρι την ολοκλήρωση του τρέχοντος αιτήματος να απορρίπτονται. Το γεγονός αυτό, σε συνδυασμό με το ότι το VRP πρόβλημα που χρησιμοποιήθηκε απαιτεί ένα εύλογο χρονικό διάστημα προκειμένου να επιλυθεί, συνετέλεσαν στην απόρριψη πολύ μεγάλου αριθμού αιτημάτων, επιβεβαιώνοντας την αρχική πρόβλεψη πως μία τέτοια αρχιτεκτονική αδυνατεί να εγγυηθεί τη διαχείριση μεγάλου όγκου από requests σε πραγματικό χρόνο. Παρακάτω παρατίθενται τα στατιστικά που προέκυψαν από την εκτέλεση της διαδικασίας, όπου πρώτα απεστάλησαν 1 και έπειτα 10, 100 και 1000 πανομοιότυπα αιτήματα, αντίστοιχα:

ΑΡΧΙΤΕΚΤΟΝΙΚΗ 2 (χωρίς message broker, με solver wrapper)			
Αριθμός αιτημάτων	**Αριθμός επιλυμένων αιτημάτων**	**Αποτελεσματικότητα**	**Συνολική διάρκεια**
1	1	100%	1' 19'' 210 ms
10	1	10%	1' 32'' 22 ms
100	2	2%	2' 39'' 290 ms
1000	18	1.8%	23' 54'' 385 ms

Όπως φαίνεται από τον πίνακα, η αρχιτεκτονική αυτή παρουσιάζει ιδιαίτερα χαμηλό efficiency. Έχοντας ως βάση ένα πρόβλημα το οποίο απαιτεί χρόνο επίλυσης κοντά στο ενάμιση λεπτό, προκύπτει από τις μετρικές πως η αποτελεσματικότητα είναι χαμηλότερη του 2% κατά τη μαζική, ταυτόχρονη εκτέλεση μεγάλου αριθμού από instances. Παράλληλα, η συνολική διάρκεια της διαδικασίας είναι πολύ μικρή, προσεγγίζοντας περίπου το άθροισμα των χρόνων επίλυσης για κάθε request το οποίο εισάγεται στον πυρήνα του solver και η επίλυσή του ολοκληρώνεται, εφόσον όλα τα υπόλοιπα αιτήματα απορρίπτονται σχεδόν ακαριαία. Έτσι, στο case των 10 αιτημάτων, η εκτέλεση της διαδικασίας διήρκεσε μόλις μερικά δευτερόλεπτα παραπάνω από το χρόνο επίλυσης του 1 instance. Κατ' αναλογία, στις περιπτώσεις αποστολής 100 και 1000 requests, η ολοκλήρωση της διαδικασίας απαίτησε χρόνο περίπου ίσο με την επίλυση των 2 και 18 instances που επιτεύχθηκε, αντίστοιχα. Επιπλέον, αξίζει να σημειωθεί πως η ομολογουμένως χαμηλή αποτελεσματικότητα που προκύπτει κατά την αποστολή αλλεπάλληλων, concurrent αιτημάτων συνιστά ένα σχετικά «αισιόδοξο» σενάριο, καθώς το πρόβλημα που αποστέλλεται είναι ένα γενικά σύντομο πρόβλημα, δίχως πολύ υψηλή πολυπλοκότητα και όγκο δεδομένων. Για παράδειγμα, αν αντ' αυτού χρησιμοποιούσαμε ένα πρόβλημα που απαιτεί 10 λεπτά προκειμένου να επιλυθεί με τους ίδιους υπολογιστικούς πόρους, το efficiency θα ήταν αναμενόμενα πολύ χαμηλότερο από το 1.8%, εφόσον η διαδικασία επίλυσης θα διαρκούσε σχεδόν δεκαπλάσιο χρόνο και άρα στο διάστημα αυτό θα προλάβαιναν να φτάσουν στον wrapper του επιλύτη πολύ περισσότερα αιτήματα, τα οποία όλα θα απορρίπτονταν. Ακόμη, σε περίπτωση αποστολής πολύ μεγάλου αριθμού requests, όπως της τάξεως των δεκάδων χιλιάδων, η διαχείριση όλων τους από το σύστημα είναι ανέφικτο να εξασφαλίσει την σωστή – ακόμη και αν πρόκειται για απόρριψη – αντιμετώπισή τους, με την εμφάνιση conflicts και την υπερχείλιση της μνήμης να συνιστούν πιθανά ενδεχόμενα.

Συνεπώς, η αρχιτεκτονική αυτή αδυνατεί να αποτελέσει αξιόπιστη επιλογή για μία εμπορική υπηρεσία. Στον πραγματικό κόσμο, το σημαντικότερο στοιχείο που οφείλει να έχει ένα λογισμικό προκειμένου να καταστεί επιτυχημένο είναι η εγγύηση υψηλής απόδοσης, κλιμακωσιμότητας και ανθεκτικότητας σε πιθανά server fails, με τη συγκεκριμένη αρχιτεκτονική να αποτυγχάνει και στους τρεις τομείς. Η αποτελεσματικότητα είναι αξιοσημείωτα χαμηλή, ενώ τα προβλήματα διαχείρισης μεγάλου αριθμού requests σημαίνουν πως και το scalability δεν είναι ικανοποιητικό. Παράλληλα, η μη χρήση ουρών για τη δρομολόγηση οχημάτων καθιστά ανέφικτη την κατασκευή μιας υποδομής αποθήκευσης των μηνυμάτων τα οποία συνέπεσαν χρονικά με κάποια αποτυχία του server, με συνέπεια όλα αυτά τα requests να χαθούν πλήρως από το σύστημα.

3.5.2. Αποτελέσματα τρίτης αρχιτεκτονικής

Έχοντας ολοκληρώσει την μελέτη της δεύτερης αρχιτεκτονικής στην πράξη, απομένει η διενέργεια αντίστοιχων προβλημάτων με την πλήρη αρχιτεκτονική του προβλήματος, ώστε να εξαχθούν τα απαραίτητα συμπεράσματα και να αποδειχθεί η θεωρητική πρόβλεψη πως η επίλυση των αιτημάτων που στέλνονται στο solver μέσω του message broker είναι εγγυημένη ανεξαρτήτως του πλήθους και της συχνότητας αποστολής τους. Συγκεκριμένα, στην αρχιτεκτονική αυτή το κάθε μήνυμα παίρνει πρώτα τη θέση του στη FIFO ουρά του RabbitMQ, από όπου καταναλώνεται από το service του consumer και έπειτα, εφόσον ο solver δεν είναι απασχολημένος με κάποιο άλλο request, προωθείται για να επιλυθεί. Γίνεται λοιπόν φανερό πως σε περίπτωση μαζικής αποστολής αιτημάτων, ο πυρήνας του solver θα δεσμεύει το κάθε πρόβλημα για όσο χρόνο απαιτείται, μόλις ελευθερώνεται θα δεσμεύει το επόμενο request κ.ο.κ, με συνέπεια ο συνολικός χρόνος επίλυσης των προβλημάτων να είναι ισοδύναμος του αθροίσματος του χρόνου κάθε αιτήματος ξεχωριστά. Προκειμένου λοιπόν να διεξαχθούν οι απαραίτητες μετρήσεις εντός ενός εύλογου διαστήματος, επιλέχθηκε μικρότερος όγκος μηνυμάτων συγκριτικά με την προηγούμενη αρχιτεκτονική. Παρακάτω παρατίθενται τα στατιστικά όπως προέκυψαν από την επαναληπτική εκτέλεση της διαδικασίας για 1, 2 και 10 πανομοιότυπα requests, αντίστοιχα:

ΑΡΧΙΤΕΚΤΟΝΙΚΗ 2 (χωρίς message broker, με solver wrapper)			
Αριθμός αιτημάτων	**Αριθμός επιλυμένων αιτημάτων**	**Αποτελεσματικότητα**	**Συνολική διάρκεια**
1	1	100%	1' 24" 841 ms
2	2	100%	3' 02" 347 ms
10	10	100%	12' 47" 149 ms

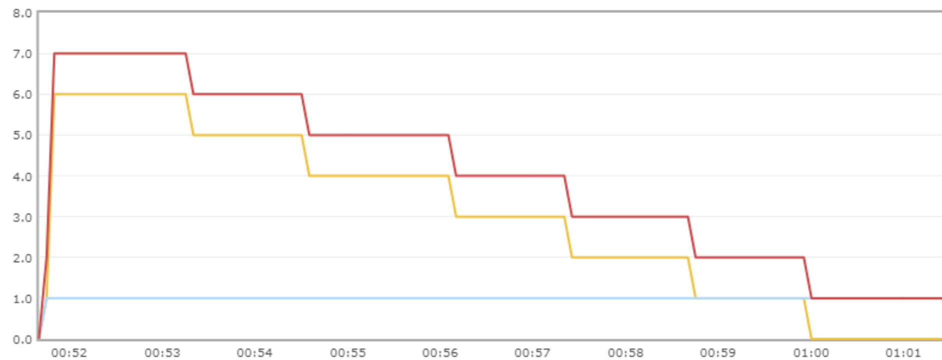
Όπως έχει αναλυθεί σε προηγούμενη ενότητα, όταν αποστέλλονται μαζικά κάποια αιτήματα προς επίλυση, εισάγονται πρώτα όλα μαζί στο exchange της RabbitMQ και κατ' επέκταση στο queue που αντιστοιχεί στο συγκεκριμένο τύπο προβλήματος, και έπειτα απορροφώνται ένα – ένα από τον solver dispatcher, ο οποίος αναλαμβάνει να τα προωθήσει στο solver wrapper. Όταν φτάσουν εκεί, θα περιμένουν ώσπου να ολοκληρωθεί η επίλυση του αμέσως προηγούμενου μηνύματος ώστε να πάρουν τη θέση τους στο solver core. Όλη αυτή η διαδικασία απεικονίζεται στα παρακάτω, τα οποία εξήχθησαν από το GUI της RabbitMQ σε πραγματικό χρόνο. Συγκεκριμένα, πρόκειται για τη γραφική αναπαράσταση επίλυσης 7 αιτημάτων τα οποία απεστάλησαν μαζικά από έναν υποθετικό client. Σε κάθε εικόνα, το πρώτο διάγραμμα αναπαριστά τα queued messages συναρτήσει του χρόνου, όπου εντοπίζονται τρεις διαφορετικοί χρωματισμοί:

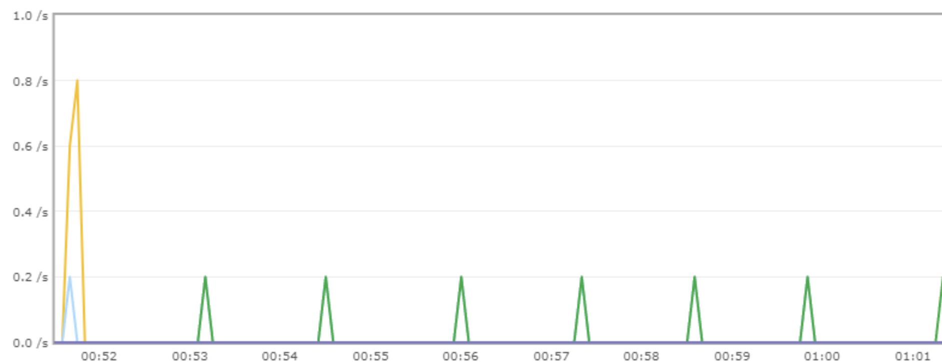
- με κόκκινο χρώμα φαίνεται το σύνολο των μηνυμάτων,
- με μπλε χρώμα απεικονίζονται τα unacknowledged messages, δηλαδή αυτά τα οποία βρίσκονται σε διαδικασία επίλυσης και ακόμα δεν έχουν καταχωριστεί ως ολοκληρωμένα (acknlowledged)

- με κίτρινο χρωματίζονται τα μηνύματα που περιμένουν πίσω στην ουρά προκειμένου να καταναλωθούν.

Όσον αφορά το δεύτερο διάγραμμα, αυτό δείχνει τα message rates, δηλαδή τους ρυθμούς ορισμένων ενεργειών ανά το χρόνο, σε κλίμακα δευτερολέπτου. Σε αυτήν την περίπτωση, οι χρωματισμοί έχουν την ακόλουθη ερμηνεία:

- με κίτρινο χρώμα υποδηλώνεται το publish των μηνυμάτων
- με γαλανό χρώμα απεικονίζεται το deliver (manual acknowledgement) ενός μηνύματος, δηλαδή η σήμανση ολοκλήρωσής του
- με πράσινο χρωματίζεται το consumer acknowledgement, δηλαδή η επιτυχής επεξεργασία και διαχείριση του μηνύματος από τον solver dispatcher.





Εικόνα 26. Τα μηνύματα της ουράς και διάφορα message rates κατά την εκτέλεση προβλημάτων Δρομολόγησης Οχημάτων.

Όπως είναι λογικό, η διαδικασία εισαγωγής και αφαίρεσης κάθε μηνύματος στις ουρές της RabbitMQ έχει ως συνέπεια τη μικρή αύξηση του απαιτούμενου χρόνου για την εκπόνηση του κάθε αιτήματος. Το γεγονός αυτό επιβεβαιώνεται από τη σύγκριση της προηγούμενης αρχιτεκτονικής με την τρέχουσα ως προς τη διάρκεια επίλυσης στην απλούστερη περίπτωση του ενός instance. Συγκεκριμένα, η brokerless αρχιτεκτονική χρειάστηκε 1' 19'' 210 ms για την επίλυση ενός προβλήματος, ενώ στην περίπτωση της

broker-based αρχιτεκτονικής απαιτήθηκαν 1’ 24” 841 ms, δηλαδή περίπου 5.5 περισσότερα δευτερόλεπτα.

Γνωρίζοντας ήδη πως κάθε αίτημα απαιτεί χρόνο μεταξύ ενός και ενάμιση λεπτού για να επιλυθεί, επιβεβαιώνεται πως, σε κάθε case που διερευνάται, η συνολική διάρκεια εκτέλεσής του ισοδυναμεί περίπου με το γινόμενο του πλήθους των requests επί του χρόνου επίλυσης του κάθε request. Για το λόγο αυτό, η διαδικασία απαιτεί πολλαπλάσιο χρόνο σε σχέση με την προηγούμενη αρχιτεκτονική για να αντιμετωπίσει τον ίδιο αριθμό αιτημάτων, ο οποίος μπορεί να υπολογισθεί προσεγγιστικά από την κάτωθι φόρμουλα:

$$\frac{\text{Χρόνος n αιτημάτων πλήρους αρχιτεκτονικής}}{\text{Χρόνος n αιτημάτων προηγούμενης αρχιτεκτονικής}} = \frac{\mathrm{n} \cdot \mathrm{t_s}}{\mathrm{x} \cdot \mathrm{t_s} + (\mathrm{n} - \mathrm{x})\, \mathrm{t_r}}$$

, όπου t_s ο χρόνος που απαιτεί κάθε request, x το πλήθος των requests που εισήχθησαν στον solver και επιλύθηκαν στην δεύτερη αρχιτεκτονική, και t_r το άθροισμα των σχεδόν αμελητέων διαστημάτων απόρριψης των $(n - x)$ αιτημάτων που βρήκαν τον solver δεσμευμένο.

4. Συμπεράσματα – Μελλοντική Εργασία

4.1. Συμπεράσματα

Στην παρούσα διπλωματική εργασία, κατασκευάστηκε από την αρχή ένα σύστημα επίλυσης προβλημάτων το οποίο υλοποιεί τον ευρέως διαδεδομένο message broker RabbitMQ για τη δρομολόγηση των αιτημάτων που δέχεται και εγγυάται την επιτυχημένη ολοκλήρωση της επίλυσης ανεξαρτήτως του φόρτου εργασίας που συσσωρεύεται σε πραγματικό χρόνο. Δημιουργήθηκαν δύο ξεχωριστά APIs, όπου το ένα αναλαμβάνει την προώθηση των μηνυμάτων στις ουρές του message broker και το άλλο είναι υπεύθυνο για την μεταφορά του επόμενου στη σειρά αιτήματος προς επίλυση, όταν ολοκληρωθεί η διαδικασία για το αμέσως προηγούμενο request. Έτσι, χρησιμοποιήθηκαν δύο ξεχωριστοί servers, ένας για το κάθε API, γεγονός που συνέβαλε στην προσομοίωση του συνόλου της διαδικασίας σε μία commercial εφαρμογή, από το πρώτο στάδιο όπου ο client αιτείται την επίλυση του προβλήματος μέχρι να φιλοξενηθεί στις ουρές του message broker, να επιλυθεί από τον πυρήνα του solver και τελικά να επιστρέψει πίσω στο χρήστη.

Το σημαντικότερο προτέρημα που διαθέτει ένα τέτοιου είδους λογισμικό είναι η δυνατότητα να ακολουθήσει τις αρχές της «Software as a Service» αρχιτεκτονικής, στοιχείο το οποίο απουσιάζει από τους περισσότερους καθιερωμένους solvers στον επιχειρηματικό κόσμο, όπως το GAMS και το Gurobi. Συγκεκριμένα, η εφαρμογή αναπτύχθηκε με τέτοιον τρόπο ώστε να μην απαιτείται η εμπορική εξαγορά της από το χρήστη, αλλά να μπορεί ο εκάστοτε ενδιαφερόμενος να αγοράσει, οποτεδήποτε χρειάζεται, έναν καθορισμένο αριθμό προβλημάτων τα οποία επιτρέπεται να στείλει προς επίλυση. Με αυτόν τον τρόπο, παρατηρείται πως το λογισμικό σύστημα που κατασκευάστηκε μπορεί να αποτελέσει ελκυστική επιλογή στις συνθήκες τις αγοράς, ιδιαίτερα σε μικρότερες εταιρείες οι οποίες δε διαθέτουν τη δυνατότητα καταβολής των μεγάλων ποσών όπου κοστολογούνται οι commercial solvers και αναζητούν μία εναλλακτική, value-for-money λύση για την εκτέλεση ενός πεπερασμένου αριθμού προβλημάτων. Μάλιστα, το γεγονός πως σε όλα τα σενάρια τα οποία υλοποιήθηκαν, δηλαδή ακόμα και για συσσωρευμένο φορτίο χιλιάδων αιτημάτων, το efficiency του υπολογιστικού μοντέλου ισούνταν πάντοτε με 100% οδηγεί στο συμπέρασμα πως οι message brokers αποτελούν μία πολύ αξιόπιστη και πιθανότατα απαραίτητη λύση για κάθε web-based πλατφόρμα η οποία αναμένεται να αντιμετωπίσει πολύ μεγάλο φορτίο εργασίας.

Ωστόσο, η προσέγγιση ενός λογισμικού που λειτουργεί ως υπηρεσία δεν μπορεί να αντιμετωπίσει επάξια σε κάθε τομέα τους εμπορικούς επιλύτες επί πληρωμή, με το βασικότερο μειονέκτημα που παρουσιάζει να είναι η μεγάλη χρονική διάρκεια που προϋποθέτει. Ειδικότερα, μέσα από την εκτενή διερεύνηση της as-a-service αρχιτεκτονικής διαπιστώθηκε πως, αν χρησιμοποιείται ένα solver instance, η εκτέλεση n requests όπου το καθένα απαιτεί t χρόνο επεξεργασίας συνεπάγεται συνολικό χρόνο απασχόλησης του solver ίσο με $n \cdot t$. Επομένως, παρότι το υπολογιστικό μοντέλο που κατασκευάστηκε πράγματι

διασφαλίζει την επίλυση όλων των αιτημάτων, δεν ενδείκνυται για περιπτώσεις χρήσης όπου ο client πρέπει να επιλύσει πολλά requests σε μικρό χρονικό περιθώριο. Βέβαια, αξίζει να σημειωθεί πως υπάρχουν τεχνικές για την καλύτερη διαχείριση μεγάλου όγκου αιτημάτων και κατ' επέκταση τη βελτίωση της ταχύτητας στο λογισμικό, αλλά δεν κρίθηκε απαραίτητη η μελέτη τους για τους σκοπούς της παρούσας εργασίας.

Επιπλέον, η δυνατότητα συγγραφής του πηγαίου κώδικα σε .NET οικοσύστημα, δηλαδή σε ένα Microsoft-oriented περιβάλλον και εκτέλεσης των runtimes σε Linux Ubuntu διακομιστή, αναδεικνύει άλλο ένα σημαντικό συμπέρασμα: την αυξημένη ευελιξία του .NET Framework. Ενώ μέχρι πριν μερικά χρόνια επέτρεπε μόνο τον προγραμματισμό σε Windows λειτουργικά, πλέον υποστηρίζεται απρόσκοπτα και με ευκολία το deployment του κώδικα σε άλλα συστήματα, γεγονός που επιβεβαιώθηκε και κατά την λειτουργία των δύο APIs στην παρούσα εργασία.

4.2. Μελλοντικές επεκτάσεις

Η περάτωση της παρούσας διπλωματικής εργασίας σηματοδοτεί την υλοποίηση μίας λειτουργικής έκδοσης ενός υπολογιστικού μοντέλου για την as-a-service επίλυση προβλημάτων, ωστόσο υπάρχουν πολλές ιδέες που θα μπορούσαν να αποτελέσουν μελλοντικές επεκτάσεις για την αναβάθμιση του συστήματος. Αρχικά, μία εφικτή και συνάμα απαραίτητη δράση είναι η εφαρμογή τεχνικών load balancing για τη μείωση του χρόνου παραμονής των αιτημάτων στις ουρές και κατ' επέκταση της επίδοσης του μοντέλου. Συγκεκριμένα, η as-a-service υλοποίηση του consumer των μηνυμάτων που εισάγονται στις ουρές της RabbitMQ επιτρέπει τον πολλαπλασιασμό των consumer instances μόνο μέσω του «τρεξίματος» της ίδιας διαδικασίας, δίχως καμία ανάγκη διαφοροποίησης του κώδικα ή της παραμετροποίησης των αρχείων. Έτσι, σε περίπτωση που παρατηρείται απότομη αύξηση των requests από τους διάφορους χρήστες, μπορεί να επιτευχθεί άκοπα μία σημαντική βελτίωση του χρόνου επίλυσης του εκάστοτε υπολογιστικού προβλήματος, καθώς περισσότερα instances καταναλωτών συνεπάγονται ταχύτερη εξυπηρέτηση. Ακόμη, προτείνεται η δημιουργία νέων endpoints στο solver controller τα οποία θα έχουν πανομοιότυπη λειτουργία με το τρέχον endpoint που προωθεί τα μηνύματα στις ουρές, και η υλοποίηση ενός round-robin load balancer ο οποίος θα μεταφέρει τα μηνύματα στο πρώτο διαθέσιμο endpoint. Με αυτόν τον τρόπο, θα επιτευχθεί η ταχύτερη εισαγωγή των requests στις ουρές του message broker. Παράλληλα, το «τύλιγμα» του επιλύτη γύρω από ένα web API επιτρέπει την εκτέλεση πολλαπλών instances του solver αντί για ένα, γεγονός που σε συνδυασμό με την ένταξη ενός δυναμικού load balancer στο solver wrapper API ο οποίος πιθανώς θα υλοποιεί τον Αλγόριθμο Ελαχίστων Συνδέσεων, καθιστά εφικτή την εξισορρόπηση του φορτίου στέλνοντας τα requests που καταναλώνονται από τον consumer στον λιγότερο απασχολημένο πυρήνα.

Απαραίτητη μελλοντική εργασία συνιστά η σύγκριση ανάμεσα στην περίπτωση τοπικής εγκατάστασης του message broker και της προσέγγισης, όπου τόσο το υπολογιστικό μοντέλο όσο και ο διαμεσολαβητής μηνυμάτων αποτελούν SaaS προγράμματα. Καθώς η υλοποίηση ενός SaaS υπολογιστικού μοντέλου προϋποθέτει υψηλή αντοχή στην κλιμακωσιμότητα (scalability), δηλαδή την αυξομείωση του μεγέθους και του όγκου των δεδομένων προς επίλυση, η χρήση του RabbitMQ ως υπηρεσία θα μπορούσε να προσδώσει σημαντική βελτίωση στο performance του μοντέλου. Αυτό μπορεί να συμβεί μέσω του

CloudAMQP, το οποίο παρέχει fully managed RabbitMQ διακομιστές σε πολλές Cloud πλατφόρμες, δίχως να απαιτείται η εγκατάσταση της τοπικά στον server ως βιβλιοθήκη. Με αυτόν τον τρόπο, αυτοματοποιείται ολόκληρο το setup, η λειτουργία αλλά και το scaling των RabbitMQ clusters, δηλαδή των λογικών ομαδοποιήσεων ενός η περισσότερων κόμβων, queues, exchanges κ.α., υποστηρίζοντας παράλληλα όλα τα πρωτόκολλα που παρέχει και η τοπική έκδοση του RabbitMQ. Στο CloudAMQP καθίσταται εφικτή η δυναμική αλλαγή μεγέθους των clusters σε κάποιο μεγαλύτερο ή μικρότερο instance *κατ' απαίτηση* (on demand), επιτρέποντας τη χρήση του κατά τη διάρκεια του re-configuration στο background [55]. Ως αποτέλεσμα, η διαδικασία εισαγωγής και κατανάλωσης μεγάλου πλήθους μηνυμάτων στις ουρές μπορεί να διευκολυνθεί μέσω της αυτοματοποίησης και συγχρόνως να επιταχυνθεί σημαντικά.

Τέλος, άλλη μία επέκταση που έχει νόημα να επισημανθεί είναι η υλοποίηση του solver ως υπηρεσία. Αν και το Google OR-Tools ακόμη δε διατίθεται στο Cloud μέσω κάποιου dedicated API για προβλήματα Επιχειρησιακής Έρευνας, παρέχεται η δυνατότητα φιλοξενίας ενός API σε κάποιον managed server, όπως το Google Kubernetes Engine στο Google Cloud Platform (GCP), πληρώνοντας βέβαια τη χρήση του infrastructure. Έτσι, τόσο ο solver wrapper όσο και ο solver core μπορούν να «πακεταριστούν» σε *container images*, δηλαδή σε lightweight, stand-alone εκτελέσιμα πακέτα λογισμικού που περιλαμβάνουν όλα τα απαραίτητα στοιχεία προκειμένου να λειτουργήσουν όπως πρέπει. Το γεγονός αυτό επιτρέπει το αυτοματοποιημένο deployment και management της εφαρμογής, την υψηλή ασφάλεια που προσφέρει η Google καθώς και τη μεγάλη διαθεσιμότητα μέσω συνεχών ελέγχων υγείας στα containers. Επιπλέον, διασφαλίζεται η εύκολη, αυτοματοποιημένη κλιμάκωση των containers ανάλογα με τη ζήτηση, δημιουργώντας πολλαπλά instances που βασίζονται στο ίδιο container image, ενώ παράλληλα τρέχουν πολλοί αλγόριθμοι load balancing για τη βελτιστοποίηση της κατανομής του φορτίου [56]. Επομένως, η υλοποίηση του solver as-a-service μπορεί να αποτελέσει ουσιώδη επέκταση, αναβαθμίζοντας ποιοτικά το λογισμικό σε όλους τους τομείς και καθιστώντας το ικανό να διαχειριστεί αποτελεσματικά και αυτοματοποιημένα πολύ μεγάλο όγκο αιτημάτων.

Βιβλιογραφία

[1] Wikipedia, «Solver,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Solver. [Πρόσβαση Φεβρουάριος 2024].

[2] GAMS, «GAMS,» [Ηλεκτρονικό]. Available: https://www.gams.com/. [Πρόσβαση Φεβρουάριος 2024].

[3] Wikipedia, «Generic Algebraic Modeling System,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/General_algebraic_modeling_system. [Πρόσβαση Φεβρουάριος 2024].

[4] Wikipedia, «Gurobi Optimizer,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Gurobi_Optimizer. [Πρόσβαση Φεβρουάριος 2024].

[5] Gurobi, «Gurobi,» [Ηλεκτρονικό]. Available: https://www.gurobi.com/. [Πρόσβαση Φεβρουάριος 2024].

[6] Wikipedia, «API,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/API. [Πρόσβαση Φεβρουάριος 2024].

[7] Cloud Studio, «Journey Of API,» Οκτώβριος 2021. [Ηλεκτρονικό]. Available: https://cloudstudio.com.au/2021/10/30/journey-of-api/. [Πρόσβαση Φεβρουάριος 2024].

[8] RESTful API Tutorial, Δεκέμβριος 2023. [Ηλεκτρονικό]. Available: https://restfulapi.net/. [Πρόσβαση Φεβρουάριος 2024].

[9] IBM, « What is a REST API?,» [Ηλεκτρονικό]. Available: https://www.ibm.com/topics/rest-apis. [Πρόσβαση Φεβρουάριος 2024].

[10] F. L., «getstream.io,» Απρίλιος 2023. [Ηλεκτρονικό]. Available: https://getstream.io/blog/api-protocols/. [Πρόσβαση Φεβρουάριος 2024].

[11] Great Learning, Ιούλιος 2022. [Ηλεκτρονικό]. Available: https://www.mygreatlearning.com/blog/stateful-vs-stateless/. [Πρόσβαση Φεβρουάριος 2024].

[12] Graffersid, «Difference between Stateful vs Stateless API,» Νοέμβριος 2023. [Ηλεκτρονικό]. Available: https://graffersid.com/difference-between-stateful-and-stateless-api/. [Πρόσβαση Φεβρουάριος 2024].

[13] B. M. Linda Rosencrance, «Tech Target,» [Ηλεκτρονικό]. Available: https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC.

[14] Wikipedia - Remote Procedure Call, Φεβρουάριος 2024. [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Remote_procedure_call. [Πρόσβαση Φεβρουάριος 2024].

[15] IT Release, «What is Remote Procedure Call in Operating System,» Ιούνιος 2021. [Ηλεκτρονικό]. Available: https://www.itrelease.com/2021/06/what-is-remote-procedure-call-rpc-in-operating-system/. [Πρόσβαση Φεβρουάριος 2024].

[16] P. Sturgeon, «Smashing Magazine,» Σεπτέμβριος 2016. [Ηλεκτρονικό]. Available: https://www.smashingmagazine.com/2016/09/understanding-rest-and-rpc-for-http-apis/. [Πρόσβαση Φεβρουάριος 2024].

[17] gRPC, «About gRPC - Who is using gRPC and why,» [Ηλεκτρονικό]. Available: https://grpc.io/about/. [Πρόσβαση Φεβρουάριος 2024].

[18] M. J. Isabella Olivos, «Comparative Study of REST and gRPC for Microservices in Established Software Architectures,» 2023. [Ηλεκτρονικό]. Available: https://www.diva-portal.org/smash/get/diva2:1772587/FULLTEXT01.pdf.

[19] Wikipedia, «Protocol Buffers,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Protocol_Buffers. [Πρόσβαση Φεβρουάριος 2024].

[20] Mulesoft, «Understanding the Essentials of gRPC,» [Ηλεκτρονικό]. Available: https://www.mulesoft.com/api-university/understanding-essentials-grpc. [Πρόσβαση Φεβρουάριος 2024].

[21] D. Kuruppu, «The New Stack,» Αύγουστος 2021. [Ηλεκτρονικό]. Available: https://thenewstack.io/grpc-a-deep-dive-into-the-communication-pattern/. [Πρόσβαση Φεβρουάριος 2024].

[22] DreamFactory API Builder - Blog, Ιανουάριος 2024. [Ηλεκτρονικό]. Available: https://blog.dreamfactory.com/grpc-vs-rest-how-does-grpc-compare-with-traditional-rest-apis/. [Πρόσβαση Φεβρουάριος 2024].

[23] Code Reliant, «gRPC vs REST,» Σεπτέμβριος 2023. [Ηλεκτρονικό]. Available: https://www.codereliant.io/grpc-vs-rest/. [Πρόσβαση Ιανουάριος 2024].

[24] J. Muech, Ιανουάριος 2022. [Ηλεκτρονικό]. Available: https://betterprogramming.pub/why-do-we-need-message-broker-7382ce0e46c6. [Πρόσβαση Φεβρουάριος 2024].

[25] TIBCO Enterprise Message Service User - Documentation, «TIBCO,» 2021. [Ηλεκτρονικό]. Available: https://docs.tibco.com/pub/ems/8.6.0/doc/html/GUID-8FA402D4-3150-4999-A920-D1EC1D764FCC.html. [Πρόσβαση Φεβρουάριος 2024].

[26] G. R. B. C. Mihaela Ion, «Supporting Publication and Subscription Confidentiality in Pub/Sub Networks,» 2010. [Ηλεκτρονικό]. Available: https://eudl.eu/pdf/10.1007/978-3-642-16161-2_16.

[27] Wikipedia, «RabbitMQ,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/RabbitMQ. [Πρόσβαση Φεβρουάριος 2024].

[28] RabbitMQ, «RabbitMQ,» [Ηλεκτρονικό]. Available: https://www.rabbitmq.com/. [Πρόσβαση Φεβρουάριος 2024].

[29] Wikipedia, «Transmission Control Protocol,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Transmission_Control_Protocol. [Πρόσβαση Φεβρουάριος 2024].

[30] L. Johansson, «CloudAMQP,» Νοέμβριος 2019. [Ηλεκτρονικό]. Available: https://www.cloudamqp.com/blog/the-relationship-between-connections-and-channels-in-rabbitmq.html. [Πρόσβαση Φεβρουάριος 2024].

[31] Packt, «Establishing a solid connection to RabbitMQ,» [Ηλεκτρονικό]. Available: https://subscription.packtpub.com/book/business-and-other/9781789131666/2/ch02lvl1sec13/establishing-a-solid-connection-to-rabbitmq. [Πρόσβαση Φεβρουάριος 2024].

[32] Baeldung, «Exchanges, Queues, and Bindings in RabbitMQ,» Ιανουάριος 2024. [Ηλεκτρονικό]. Available: https://www.baeldung.com/java-rabbitmq-exchanges-queues-bindings. [Πρόσβαση Φεβρουάριος 2024].

[33] RabbitMQ, «Documentation: Table of Contents,» [Ηλεκτρονικό]. Available: https://www.rabbitmq.com/docs. [Πρόσβαση Φεβρουάριος 2024].

[34] L. Johansson, Σεπτέμβριος 2019. [Ηλεκτρονικό]. Available: https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html. [Πρόσβαση Φεβρουάριος 2024].

[35] VMWare Docs, «RabbitMQ tutorial - Remote procedure call (RPC) SUPPRESS-RHS,» Νοέμβριος 2023. [Ηλεκτρονικό]. Available: https://docs.vmware.com/en/VMware-RabbitMQ-for-

Kubernetes/1/rmq/tutorials-tutorial-six-dotnet.html. [Πρόσβαση Φεβρουάριος 2024].

[36] Wikipedia, «.NET,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/.NET. [Πρόσβαση Φεβρουάριος 2024].

[37] Javatpoint, «.NET Common Language Runtime (CLR),» [Ηλεκτρονικό]. Available: https://www.javatpoint.com/net-common-language-runtime. [Πρόσβαση Ιανουάριος 2024].

[38] Microsoft, «NuGet Documentation,» [Ηλεκτρονικό]. Available: https://learn.microsoft.com/en-us/nuget/. [Πρόσβαση Φεβρουάριος 2024].

[39] K. Beladiya, «The One Technologies,» Απρίλιος 2022. [Ηλεκτρονικό]. Available: https://theonetechnologies.com/blog/post/big-think-in-dotnet-framework-for-web-development. [Πρόσβαση Φεβρουάριος 2024].

[40] CloudAMQP, «CloudAMQP with .NET,» [Ηλεκτρονικό]. Available: https://www.cloudamqp.com/docs/dotnet.html. [Πρόσβαση Φεβρουάριος 2024].

[41] Newtonsoft, «JSON.NET Documentation,» [Ηλεκτρονικό]. Available: https://www.newtonsoft.com/json/help/html/Performance.htm. [Πρόσβαση Ιανουάριος 2024].

[42] LogicMonitor, «9 reasons Linux is a popular choice for servers,» Φεβρουάριος 2023. [Ηλεκτρονικό]. Available: https://www.logicmonitor.com/blog/9-reasons-linux-is-a-popular-choice-for-servers. [Πρόσβαση Φεβρουάριος 2024].

[43] Wikipedia, «Microkernel,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Microkernel. [Πρόσβαση Φεβρουάριος 2024].

[44] VMware, «What is software load balancing?,» [Ηλεκτρονικό]. Available: https://www.vmware.com/topics/glossary/content/software-load-balancing.html. [Πρόσβαση Φεβρουάριος 2024].

[45] Nginx, «What Is Load Balancing?,» [Ηλεκτρονικό]. Available: https://www.nginx.com/resources/glossary/load-balancing/. [Πρόσβαση Φεβρουάριος 2024].

[46] Hepto Technologies, «Static vs Dynamic Load Balancing,» [Ηλεκτρονικό]. Available: https://www.heptotechnologies.com/blog/static-vs-dynamic-load-balancing. [Πρόσβαση Φεβρουάριος 2024].

[47] Cloudflare, «Types of load balancing algorithms,» [Ηλεκτρονικό]. Available: https://www.cloudflare.com/learning/performance/types-of-load-balancing-algorithms/. [Πρόσβαση Φεβρουάριος 2024].

[48] MBA-H2040 Quantitative Techniques for Managers , «Introduction to Operations Research,» [Ηλεκτρονικό]. Available: https://www.bbau.ac.in/dept/UIET/EMER-601%20Operation%20Research%20Queuing%20theory.pdf.

[49] Wikipedia, «Combinatorial Optimization,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Combinatorial_optimization. [Πρόσβαση Φεβρουάριος 2024].

[50] Wikipedia, «Race Condition,» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Race_condition. [Πρόσβαση Φεβρουάριος 2024].

[51] MathWorks, «What Are Deadlocks and How Do You Prevent Them During Software Development?,» Μάιος 2020. [Ηλεκτρονικό]. Available: https://www.mathworks.com/products/polyspace/static-analysis-notes/what-deadlocks-how-prevent-during-software-development.html. [Πρόσβαση Φεβρουάριος 2024].

[52] MIT 6.005 - Software Construction, «Reading 23: Locks and Synchronization,» Σεπτέμβριος 2015. [Ηλεκτρονικό]. Available: https://web.mit.edu/6.005/www/fa15/classes/23-locks/. [Πρόσβαση Φεβρουάριος 2024].

[53] Wikipedia, «Lock (Computer Science),» [Ηλεκτρονικό]. Available: https://en.wikipedia.org/wiki/Lock_(computer_science).

[54] Microsoft Learn, Μάιος 2023. [Ηλεκτρονικό]. Available: https://learn.microsoft.com/en-us/aspnet/core/web-api/?view=aspnetcore-8.0#attribute-routing-requirement. [Πρόσβαση Φεβρουάριος 2024].

[55] J. Rhodin, «CloudAMQP,» Αύγουστος 2016. [Ηλεκτρονικό]. Available: https://www.cloudamqp.com/blog/cloudamqp-vs-RabbitMQ-DIY.html. [Πρόσβαση Φεβρουάριος 2024].

[56] Google, «Kubernetes Engine Overview,» [Ηλεκτρονικό]. Available: https://cloud.google.com/kubernetes-engine/docs/concepts/kubernetes-engine-overview. [Πρόσβαση Φεβρουάριος 2024].