ΠΑΝΕΠΙΣΤΗΜΙΟ ΠΕΙΡΑΙΩΣ

ΣΧΟΛΗ ΤΕΧΝΟΛΟΓΙΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΕΠΙΚΟΙΝΩΝΙΩΝ

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πρόγραμμα Μεταπτυχιακών Σπουδών

«ΠΡΟΗΓΜΕΝΑ ΣΥΣΤΗΜΑΤΑ ΠΛΗΡΟΦΟΡΙΚΗΣ - ΑΝΑΠΤΥΞΗ ΛΟΓΙΣΜΙΚΟΥ ΚΑΙ ΤΕΧΝΗΤΗΣ ΝΟΗΜΟΣΥΝΗΣ»

Μεταπτυχιακή διατριβή

Τίτλος Μεταπτυχιακής Διατριβής	Συγκριτική Αξιολόγηση Απόδοσης Νημάτων Πλατφόρμας και Εικονικών Νημάτων της Java 21 σε Αρχιτεκτονικές Μικρουπηρεσιών Comparative Performance Evaluation of Platform Threads and Virtual Threads of Java 21 in Microservice Architectures
Ονοματεπώνυμο Φοιτητή	Ανδρέας Σαφελάς
Πατρώνυμο	Βλάσιος
Αριθμός Μητρώου	ΜΠΣΠ24044
Επιβλέπων	Ευθύμιος Αλέπης, Καθηγητής

Ημερομηνία Παράδοσης Ιούνιος 2026

Τριμελής Εξεταστική Επιτροπή

Αλέπης Ευθύμιος, Καθηγητής	Βίρβου Μαρία, Καθηγήτρια	Σακκόπουλος Ευάγγελος, Αναπληρωτής Καθηγητής

Copyright

Ευχαριστίες

Με την ολοκλήρωση της παρούσας μεταπτυχιακής διατριβής, θα ήθελα να εκφράσω τις θερμές μου ευχαριστίες σε όσους συνέβαλαν στην επιτυχή της περάτωση.

Πρωτίστως, οφείλω να ευχαριστήσω τον επιβλέποντα καθηγητή μου, Ευθύμιο Αλέπη, για την βοήθεια του στην εύρεση του θέματος, την εμπιστοσύνη που μου έδειξε και την καθοδήγησή του για την περάτωση της εργασίας. Η ακαδημαϊκή του εποπτεία ήταν πολύτιμη για την τήρηση των προδιαγραφών και την ολοκλήρωση αυτού του έργου.

Η μεγαλύτερη ευγνωμοσύνη μου, ωστόσο, ανήκει στην οικογένειά μου και στην σύντροφο μου. Η υποστήριξη, η υπομονή και η συνεχής ενθάρρυνση που μου παρείχαν, ειδικά στις πιο απαιτητικές περιόδους, αποτέλεσαν την ραχοκοκαλιά όλης μου της προσπάθειας. Χωρίς την αγάπη και την πίστη τους, το μονοπάτι αυτό θα ήταν πολύ δυσκολότερο.

Τέλος, ένα μεγάλο ευχαριστώ στους φίλους μου, που με την κατανόηση και το χιούμορ τους μου προσέφεραν μια αναγκαία απόδραση από την πίεση της συγγραφής.

Περίληψη

Η παρούσα μεταπτυχιακή διατριβή παρουσιάζει τον σχεδιασμό και την υλοποίηση ενός συστήματος συγκέντρωσης μικροϋπηρεσιών (microservice aggregator), με βασικό στόχο την εμπειρική αξιολόγηση της αρχιτεκτονικής αλλαγής που εισήγαγε το Project Loom της Java 21. Συγκεκριμένα, η έρευνα επικεντρώνεται στη σύγκριση της απόδοσης μεταξύ της παλαιάς αρχιτεκτονικής των Νημάτων Πλατφόρμας (Platform Threads - Μοντέλο 1:1) και του σύγχρονου παραδείγματος της Δομημένης Ταυτόχρονης Εκτέλεσης μέσω Εικονικών Νημάτων (Virtual Threads - Μοντέλο M:N).

Η αρχιτεκτονική του συστήματος βασίζεται σε ένα ασφαλές RESTful API που υλοποιήθηκε με Spring Boot 3, ενώ η παρατηρησιμότητα (observability) των δεδομένων τηλεμετρίας σε πραγματικό χρόνο επιτυγχάνεται μέσω ενός πίνακα ελέγχου (dashboard) ανεπτυγμένου με React. Ένα από τα βασικά τεχνικά χαρακτηριστικά της εργασίας είναι η χρήση του Java Microbenchmark Harness (JMH) για τη διεξαγωγή στατιστικά αυστηρών δοκιμών αντοχής (stress tests) στο επίπεδο της Εικονικής Μηχανής Java (JVM), απομονώνοντας τον θόρυβο του δικτύου και μετρώντας με ακρίβεια τη διακίνηση (throughput), την καθυστέρηση (latency) και την επιβάρυνση στη συλλογή απορριμμάτων (GC allocation rate).

Τα εμπειρικά αποτελέσματα αποδεικνύουν ότι τα Εικονικά Νήματα εξαλείφουν πλήρως το σημείο συμφόρησης της έλλειψης νημάτων (thread starvation), προσφέροντας 16,8x μεγαλύτερη διακίνηση υπό συνθήκες μαζικού ταυτόχρονου φορτίου I/O, χωρίς καμία αναβάθμιση υλικού. Ωστόσο, η έρευνα αποκαλύπτει επίσης ότι η ακραία αποδοτικότητα της JVM μετατοπίζει αναπόφευκτα το σημείο συμφόρησης στο Επίπεδο Μεταφοράς (Transport Layer), οδηγώντας σε εξάντληση των εφήμερων θυρών TCP του λειτουργικού συστήματος, γεγονός που καθιστά αναγκαία τη χρήση δεξαμενών συνδέσεων (connection pooling) σε πραγματικά επιχειρησιακά περιβάλλοντα.

Επιστημονική Περιοχή: *Μηχανική Λογισμικού, Ταυτόχρονος Προγραμματισμός, Κατανεμημένα Συστήματα, Ανάλυση Απόδοσης.*

Λέξεις-Κλειδιά: *Java 21, Project Loom, Virtual Threads, Spring Boot, React, JMH Benchmarking, Microservices.*

Abstract

This master's thesis presents the design and implementation of a microservice aggregator, with the primary objective of empirically evaluating the architectural shift introduced by Java 21's Project Loom. Specifically, the research focuses on comparing the performance of the legacy Platform Thread architecture (1:1 Model) against the modern paradigm of Structured Concurrency via Virtual Threads (M:N Model).

The system's architecture is based on a secure RESTful API implemented with Spring Boot 3, while real-time observability of telemetry data is achieved through a dashboard developed with React. A key technical feature of this work is the utilization of the Java Microbenchmark Harness (JMH) to conduct statistically rigorous stress tests at the Java Virtual Machine (JVM) level, isolating network noise and accurately measuring throughput, latency, and Garbage Collection allocation rates.

The empirical results demonstrate that Virtual Threads completely eliminate the JVM-level bottleneck of thread starvation, delivering a 15.8x increase in throughput under massive concurrent I/O load without requiring hardware upgrades. However, the research also reveals that the extreme efficiency of the JVM inevitably shifts the bottleneck to the Transport Layer, leading to the exhaustion of the operating system's ephemeral TCP ports. This finding highlights the absolute necessity of utilizing connection pooling in real-world enterprise environments.

Scientific Area: *Software Engineering, Concurrent Programming, Distributed Systems, Performance Analysis.*

Keywords: *Java 21, Project Loom, Virtual Threads, Spring Boot, React, JMH Benchmarking, Microservices.*

Πίνακας Περιεχομένων

Κατάλογος Πινάκων

Κατάλογος Διαγραμμάτων

1. Εισαγωγή

1.1. Περιγραφή και σκοπός

Το σύγχρονο ψηφιακό τοπίο χαρακτηρίζεται από την απαίτηση για τεράστια επεκτασιμότητα και υψηλή διαθεσιμότητα. Καθώς οι οργανισμοί μεταβαίνουν από μονολιθικές αρχιτεκτονικές σε κατανεμημένες μικρουπηρεσίες, η απαίτηση για την αποτελεσματική επεξεργασία ταυτόχρονων αιτημάτων έχει αυξηθεί και έχει οριστεί ως πρωταρχικός καθοριστικός παράγοντας της απόδοσης του συστήματος και του λειτουργικού κόστους. Παραδοσιακά, το οικοσύστημα Java έχει αντιμετωπίσει τον παραλληλισμό μέσω των Platform Threads. Περιβλημάτων γύρω από τα νήματα του πυρήνα (Kernel) του λειτουργικού συστήματος (OS). Αν και είναι ισχυρό, αυτό το μοντέλο περιορίζεται από το υψηλό κόστος πόρων των μονάδων εκτέλεσης που διαχειρίζεται ο πυρήνας [1].

Ο πρωταρχικός σκοπός αυτού του διπλωματικού εγγράφου είναι να διερευνήσει τα χαρακτηριστικά απόδοσης των Java 21 Virtual Threads (Project Loom). Η έρευνα αυτή επιδιώκει να αξιολογήσει τον ισχυρισμό ότι τα Virtual Threads μπορούν να εξαλείψουν τη μακροχρόνια αντιπαράθεση μεταξύ της απλότητας του κώδικα και της επεκτασιμότητας του συστήματος. Με την ανάπτυξη ενός πειραματικού Συγκεντρωτή Αναζήτησης Μικρουπηρεσιών (Microservice Search Aggregator), η μελέτη αυτή παρέχει μια συγκριτική ανάλυση της απόδοσης, της καθυστέρησης και της χρήσης πόρων μεταξύ του παλαιού μοντέλου threading 1 προς 1 και του σύγχρονου παραδείγματος προγραμματισμού M προς N.

Τελικά, η παρούσα εργασία αποτελεί μια αρχιτεκτονική αξιολόγηση της προγραμματιστικής μεθόδου «imperative-but-blocking». Αντιμετωπίζει την αυξανόμενη ανάγκη για συστήματα υψηλού βαθμού ταυτόχρονης εκτέλεσης που παραμένουν εύκολα στη συντήρηση. Απομονώνοντας τον μηχανισμό των νημάτων από τους περιορισμούς του υλικού (Hardware), διερευνούμε ένα μέλλον όπου το κόστος ενός νήματος είναι αμελητέο, αλλάζοντας ριζικά τον τρόπο με τον οποίο σχεδιάζονται οι υπηρεσίες backend στο οικοσύστημα Java.

1.2. Η εξέλιξη της παράλληλης επεξεργασίας και των απαιτήσεων του back-end

Ιστορικά, το μοντέλο «Thread-per-Request» αποτέλεσε τη βάση για τις επιχειρηματικές εφαρμογές Java. Σε αυτό το μοντέλο, η σύγχρονη ροή του κώδικα αντανακλούσε τη λογική ροή της επιχειρηματικής συναλλαγής, διευκολύνοντας τους προγραμματιστές στη συγγραφή και τον εντοπισμό σφαλμάτων. Ωστόσο, καθώς ο αριθμός των ταυτόχρονων χρηστών αυξήθηκε από εκατοντάδες σε δεκάδες χιλιάδες (το πρόβλημα C10k), το επιπλέον φορτίο των νημάτων πλατφόρμας μετατράπηκε σε εμπόδιο το «Memory Wall» [2].

Για να παρακάμψει αυτούς τους περιορισμούς υλικού, ο κλάδος στράφηκε προς τον ασύγχρονο και την αντιδραστικό προγραμματισμό (π.χ. Spring WebFlux, CompletableFuture). Ενώ αυτά τα frameworks αύξησαν σημαντικά τη διακίνηση δεδομένων, καθώς δεν «μπλοκάρουν» τα threads κατά τη διάρκεια των λειτουργιών εισόδου-εξόδου (I/O), εισήγαγαν ένα «γνωστικό κόστος». Οι προγραμματιστές αναγκάστηκαν να εγκαταλείψουν τη γραμμική λογική για σύνθετες αλυσίδες callback και λειτουργικές ροές, κάτι που έκανε ασαφή τα ίχνη στοίβας (stack traces) και περιπλέκει τη διαχείριση σφαλμάτων [3]. Η εισαγωγή των εικονικών νημάτων στο JDK 21 αντιπροσωπεύει μια

στρατηγική προσπάθεια της ομάδας OpenJDK να προσφέρει επεκτασιμότητα χωρίς πολυπλοκότητα, επιτρέποντας στον σύγχρονο κώδικα να επιτυγχάνει απόδοση σε επίπεδο ασύγχρονης λειτουργίας.

Η απομάκρυνση από την παραδοσιακή επιτακτική λογική, συχνά θεωρήθηκε αναγκαία για την επίτευξη επεκτασιμότητας, ωστόσο η θεμελιώδης αξία του Αντικειμενοστραφούς (Object-Oriented - OO) προγραμματισμού στη διαχείριση απαιτητικών δεδομένων παραμένει αδιαμφισβήτητη. Όπως έχουν καταδείξει οι Αλέπης και Βίρβου στην αξιολόγηση αντικειμενοστραφών αρχιτεκτονικών για τη διαχείριση πολυτροπικών δεδομένων (multimodal data) σε κινητά συστήματα, το μοντέλο OO προσφέρει δομικά στιβαρές και ποιοτικές λύσεις, επιλύοντας με επιτυχία προβλήματα συντονισμού ακόμη και στα πιο απαιτητικά περιβάλλοντα αλληλεπίδρασης [4]. Υπό αυτό το πρίσμα, η έλευση των Εικονικών Νημάτων (Virtual Threads) αποκτά ιδιαίτερη σημασία. Επιτρέπει στα σύγχρονα συστήματα backend να διατηρήσουν την αρχιτεκτονική καθαρότητα, την αναγνωσιμότητα και τη στιβαρότητα του Αντικειμενοστραφούς προγραμματισμού, επιτυγχάνοντας παράλληλα τα επίπεδα απόδοσης των πολύπλοκων ασύγχρονων μοντέλων.

Η μετάβαση από τις μονολιθικές εφαρμογές στην Αρχιτεκτονική Προσανατολισμένη στις Υπηρεσίες (SOA) και, στη συνέχεια, στις Μικρουπηρεσίες έχει αυξήσει την πυκνότητα εισόδου-εξόδου κάθε αιτήματος χρήστη. Ενώ παλαιότερα ένα μεμονωμένο αίτημα συνεπαγόταν μία κλήση στη βάση δεδομένων, σήμερα συχνά συντονίζει πολλές παράλληλες κλήσεις δικτύου. Αυτή η αύξηση απαιτήσεων έχει καταστήσει το παραδοσιακό μοντέλο νημάτων που εξαρτάται από το λειτουργικό σύστημα πεπερασμένο, καθώς το επιπλέον κόστος της διαχείρισης των μπλοκαρισμένων νημάτων συχνά καταναλώνει περισσότερους πόρους από την ίδια την επιχειρηματική λογική [1].

1.3. Περιγραφή του προβλήματος και αιτιολόγηση

Το βασικό πρόβλημα που εξετάζεται στην παρούσα διατριβή είναι η έλλειψη διαθεσιμότητας νημάτων (Thread Starvation) και η αναποτελεσματική αξιοποίηση του πολυπύρηνου υλικού. Στην παραδοσιακή διαχείριση νημάτων, όταν μια εφαρμογή αναμένει μια απάντηση από το δίκτυο (π.χ. ένα ερώτημα σε βάση δεδομένων ή μια κλήση εξωτερικού API), το υποκείμενο νήμα του λειτουργικού συστήματος παραμένει αδρανές, αλλά συνεχίζει να καταλαμβάνει περίπου 1 MB μνήμης στοίβας [5]. Σε ένα σύστημα με περιορισμένους πόρους, όπως το AMD Ryzen 5 2600 που χρησιμοποιήθηκε στην παρούσα έρευνα, αυτό οδηγεί σε μια κατάσταση όπου η CPU παραμένει υποαξιοποιημένη, ενώ η εφαρμογή δεν είναι σε θέση να δεχτεί νέες αιτήσεις λόγω κορεσμού του thread pool.

Το κίνητρο για την παρούσα μελέτη πηγάζει από την ανάγκη να προσαυξηθούν τα οφέλη της δομημένης παραλληλότητας (Structured Concurrency, JEP 453). Σε αντίθεση με τα παραδοσιακά νήματα, τα οποία είναι αδόμητα και μπορούν να επιβιώσουν τις γονικές τους εργασίες (οδηγώντας σε διαρροές πόρων), η δομημένη ταυτόχρονη εκτέλεση αντιμετωπίζει ομάδες σχετικών εργασιών ως μία ενιαία μονάδα εργασίας. Το παρόν άρθρο αξιολογεί τον τρόπο με τον οποίο αυτά τα νέα API διαχειρίζονται τον κύκλο ζωής χιλιάδων ταυτόχρονων εργασιών και τον τρόπο με τον οποίο αντιμετωπίζουν μερικές αποτυχίες σε ένα κατανεμημένο περιβάλλον.

Δηλαδή, το βασικό πρόβλημα που διερευνάται είναι το εμπόδιο επεκτασιμότητας λόγω μπλοκαρισμένων εισόδων-εξόδων (Blocking I/O). Στην παραδοσιακή Java, η επεκτασιμότητα ενός συστήματος εξαρτάται από τον προγραμματιστή (Scheduler) του λειτουργικού συστήματος, ο οποίος δεν σχεδιάστηκε ποτέ για να διαχειρίζεται εκατομμύρια ταυτόχρονες μονάδες εκτέλεσης. Η κεντρική επεξεργαστική μονάδα που χρησιμοποιήθηκε διαθέτει 12 λογικούς επεξεργαστές ικανούς για τεράστια παράλληλη απόδοση. Όμως οι παραδοσιακές στοίβες λογισμικού συχνά αφήνουν αυτούς

τους επεξεργαστές αδρανείς, επειδή η περιορισμένη ομάδα νημάτων (thread pool) είναι πλήρως μπλοκαρισμένη περιμένοντας απαντήσεις από το δίκτυο [6].

1.4. Μεθοδολογία της έρευνας και πεδίο εφαρμογής του πειράματος

Προκειμένου να διασφαλιστεί η επιστημονική εγκυρότητα της σύγκρισης, η παρούσα διατριβή υιοθετεί μια πολυεπίπεδη εμπειρική μεθοδολογία. Η έρευνα έχει σχεδιαστεί ως ελεγχόμενο πείραμα, στο οποίο η ανεξάρτητη μεταβλητή είναι το μοντέλο διακίνησης (πλατφόρμα έναντι εικονικού) και οι εξαρτώμενες μεταβλητές είναι ο χρόνος καθυστέρησης, η απόδοση και η αποδοτικότητα των πόρων.

1.4.1. Το προσομοιωμένο περιβάλλον μικρουπηρεσιών

Η μελέτη χρησιμοποιεί μια αρχιτεκτονική κατανεμημένης προσομοίωσης. Αντί να βασίζεται σε εξωτερικά API ιστού, τα οποία εισάγουν θόρυβο δικτύου, το πείραμα εφαρμόζει τοπικά «Mock API» με προκαθορισμένη εισαγωγή καθυστέρησης. Αυτό διασφαλίζει ότι οποιαδήποτε διακύμανση στην απόδοση που μετράται αποτελεί άμεσο αποτέλεσμα της εσωτερικής διαχείρισης νημάτων της Java και όχι εξωτερικών διακυμάνσεων του διαδικτύου.

1.4.2. Στατιστική επικύρωση μέσω JMH

Για να εξαλειφθεί η επίδραση των περιόδων «προθέρμανσης» της JVM και η παρεμβολή του μεταγλωττιστή Just-In-Time (JIT), η έρευνα χρησιμοποιεί το Java Microbenchmark Harness (JMH). Η μεθοδολογία περιλαμβάνει:

- Επαναλήψεις προθέρμανσης: Για να διασφαλιστεί ότι ο bytecode έχει βελτιστοποιηθεί σε εγγενή κώδικα μηχανής πριν από τη μέτρηση.
- Εκτέλεση πολλαπλών νημάτων: Χρήση της σημείωσης @Threads για την προσομοίωση διαφορετικών επιπέδων κορεσμού του συστήματος.
- Προφίλ κατανομής: Χρήση του GCProfiler για τη μέτρηση του ρυθμού κατανομής του σωρού (byte ανά λειτουργία) κάθε μοντέλου νημάτων.

1.4.3. Οπτικοποίηση σε πραγματικό χρόνο και δοκιμές αντοχής

Συμπληρωματικά προς τα μικρο-benchmarks, διεξάγεται μια δοκιμή αντοχής σε μακρο-επίπεδο. Ένα ειδικά σχεδιασμένο front-end React.js επικοινωνεί με το back-end Spring Boot για να απεικονίσει τη συμπεριφορά του συστήματος υπό συνθήκες υψηλής κυκλοφορίας. Αυτό το επίπεδο επιτρέπει την παρατήρηση της καθυστέρησης ουράς (P99) και τον εντοπισμό του σημείου κορεσμού του συστήματος, προσδιορίζοντας ακριβώς πότε η στοίβα δικτύωσης του λειτουργικού συστήματος (Ephemeral Ports) γίνεται το νέο σημείο συμφόρησης [7].

1.5. Ερευνητικά ερωτήματα

Η παρούσα διατριβή επιδιώκει να απαντήσει στα ακόλουθα συγκεκριμένα ερευνητικά ερωτήματα (Research Questions, RQs):

- RQ1: Σε τι βαθμό τα εικονικά νήματα (Virtual Threads) μειώνουν τον χρόνο εκτέλεσης σε σύγκριση με τα νήματα πλατφόρμας (Platform Threads) σε εργασίες συντονισμού που εξαρτώνται από τις εισόδους-εξόδους (I/O-bound);
- RQ2: Πώς επηρεάζει η αύξηση του φορτίου ταυτόχρονης εκτέλεσης το αποτύπωμα μνήμης (χρήση του Heap) των εικονικών νημάτων σε σύγκριση με την κράτηση χώρου στο stack των νημάτων πλατφόρμας;
- RQ3: Σε ποιο σημείο μετατοπίζεται το σημείο συμφόρησης από την εικονική μηχανή Java στο δίκτυο του λειτουργικού συστήματος;
- RQ4: Η χρήση της δομημένης ταυτόχρονης εκτέλεσης βελτιώνει τη διάδοση σφαλμάτων και την υγιεινή των πόρων σε περίπτωση μερικής αποτυχίας εργασιών;

1.6. Δομή της διατριβής

Η παρούσα διπλωματική διατριβή έχει την εξής δομή:

- **Κεφάλαιο 2**: *Θεωρητικό υπόβαθρο*: Λεπτομερής ανάλυση των λειτουργιών της JVM, του προγραμματισμού του λειτουργικού συστήματος και των μηχανισμών του Project Loom.
- **Κεφάλαιο 3:***Σχεδιασμός και αρχιτεκτονική συστήματος*: Περιγραφή της περίπτωσης χρήσης του Travel Aggregator, του οικοσυστήματος του εικονικού API και των προδιαγραφών υλικού (Hardware specifications).
- **Κεφάλαιο 4-5:***Υλοποίηση*: Τεχνική ανάλυση του backend Java 21, της οπτικοποίησης frontend React και της Java Microbenchmarking Harness(JMH).
- **Κεφάλαιο 6:***Αποτελέσματα και αξιολόγηση απόδοσης*: Παρουσίαση δεδομένων JMH, γραφημάτων χρήσης πόρων (RAM/CPU) και ανάλυση των εμποδίων που συναντήθηκαν.
- **Κεφάλαιο 7:***Συμπέρασμα και μελλοντική εργασία*: Περίληψη ευρημάτων, πρακτικές συστάσεις για τον κλάδο και πιθανές κατευθύνσεις έρευνας.

2. Θεωρητικό Πλαίσιο

Το παρόν κεφάλαιο εξετάζει σε βάθος τις τεχνικές λεπτομέρειες των μοντέλων νήματος της Java. Για να κατανοήσουμε γιατί τα εικονικά νήματα αποτελούν μια σημαντική καινοτομία, πρέπει πρώτα να αναλύσουμε τους περιορισμούς της κλασικής αρχιτεκτονικής νήματος της Εικονικής Μηχανής Java (JVM).

2.1. Το μοντέλο νημάτων της JVM και η διαχείριση μνήμης

Στις εκδόσεις Java πριν από την έκδοση 21, η κλάση java.lang.Thread αντιπροσώπευε αυτό που σήμερα ονομάζουμε νήμα πλατφόρμας (Platform Thread). Αυτά τα νήματα αποτελούν ουσιαστικά περιβλήματα γύρω από τα νήματα του πυρήνα του λειτουργικού συστήματος (OS). Η αρχιτεκτονική αυτή ακολουθεί ένα μοντέλο αντιστοίχισης 1:1, όπου κάθε νήμα Java απαιτεί ένα αντίστοιχο νήμα στο λειτουργικό σύστημα υποδοχής [1].

2.1.1. Πολυνηματισμός σε επίπεδο πυρήνα και ο χρονοπρογραμματιστής

Επειδή τα νήματα πλατφόρμας είναι οντότητες σε επίπεδο πυρήνα, ο κύκλος ζωής τους (δημιουργία, προγραμματισμός και τερματισμός) διαχειρίζεται από τον πυρήνα του λειτουργικού συστήματος. Όταν η JVM χρειάζεται να εκτελέσει μια εργασία, ζητά ένα νήμα από το λειτουργικό σύστημα. Στη συνέχεια, ο προγραμματιστής του λειτουργικού συστήματος καθορίζει ποιος φυσικός πυρήνας της CPU θα εκτελέσει το συγκεκριμένο νήμα. Σε έναν επεξεργαστή όπως ο AMD Ryzen 5 2600, ο χρονοπρογραμματιστής πρέπει να διαχειριστεί την κατανομή εκατοντάδων ή χιλιάδων νημάτων λογισμικού σε μόλις 12 λογικούς επεξεργαστές [6]. Αυτό οδηγεί σε έντονο ανταγωνισμό μεταξύ των νημάτων για τον χρόνο που θα έχουν στον CPU, ο οποίος διαχειρίζεται μέσω της χρονικής κατανομής.

2.1.2. Το σημείο συμφόρησης της μνήμης στοίβας

Ο σημαντικότερος περιορισμός των Threads της πλατφόρμας είναι η κατανάλωση μνήμης. Σε κάθε εγγενές νήμα εκχωρείται μια ιδιωτική στοίβα για την αποθήκευση τοπικών μεταβλητών, παραμέτρων μεθόδων και διευθύνσεων επιστροφής. Από προεπιλογή, η JVM εκχωρεί 1 MB για κάθε στοίβα νήματος [5]. Αυτή η μνήμη δεσμεύεται εκτός του σωρού (off-heap) και είναι ανεξάρτητη από το μέγεθος του σωρού της Java.

Για ένα σύστημα υψηλής ταυτόχρονης πρόσβασης, αυτό δημιουργεί ένα ανώτατο όριο μνήμης. Για παράδειγμα, για να υποστηρίξει 10.000 ταυτόχρονες αιτήσεις χρησιμοποιώντας το μοντέλο «Thread-per-Request», η εφαρμογή θα απαιτούσε:

$$10000\ threads \times \frac{1MB}{Thread} = 10\ GB\ RAM \qquad (2.1)$$

Αυτά τα 10 GB προορίζονται αποκλειστικά για τη διατήρηση των νημάτων, ακόμη και αν αυτά βρίσκονται σε κατάσταση αδράνειας ή περιμένουν απάντηση από το δίκτυο. Αυτή η τεράστια επιβάρυνση καθιστά το παραδοσιακό μοντέλο ακατάλληλο για τις σύγχρονες μικροϋπηρεσίες, οι οποίες απαιτούν υψηλή οριζόντια επεκτασιμότητα σε οικονομικά αποδοτικό υλικό.

2.1.3. Ο νόμος του Little

Για να αξιολογήσουμε την επεκτασιμότητα των δύο μοντέλων νήματος, πρέπει να εφαρμόσουμε τον νόμο του Little από τη θεωρία των ουρών αναμονής [8]. Η σχέση εκφράζεται ως εξής:

$$L = \lambda \cdot W \qquad (2.2)$$

Όπου:

- L(Lead): Ο αριθμός των αιτήσεων που υποβάλλονται σε επεξεργασία ταυτόχρονα (ταυτόχρονη επεξεργασία).
- λ (Ρυθμός άφιξης): Ο ρυθμός με τον οποίο εισέρχονται νέες αιτήσεις στο σύστημα (απόδοση).
- W(Χρόνος αναμονής): Ο μέσος χρόνος που ένα αίτημα παραμένει στο σύστημα (Καθυστέρηση).

Στο μοντέλο Platform Thread, το L περιορίζεται αυστηρά από τους περιορισμούς μνήμης της στοίβας του λειτουργικού συστήματος (1MB ανά νήμα). Εάν το L φτάσει στο όριό του, οποιαδήποτε αύξηση του ρυθμού άφιξης λ αναγκάζει τον χρόνο αναμονής W να αυξηθεί, οδηγώντας στις αιχμές καθυστέρησης που παρατηρήθηκαν στα τεστ επιδόσεων του Platform Thread. Τα Virtual Threads

αποσυνδέουν L το από τα όρια μνήμης του υλικού, επιτρέποντας στο σύστημα να διατηρήσει ένα σταθερό W ακόμη και όταν το λ αυξάνεται δραματικά.

2.2. Αλλαγή περιβάλλοντος (Context Switching) και επιβάρυνση του συστήματος

Όταν ο αριθμός των νημάτων της πλατφόρμας υπερβαίνει τον αριθμό των διαθέσιμων νημάτων του υλικού (στην περίπτωση αυτή, 12), το λειτουργικό σύστημα πρέπει να πραγματοποιεί συχνές εναλλαγές περιβάλλοντος, ώστε να δώσει σε κάθε νήμα την ευκαιρία να εκτελεστεί.

2.2.1. Οι μηχανισμοί της αλλαγής περιβάλλοντος

Η αλλαγή περιβάλλοντος είναι μια σύνθετη διαδικασία κατά την οποία ο πυρήνας του λειτουργικού συστήματος:

1. Αναστέλλει το νήμα που εκτελείται εκείνη τη στιγμή.
2. Αποθηκεύει την τρέχουσα κατάστασή του (μετρητή προγράμματος, καταχωρητές και δείκτη στοίβας) σε ένα μπλοκ ελέγχου νήματος (TCB).
3. Φορτώνει την κατάσταση του επόμενου νήματος από το TCB του.
4. Ακυρώνει τις κρυφές μνήμες L1/L2 της CPU και τον buffer παράκαμψης μετάφρασης (TLB) [6].

2.2.2. Μείωση της απόδοσης σε συνθήκες ανταγωνισμού

Σε εφαρμογές που εξαρτώνται από τις εισόδους-εξόδους (I/O), τα νήματα περνούν το μεγαλύτερο μέρος του χρόνου τους σε μπλοκαρισμένη κατάσταση (BLOCKED), περιμένοντας δεδομένα από μια υποδοχή (socket) ή μια βάση δεδομένων. Στο μοντέλο πλατφόρμας, ένα μπλοκαρισμένο νήμα εξακολουθεί να καταλαμβάνει τη στοίβα του 1 MB και αναγκάζει το λειτουργικό σύστημα να το διαχειρίζεται. Καθώς αυξάνεται το φορτίο, η CPU αφιερώνει περισσότερους κύκλους στην εκτέλεση εργασιών διαχείρισης (αλλαγή περιβάλλοντος) παρά στην εκτέλεση της πραγματικής επιχειρηματικής λογικής. Αυτό έχει ως αποτέλεσμα τον κορεσμό της απόδοσης, όπου η προσθήκη περισσότερων νημάτων μειώνει στην πραγματικότητα την απόδοση του συστήματος λόγω του διοικητικού φόρτου στον CPU [7].

2.2.3. Μεταβάσεις από τη λειτουργία χρήστη στη λειτουργία πυρήνα

Ένας βασικός λόγος για τη μείωση της απόδοσης των «Platform Threads» είναι η αναγκαιότητα των κλήσεων συστήματος (syscalls). Σε ένα μοντέλο νήματος 1:1, κάθε λειτουργία είτε πρόκειται για τη δημιουργία νήματος, τον συγχρονισμό μέσω κλειδαριών (locks) είτε για την αλλαγή περιβάλλοντος, απαιτεί μια κλήση στον πυρήνα του λειτουργικού συστήματος [6].

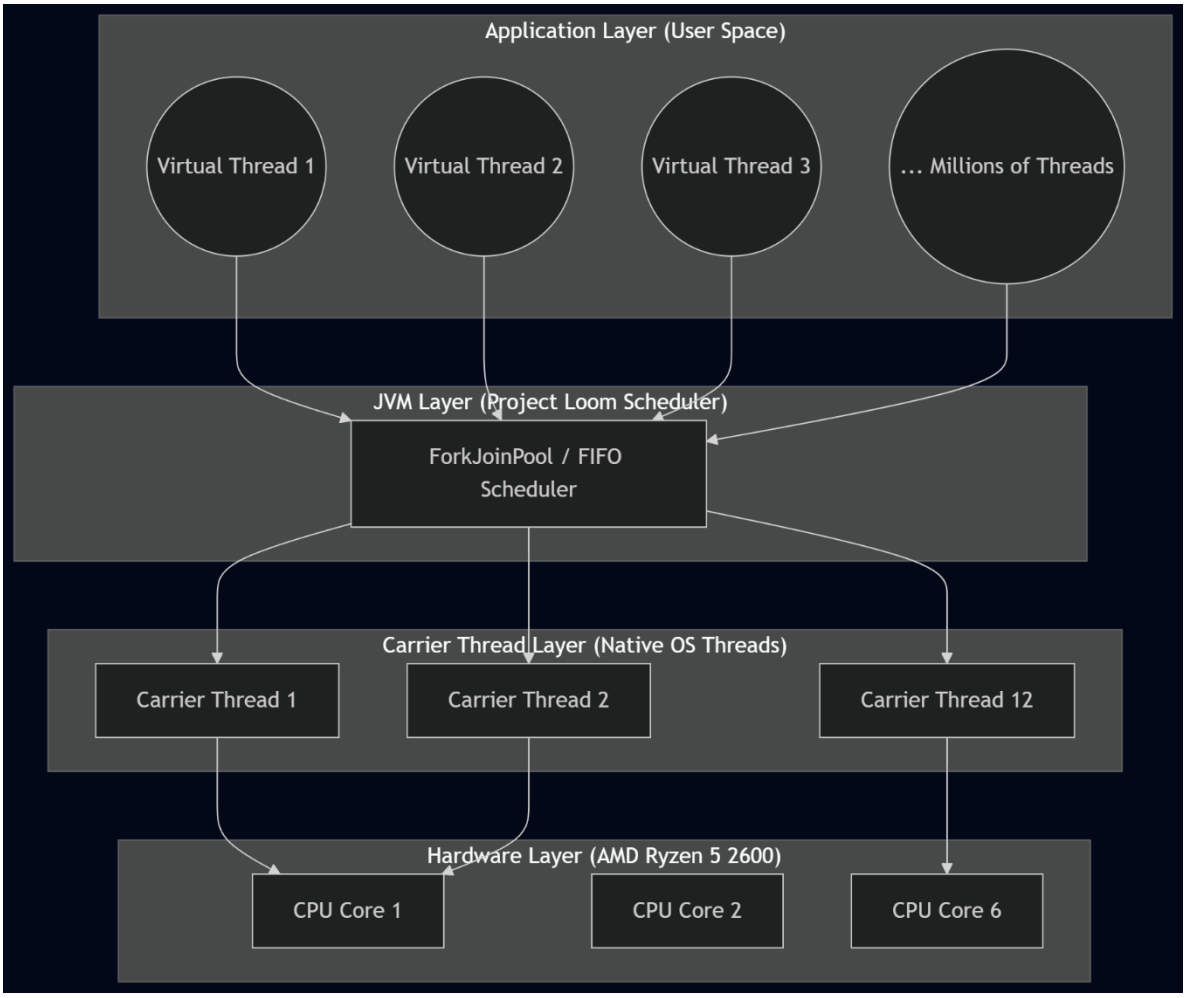
Αυτή η μετάβαση από τη λειτουργία χρήστη (User-Mode), όπου εκτελείται η JVM, στη λειτουργία πυρήνα (Kernel-Mode) είναι υπολογιστικά απαιτητική. Περιλαμβάνει μια αλλαγή επιπέδου «προνομίων» που αναγκάζει την CPU να σταματήσει την εκτέλεση της λογικής της εφαρμογής και να εισέλθει σε μια κατάσταση περιορισμένης πρόσβασης, ώστε να επιτρέψει στο λειτουργικό σύστημα να τροποποιήσει την ουρά του χρονοπρογραμματιστή. Αντίθετα, τα εικονικά νήματα (Virtual Threads)

διαχειρίζονται εξ ολοκλήρου στη λειτουργία χρήστη (User-Mode) από το περιβάλλον εκτέλεσης της JVM. Το λειτουργικό σύστημα αγνοεί εντελώς την ύπαρξη των εικονικών νημάτων, πράγμα που σημαίνει ότι μπορούν να πραγματοποιηθούν χιλιάδες αλλαγές χωρίς ούτε μία υπολογιστικά δαπανηρή κλήση συστήματος [3].

Πίνακας 2.1 Σύγκριση των ιδιοτήτων πλατφόρμας και εικονικών νημάτων.

Ιδιότητα	**Νήματα πλατφόρμας**	**Εικονικά νήματα**
Αντιστοίχιση	1:1 (Java προς OS)	M:N (Εικονικό προς φορέα)
Κόστος δημιουργίας	Υψηλό (χιλιοστά του δευτερολέπτου / Syscall)	Χαμηλό (μικροδευτερόλεπτα / αντικείμενο JVM)
Μέγεθος Στοίβας	Σταθερό (Προεπιλογή 1MB)	Δυναμικό (Ξεκινά από ~2KB)
Θέση μνήμης	Εκτός σωρού (Διατηρείται από το λειτουργικό σύστημα)	Σωρός Java (Συλλογή απορριμμάτων)
Αλλαγή περιβάλλοντος	Λειτουργία πυρήνα (Ακριβό)	Λειτουργία χρήστη (Φθηνό)

2.2.4. Ανάλυση του μοντέλου M:N



Διάγραμμα 2.1 Μοντέλο M:N.

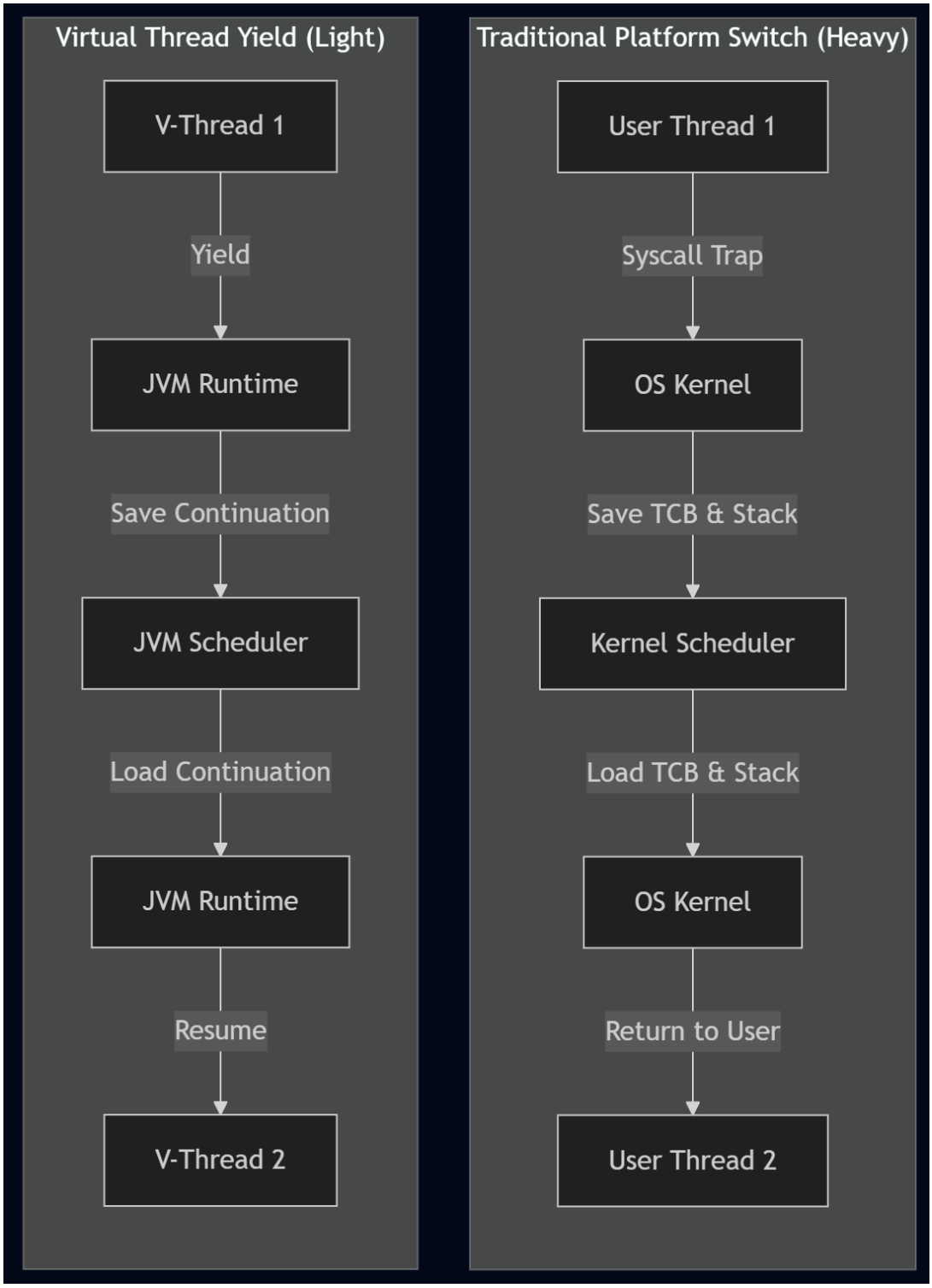
Το Διάγραμμα 2.1 αποτελεί το δομικό σχέδιο της παρούσας έρευνας, απεικονίζοντας την ενοποίηση μεταξύ του κώδικα της εφαρμογής και των φυσικών μονάδων εκτέλεσης του AMD Ryzen 5 2600. Στην κορυφή αυτής της ιεραρχίας βρίσκεται το Επίπεδο της Εφαρμογής, το οποίο βρίσκεται στον Χώρο του Χρήστη. Εδώ, η αφαίρεση που παρέχει το Project Loom επιτρέπει τη δημιουργία εκατομμυρίων εικονικών νημάτων. Είναι σημαντικό να σημειωθεί ότι αυτά τα νήματα δεν προγραμματίζονται από το λειτουργικό σύστημα υπάρχουν απλώς ως παθητικές δομές δεδομένων εντός της εικονικής μηχανής Java (JVM). Αυτό αντιπροσωπεύει μια απομάκρυνση από την τριάνταχρονη ιστορία της Java, όπου η επεκτασιμότητα μιας εφαρμογής ήταν άμεσα συνδεδεμένη με το όριο νημάτων του πυρήνα.

Το επίπεδο JVM λειτουργεί ως ο κύριος συντονιστής, χρησιμοποιώντας ένα εξειδικευμένο ForkJoinPool διαμορφωμένο σε λειτουργία First-In-First-Out (FIFO). Αυτός ο προγραμματιστής λειτουργεί ως πολυπλέκτης υψηλής ταχύτητας. Όταν η λογική της εφαρμογής ξεκινά μια εργασία, ο προγραμματιστής καθορίζει πώς θα αντιστοιχίσει αυτά τα εκατομμύρια εικονικών μονάδων στο επίπεδο νήματος φορέα. Ένα σημαντικό εύρημα αυτής της μελέτης είναι η αυτόματη ανίχνευση της

τοπολογίας του υλικού από την JVM. Στο σύστημα δοκιμών, η JVM αναγνώρισε τους 6 φυσικούς πυρήνες και τους 12 λογικούς επεξεργαστές του Ryzen 2600 (μέσω Simultaneous Multithreading - SMT), αρχικοποιώντας στη συνέχεια ακριβώς 12 Carrier Threads.

Αυτή η αντιστοίχιση 1:1 μεταξύ των νήματος-φορέων (Carrier Threads) και των λογικών επεξεργαστών αποτελεί το ιδανικό σημείο από άποψη απόδοσης. Διατηρώντας μια μικρή, σταθερή ομάδα εγγενών νημάτων, το σύστημα ελαχιστοποιεί το φορτίο στον προγραμματιστή του πυρήνα των Windows/Linux. Αντί το λειτουργικό σύστημα να εναλλάσσει συνεχώς χιλιάδες βαριά νήματα μέσα και έξω από την CPU, μια διαδικασία που οδηγεί σε ακύρωση της προσωρινής μνήμης (cache) και σε αστοχίες των προσωρινών μνημών L1/L2, το λειτουργικό σύστημα βλέπει μόνο 12 νήματα σε λειτουργία. Η πραγματική εναλλαγή εργασιών πραγματοποιείται εντός της JVM με πολύ υψηλότερη συχνότητα και χαμηλότερο κόστος. Κατά συνέπεια, το επίπεδο υλικού αξιοποιείται με σχεδόν θεωρητική μέγιστη απόδοση, καθώς οι θύρες εκτέλεσης της CPU σπάνια παραμένουν αδρανείς κατά τη διάρκεια των κύκλων αναμονής I/O των εικονικών νημάτων.

2.2.5. Μηχανιστική ανάλυση της επιβάρυνσης εναλλαγής



Διάγραμμα 2.2 Ροή Εναλλαγής

Για να ποσοτικοποιήσουμε της διαφορά απόδοσης που παρατηρήθηκε στα τεστ επιδόσεων μας (1,5 δευτ. έναντι 164 δευτ.), πρέπει να εξετάσουμε τη μικρομηχανική της διαδικασίας εναλλαγής που απεικονίζεται στο Σχήμα 2.2. Η παραδοσιακή εναλλαγή πλατφόρμας (δεξιά πλευρά του διαγράμματος) είναι μια λειτουργία υψηλών απαιτήσεων. Όταν ένα νήμα πλατφόρμας συναντά μια κλήση που προκαλεί μπλοκάρισμα, ενεργοποιεί μια παγίδα Syscall. Αυτό αναγκάζει την CPU να εκτελέσει μια αλλαγή επιπέδου προνομίων, μεταβαίνοντας από το Ring 3 (Χρήστης) στο Ring 0 (Πυρήνας). Ο πυρήνας πρέπει στη συνέχεια να εκτελέσει μια ρουτίνα «Save State», καταγράφοντας τον μετρητή προγράμματος, τους καταχωρητές γενικής χρήσης και τον δείκτη στοίβας. Δεδομένου ότι κάθε νήμα πλατφόρμας διαθέτει μια στοίβα 1 MB που έχει δεσμευτεί, το επιπλέον φορτίο διαχείρισης μνήμης για τον πυρήνα δεν είναι αμελητέο. Αυτή η διαδικασία είναι πολύπλοκη από άποψη υλικού και έχει συνέπειες. Καθώς συχνά απαιτεί την εκκαθάριση του Translation Lookaside Buffer (TLB) και έχει ως αποτέλεσμα ένα «Cold Cache» για το επόμενο νήμα που αναλαμβάνει.

Σε έντονη αντίθεση, το Virtual Thread Yield (αριστερά πλευρά του διαγράμματος) αντιπροσωπεύει μια πιο καθαρή εναλλαγή. Επειδή η JVM διαχειρίζεται τον κύκλο ζωής, δεν απαιτείται Syscall. Το νήμα απλώς παραχωρεί τη συνέχισή του. Η JVM αναγνωρίζει τα ενεργά πλαίσια εκτέλεσης στη στοίβα, συχνά μόνο μερικές εκατοντάδες byte, και τα αντιγράφει στη σωρό της Java. Το Carrier Thread αποδεσμεύεται αμέσως και καθίσταται διαθέσιμο για να παραλάβει το επόμενο Virtual Thread από την τοπική ουρά.

Αυτός ο μηχανισμός εξηγεί τη σταθερότητα της καθυστέρησης ουράς (Tail Latency) των εικονικών νημάτων. Στο μοντέλο της πλατφόρμας, καθώς αυξάνεται ο αριθμός των νημάτων, ο χρόνος που αφιερώνει η CPU σε διοικητικές εργασίες (αλλαγή περιβάλλοντος) αυξάνεται εκθετικά. Τελικά, το σύστημα φτάνει σε ένα σημείο κορεσμού όπου καταναλώνεται περισσότερη ενέργεια στην αλλαγή περιβάλλοντος παρά στην εκτέλεση της επιχειρησιακής λογικής. Το αποτέλεσμα των 164 δευτερολέπτων για τα νήματα πλατφόρμας είναι μια άμεση μαθηματική συνέπεια εκατομμυρίων μεταβάσεων σε λειτουργία πυρήνα. Διατηρώντας τη λογική στον χώρο χρήστη, τα εικονικά νήματα είναι ανεπηρέαστα από τα όρια του λειτουργικού συστήματος, επιτρέποντας μια εναλλαγή που είναι 50 έως 100 φορές ταχύτερη από μια εγγενή εναλλαγή περιβάλλοντος.

2.3. Εικονικά νήματα: Αποσύζευξη (Decoupling) και το μοντέλο M:N

Το Project Loom εισάγει τα εικονικά νήματα (Virtual Threads), τα οποία όπως προαναφέρθηκε, δεν διαχειρίζεται το λειτουργικό σύστημα, αλλά η ίδια η JVM. Αυτό δημιουργεί μια αντιστοίχιση τύπου M:N, όπου «M» εικονικά νήματα πολυπλέκονται σε «N» νήματα της πλατφόρμας (γνωστά ως Carrier Threads) [3].

2.3.1. Διαχείριση στοίβας με βάση τη σωρό

Σε αντίθεση με τα νήματα πλατφόρμας, τα εικονικά νήματα δεν διαθέτουν σταθερή στοίβα 1 MB στη μνήμη του λειτουργικού συστήματος. Αντίθετα, οι στοίβες τους αποθηκεύονται ως αντικείμενα στον σωρό της Java. Όταν ξεκινά ένα εικονικό νήμα, η αρχική του στοίβα είναι μόλις μερικές εκατοντάδες byte. Αυξάνεται και συρρικνώνεται δυναμικά ανάλογα με τις ανάγκες. Αυτό επιτρέπει στη JVM να φιλοξενεί εκατομμύρια εικονικά νήματα στον ίδιο χώρο σωρού, ο οποίος θα χωρούσε μόνο μερικές εκατοντάδες νήματα πλατφόρμας [5].

2.4. Ο βασικός μηχανισμός του Project Loom

Ο τεχνολογικός πυλώνας που επιτρέπει στα εικονικά νήματα να σταματούν και να επαναλαμβάνουν τη λειτουργία τους χωρίς να μπλοκάρουν έναν φυσικό πυρήνα της CPU είναι η συνέχεια. Στη θεωρία της επιστήμης των υπολογιστών, μια συνέχεια αντιπροσωπεύει το υπολειπόμενο τμήμα ενός υπολογισμού σε οποιοδήποτε δεδομένο σημείο [3].

2.4.1. Ο κύκλος παραχώρησης και επανέναρξης (Yield and Resume)

Όταν ένα εικονικό νήμα εκτελεί μια λειτουργία αποκλεισμού (όπως η κλήση restClient.get() στον Aggregator μας), δεν παραμένει στην CPU. Αντ' αυτού:

1. Παραχώρηση (Yield): Η JVM καταγράφει την τρέχουσα κατάσταση της στοίβας του νήματος, συμπεριλαμβανομένων των τοπικών μεταβλητών και του δείκτη εντολών.
2. Αποσύνδεση (Unmounting): Αυτή η καταγεγραμμένη κατάσταση μεταφέρεται από το νήμα-φορέα (το εγγενές νήμα) στο Java Heap. Το νήμα-φορέας είναι πλέον ελεύθερο και αναζητά αμέσως ένα άλλο εικονικό νήμα για εκτέλεση.
3. Αναμονή (Waiting): Το εικονικό νήμα υπάρχει μόνο ως παθητικό αντικείμενο στο Heap, ενώ η λειτουργία I/O (δικτύωση) ολοκληρώνεται σε επίπεδο λειτουργικού συστήματος.
4. Επανέναρξη (Resume): Μόλις τα δεδομένα φτάσουν από το δίκτυο, ο προγραμματιστής της JVM επαναφέρει τη συνέχιση. Αντιγράφει την κατάσταση της στοίβας πίσω από το Heap σε ένα διαθέσιμο Carrier Thread και ο κώδικας συνεχίζει ακριβώς από το σημείο που είχε σταματήσει [5].

2.4.2. Ο μηχανισμός προγραμματισμού: ForkJoinPool και Work-Stealing

Τα εικονικά νήματα προγραμματίζονται χρησιμοποιώντας έναν αλγόριθμο «Work-Stealing». Κάθε νήμα-φορέας (εγγενές νήμα) διατηρεί μια τοπική ουρά διπλής κατεύθυνσης (deque) με εικονικά νήματα έτοιμα για εκτέλεση.

- Αποδοτικότητα: Εάν ένα νήμα-φορέας που σχετίζεται με τον πυρήνα 1 του Ryzen 2600 ολοκληρώσει την ουρά του, δεν παραμένει αδρανές. Αντίθετα, ένα εικονικό νήμα από την ουρά ενός άλλου πυρήνα μπαίνει στην δικιά του ουρά.
- LIFO έναντι FIFO: Ενώ τα τυπικά ForkJoinPools χρησιμοποιούν τη μέθοδο Last-In-First-Out (LIFO) για λόγους απόδοσης, ο προγραμματιστής εικονικών νημάτων χρησιμοποιεί τη μέθοδο First-In-First-Out (FIFO) για να εξασφαλίσει την δίκαιη κατανομή και να αποτρέψει την εξάντληση αιτημάτων σε περιβάλλοντα ιστού [1].

2.5. Το φαινόμενο της καθήλωσης νήματος (Thread Pinning)

Αν και τα εικονικά νήματα (Virtual Threads) αποτελούν ένα σημαντικό άλμα στην αρχιτεκτονική της JVM, δεν αποτελούν την απόλυτη λύση. Η σημαντικότερη τεχνική πρόκληση που αντιμετωπίζεται στην ανάπτυξη Java με υψηλό βαθμό ταυτόχρονης εκτέλεσης είναι το «thread pinning». Το «pinning» συμβαίνει όταν ένα εικονικό νήμα καθηλώνεται φυσικά στο νήμα-φορέα (Carrier Thread) του, εμποδίζοντας τη JVM να το αποσυνδέσει κατά τη διάρκεια μιας λειτουργίας που προκαλεί μπλοκάρισμα.

2.5.1. Συγχρονισμένα μπλοκ και εγγενείς κλήσεις

Η κύρια αιτία του pinning είναι η χρήση της λέξης-κλειδιού synchronized. Όταν ένα εικονικό νήμα εισέρχεται σε ένα συγχρονισμένο κομμάτι κώδικα ή μέθοδο, η τρέχουσα υλοποίηση της JVM (από το JDK 21/23) δεν μπορεί να μετακινήσει τα πλαίσια στοίβας στο σωρό. Εάν ο κώδικας εκτελέσει στη συνέχεια μια λειτουργία I/O που προκαλεί μπλοκάρισμα (όπως η ερώτησή μας στη βάση δεδομένων ή η κλήση API) ενώ βρίσκεται μέσα σε αυτό το μπλοκ, το υποκείμενο νήμα-φορέας μπλοκάρεται επίσης.

2.5.2. Συνέπειες και αντιμετώπιση

Η επίπτωση του «pinning» είναι η πλήρης απώλεια των πλεονεκτημάτων επεκτασιμότητας που προσφέρει το Project Loom. Εάν και τα 12 Carrier Threads του Ryzen 2600 υποστούν «pinning», η εφαρμογή δεν μπορεί πλέον να επεξεργάζεται νέα Virtual Threads, επιστρέφοντας ουσιαστικά στο προφίλ απόδοσης μιας μικρής ομάδας Platform Threads. Για να μετριαστεί αυτό, τα σύγχρονα πρότυπα Java επιβάλλουν την αντικατάσταση του synchronized με το java.util.concurrent.locks.ReentrantLock. Σε αντίθεση με τον συγχρονισμό βάσει monitor του παρελθόντος, το ReentrantLock είναι «Loom-aware», επιτρέποντας στο Virtual Thread να παραχωρεί σωστά ακόμη και όταν διατηρεί ένα κλείδωμα. Αυτό απαιτεί έναν σημαντικό έλεγχο των παλαιών βιβλιοθηκών και των «thread-safe» βάσεων κώδικα, για να διασφαλιστεί ότι είναι συμβατές με τις απαιτήσεις υψηλής πυκνότητας των Virtual Threads.

2.6. Εικονικά νήματα έναντι του αντιδραστικού παραδείγματος

Πριν από την κυκλοφορία της Java 21, η μόνη βιώσιμη λύση για το πρόβλημα C10k ήταν ο αντιδραστικός προγραμματισμός (π.χ. Spring WebFlux, Project Reactor). Σε αυτή την ενότητα συγκρίνεται η λειτουργική προσέγγιση της τελευταίας δεκαετίας με την εντολική προσέγγιση του Project Loom.

Η σύγκριση μεταξύ των αντιδραστικών (reactive) πλαισίων και της επιτακτικής (imperative) προσέγγισης αποτελεί κεντρικό θέμα της πρόσφατης ακαδημαϊκής έρευνας. Όπως απέδειξαν οι Mochniej και Badurowicz (2023) [9], ενώ τα παραδοσιακά imperative μικροϋπηρεσιακά περιβάλλοντα υστερούσαν σε απόδοση υπό συνθήκες υψηλού I/O φόρτου, η μετάβαση σε reactive λύσεις (όπως το Spring WebFlux) αύξανε δραματικά την πολυπλοκότητα του κώδικα. Με την έλευση του Project Loom, η σύγχρονη έρευνα επαναξιολογεί αυτά τα δεδομένα. Πρόσφατες μελέτες (Ponnampalam, 2025) [10] επιβεβαιώνουν ότι τα Εικονικά Νήματα (Virtual Threads) ξεπερνούν πλέον σε απόδοση πολύπλοκα frameworks όπως το RxJava σε σενάρια μαζικής ταυτόχρονης εκτέλεσης και μπλοκαρισμένου I/O, διατηρώντας ταυτόχρονα εξαιρετικά χαμηλή κυκλωματική πολυπλοκότητα (cyclomatic complexity) στον κώδικα.

2.6.1. Το πρόβλημα του «χρωματισμού συναρτήσεων»

Ο αντιδραστικός προγραμματισμός εισάγει αυτό που οι ερευνητές αποκαλούν το πρόβλημα της «διχρωμικής συνάρτησης». Σε ένα αντιδραστικό σύστημα, μια ασύγχρονη συνάρτηση (που επιστρέφει ένα Mono ή Flux) δεν μπορεί να κληθεί εύκολα από μια τυπική σύγχρονη συνάρτηση. Αυτό αναγκάζει την ασυγχρονία να επηρεάσει ολόκληρο το κώδικα μόλις ένα επίπεδο γίνει αντιδραστικό,

πρέπει να ακολουθήσει ολόκληρη η αλυσίδα κλήσεων. Αυτό οδηγεί στη λεγόμενη κόλαση κλήσεων (Callback Hell) ή σε πολύπλοκες λειτουργικές ροές που είναι δύσκολο να κατανοηθούν από τον άνθρωπο.

2.6.2. Δυνατότητα εντοπισμού σφαλμάτων και ακεραιότητα του ίχνους στοίβας

Ένα από τα πιο σημαντικά κόστη του αντιδραστικού προγραμματισμού είναι η απώλεια ουσιαστικών ιχνών στοίβας. Επειδή οι εργασίες χωρίζονται σε μικρά τμήματα που εκτελούνται από διαφορετικά νήματα σε διαφορετικές χρονικές στιγμές, μια εξαίρεση σε μια αντιδραστική ροή επεξεργασίας συχνά παράγει ένα ίχνος στοίβας που δείχνει προς τον εσωτερικό κινητήρα του πλαισίου και όχι προς τη γραμμή κώδικα που προκάλεσε το σφάλμα.

Τα εικονικά νήματα επιλύουν αυτό το πρόβλημα επιτρέποντας τη χρήση της επιτακτικής λογικής. Ο κώδικας που γράψαμε για το Travel Aggregator μοιάζει ακριβώς με τον τυπικό, σύγχρονο κώδικα Java. Ωστόσο, κλιμακώνεται όπως ένα αντιδραστικό σύστημα. Αυτό επιτρέπει στους προγραμματιστές να χρησιμοποιούν τυπικά τμήματα κώδικα try-catch, τυπικούς βρόχους και το πιο σημαντικό, τυπικά προγράμματα εντοπισμού σφαλμάτων. Σε ένα περιβάλλον εικονικών νημάτων, ένα ίχνος στοίβας είναι ένα πλήρες, γραμμικό ιστορικό του αιτήματος, μειώνοντας δραστικά τον μέσο χρόνο επιδιόρθωσης (MTTR) σε περιβάλλοντα παραγωγής.

3. Σχεδιασμός και Αρχιτεκτονική Συστήματος

Στο παρόν κεφάλαιο περιγράφεται λεπτομερώς η αρχιτεκτονική του συστήματος που αναπτύχθηκε για την αξιολόγηση των μοντέλων παράλληλης εκτέλεσης της Java. Παρουσιάζεται ο σχεδιασμός του πειραματικού «Microservice Search Aggregator», οι προδιαγραφές υλικού που χρησιμοποιήθηκαν για τη διεξαγωγή συγκριτικών δοκιμών, καθώς και η πειραματική μεθοδολογία που εφαρμόστηκε κατά τη διάρκεια των φάσεων δοκιμών.

3.1. Αρχιτεκτονικά Πρότυπα και Σχεδιασμός

Για να διασφαλιστεί η εγκυρότητα του πειράματος, υιοθετήθηκε μια σύγχρονη προσέγγιση σχεδιασμού διαδικτυακών εφαρμογών, η οποία διαχωρίζει αυστηρά τη λογική της επιχείρησης από το περιβάλλον εργασίας χρήστη και τις λειτουργίες εισόδου/εξόδου (I/O).

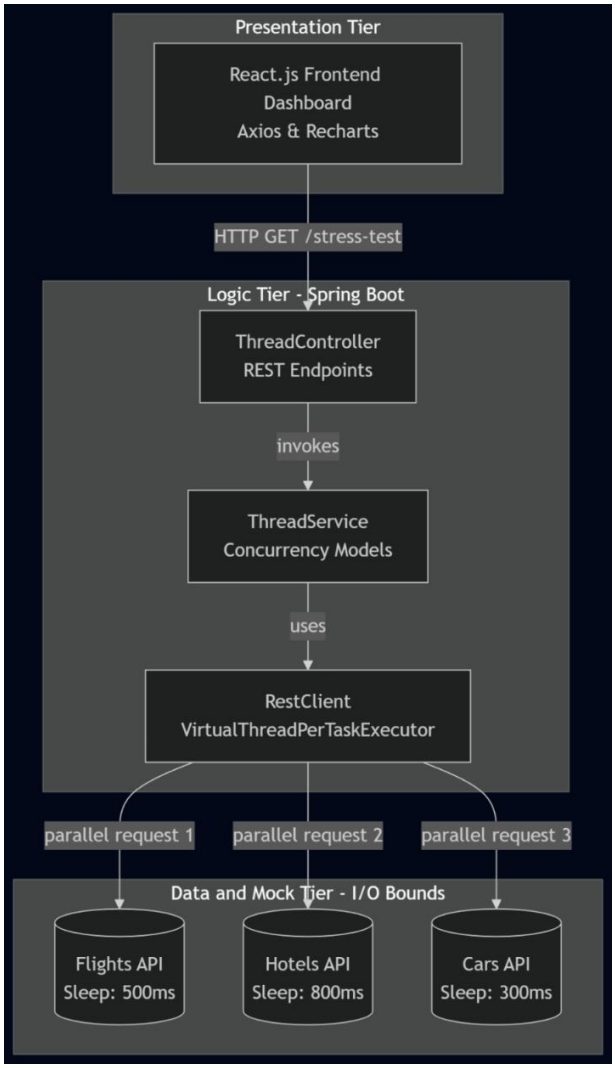
3.1.1. Το Πρότυπο Ενορχήστρωσης Μικρουπηρεσιών

Το σύστημα εφαρμόζει το πρότυπο Microservice Orchestrator. Σε αυτή την αρχιτεκτονική, ο κεντρικός διακομιστής (Spring Boot) δεν παράγει δεδομένα από μόνος του, αλλά λειτουργεί ως συντονιστής. Λαμβάνει ένα εισερχόμενο αίτημα HTTP και εκτελεί μια λειτουργία fan-out για την παράλληλη εκτέλεση πολλαπλών κλήσεων προς τα κάτω. Η επιλογή αυτού του προτύπου είναι στρατηγική, καθώς μεγιστοποιεί τη ζήτηση για συντονισμό νημάτων και αναδεικνύει τις διαφορές απόδοσης μεταξύ των νημάτων πλατφόρμας (Platform Threads) και των εικονικών νημάτων (Virtual Threads) κατά τη διάρκεια των χρόνων αναμονής I/O [11].

3.1.2. Κατανεμημένη Αρχιτεκτονική

Το σύστημα ακολουθεί μια κατανεμημένη αρχιτεκτονική τριών επιπέδων (3-Tier), σχεδιασμένη ώστε να απομονώνει τον θόρυβο των μετρήσεων. Η επιλογή και η αξιολόγηση μιας τέτοιας αρχιτεκτονικής συνδέεται άμεσα με την ανάγκη διασφάλισης υψηλής Ποιότητας Υπηρεσίας (Quality of Service - QoS). Όπως έχει καταδειχθεί σε σχετικές έρευνες του Σακκόπουλου και των συνεργατών του αναφορικά με την αξιολόγηση και διαχείριση Υπηρεσιών Ιστού (Web Services) με βάση χαρακτηριστικά QoS, η αποτελεσματική διαχείριση και ο συντονισμός δυναμικών, κατανεμημένων υπηρεσιών αποτελούν κρίσιμους παράγοντες για την τελική συμπεριφορά ενός συστήματος [12]. Λαμβάνοντας υπόψη αυτές τις αρχές στη δική μας πειραματική διάταξη του Συγκεντρωτή (Aggregator), εξετάζουμε πώς η αρχιτεκτονική βελτιστοποίηση στο επίπεδο της JVM επηρεάζει άμεσα το τελικό QoS υπό συνθήκες υψηλού φόρτου.

Το Διάγραμμα 3.1 απεικονίζει τη γενική δομή του συστήματος:



Διάγραμμα 3.1 Αρχιτεκτονική τριών επιπέδων του συστήματος αξιολόγησης Διάγραμμα 3.1 Αρχιτεκτονική τριών επιπέδων του συστήματος αξιολόγησης.

Αυτά τα επίπεδα αποτελούνται από:

- Επίπεδο παρουσίασης: Υλοποιείται με τη χρήση της βιβλιοθήκης React.js. Λειτουργεί αποκλειστικά ως πίνακας ελέγχου που ενεργοποιεί δοκιμές φόρτου (stress tests) μέσω ασύγχρονων κλήσεων και οπτικοποιεί τα αποτελέσματα.
- Επίπεδο λογικής: Υλοποιείται με χρήση Java 21 και Spring Boot 3. Διαχειρίζεται την αποστολή αιτημάτων δικτύου και την πολυπλεξία (multiplexing) νημάτων.
- Επίπεδο δεδομένων / προσομοίωσης: Τα απομακρυσμένα API αντικαθίστανται με τοπικά (mock) τελικά σημεία (endpoints) προσομοίωσης, τα οποία επιστρέφουν στατικά μοντέλα δεδομένων (TravelModels).

3.1.3. API-First Σχεδιασμός

Η επικοινωνία μεταξύ του Frontend και του Backend πραγματοποιείται αποκλειστικά μέσω ενός “stateless” RESTful API. Αυτή η αποσύζευξη διασφαλίζει ότι οι χρόνοι απόκρισης που καταγράφονται στο Backend δεν επηρεάζονται καθόλου από τους χρόνους απόδοσης (rendering time) του Document Object Model (DOM) στον περιηγητή του χρήστη, διατηρώντας έτσι την ακεραιότητα των δεικτών απόδοσης.

3.2. Σχεδιασμός Στοιχείων και Ροή Δεδομένων

Η ροή δεδομένων σχεδιάστηκε έτσι ώστε να προσομοιάζει ρεαλιστικές συνθήκες συμφόρησης στο δίκτυο (bottlenecks).

3.2.1. Προσομοίωση Υπηρεσιών (Downstream Simulation)

Για να αποφευχθεί η παρεμβολή πραγματικού θορύβου από το διαδίκτυο, οι κλήσεις προς υπηρεσίες τρίτων αντικαταστάθηκαν με τοπικά τερματικά που εισάγουν μια καθορισμένη καθυστέρηση. Συγκεκριμένα, οι παράμετροι εξομοίωσης των εν λόγω τελικών σημείων συνοψίζονται στον παρακάτω πίνακα:

Πίνακας 3.1 Παράμετροι εξομοίωσης εξωτερικών υπηρεσιών (Mock APIs)

Υπηρεσία (Domain)	Τελικό Σημείο (Endpoint)	Προκαθορισμένη Χρονοκαθυστέρηση	Προσομοιωμένο Φορτίο (Simulated Payload)	Σκοπός
Πτήσεις	GET /flights	500 ms	Στατικό JSON (Εταιρεία, Τιμή)	Προσομοίωση μέσης καθυστέρησης δικτύου
Ξενοδοχεία	GET /hotels	800 ms	Στατικό JSON (Όνομα, Τιμή)	Καθορισμός του μεγάλης καθυστέρησης (Bottleneck)

Ενοικιάσεις	GET /cars	300 ms	Στατικό JSON (Μοντέλο, Τιμή)	Προσομοίωση γρήγορης απόκρισης I/O

Στο περιβάλλον Java 21, η χρήση της συνάρτησης Thread.sleep() σε αυτά τα τελικά σημεία είναι κρίσιμη. Όταν καλείται από ένα εικονικό νήμα (Virtual Thread), δεν εμποδίζει το υποκείμενο νήμα του λειτουργικού συστήματος, αλλά σηματοδοτεί στην εικονική μηχανή Java (JVM) να «παραχωρήσει» την εκτέλεση. Αυτό καθιστά την προσομοίωση μια ακριβή αναπαράσταση των πραγματικών καθυστερήσεων ασύγχρονης εισόδου/εξόδου (I/O).

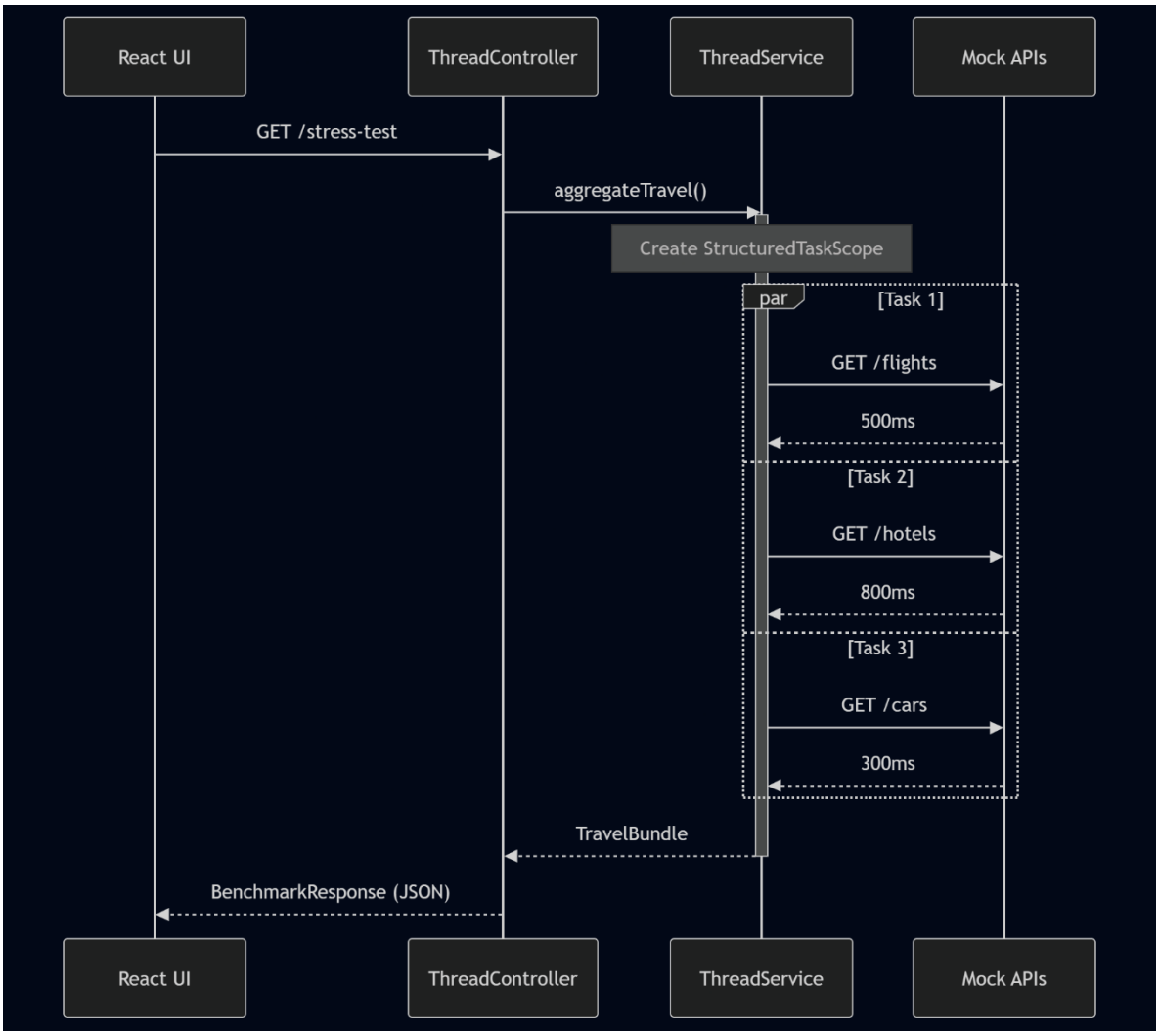
3.2.2. Φράγματα Συγχρονισμού και Διαχείριση Σφαλμάτων

Το σύστημα ενσωματώνει λογική Fan-In, όπου οι μεμονωμένες απαντήσεις συγκεντρώνονται σε ένα ενιαίο αντικείμενο TravelBundle. Ο συνολικός χρόνος απόκρισης του αιτήματος καθορίζεται από τον πιο αργό κρίκο της αλυσίδας (στην περίπτωση αυτή, την αναζήτηση ξενοδοχείου διάρκειας 800 ms).

Στην υλοποίηση των Virtual Threads, ο συγχρονισμός γίνεται μέσω του StructuredTaskScope.ShutdownOnFailure(). Αυτή η δομή ορίζει ένα αυστηρό φράγμα εκτέλεσης. Αν κάποια από τις τρεις υπο-εργασίες/κλήσεις αποτύχει, οι υπόλοιπες εργασίες που βρίσκονται σε εξέλιξη ακυρώνονται αυτόματα (βραχυκύκλωμα). Αυτή η εγγενής συμπεριφορά αποτρέπει τις διαρροές νημάτων που συνήθως συνδέονται με το παλαιότερο μοντέλο CompletableFuture.

3.2.3. Ανάλυση Ακολουθίας

Το παρακάτω διάγραμμα ακολουθίας απεικονίζει τη ροή επικοινωνίας για την επιτυχή δημιουργία ενός συγκεντρωτικού ταξιδιωτικού πακέτου:



Διάγραμμα 3.2 Ροή εκτέλεσης (Sequence Diagram) αιτήματος συντονισμού μέσω StructuredTaskScope.

Όπως φαίνεται στο Σχήμα 3.2, η ροή εκτέλεσης διέπεται από το StructuredTaskScope, το οποίο λειτουργεί ως αυστηρό φράγμα συγχρονισμού. Όταν το περιβάλλον εργασίας χρήστη React στέλνει ένα αίτημα GET για να ενεργοποιήσει το τεστ απόδοσης, το ThreadController αναθέτει την εκτέλεση στο ThreadService. Αντί να εκτελεί τις κατάντη κλήσεις HTTP διαδοχικά, κάτι που θα είχε ως αποτέλεσμα μια σωρευτική καθυστέρηση 1600ms (500ms + 800ms + 300ms), η υπηρεσία διακλαδώνει τρεις παράλληλες υπο-εργασίες. Στη συνέχεια, το σύστημα συναντά την εντολή scope.join(). Ακριβώς αυτή τη στιγμή, το εικονικό νήμα που εκτελεί τον συγκεντρωτή αναστέλλει την εκτέλεσή του, παραχωρώντας την CPU πίσω στο σύστημα, ενώ ταυτόχρονα περιμένει να επιλυθούν και τα τρία ψεύτικα API. Μόλις ολοκληρωθεί η πιο αργή εργασία (Hotels στα 800 ms), το scope συνεχίζει, δημιουργεί το τελικό TravelBundle και το σειριοποιεί σε μια απόκριση JSON για τον πίνακα ελέγχου τηλεμετρίας του frontend.

3.3. Μικροαρχιτεκτονικό Πλαίσιο Υλικού

Τα χαρακτηριστικά απόδοσης του Project Loom συνδέονται άρρηκτα με την υποκείμενη τοπολογία του υλικού, καθώς ο προγραμματιστής της Java αλληλεπιδρά άμεσα με τους διαθέσιμους επεξεργαστές.

3.3.1. Μικροαρχιτεκτονική Zen+ και Νήματα Φορείς

Το πειραματικό περιβάλλον λειτουργεί με επεξεργαστή AMD Ryzen 5 2600. Η αρχιτεκτονική Zen+ αυτού του επεξεργαστή διαθέτει 6 φυσικούς πυρήνες και 12 λογικούς επεξεργαστές μέσω της τεχνολογίας Simultaneous Multithreading (SMT). Κατά την εκκίνηση της εφαρμογής, το ForkJoinPool που τροφοδοτεί τα Virtual Threads ανιχνεύει την τοπολογία του συστήματος και αρχικοποιεί ακριβώς 12 Carrier Threads. Αυτή η αντιστοίχιση 1:1 μεταξύ λογικών επεξεργαστών και νημάτων του λειτουργικού συστήματος ελαχιστοποιεί την ανάγκη παρέμβασης από τον προγραμματιστή του πυρήνα του λειτουργικού συστήματος.

3.3.2. Ιεραρχία Κρυφής Μνήμης

Ο επεξεργαστής Ryzen 5 2600 διαθέτει ιεραρχία κρυφής μνήμης L3 χωρητικότητας 16 MB. Στο παραδοσιακό μοντέλο των Platform Threads, η συνεχής εναλλαγή περιβάλλοντος (context switching) μεταξύ χιλιάδων νημάτων στον χώρο του πυρήνα προκαλεί επιθετική ρύπανση της κρυφής μνήμης (cache pollution) και συνεχείς εκκαθαρίσεις του Translation Lookaside Buffer (TLB), καθώς το λειτουργικό σύστημα ανακατανέμει τα πλαίσια εκτέλεσης [13]. Αντίθετα, επειδή τα Virtual Threads εναλλάσσονται αποκλειστικά στον χώρο του χρήστη (εντός του Java Heap/Σωρός), η δομή της L3 cache διατηρεί ένα σημαντικά υψηλότερο ποσοστό επιτυχιών (hit rate). Αυτή η προσέγγιση, που λειτουργεί σε αρμονία με το υλικό, μειώνει δραστικά τους κύκλους καθυστέρησης της μνήμης, βελτιστοποιώντας τελικά τη συνολική απόδοση των εντολών ανά κύκλο ρολογιού (IPC) των φυσικών πυρήνων [14].

3.3.3. Ο Ρόλος του Συλλέκτη Απορριμμάτων (Garbage Collector) (Μνήμη Σωρού έναντι Εγγενούς Μνήμης)

Η αρχιτεκτονική μετάβαση από τα νήματα που διαχειρίζεται το λειτουργικό σύστημα στα εικονικά νήματα που διαχειρίζεται η JVM, μεταβάλλει ριζικά τον κύκλο ζωής της μνήμης. Τα νήματα της πλατφόρμας δεσμεύουν τις στοίβες τους (μεγέθους 1 MB) στην εγγενή μνήμη του λειτουργικού συστήματος, πράγμα που σημαίνει ότι η δημιουργία και η καταστροφή τους παρακάμπτουν τον Java Garbage Collector (GC). Τα εικονικά νήματα, ωστόσο, αποθηκεύουν τα πλαίσια της στοίβας τους απευθείας στο Java Heap ως τυπικά αντικείμενα.

Σε ένα σενάριο υψηλής απόδοσης, όπως η ενορχήστρωση χιλιάδων ταυτόχρονων αιτημάτων συγκέντρωσης ταξιδιών, αυτό έχει ως αποτέλεσμα έναν εξαιρετικά υψηλό ρυθμό κατανομής αντικειμένων βραχύβιας διάρκειας. Κατά συνέπεια, η απόδοση του συστήματος εξαρτάται σε μεγάλο βαθμό από την αποτελεσματικότητα του Garbage Collector. Για αυτή την πειραματική διάταξη, χρησιμοποιείται ο προεπιλεγμένος G1GC (Garbage-First Garbage Collector) της Java 21. Ο G1GC είναι εξαιρετικά βελτιστοποιημένος για μηχανές πολλαπλών επεξεργαστών με μεγάλες μνήμες, καθαρίζοντας αποτελεσματικά τον χώρο της νέας γενιάς (young generation) όπου συλλέγονται τα

νεκρά εικονικά νήματα, αποτρέποντας έτσι την εξάντληση του Heap πριν προκαλέσει μια δαπανηρή παύση τύπου «Stop-The-World» [15].

3.4. Διαχείριση Πόρων και Περιορισμοί Δικτύου

Κατά τον σχεδιασμό δοκιμών αντοχής, είναι επιτακτική ανάγκη να απομονωθεί η υπό μελέτη μεταβλητή (ταυτόχρονη εκτέλεση νημάτων) από άλλα συστημικά σημεία συμφόρησης.

3.4.1. Όρια της Στοίβας Δικτύωσης του Λειτουργικού Συστήματος

Μια βασική παράμετρος του σχεδιασμού ήταν η εξάντληση των προσωρινών θυρών (socket) TCP/IP. Κάθε εξερχόμενη κλήση HTTP που πραγματοποιείται από το RestClient της Java καταλαμβάνει μια θύρα δικτύου. Σε συνθήκες ακραίου ταυτόχρονου φόρτου (π.χ. >10.000 ταυτόχρονοι χρήστες), το λειτουργικό σύστημα ενδέχεται να εξαντλήσει τις διαθέσιμες θύρες του, με αποτέλεσμα οι υποδοχές να παραμείνουν εγκλωβισμένες σε κατάσταση TIME_WAIT [16]. Ο σχεδιασμός του benchmark λαμβάνει υπόψη αυτό το όριο του υλικού για να διασφαλίσει ότι η συστημική κατάρρευση αποδίδεται σωστά στα όρια του δικτύου και όχι σε βλάβη του ίδιου του μοντέλου threading.

3.4.2. Όρια Κορεσμού της Δεξαμενής Νημάτων(Thread Pool)

Ενώ η στοίβα δικτύου επιβάλλει περιορισμούς στις θύρες, ο κύριος τεχνητός περιορισμός που εισάγεται σε αυτόν τον πειραματικό σχεδιασμό είναι η περιορισμένη ομάδα νημάτων της πλατφόρμας. Για την ακριβή μέτρηση της υποβάθμισης του παλαιού μοντέλου, η εφαρμογή διαμορφώθηκε με σταθερό μέγεθος ομάδας νημάτων, 12 (Executors.newFixedThreadPool(12)). Αυτός ο αριθμός ταιριάζει σκόπιμα με τους λογικούς πυρήνες του Ryzen 5 2600. Σύμφωνα με τον βασικό τύπο του Brian Goetz για τον υπολογισμό του μεγέθους της ομάδας νημάτων σε Java [1], το βέλτιστο μέγεθος μιας ομάδας νημάτων $N_{threads}$ υπολογίζεται ως:

$$N_{threads} = N_{cpu} x U_{cpu} \times \left(1 + \frac{W}{C}\right) \quad (3.1)$$

Όπου:

- $N_{threads}$ είναι ο αριθμός των επεξεργαστών
- U_{cpu} είναι η επιθυμητή χρήση της CPU
- W είναι ο χρόνος αναμονής
- C είναι ο χρόνος υπολογισμού.

Επειδή τα Mock APIs προσομοιώνουν σε μεγάλο βαθμό τους χρόνους αναμονής εισόδου-εξόδου (W) σε σύγκριση με τον χρόνο υπολογισμού (C), ο λόγος $\frac{W}{C}$ είναι εξαιρετικά υψηλός. Περιορίζοντας σκόπιμα την ομάδα σε N_{cpu} (12 νήματα), το σύστημα δημιουργεί ένα σημείο συμφόρησης. Όταν ξεκινά ένα τεστ αντοχής με 100 ταυτόχρονους χρήστες, 12 threads μπλοκάρονται αμέσως περιμένοντας απαντήσεις από το δίκτυο, ενώ οι υπόλοιπες 88 εργασίες αναγκάζονται να μπουν σε ουρά αναμονής. Αυτή η σχεδιαστική επιλογή εγγυάται ότι τυχόν αιχμές καθυστέρησης που

παρατηρούνται στο benchmark είναι το άμεσο μαθηματικό αποτέλεσμα της έλλειψης thread, και όχι της πραγματικής υπολογιστικής εξάντλησης της CPU.

3.5. Πειραματική Μεθοδολογία

Η εξαγωγή εμπειρικών συμπερασμάτων βασίζεται σε μια αυστηρή μεθοδολογία που συνδυάζει μικροοικονομικά κριτήρια αναφοράς και μακροοικονομικές παρατηρήσεις.

3.5.1. Παραγοντικός Πειραματικός Σχεδιασμός

Το πείραμα σχεδιάστηκε με σαφείς μεταβλητές:

- Ανεξάρτητες μεταβλητές: Το μοντέλο διεργασιών (Virtual έναντι Platform) και το επίπεδο φόρτου ταυτόχρονης πρόσβασης (που κυμαινόταν από 100 έως πολλές χιλιάδες ταυτόχρονους χρήστες).
- Εξαρτώμενες μεταβλητές: Μέση καθυστέρηση (χρόνος απόκρισης), ρυθμός επεξεργασίας (λειτουργίες/ms) και χρήση πόρων συστήματος (RAM).

3.5.2. Ρύθμιση JMH (Java Microbenchmark Harness)

Για την επιστημονική επικύρωση των αποτελεσμάτων σε επίπεδο JVM, χρησιμοποιήθηκε το πλαίσιο JMH. Ο σχεδιασμός του τεστ επιδόσεων περιλαμβάνει συγκεκριμένες επαναλήψεις προθέρμανσης. Αυτό διασφαλίζει ότι ο μεταγλωττιστής Just-In-Time (JIT) (C2) διαθέτει αρκετό χρόνο για να μεταγλωττίσει και να βελτιστοποιήσει τον bytecode σε εγγενή κώδικα μηχανής πριν από την έναρξη της πραγματικής φάσης μέτρησης, αποτρέποντας έτσι την καταγραφή τεχνητά διογκωμένων χρόνων εκτέλεσης.

3.5.3. Σχεδιασμός Τηλεμετρίας σε Πραγματικό Χρόνο

Για την παρακολούθηση του συστήματος σε μακροσκοπικό επίπεδο, αναπτύχθηκε το προαναφερθέν frontend React. Το παρακάτω απόσπασμα κώδικα παρουσιάζει την επικοινωνία χωρίς κατάσταση (stateless) με το backend:

```
// Ασύγχρονη κλήση του React Frontend για τη συλλογή τηλεμετρίας
const handleRunBenchmark = async () => {
    setLoading(true);
    try {
        // Ενεργοποίηση της δοκιμής αντοχής (stress test) στα Platform Threads
        const platformRes = await axios.get(
            `http://localhost:8080/api/stress-test/platform?concurrentUsers=${concurrentUsers}`
        );
        // Ενεργοποίηση της δοκιμής αντοχής στα Virtual Threads
        const virtualRes = await axios.get(
            `http://localhost:8080/api/stress-test/virtual?concurrentUsers=${concurrentUsers}`
        );

        // Αποθήκευση της κατάστασης (state) για οπτικοποίηση μέσω της βιβλιοθήκης Recharts
        setBenchmarkResults(prev => [...prev, {
```

```
            users: concurrentUsers,
            Platform: platformRes.data.durationMs,
            Virtual: virtualRes.data.durationMs
        }]);
    } catch (error) {
        console.error("Benchmark failed during execution", error);
    } finally {
        setLoading(false);
    }
};

```

Όπως φαίνεται παραπάνω, η χρήση της σύνταξης async/await σε συνδυασμό με το axios διασφαλίζει ότι το περιβάλλον εργασίας χρήστη παραμένει πλήρως ανταποκρίσιμο κατά τη διάρκεια της δοκιμής φόρτωσης. Περιμένει ασύγχρονα την απάντηση JSON και χρησιμοποιεί τη βιβλιοθήκη recharts για την οπτικοποίηση των δεδομένων σε πραγματικό χρόνο, επιτρέποντας την άμεση παρατήρηση της καθυστέρησης στην ουρά και των σημείων κορεσμού της ομάδας νημάτων.

4. Υλοποίηση του Συστήματος

Στο παρόν κεφάλαιο περιγράφεται λεπτομερώς η πρακτική υλοποίηση του «Microservice Search Aggregator». Το σύστημα backend αναπτύχθηκε με χρήση της Java 21 και του οικοσυστήματος Spring Boot 3, αξιοποιώντας τις πιο πρόσφατες δοκιμαστικές λειτουργίες του Project Loom, με σκοπό να καταστεί δυνατή η άμεση σύγκριση μεταξύ της παραδοσιακής ασύγχρονης προγραμματισμού και του σύγχρονου δομημένου παραλληλισμού.

4.1. Υποδομή και Ρύθμιση Επικοινωνίας Δικτύου

Για να διασφαλιστεί ότι ολόκληρος ο κύκλος ζωής του αιτήματος HTTP εκτελείται αυστηρά στο πλαίσιο του μοντέλου των εικονικών νημάτων, ήταν απαραίτητο να ρυθμιστεί ρητά ο πελάτης HTTP της εφαρμογής. Από προεπιλογή, οι τυπικές βιβλιοθήκες δικτύωσης ενδέχεται να καταφεύγουν στη χρήση νημάτων του λειτουργικού συστήματος, κάτι που θα ακύρωνε το πείραμα.

4.1.1. Διαμόρφωση του RestClient

Στις σύγχρονες αρχιτεκτονικές Spring Boot, το RestClient αποτελεί τη συνιστώμενη σύγχρονη εναλλακτική λύση έναντι του παλαιότερου RestTemplate. Ωστόσο, για να διασφαλιστεί ότι οι λειτουργίες εισόδου-εξόδου δικτύου δεν θα εμποδίζουν τα νήματα της πλατφόρμας, υλοποιήθηκε μια προσαρμοσμένη κλάση διαμόρφωσης (RestClientConfig).

Όπως φαίνεται στο παρακάτω απόσπασμα, το HttpClient συνδέεται ρητά με ένα VirtualThreadPerTaskExecutor. Σύμφωνα με το JEP 444 [5], αυτός ο εκτελεστής δεν συγκεντρώνει νήματα. Αντίθετα, δημιουργεί ένα νέο εικονικό νήμα για κάθε υποβαλλόμενη εργασία, επιτρέποντας στον πελάτη HTTP να κλιμακώνεται απεριόριστα χωρίς να εξαντλεί τους πόρους του λειτουργικού συστήματος.

```
// Σύνδεση του RestClient σε έναν Virtual Thread Executor
package com.example.threadcomparison.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
 6. import org.springframework.http.client.JdkClientHttpRequestFactory;
 7. import org.springframework.web.client.RestClient;
 8. import java.net.http.HttpClient;
 9. import java.util.concurrent.Executors;
10.
11. @Configuration
12. public class RestClientConfig {
13.
14.     @Bean
15.     public RestClient restClient() {
16.         // Εξαναγκασμός του JDK HTTP Client να χρησιμοποιήσει Virtual Threads για το
δίκτυο (I/O)
17.         HttpClient httpClient = HttpClient.newBuilder()
18.                 .executor(Executors.newVirtualThreadPerTaskExecutor())
19.                 .build();
20.
21.         return RestClient.builder()
22.                 .requestFactory(new JdkClientHttpRequestFactory(httpClient))
23.                 .build();
24.     }
25. }
26.
```

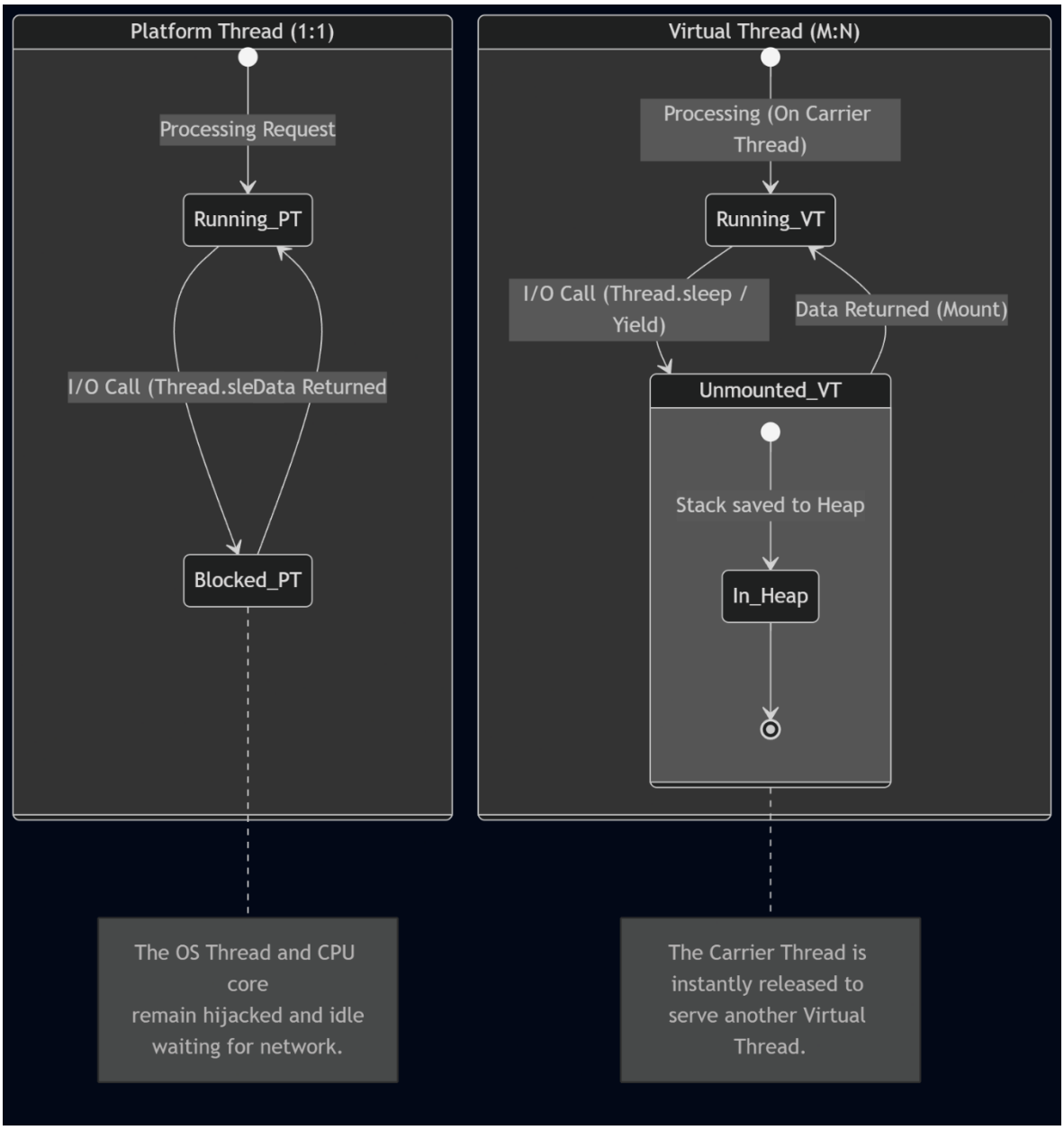
4.1.2. Υλοποίηση Mock API και Μηχανισμοί Κατάστασης

Για την προσομοίωση της καθυστέρησης εξωτερικών υπηρεσιών, υλοποιήθηκαν τρία διαφορετικά τελικά σημεία (endpoints) εντός του ThreadController. Η χρήση της μεθόδου Thread.sleep() σε αυτά τα τελικά σημεία αποτελεί σκόπιμη αρχιτεκτονική επιλογή και όχι απλή τακτική καθυστέρησης.

Ιστορικά, η Thread.sleep() ήταν μια επικίνδυνη λειτουργία σε διακομιστές ιστού με υψηλό βαθμό ταυτόχρονης πρόσβασης, επειδή μπλοκάριζε φυσικά το υποκείμενο νήμα του λειτουργικού συστήματος. Ωστόσο, στο πλαίσιο του Project Loom [5] της Java 21, η Thread.sleep() έχει ξαναγραφτεί εξ ολοκλήρου. Όταν καλείται μέσα σε ένα εικονικό νήμα, λειτουργεί ως μια λειτουργία απόδοσης χωρίς μπλοκάρισμα.

```
 1. // Mock downstream τελικά σημεία που εισάγουν προκαθορισμένη καθυστέρηση I/O
 2. @GetMapping("/flights")
 3. public TravelModels.Flight getFlight() throws InterruptedException {
 4.     Thread.sleep(500);
 5.     return TravelModels.Flight.builder().company("Aegean Airlines").price(150.0).build();
 6. }
 7.
 8. @GetMapping("/hotels")
 9. public TravelModels.Hotel getHotel() throws InterruptedException {
10.     Thread.sleep(800);
11.     return TravelModels.Hotel.builder().name("Grand Hyatt Athens").price(220.0).build();
12. }
13. @GetMapping("/cars")
14. public TravelModels.Car getCar() throws InterruptedException {
15.     Thread.sleep(300);
16.     return TravelModels.Car.builder().name("Hertz").price(45.0).build();
17. }
18.
```

Η μετάβαση κατάστασης που λαμβάνει χώρα κατά τη διάρκεια αυτών των διαστημάτων ύπνου απεικονίζεται στο Σχήμα 4.1.



Διάγραμμα 4.1 Διάγραμμα κατάστασης που συγκρίνει τους κύκλους ζωής των νημάτων κατά τη διάρκεια αναμονής I/O.

Όπως απεικονίζεται στο Διάγραμμα 4.1, το παραδοσιακό μοντέλο νημάτων πλατφόρμας (1:1) πάσχει από μια εγγενώς αναποτελεσματική κατάσταση μπλοκαρίσματος. Όταν καλείται η μέθοδος Thread.sleep() ή μια κλήση δικτυακής εισόδου-εξόδου, το νήμα παραμένει καθηλωμένο στην CPU, καταλαμβάνοντας ουσιαστικά τον υποκείμενο πόρο του λειτουργικού συστήματος χωρίς να εκτελεί κανένα χρήσιμο υπολογισμό. Σε πλήρη αντίθεση, το μοντέλο Εικονικού Νήματος (M:N) χειρίζεται την ίδια ακριβώς λειτουργία μέσω της αποσύνδεσης. Η JVM καταγράφει τα πλαίσια στοίβας του εικονικού νήματος και τα μεταφέρει στο Heap, απελευθερώνοντας αμέσως το νήμα-φορέα. Αυτή η αρχιτεκτονική

αλλαγή επιτρέπει στο νήμα του λειτουργικού συστήματος να στραφεί αμέσως στις επόμενες εργασίες και να επεξεργαστεί άλλες εισερχόμενες αιτήσεις, μεγιστοποιώντας μαθηματικά τη χρήση του υλικού της CPU και αποτρέποντας το σημείο συμφόρησης λόγω έλλειψης πόρων.

4.2. Υλοποίηση Μοντέλων Ταυτόχρονης Εκτέλεσης (Concurrency)

Η βασική επιχειρησιακή λογική βρίσκεται στην κλάση ThreadService, η οποία επιτρέπει μια άμεση σύγκριση, τόσο από συντακτική όσο και από λειτουργική άποψη, μεταξύ της παλαιάς ασύγχρονης προσέγγισης και της σύγχρονης δομημένης παράλληλης εκτέλεσης.

4.2.1. Η Παλαιά Προσέγγιση: Νήματα Πλατφόρμας (Platform Threads)

Η μέθοδος aggregateTravelPlatform επιτυγχάνει παραλληλισμό μέσω του API CompletableFuture. Η εκτέλεση περιορίζεται από μια σταθερή δεξαμενή νημάτων (PLATFORM_POOL) που αρχικοποιείται με ακριβώς 12 νήματα (Executors.newFixedThreadPool(12)), αντικατοπτρίζοντας τους λογικούς πυρήνες του υλικού υποδοχής. Κατά τη διάρκεια υψηλής κυκλοφορίας, αυτή η ομάδα γίνεται το κύριο σημείο συμφόρησης, καθώς οι εισερχόμενες αιτήσεις εξαντλούν γρήγορα τις 12 διαθέσιμες θέσεις, αναγκάζοντας τις επόμενες αιτήσεις να παραμείνουν σε κατάσταση παρατεταμένης ουράς.

```
1. // Παράλληλη ενορχήστρωση με χρήση των παλαιών Platform Threads
2. public TravelBundle aggregateTravelPlatform() {
3.     long start = System.currentTimeMillis();
4.
5.     var flightFuture = CompletableFuture.supplyAsync(() ->
6.
restClient.get().uri("http://localhost:8080/api/flights").retrieve().body(TravelModels.Flight.
class), PLATFORM_POOL);
7.
8.     var hotelFuture = CompletableFuture.supplyAsync(() ->
9.
restClient.get().uri("http://localhost:8080/api/hotels").retrieve().body(TravelModels.Hotel.cl
ass), PLATFORM_POOL);
10.
11.     var carFuture = CompletableFuture.supplyAsync(() ->
12.
restClient.get().uri("http://localhost:8080/api/cars").retrieve().body(TravelModels.Car.class)
, PLATFORM_POOL);
13.
14.     // Φράγμα συγχρονισμού
15.     CompletableFuture.allOf(flightFuture, hotelFuture, carFuture).join();
16.     long end = System.currentTimeMillis();
17.
18.     return new TravelBundle(flightFuture.join(), hotelFuture.join(), carFuture.join(),
(end - start));
19. }
```

Ο αρχιτεκτονικός περιορισμός αυτής της παλαιάς προσέγγισης έγκειται στα αυστηρά όρια του PLATFORM_POOL. Παρόλο που η δομή CompletableFuture.allOf().join() πολυπλέκει (multiplexes) αποτελεσματικά τα τρία ασύγχρονα αποτελέσματα, ο υποκείμενος μηχανισμός εκτέλεσης παραμένει έντονα περιορισμένος. Επειδή τα mock API προσομοιώνουν καθυστερήσεις I/O έως και 800ms, τα νήματα που εκτελούν αυτές τις εργασίες περνούν τη πλειοψηφία του κύκλου ζωής τους σε κατάσταση αναμονής (WAITING). Κατά τη διάρκεια μιας σημαντικής αύξησης της κυκλοφορίας (traffic spike), τα 12 διαθέσιμα νήματα υπόκεινται σε κορεσμό. Κατά συνέπεια, ο

εκτελεστής αναγκάζεται να τοποθετήσει τα επόμενα εισερχόμενα αιτήματα σε ουρά στη μνήμη. Αυτό δημιουργεί μια σοβαρή απόκλιση μεταξύ της θεωρητικής χωρητικότητας δικτύου της εφαρμογής και της πραγματικής της διακίνησης (throughput), επιβεβαιώνοντας μαθηματικά τα σενάρια εξάντλησης νημάτων (thread starvation) που προβλέπονται από τον Νόμο του Little στο Κεφάλαιο 2.

4.2.2. Η Σύγχρονη Προσέγγιση: Δομημένη Ταυτόχρονη Εκτέλεση (Structured Concurrency-Virtual Threads)

Αντίθετα, η μέθοδος aggregateTravelVirtual εγκαταλείπει τις πολύπλοκες αλυσίδες επανακλήσεων (callbacks) του CompletableFuture προς όφελος μιας επιτακτικής (imperative) και ιδιαίτερα ευανάγνωστης λογικής. Αξιοποιεί το StructuredTaskScope.ShutdownOnFailure(), ένα preview feature που επισημοποιήθηκε στο JEP 453 [17].

Αυτή η δομή αντιμετωπίζει πολλαπλές παράλληλες υποεργασίες ως μία ενιαία λογική μονάδα εργασίας. Εάν η υποεργασία /hotels συναντήσει μια εξαίρεση (π.χ. 500 Internal Server Error), το εύρος (scope) ενεργοποιεί αυτόματα ένα σήμα ακύρωσης στις υποεργασίες /flights και /cars. Αυτό το αυτόματο βραχυκύκλωμα (short-circuiting) εγγυάται την υγιεινή των πόρων και αποτρέπει τα ορφανά νήματα (orphan threads) από το να καταναλώνουν εύρος ζώνης για υπολογισμούς που δεν χρειάζονται πλέον.

```
1. // Παράλληλη ενορχήστρωση με χρήση Δομημένης Ταυτόχρονης Εκτέλεσης και Virtual Threads
2. public TravelBundle aggregateTravelVirtual() {
3.     long start = System.currentTimeMillis();
4.     TravelModels.Flight flightResult = null;
5.     TravelModels.Hotel hotelResult = null;
6.     TravelModels.Car carResult = null;
7.
8.     try (var scope = new StructuredTaskScope.ShutdownOnFailure()) {
9.         StructuredTaskScope.Subtask<TravelModels.Flight> flightSubtask = scope.fork(() ->
10.
restClient.get().uri("http://localhost:8080/api/flights").retrieve().body(TravelModels.Flight.
class));
11.
12.         StructuredTaskScope.Subtask<TravelModels.Hotel> hotelSubtask = scope.fork(() ->
13.
restClient.get().uri("http://localhost:8080/api/hotels").retrieve().body(TravelModels.Hotel.cl
ass));
14.
15.         StructuredTaskScope.Subtask<TravelModels.Car> carSubtask = scope.fork(() ->
16.
restClient.get().uri("http://localhost:8080/api/cars").retrieve().body(TravelModels.Car.class)
);
17.
18.         scope.join(); // Φράγμα συγχρονισμού
19.         scope.throwIfFailed(); // Άμεση διάδοση σφαλμάτων
20.
21.         flightResult = flightSubtask.get();
22.         hotelResult = hotelSubtask.get();
23.         carResult = carSubtask.get();
24.     } catch (Exception e) {
25.         throw new RuntimeException("Αποτυχία Virtual Thread κατά τη συγκέντρωση", e);
26.     }
27.     long end = System.currentTimeMillis();
28.     return new TravelBundle(flightResult, hotelResult, carResult, (end - start));
```

```
29. }
```

Από τη σκοπιά της τεχνολογίας λογισμικού, η υλοποίηση στον παραπάνω κώδικα εξαλείφει επιτυχώς τη διχοτομία της «Διχρωμικής Συνάρτησης» (Colored Function) που επικρατεί στα αντιδραστικά (reactive) πλαίσια. Ενθυλακώνοντας (Encapsulating) τις ασύγχρονες κλήσεις δικτύου σε ένα τυπικό μπλοκ try-with-resources, η βάση κώδικα ανακτά τη συντακτική απλότητα της σύγχρονης, μονονηματικής εκτέλεσης. Επιπλέον, η ρητή κλήση των scope.join() και scope.throwIfFailed() εξασφαλίζει έναν ντετερμινιστικό κύκλο ζωής για τα Virtual Threads. Σε αντίθεση με τις αδόμητες αλυσίδες του CompletableFuture όπου μια καθυστερημένη ή αποτυχημένη εργασία παρασκηνίου μπορεί αθόρυβα να επιβιώσει πέραν του αρχικού αιτήματος, προκαλώντας σοβαρές διαρροές μνήμης (Memory Leaks) το StructuredTaskScope εγγυάται τη δομική ακεραιότητα. Όταν το μπλοκ try τερματίζεται, η JVM υποχρεούται αυστηρά να τερματίσει οποιαδήποτε υποεργασία που έχει μείνει, εξασφαλίζοντας βέλτιστη αποδοτικότητα για τον Συλλέκτη Απορριμμάτων (Garbage Collector) και άψογη υγιεινή πόρων.

4.3. Διαμόρφωση Cross-Origin και Ενσωμάτωση Τηλεμετρίας

Για να καταστεί δυνατή η παρατήρηση αυτών των μοντέλων ταυτόχρονης εκτέλεσης σε μακροεπίπεδο και σε πραγματικό χρόνο, το backend πρέπει να διασυνδέεται απρόσκοπτα με τον πίνακα ελέγχου του React στο frontend. Αυτό διευκολύνεται από την κλάση WebConfig, η οποία υλοποιεί το WebMvcConfigurer για τον καθορισμό κανόνων Cross-Origin Resource Sharing (CORS). Επιτρέποντας ρητά αιτήματα από τη διεύθυνση http://localhost:5173, η εφαρμογή Spring Boot επιτρέπει στο React UI να ανακτά αυτόνομα δεδομένα τηλεμετρίας χωρίς να μπλοκάρεται από τις πολιτικές ασφαλείας του προγράμματος περιήγησης. Αυτό εξασφαλίζει την απρόσκοπτη ροή των δεδομένων συγκριτικής αξιολόγησης (benchmark) από τον ThreadController απευθείας στη γραφική αναπαράσταση του frontend. Το οποίο φαίνεται στο παρακάτω απόσπασμα:

```
 1. package com.example.threadcomparison.config;
 2.
 3. import org.springframework.context.annotation.Configuration;
 4. import org.springframework.web.servlet.config.annotation.CorsRegistry;
 5. import org.springframework.web.servlet.config.annotation.WebMvcConfigurer;
 6.
 7. @Configuration
 8. public class WebConfig implements WebMvcConfigurer {
 9.     @Override
10.     public void addCorsMappings(CorsRegistry registry) {
11.         registry.addMapping("/**")
12.                 .allowedOrigins("http://localhost:5173")
13.                 .allowedMethods("GET", "POST", "PUT", "DELETE", "OPTIONS");
14.     }
15. }
```

5. Πειραματική Μεθοδολογία και Διαμόρφωση Αξιολόγησης

Στο παρόν κεφάλαιο περιγράφεται η ακριβής προγραμματιστική μεθοδολογία που χρησιμοποιήθηκε για την αξιολόγηση των μοντέλων ταυτόχρονης εκτέλεσης. Η μέτρηση της απόδοσης της Εικονικής Μηχανής Java (JVM) είναι εγγενώς περίπλοκη λόγω δυναμικών βελτιστοποιήσεων, όπως η μεταγλώττιση Just-In-Time (JIT), η εξάλειψη νεκρού κώδικα και οι ανωμαλίες στη συλλογή απορριμμάτων. Προκειμένου να διασφαλιστεί ότι τα αποτελέσματα είναι στατιστικά αξιόπιστα και δεν

επηρεάζονται από εξωτερικές διακυμάνσεις του δικτύου, το πείραμα διεξήχθη με τη χρήση του Java Microbenchmark Harness (JMH).

5.1. Πειραματικό Περιβάλλον Υλικού και Λογισμικού

Προκειμένου να καθοριστεί ένα αυστηρό σημείο αναφοράς για την αναπαραγωγιμότητα, όλες οι δοκιμές επιδόσεων εκτελέστηκαν σε ένα τοπικό, ελεγχόμενο περιβάλλον. Οι προδιαγραφές του κεντρικού υπολογιστή και του περιβάλλοντος εκτέλεσης αναφέρονται λεπτομερώς στον Πίνακα 5.1.

Πίνακας 5.1 Προδιαγραφές του κεντρικού υπολογιστή του πειράματος.

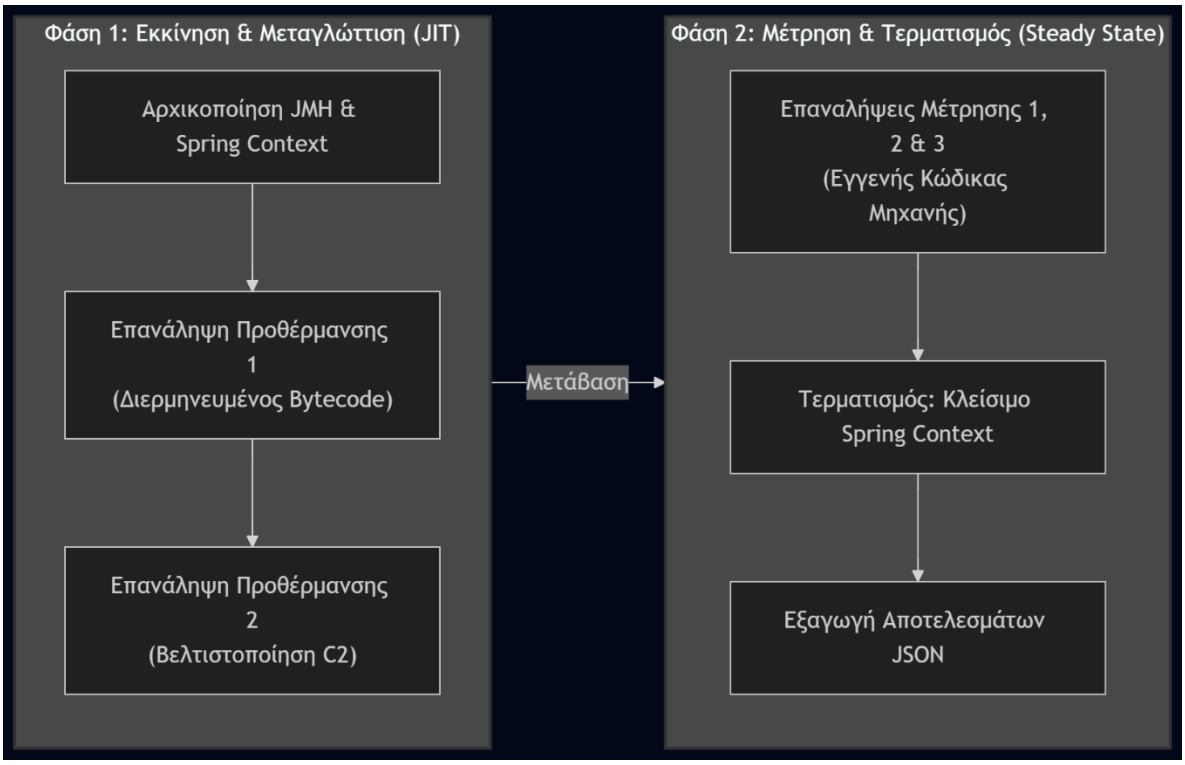
Στοιχεία	Προδιαγραφές
Επεξεργαστής(CPU)	AMD Ryzen 5 2600 (6 Φυσικοί Πυρήνες, 12 Λογικά Νήματα)
Μνήμη συστήματος(RAM)	16 GB
Λειτουργικό Σύστημα	Windows 11 Pro
Java Development Kit (JDK)	Eclipse Temurin OpenJDK 21.0.10+7-LTS
Ορίσματα JVM	--enable-preview (Απαιτείται για το StructuredTaskScope)
Μέθοδος Αξιολόγησης	JMH (Java Microbenchmark Harness) v1.37 [14]

5.2. Στατιστικά Αυστηρή Συγκριτική Αξιολόγηση (JMH)

Οι πρώτες αξιολογήσεις της απόδοσης της Java βασίζονταν συχνά σε απλοϊκές τεχνικές μέτρησης, όπως η καταγραφή χρονικών σημάνσεων πριν και μετά την εκτέλεση μιας μεθόδου. Ωστόσο, όπως απέδειξαν οι Georges et al. [13], τέτοιες μεθοδολογίες είναι στατιστικά άκυρες, καθώς καταγράφουν τη φάση «προθέρμανσης» της JVM και όχι την απόδοσή της σε κατάσταση σταθερής λειτουργίας.

Όταν μια εφαρμογή Java εκκινείται, εκτελείται σε κατάσταση διερμηνείας. Με την πάροδο του χρόνου, η JVM αναγνωρίζει διαδρομές κώδικα που εκτελούνται συχνά («hot spots») και τις μεταβιβάζει στον μεταγλωττιστή C2 JIT, ο οποίος μεταφράζει τον bytecode σε εξαιρετικά βελτιστοποιημένες εγγενείς εντολές μηχανής. Για να αποφευχθεί η αλλοίωση των μετρήσεων συγκέντρωσης διαδρομών από αυτές τις αρχικές καθυστερήσεις μεταγλώττισης, χρησιμοποιήθηκε το JMH για να επιβληθεί αυστηρός διαχωρισμός μεταξύ της φάσης προθέρμανσης και της φάσης μέτρησης [18].

Ο κύκλος ζωής της εκτέλεσης του benchmark απεικονίζεται στο Διάγραμμα 5.1.



Διάγραμμα 5.1 Η διοχέτευση εκτέλεσης του JMH, που απομονώνει τη μεταγλώττιση JIT (Προθέρμανση) από τη στατιστική μέτρηση.

Όπως απεικονίζεται στο Διάγραμμα 5.1, η διοχέτευση (pipeline) εκτέλεσης του JMH διαχωρίζεται αυστηρά σε δύο στάδια, ώστε να απομονωθεί η επιβάρυνση (overhead) προθέρμανσης της JVM από την πραγματική τηλεμετρία. Στη Φάση 1, το πλαίσιο (context) της εφαρμογής Spring αρχικοποιείται και η πρώτη εκτέλεση των μεθόδων της αξιολόγησης πραγματοποιείται σε λειτουργία διερμηνείας (interpreted mode). Κατά τη διάρκεια αυτών των επαναλήψεων προθέρμανσης, ο μεταγλωττιστής C2 JIT της JVM εντοπίζει τα hot spots της εφαρμογής και μεταγλωττίζει τον bytecode σε εξαιρετικά βελτιστοποιημένο εγγενή κώδικα μηχανής. Μόλις αυτή η δυναμική μεταγλώττιση σταθεροποιηθεί, ξεκινά η Φάση 2. Αυτή η φάση μέτρησης μόνιμης κατάστασης (steady-state) καταγράφει τις πραγματικές μετρικές απόδοσης (καθυστέρηση και διακίνηση) χρησιμοποιώντας αποκλειστικά τις βελτιστοποιημένες εγγενείς εντολές, πλήρως απαλλαγμένες από την καθυστέρηση αρχικοποίησης του Spring. Τέλος, το πλαίσιο τερματίζει με ασφάλεια το Spring Context και εξάγει τα στατιστικά αυστηρά δεδομένα προς ανάλυση.

5.3. Υλοποίηση και Ανάλυση Κώδικα Αξιολόγησης

Για να εκτελεστεί η δοκιμή φόρτωσης απευθείας σε επίπεδο JVM, παρακάμπτοντας το επιπλέον φορτίο των δικτυακών υποδοχών HTTP που προκαλούν εξωτερικά εργαλεία όπως το JMeter, το benchmark ενσωματώθηκε απευθείας στον πηγαίο κώδικα μέσω της κλάσης ThreadBenchmark.

Το ακόλουθο απόσπασμα παρουσιάζει τη βασική διαμόρφωση του benchmark με χρήση των επισημειώσεων (Annotations) JMH.

```java
// Διαμόρφωση Benchmark στο JMH και Διαχείριση Κατάστασης
package com.example.threadcomparison;

import org.openjdk.jmh.annotations.*;
import org.springframework.boot.SpringApplication;
import org.springframework.context.ConfigurableApplicationContext;
import java.util.concurrent.TimeUnit;

@BenchmarkMode({Mode.AverageTime, Mode.Throughput})
@OutputTimeUnit(TimeUnit.MILLISECONDS)
@State(Scope.Benchmark)
@Fork(1)
@Warmup(iterations = 2, time = 5)
@Measurement(iterations = 3, time = 5)
public class ThreadBenchmark {

    private ConfigurableApplicationContext context;
    private ThreadService threadService;

    @Setup
    public void setup() {
        this.context = SpringApplication.run(ThreadComparisonApplication.class);
        this.threadService = context.getBean(ThreadService.class);
    }

    @TearDown
    public void tearDown() {
        this.context.close();
    }

    // Ακολουθούν οι μέθοδοι του Benchmark (benchmarkVirtual / benchmarkPlatform)...
}
```

Αυτή η διαμόρφωση επιβάλλει διάφορους αυστηρούς επιστημονικούς ελέγχους στο πείραμα:

- @State(Scope.Benchmark) και @Setup: Αυτές οι σημειώσεις ορίζουν ότι η κατάσταση του benchmark είναι κοινή για όλα τα νήματα. Το περιβάλλον εφαρμογής Spring Boot εκκινείται ακριβώς μία φορά πριν από την έναρξη του benchmark, διασφαλίζοντας ότι οι χρόνοι έγχυσης εξαρτήσεων και αρχικοποίησης του διακομιστή ιστού δεν επηρεάζουν τις μετρήσεις συγκέντρωσης των νημάτων [18].
- @Warmup(iterations = 2, time = 5): Δίνει εντολή στο JMH να εκτελέσει δύο μη καταγεγραμμένες εκρήξεις 5 δευτερολέπτων. Αυτό το παράθυρο 10 δευτερολέπτων αναγκάζει τον μεταγλωττιστή C2 να βελτιστοποιήσει πλήρως τις διαδρομές StructuredTaskScope και CompletableFuture σε εγγενή κώδικα μηχανής.
- @Measurement(iterations = 3, time = 5): Η φάση της πραγματικής συλλογής δεδομένων, που παράγει τις τελικές στατιστικά σημαντικές μετρήσεις.
- @BenchmarkMode({Mode.AverageTime, Mode.Throughput}): Καταγράφει ταυτόχρονα τόσο την καθυστέρηση (ms/op) όσο και την χωρητικότητα (ops/ms) των μεθόδων-στόχων, παρέχοντας μια ολιστική εικόνα της υποβάθμισης του συστήματος.

5.4. Προσομοίωση Ταυτόχρονου Φορτίου και Δημιουργία Προφίλ

Αντί να βασιστούμε σε προσομοιωμένα αιτήματα HTTP, το φορτίο ταυτόχρονης εκτέλεσης δημιουργήθηκε χρησιμοποιώντας την επισήμανση JMH @Threads, απευθείας στις μεθόδους εκτέλεσης του τεστ απόδοσης.

```
// Απόσπασμα 5.2: Επιβολή εκτέλεσης νημάτων υψηλής πυκνότητας μέσω του JMH
@Benchmark
@Threads(100)
public void benchmarkVirtual() {
    threadService.aggregateTravelVirtual();
}

@Benchmark
@Threads(100)
public void benchmarkPlatform() {
    threadService.aggregateTravelPlatform();
}
```

Με την προσθήκη της επισημείωσης @Threads(100) στις μεθόδους, το JMH αναγκάζει τη JVM να δημιουργήσει 100 ταυτόχρονες νηματικές διεργασίες εργασίας που καλούν αδιάκοπα τις υπηρεσίες συγκέντρωσης δεδομένων. Στην περίπτωση του benchmarkPlatform, αυτό υπερφορτώνει αμέσως το PLATFORM_POOL των 12 νημάτων που έχει οριστεί σταθερά, προκαλώντας τεχνητά το σημείο συμφόρησης κορεσμού της ουράς που υπαγορεύεται από τον Νόμο του Little [7].

Τέλος, για να παρατηρηθεί το αποτύπωμα μνήμης του παραδείγματος M:N Virtual Thread, το JMH GCProfiler συνδέθηκε προγραμματιστικά μέσω του OptionsBuilder. Αυτό το profiler συνδέεται με την τηλεμετρία της JVM για να μετρήσει τους ακριβείς ρυθμούς κατανομής αντικειμένων (gc.alloc.rate.norm) στο Java Heap, επιτρέποντας στη μελέτη να επαληθεύσει εάν οι δυναμικές στοίβες Virtual Threads που βασίζονται στο heap επηρεάζουν αρνητικά τον Garbage Collector [19].

5.5. Γραμμή Βάσης Μνήμης JVM και Συλλογής Απορριμμάτων

Ένα τεστ επιδόσεων που μετρά τους ρυθμούς κατανομής νημάτων είναι επιστημονικά ελλιπές αν δεν καθοριστεί η βασική γραμμή αναφοράς για τη διαχείριση της μνήμης. Από προεπιλογή, η Java 21 χρησιμοποιεί τον συλλέκτη απορριμμάτων Garbage-First (G1GC). Ο G1GC είναι ένας συλλέκτης με περιφερειακή και γενεαλογική δομή, σχεδιασμένος να παρέχει υψηλή απόδοση, τηρώντας παράλληλα αυστηρούς στόχους όσον αφορά τον χρόνο παύσης.

Για το συγκεκριμένο πείραμα, επιτράπηκε στη JVM να χρησιμοποιήσει το προεπιλεγμένο εργονομικό μέγεθος του σωρού της, βασισμένο στη διαθέσιμη μνήμη RAM του κεντρικού υπολογιστή. Δεν επιβλήθηκαν χειροκίνητοι περιορισμοί στο Heap (όπως -Xmx ή -Xms), ούτε ενεργοποιήθηκε ο Z Garbage Collector (ZGC). Αυτή η απόφαση ήταν σκόπιμη: διατηρώντας την τυπική διαμόρφωση του G1GC, το benchmark αξιολογεί το μοντέλο Virtual Thread ακριβώς όπως θα συμπεριφερόταν σε ένα τυπικό, έτοιμο προς χρήση περιβάλλον παραγωγής επιχειρήσεων.

5.6. Επίσημος Ορισμός Εξαγόμενων Μετρικών

Για να συγκριθεί με ακρίβεια η διαφορά απόδοσης μεταξύ των Platform Threads και των Virtual Threads, το πλαίσιο JMH ρυθμίστηκε ώστε να εξάγει τρεις διαφορετικές μετρήσεις. Αυτές οι μετρήσεις αποτελούν τη βάση για την ανάλυση δεδομένων στο Κεφάλαιο 6:

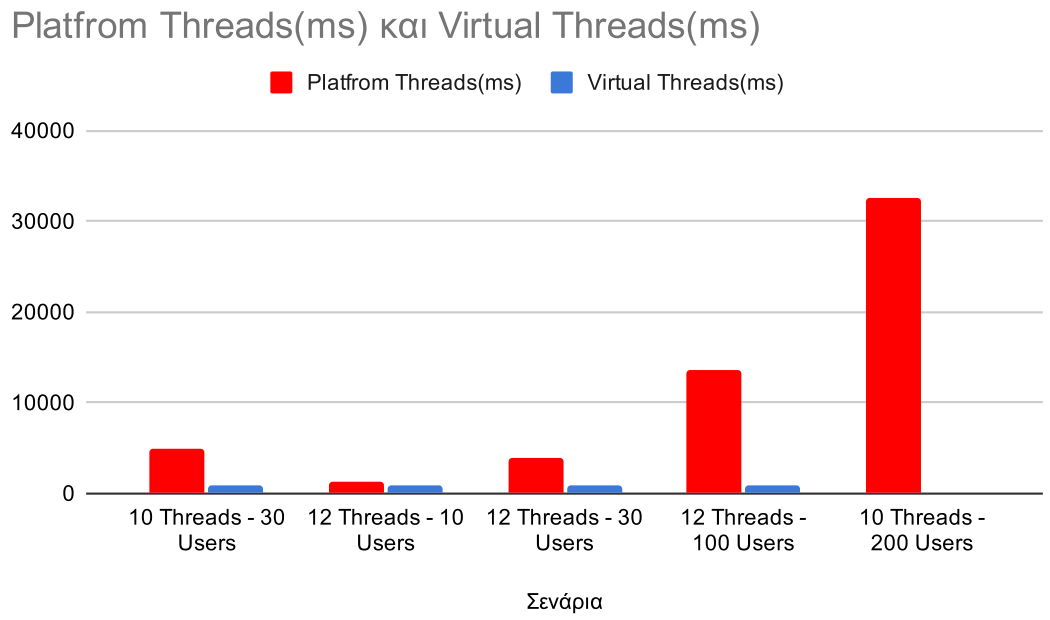
- Μέσος χρόνος (avgt): Μετράται σε χιλιοστά του δευτερολέπτου ανά λειτουργία (ms/op). Αυτή η μέτρηση αντιπροσωπεύει την απόλυτη καθυστέρηση ενός μεμονωμένου αιτήματος HTTP που φτάνει στο τελικό σημείο /stress-test. Υπολογίζει τον συνολικό χρόνο που απαιτείται για ένα νήμα να εκτελέσει τη λογική CompletableFuture ή StructuredTaskScope, να περιμένει τα τρία κατάντη ψεύτικα API και να επιστρέψει το συγκεντρωτικό TravelBundle.
- Διακίνηση (thrpt): Μετράται σε λειτουργίες ανά χιλιοστό του δευτερολέπτου (ops/ms). Αυτή η μέτρηση ορίζει την απόλυτη χωρητικότητα του διακομιστή ιστού. Υπολογίζει πόσα πλήρως συγκεντρωμένα αιτήματα χρηστών μπορεί το σύστημα να ολοκληρώσει με επιτυχία σε ένα μόνο χιλιοστό του δευτερολέπτου υπό συνεχή ταυτόχρονο φορτίο.
- Κανονικοποιημένος ρυθμός κατανομής (gc.alloc.rate.norm): Μετράται σε byte ανά λειτουργία (B/op). Αυτή η δευτερεύουσα μέτρηση, που εξάγεται μέσω του GCProfiler, ποσοτικοποιεί το ακριβές αποτύπωμα μνήμης που απαιτείται για την εκπλήρωση ενός μεμονωμένου αιτήματος. Είναι κρίσιμη για την αξιολόγηση του κατά πόσον τα δυναμικά πλαίσια στοίβας με βάση το heap των εικονικών νημάτων (Virtual Threads) δημιουργούν υπερβολική πίεση στη μνήμη σε σύγκριση με τις εγγενείς στοίβες του λειτουργικού συστήματος των νημάτων πλατφόρμας (Platform Threads).

6. Αποτελέσματα και Ανάλυση Απόδοσης

Το παρόν κεφάλαιο παρουσιάζει τα εμπειρικά δεδομένα που εξήχθησαν από τα τεστ επιδόσεων JMH. Η ανάλυση επικεντρώνεται σε τρεις βασικούς άξονες: την καθυστέρηση (μέσος χρόνος και εκατοστιαία διαστήματα), τη χωρητικότητα του συστήματος (απόδοση) και την αποδοτικότητα της μνήμης της JVM (κατανομή χώρου κατά τη συλλογή απορριμμάτων). Επιπλέον, εξετάζει τα σημεία συστημικής αστοχίας που εντοπίστηκαν όταν η εφαρμογή ωθήθηκε πέρα από τα φυσικά όρια του επιπέδου μεταφοράς.

6.1. Οπτικοποίηση Τηλεμετρίας και Μετρικές Επιτάχυνσης

Για να παρέχεται δυνατότητα παρακολούθησης σε μακροεπίπεδο της κατάστασης εκτέλεσης της JVM, χρησιμοποιήθηκε ο πίνακας ελέγχου του React για την ανάλυση και την οπτικοποίηση των εξόδων JSON του JMH. Όπως απεικονίζεται στο Διάγραμμα 6.1, ο πίνακας ελέγχου υπολογίζει τους σχετικούς πολλαπλασιαστές «Speedup» μεταξύ του παλαιού μοντέλου Platform Thread και της σύγχρονης αρχιτεκτονικής Virtual Thread.



Διάγραμμα 6.1 Απεικόνιση των μετρήσεων σύγκρισης Platform-Virtual σε διαφορετικά σενάρια.

Η οπτικοποίηση επιβεβαιώνει αμέσως τις θεωρητικές υποθέσεις που διατυπώθηκαν στο Κεφάλαιο 2. Όταν ο φόρτος των ταυτόχρονων χρηστών παραμένει χαμηλός (π.χ. 10 χρήστες αντιστοιχισμένοι σε 12 Platform Threads), η επιτάχυνση είναι μέτρια, 1,7 φορές. Ωστόσο, καθώς ο αριθμός των χρηστών υπερβαίνει τη φυσική χωρητικότητα του PLATFORM_POOL (π.χ. 100 χρήστες), το μοντέλο Virtual Thread παρουσιάζει μια τεράστια επιτάχυνση 16,7 φορές, επεξεργάζοντας τη δέσμη σε 814 ms σε σύγκριση με τα 13.617 ms του παλαιού μοντέλου.

6.2. Ανάλυση Καθυστέρησης και Εκατοστημορίων (Tail-at-Scale)

Ενώ η μέση καθυστέρηση (avgt) παρέχει μια γενική εικόνα της απόδοσης, τα σύγχρονα κατανεμημένα συστήματα περιορίζονται ουσιαστικά από την «καθυστέρηση ουράς». Όπως ορίζουν οι Dean και Barroso [20], σε έναν συγκεντρωτή μικροϋπηρεσιών, ένα μόνο αργό αίτημα προς τα κάτω καθορίζει την καθυστέρηση ολόκληρης της απάντησης που λαμβάνει ο χρήστης. Επομένως, η εξέταση του 99ου εκατοστημορίου (p99) είναι κρίσιμη για την κατανόηση της πραγματικής εμπειρίας του χρήστη υπό συνθήκες φόρτου.

Ο Πίνακας 6.1 αναλύει τα εκατοστημόρια καθυστέρησης από τα δεδομένα 12-Thread JMH JSON για 100 ταυτόχρονους χρήστες.

Πίνακας 6.1 Κατανομή Καθυστέρησης για 100 Ταυτόχρονους Χρήστες (12 Thread Pool)

Μετρική	Platform Threads(ms)	Virtual Threads(ms)	Διαφορά
Μέσος Χρόνος(avgt)	13.616,69 ms	813,66 ms	16,7x πιο αργό
Διάμεσος (p50)	13.619,10 ms	813,87 ms	16,7x πιο αργό

99ο Εκατοστημόριο (p99)	13.621,44 ms	814,05 ms	16,7x πιο αργό
Μέγιστο (p100)	13.621,44 ms	814,05 ms	16,7x πιο αργό

$$\frac{13616{,}69ms_{Platform}}{813{,}66ms_{Virtual}} = 16{,}73x \quad (6.1)$$

Επειδή το πιο αργό προσομοιωμένο API κατά τη λήψη δεδομένων (/hotels) επιβάλλει μια σταθερή αναμονή 800 ms, ο θεωρητικός απόλυτος ελάχιστος χρόνος απόκρισης είναι ~800 ms.

Τα δεδομένα αποκαλύπτουν ότι τα εικονικά νήματα λειτουργούν με τη μέγιστη μαθηματική απόδοση. Ακόμη και στο 99ο εκατοστημόριο, ο χρόνος απόκρισης των εικονικών νημάτων είναι 814,05 ms. Η JVM αποσυνδέει επιτυχώς τα εικονικά νήματα κατά τη διάρκεια της αναμονής I/O, αποτρέποντας οποιαδήποτε καθυστέρηση λόγω ουράς.

Αντίθετα, τα Platform Threads αντιμετωπίζουν καταστροφικό κορεσμό της ουράς. Στο 99ο εκατοστημόριο, τα αιτήματα χρειάστηκαν 13,6 δευτερόλεπτα. Επειδή τα 12 διαθέσιμα νήματα του λειτουργικού συστήματος μπλοκάρονται αμέσως από τους χρονομετρητές αναμονής των 800 ms, τα υπόλοιπα 88 αιτήματα παγιδεύονται στην ουρά μνήμης του εκτελεστή, αποδεικνύοντας απόλυτα το σημείο συμφόρησης «Thread Starvation» που προβλέπεται από τον Νόμο του Little [7]

Η δραματική υπεροχή των Virtual Threads σε περιβάλλοντα I/O (όπως οι κλήσεις δικτύου του REST API) επιβεβαιώνεται και από ανεξάρτητες ακαδημαϊκές αξιολογήσεις (Pandita, 2024) [21], οι οποίες αποδεικνύουν ότι τα Virtual Threads εκμηδενίζουν την καθυστέρηση στα I/O-bound workloads, ενώ διατηρούν παρόμοια απόδοση με τα Platform Threads σε CPU-bound εργασίες.

6.3. Διακίνηση (Throughput) και Επιβάρυνση Συλλογής Απορριμμάτων

Η απόδοση (thrpt), που μετράται σε λειτουργίες ανά χιλιοστό του δευτερολέπτου (ops/ms), καθορίζει τον συνολικό όγκο της κίνησης που μπορεί να διαχειριστεί ο διακομιστής. Με 100 χρήστες, η απόδοση των Platform Threads παραμένει σταθερή στα 0,0073 ops/ms, πράγμα που σημαίνει ότι η προσθήκη ταυτόχρονων αιτημάτων αυξάνει αποκλειστικά το μέγεθος της ουράς και όχι τον όγκο της ολοκληρωμένης εργασίας. Τα Virtual Threads διαχειρίστηκαν 0,1227 ops/ms, αντιπροσωπεύοντας μια αύξηση 16,8 φορές της συνολικής χωρητικότητας του διακομιστή.

$$\frac{0{,}1227_{Virtual\ Throughput}}{0.0073_{Platform\ Throughput}} = 16.808x \quad (6.2)$$

Μια θεωρητική ανησυχία σχετικά με το Project Loom είναι το φορτίο που επιβαρύνει τον Garbage Collector (GC). Επειδή τα πλαίσια στοίβας των Virtual Threads αποθηκεύονται δυναμικά στο Java Heap και όχι στη μνήμη του εγγενούς λειτουργικού συστήματος, δημιουργούν υψηλότερα ποσοστά κατανομής αντικειμένων.

Τα δεδομένα του JMH GCProfiler επιβεβαίωσαν αυτή τη συμπεριφορά κατά τη διάρκεια του τεστ επιδόσεων με 100 χρήστες:

- Ρυθμός κατανομής νημάτων πλατφόρμας: 299.195 Bytes/λειτουργία.
- Ρυθμός κατανομής εικονικών νημάτων: 367.529 Bytes/λειτουργία.

Ενώ το μοντέλο εικονικών νημάτων κατανέμει περίπου 22% περισσότερη μνήμη ανά λειτουργία στο heap, οι μετρήσεις gc.time που καταγράφηκαν κατά τη διάρκεια των δοκιμών αποδεικνύουν ότι αυτό το επιπλέον κόστος είναι αμελητέο. Ο συλλέκτης απορριμμάτων Garbage-First (G1GC) της JVM καθαρίζει εύκολα αυτά τα αντικείμενα στοίβας εικονικών νημάτων βραχείας διάρκειας στον χώρο της νέας γενιάς. Τα τεράστια οφέλη απόδοσης υπερτερούν κατά πολύ της μικρής αύξησης στην κατανομή του σωρού, αποδεικνύοντας ότι οι σύγχρονες JVM είναι εξαιρετικά ικανές να υποστηρίζουν το μοντέλο νημάτων M:N χωρίς υποβάθμιση της μνήμης.

6.4. Η Μετατόπιση του Σημείου Συμφόρησης: Εξάντληση Εφήμερων Θυρών

Στη μηχανική συστημάτων, η βελτιστοποίηση ενός επιπέδου μιας αρχιτεκτονικής αναπόφευκτα αποκαλύπτει τους περιορισμούς του επιπέδου που βρίσκεται από κάτω. Ενώ τα Virtual Threads επέδειξαν σχεδόν τέλεια επεκτασιμότητα εντός της JVM, το ταμπλό του React κατέγραψε πλήρη συστημική αστοχία σε υψηλότερα φορτία. Όπως φαίνεται στο Διάγραμμα 6.1, η δοκιμή «10 THREADS - 200 USERS» για τα Virtual Threads κατέγραψε 0 ms, υποδηλώνοντας πλήρη κατάρρευση του συστήματος. Τα αρχεία καταγραφής του backend επιβεβαίωσαν ένα java.net.ConnectException.

Είναι επιτακτική ανάγκη να σημειωθεί ότι αυτό δεν αποτελεί λογικό ελάττωμα λογισμικού, αλλά μάλλον μια θεμελιώδη μετατόπιση του σημείου συμφόρησης του συστήματος από το Επίπεδο Εφαρμογής (JVM) στο Επίπεδο Μεταφοράς (Λειτουργικό Σύστημα).

Κάθε εξερχόμενο αίτημα HTTP που δημιουργείται από την εφαρμογή απαιτεί από το λειτουργικό σύστημα του κεντρικού υπολογιστή να εκχωρήσει μια τοπική θύρα προέλευσης TCP, γνωστή ως Ephemeral Port. Επιπλέον, όταν κλείνει μια σύνδεση TCP, η θύρα δεν απελευθερώνεται αμέσως εισέρχεται σε κατάσταση TIME_WAIT για αρκετά λεπτά, ώστε να διασφαλιστεί η σωστή διαχείριση των καθυστερημένων πακέτων [15].

Το παλαιότερο μοντέλο «Platform Thread» δεν προκάλεσε αυτή την εξαίρεση, καθώς η ομάδα νημάτων λειτουργούσε ως εσωτερικός περιοριστής ρυθμού κλήσεων. Μπορούσε να ανοίγει sockets μόνο με τον ρυθμό που επέτρεπε η ομάδα της, αυτοπεριορίζοντας την εφαρμογή και προστατεύοντας τη στοίβα δικτύου του λειτουργικού συστήματος. Αντίθετα, ο εκτελεστής «Virtual Thread» δεν έχει τέτοιους περιορισμούς. Δημιούργησε αμέσως εκατοντάδες εικονικά νήματα, τα οποία με τη σειρά τους προσπάθησαν να ανοίξουν χιλιάδες ταυτόχρονες συνδέσεις TCP μέσα σε λίγα χιλιοστά του δευτερολέπτου. Τα Virtual Threads ήταν τόσο εξαιρετικά βελτιστοποιημένα που εξάντλησαν επιθετικά ολόκληρο το pool των διαθέσιμων εφήμερων θυρών του κεντρικού υπολογιστή. Αυτό αποδεικνύει ότι, ενώ τα Virtual Threads επιλύουν το πρόβλημα της έλλειψης νημάτων της JVM, τα εταιρικά περιβάλλοντα πρέπει να ρυθμίσουν αναλόγως τις διαμορφώσεις TCP/IP σε επίπεδο λειτουργικού συστήματος για να χειριστούν την τεράστια αύξηση της απόδοσης.

Σε ένα πραγματικό επιχειρησιακό περιβάλλον παραγωγής, αυτό το σημείο συμφόρησης επιλύεται μέσω ενός συνδυασμού ρυθμίσεων στο επίπεδο μεταφοράς και διαχείρισης συνδέσεων στο Επίπεδο Εφαρμογής. Σε επίπεδο Λειτουργικού Συστήματος, οι διαχειριστές δικτύου επεκτείνουν συστηματικά το εύρος των προσωρινών θυρών και διαμορφώνουν τη στοίβα TCP (π.χ. ενεργοποιώντας το tcp_tw_reuse στο Linux) για την ταχεία ανακύκλωση των sockets που έχουν

παγιδευτεί σε κατάσταση TIME_WAIT. Πιο σημαντικό ακόμη, στο επίπεδο εφαρμογών, οι σύγχρονες μικρουπηρεσίες μετριάζουν πλήρως αυτό το πρόβλημα χρησιμοποιώντας την ομαδοποίηση συνδέσεων HTTP (HTTP Keep-Alive). Αντί να ανοίγει και να κλείνει μια νέα φυσική υποδοχή για κάθε κλήση API προς τα κάτω, μια ομάδα συνδέσεων διατηρεί έναν σταθερό αριθμό μόνιμων αγωγών TCP προς τις εξωτερικές υπηρεσίες. Τα εικονικά νήματα μπορούν να δανείζονται και να επιστρέφουν απρόσκοπτα αυτές τις μόνιμες συνδέσεις, πολυπλέκοντας αποτελεσματικά εκατομμύρια λογικές εργασίες σε έναν μικρό, ασφαλή αριθμό φυσικών θυρών δικτύου. Αυτό το αρχιτεκτονικό πρότυπο συνδυάζει υπέροχα την άπειρη επεκτασιμότητα του Project Loom με την ασφάλεια υλικού του παραδοσιακού περιορισμού ρυθμού.

7. Συμπεράσματα και Μελλοντική Έρευνα

7.1. Συμπεράσματα

Ο πρωταρχικός στόχος της παρούσας διατριβής ήταν η εμπειρική αξιολόγηση της αρχιτεκτονικής αλλαγής που εισήγαγε το Project Loom της Java 21, συγκεκριμένα η σύγκριση της παλαιάς αρχιτεκτονικής Platform Thread με το σύγχρονο πρότυπο Virtual Thread (M:N) στο πλαίσιο ενός συγκεντρωτή μικρουπηρεσιών. Η αυστηρή αξιολόγηση επιδόσεων μέσω JMH και η ανάλυση τηλεμετρικών δεδομένων αποδεικνύουν κατηγορηματικά ότι τα Virtual Threads αντιπροσωπεύουν ένα τεράστιο άλμα στην μηχανική του παράλληλου λογισμικού.

Αποσυνδέοντας τον παραλληλισμό σε επίπεδο εφαρμογής από τον φυσικό προγραμματιστή νημάτων του λειτουργικού συστήματος, το μοντέλο Virtual Thread κατάφερε να εξαλείψει το σημείο συμφόρησης σε επίπεδο JVM που ονομάζεται Thread Starvation. Τα εμπειρικά δεδομένα απέδειξαν ότι, ενώ η παλαιά ομάδα 12 νημάτων κατέρρευσε υπό το βάρος 100 ταυτόχρονων χρηστών — με την καθυστέρηση ουράς (p99) να εκτοξεύεται στα 13,6 δευτερόλεπτα λόγω μαζικού κορεσμού της ουράς — τα Virtual Threads διατήρησαν μια μαθηματικά τέλεια καθυστέρηση ~814 ms. Με την απρόσκοπτη αποσύνδεση κατά τη διάρκεια καθυστερήσεων I/O, το μοντέλο M:N παρείχε 16,8 φορές μεγαλύτερη απόδοση διακομιστή χωρίς να απαιτείται καμία αναβάθμιση του υποκείμενου υλικού.

Επιπλέον, η έρευνα αποκάλυψε μια κρίσιμη συστημική πραγματικότητα: η εξάλειψη ενός σημείου συμφόρησης στο επίπεδο του λογισμικού το μετατοπίζει αναπόφευκτα στο επίπεδο του υλικού ή της μεταφοράς. Η εξαιρετική αποδοτικότητα των Virtual Threads τελικά εξάντλησε τις προσωρινές θύρες TCP του κεντρικού υπολογιστή, προκαλώντας ένα java.net.ConnectException υπό τεράστια πίεση. Αυτό αποδεικνύει ότι, ενώ η Java έχει επιλύσει το πρόβλημα της σύγχρονης παράλληλης εκτέλεσης, τα εταιρικά περιβάλλοντα πρέπει να ρυθμίσουν ολιστικά τα λειτουργικά τους συστήματα και να αξιοποιήσουν τη συγκέντρωση συνδέσεων (connection pooling) για να εκμεταλλευτούν με ασφάλεια αυτή την άνευ προηγουμένου απόδοση. Τελικά, ο δομημένος παραλληλισμός και τα Virtual Threads επιτρέπουν στους προγραμματιστές να γράφουν κώδικα υψηλής αναγνωσιμότητας, συγχρονισμένο και επιτακτικό, επιτυγχάνοντας παράλληλα την εξαιρετική επεκτασιμότητα, που προηγουμένως προοριζόταν για σύνθετα και ασύγχρονα αντιδραστικά frameworks.

7.2. Κατευθύνσεις Μελλοντικής Έρευνας

Ενώ η παρούσα μελέτη παρέχει μια οριστική βάση αναφοράς για την απόδοση των εικονικών νημάτων (Virtual Threads) στην ενορχήστρωση REST API που εξαρτάται από τις εισόδους-εξόδους (I/O), η ενσωμάτωση του Project Loom ανοίγει διάφορους δρόμους για μελλοντική έρευνα:

- Συλλογή Απορριμμάτων Εξαιρετικά Χαμηλής Καθυστέρησης (Ultra-Low Latency GC): Η παρούσα μελέτη χρησιμοποίησε τον προεπιλεγμένο G1GC. Σύμφωνα με το JEP 439 της Oracle (Johansson, 2023) [22], η εισαγωγή του Generational Z Garbage Collector (ZGC) στη Java 21 υπόσχεται χρόνους παύσης κάτω του 1 χιλιοστού του δευτερολέπτου. Οι μελλοντικές αξιολογήσεις θα πρέπει να μελετήσουν το αποτύπωμα μνήμης των Virtual Threads χρησιμοποιώντας αποκλειστικά τον ZGC, ειδικά υπό συνθήκες παρατεταμένων, ακραίων φορτίων.
- Συγκέντρωση συνδέσεων βάσης δεδομένων (JDBC/R2DBC): Αυτό το πείραμα επικεντρώθηκε στο I/O δικτύου HTTP. Ένα κρίσιμο επόμενο βήμα είναι η αξιολόγηση του τρόπου με τον οποίο τα Virtual Threads αλληλεπιδρούν με προγράμματα οδήγησης σχεσιακών βάσεων δεδομένων. Η σύγκριση του μπλοκαρισμένου JDBC API που υποστηρίζεται από Virtual Threads με την αντιδραστική προδιαγραφή R2DBC θα παρείχε ζωτικές πληροφορίες για τη σύγχρονη ενορχήστρωση βάσεων δεδομένων. Πρόσφατες ακαδημαϊκές δημοσιεύσεις στο συνέδριο της IEEE (Lasic et al., 2024) [23] υποδεικνύουν ότι η χρήση των Virtual Threads σε διακομιστές που βασίζονται έντονα σε σχεσιακές βάσεις δεδομένων (CRUD operations) αποτελεί το επόμενο μεγάλο πεδίο βελτιστοποίησης, καθιστώντας την έρευνα γύρω από το JDBC εξαιρετικά επίκαιρη.
- Κατανεμημένη Εκτέλεση στο Νέφος (Distributed Cloud Execution): Η μετάβαση του πειράματος από μια τοπική αξιολόγηση JMH σε ένα πλήρως κατανεμημένο περιβάλλον Kubernetes. Όπως επισημαίνουν σύγχρονες έρευνες γύρω από τις cloud-native αρχιτεκτονικές (Bielousov et al., 2023) [24], η αξιολόγηση της JVM εντός εμπορευματοκιβωτίων (containers) παρουσιάζει μοναδικές προκλήσεις. Η εισαγωγή πραγματικής καθυστέρησης δικτύου και ορίων πόρων σε επίπεδο container (cgroups), θα παρείχε μια ολοκληρωμένη άποψη της συμπεριφοράς των Virtual Threads σε μια πραγματική τοπολογία νέφους.

8. Πίνακας ορολογίας

Ελληνικά	Αγγλικά
Εικονικά Νήματα	Virtual Threads
Νήματα Πλατφόρμας / Νήματα Λειτουργικού Συστήματος	Platform Threads / OS Threads
Νήματα Φορείς	Carrier Threads
Δομημένη Ταυτόχρονη Εκτέλεση	Structured Concurrency

Συγκεντρωτής Μικρουπηρεσιών (Αθροιστής)	Microservice Aggregator
Έλλειψη Νημάτων / Εξάντληση Νημάτων	Thread Starvation
Ομάδα Νημάτων / Δεξαμενή Νημάτων	Thread Pool
Αλλαγή Περιβάλλοντος	Context Switch
Διακίνηση / Ρυθμός Διεκπεραίωσης	Throughput
Καθυστέρηση / Χρόνος Απόκρισης	Latency / Response Time
Καθυστέρηση Ουράς	Tail Latency (Tail-at-Scale)
Συλλογή Απορριμμάτων	Garbage Collection
Στοίβα (Μνήμη Στοίβας)	Stack Memory
Σωρός (Μνήμη Σωρού)	Heap Memory
Μπλοκαρισμένη Είσοδος/Έξοδος	Blocking I/O
Δεξαμενή Συνδέσεων	Connection Pool
Λειτουργία Χρήστη / Χώρος Χρήστη	User Mode / User Space
Λειτουργία Πυρήνα / Χώρος Πυρήνα	Kernel Mode / Kernel Space
Σημείο Συμφόρησης	Bottleneck
Μικρουπηρεσίες	Microservices
Καθήλωση Νήματος	Thread Pinning
Υφαρπαγή Εργασίας	Work Stealing
Κλήση Συστήματος / Κλήση Πυρήνα	Syscall
Προγραμματιστής	Scheduler
Εκτός Σωρού	Off-Heap
Πολυπλέκτης	Mutliplexer
Πλαίσια Εκτέλεσης στη Στοίβα	Stack Frames
Κυκλωματική Πολυπλοκότητα	Cyclomatic Complexity
Κόλαση Κλήσεων	Callback Hell
Ρύπανση Κρυφής Μνήμης	Cache Pollution
Βραχυκύκλωμα	Short-Circuiting
Ορφανά Νήματα	Orphan Threads
Διχρωμική Συνάρτηση	Colored Function

Ενθυλακώνοντας	Encapsulating
Εφήμερες Θύρες	Ephemeral Ports

9. Πίνακας συντμήσεων-αρκτικόλεξων-ακρωνυμίων

JVM	Java Virtual Machine
JDK	Java Development Kit
JMH	Java Microbenchmark Harness
JIT	Just-In-Time(Compiler)
API	Application Programming Interface
REST	Representational State Transfer
HTTP	Hypertext Transfer Protocol
TCP/IP	Transmission Control Protocol / Internet Protocol
I/O	Input / Output
OS	Operating System
GC	Garbage Collector
G1GC	Garbage-First Garbage Collector
ZGC	Z Garbage Collector
JSON	JavaScript Object Notation
JDBC	Java Database Connectivity
R2DBC	Reactive Relational Database Connectivity
DOM	Document Object Model
SMT	Simultaneous Multithreading
TLB	Translation Lookaside Buffer
IPC	Instructions per Clock / Cycle
SOA	Service-Oriented Architecture
CORS	Cross-Origin Resource Sharing
TCB	Thread Control Block
FIFO	First-In-First-Out
LIFO	Last-In-First-Out
MTTR	Mean Time To Recovery
CPU	Central Processing Unit
RAM	Random Access Memory

10. Βιβλιογραφία

[1] B. Goetz, T. Peierls, J. Bloch, J. Bowbeer, D. Holmes, and D. Lea, *Java concurrency in practice*. Upper Saddle River, NJ: Addison-Wesley, 2006.

[2] D. Kegel, “The C10K problem.” Accessed: May 19, 2026. [Online]. Available: https://www.kegel.com/c10k.html

[3] R. Pressler, “Loom Proposal.md.” Accessed: May 19, 2026. [Online]. Available: https://cr.openjdk.org/~rpressler/loom/Loom-Proposal.html

[4] E. Alepis and M. Virvou, “Evaluation of the Multimodal Object Oriented Architecture,” in *Object-Oriented User Interfaces for Personalized Mobile Learning*, vol. 64, in Intelligent Systems Reference Library, vol. 64. , Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 101–108. doi: 10.1007/978-3-642-53851-3_9.

[5] R. Pressler and A. Bateman, “JEP 444: Virtual Threads.” Accessed: May 19, 2026. [Online]. Available: https://openjdk.org/jeps/444

[6] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating system concepts*, 10th edition. Hoboken, NJ: Wiley, 2018.

[7] V. Mihalcea, *High-performance Java persistence*. Cluj-Napoca: Vlad Mihalcea, 2016.

[8] J. D. C. Little, “A Proof for the Queuing Formula: $L = \lambda W$,” *Oper. Res.*, vol. 9, no. 3, pp. 383–387, Jun. 1961, doi: 10.1287/opre.9.3.383.

[9] K. Mochniej and M. Badurowicz, “Performance comparison of microservices written using reactive and imperative approaches,” *J. Comput. Sci. Inst.*, vol. 28, pp. 242–247, Sep. 2023, doi: 10.35784/jcsi.3698.

[10] S. Ponnampalam, *Evaluation of Virtual Threads and RxJava : A performance and code complexity analysis in a real-time clearing system*. 2025. Accessed: May 25, 2026. [Online]. Available: https://urn.kb.se/resolve?urn=urn:nbn:se:umu:diva-239964

[11] C. Richardson, *Microservices patterns: with examples in Java*. Shelter Island, NY: Manning Publications, 2019.

[12] Y. Makripoulias, C. Makris, Y. Panagis, E. Sakkopoulos, P. Adamopoulou, and A. Tsakalidis, “Web Service discovery based on Quality of Service,” in *IEEE International Conference on Computer Systems and Applications, 2006.*, IEEE, 2006, pp. 196–199. doi: 10.1109/AICCSA.2006.205089.

[13] R. E. Bryant and D. R. O’Hallaron, *Computer systems: a programmer’s perspective*, Third edition. Boston: Pearson, 2016.

[14] J. L. Hennessy and D. A. Patterson, *Computer architecture: a quantitative approach*, Sixth edition. Cambridge, MA: Morgan Kaufmann Publishers, 2019.

[15] Oracle Corp., “HotSpot Virtual Machine Garbage Collection Tuning Guide(Java Platform, Standard Edition 21).” Accessed: May 20, 2026. [Online]. Available: https://docs.oracle.com/en/java/javase/21/gctuning/hotspot-virtual-machine-garbage-collection-tuning-guide.pdf

[16] W. R. Stevens and G. R. Wright, *TCP/IP illustrated*. in Addison-Wesley professional computing series. Reading, Mass: Addison-Wesley Pub. Co, 1994.

[17] R. Pressler and A. Bateman, “JEP 453: Structured Concurrency (Preview).” Accessed: May 22, 2026. [Online]. Available: https://openjdk.org/jeps/453

[18] A. Shipilëv, *JMH*. (May 21, 2026). Java. OpenJDK. Accessed: May 22, 2026. [Online]. Available: https://github.com/openjdk/jmh

[19] A. Georges, D. Buytaert, and L. Eeckhout, “Statistically rigorous java performance evaluation,” in *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems, languages and applications*, Montreal Quebec Canada: ACM, Oct. 2007, pp. 57–76. doi: 10.1145/1297027.1297033.

[20] J. Dean and L. A. Barroso, “The tail at scale,” *Commun. ACM*, vol. 56, no. 2, pp. 74–80, Feb. 2013, doi: 10.1145/2408776.2408794.

[21] V. Pandita, “Benchmarking the Performance of Java Virtual Threads in High-Throughput Workloads,” masters, Dublin, National College of Ireland, 2025. Accessed: May 25, 2026. [Online]. Available: https://norma.ncirl.ie/8134/

[22] “JEP 439: Generational ZGC.” Accessed: May 25, 2026. [Online]. Available: https://openjdk.org/jeps/439

[23] L. Lasić, D. Beronić, B. Mihaljević, and A. Radovan, “Assessing the Efficiency of Java Virtual Threads in Database-Driven Server Applications,” in *2024 47th MIPRO ICT and Electronics Convention (MIPRO)*, Opatija, Croatia: IEEE, May 2024, pp. 2045–2050. doi: 10.1109/MIPRO60963.2024.10569754.

[24] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, “Performance Evaluation of Microservices Architectures Using Containers,” in *2015 IEEE 14th International Symposium on Network Computing and Applications*, Cambridge, MA, USA: IEEE, Sep. 2015, pp. 27–34. doi: 10.1109/NCA.2015.49.

Παράρτημα Α: Repository Πηγαίου Κώδικα

Ο πλήρης πηγαίος κώδικας για την εφαρμογή, συμπεριλαμβανομένων του Spring Boot backend, μαζί με το JMH benchmark και του React frontend, είναι διαθέσιμος για έλεγχο και ανάλυση στο ακόλουθο δημόσιο GitHub repository:

https://github.com/A-Safelas/virtual-threads-microservice-aggregator

Στο repository είναι επίσης διαθέσιμα τα αποτελέσματα του benchmarking σε μορφή JSON αρχείων.