25-11-2020

ΠΡΟΓΡΑΜΜΑΤΙΣΜΟΣ ΣΥΣΤΗΜΑΤΩΝ

1ο ΠΑΡΑΔΟΤΕΟ

ΔΗΜΗΤΡΑΚΟΠΟΥΛΟΣ ΝΙΚΟΛΑΟΣ 21821

ΒΛΑΣΟΠΟΥΛΟΣ ΜΙΧΑΗΛ 21810

ΑΠΟΣΤΟΛΙΔΗΣ ΘΑΝΟΣ 21807

CONTENTS

ΕΡΩΤΗΜΑ 1

ΤΟ ΠΡΟΒΛΗΜΑ ΤΟΥ TASK OFFLOADING

Η ανάγκη για περισσότερους πόρους υπήρχε και θα υπάρχει πάντα. Με τη πρόοδο της τεχνολογίας καταφέραμε να έχουμε πρόσβαση σε μεγάλη υπολογιστική ισχύ, ενώ ξεπεράσαμε τον περιορισμό και το τεράστιο κόστος, του να πρέπει να αγοράσουμε και να συντηρούμε το infrastructure μας. Ωστόσο, ακόμα και αυτή η λύση έχει το δικό της όριο και χρειάστηκε αλλαγή ή έστω βελτιώσεις, τις οποίες ήρθε να φέρει το Cloud Computing και στη περίπτωση μας, μια τεχνική καλύτερης διαχείρισης πόρων, το task offloading.

Με τον όρο **task offloading** αναφερόμαστε στην μεταφορά ενός ή περισσοτέρων υπολογιστικών διεργασιών από μια μονάδα επεξεργαστή σε μια άλλη, ή σε μία εξωτερική πλατφόρμα ή στο cloud. Η μεταφορά αυτή βασίζεται σε μια αλγοριθμική απόφαση, η οποία παίρνεται λογίζοντας διάφορους περιορισμούς καθώς και προκλήσεις που καλούμαστε να αντιμετωπίσουμε. Ειδικότερα, αναφερόμαστε στους εξής άξονες:

- **Διαχείριση διαθέσιμων πόρων:** Είναι προφανές πως οι πόροι είναι περιορισμένοι από άποψη υλικού-οικονομικής ευχέρειας.
- **Καταμερισμός των διεργασιών:** Όσο περισσότερο εκμεταλλευόμαστε την ιδιότητα του κατεμερισμού και χρησιμοποιούμε όλο το φάσμα επιλογών του, από τερματικό χρήστη μέχρι το cloud, πρέπει να υπολογίσουμε πόσο **κόστος** θα προκύψει από τη **πολυπλοκότητα** του απαραίτητου δικτύου αλλά και πόση θα είναι η χρονική **καθυστέρηση** στην εκπλήρωση της υπηρεσίας. Στόχος στη διαχείριση αυτή αποτελεί **η σταθερότητα, η επεκτασιμότητα, η ασφάλεια και η γεωγραφική κάλυψη** όταν μιλάμε για απομακρυσμένες υπηρεσίες.
- **Απόδοση:** Σε διαχειριστικό επίπεδο, είναι αναγκαίο να παίρνουμε μια **αποδοτική απόφαση για τον καταμερισμό των διεργασιών** (τη σειρά εκτέλεσης, τη προτεραιότητα, τη μεταφορά κλπ) καθώς και σε ποιο μέρος του συστήματος θα πραγματοποιηθεί, τοπικά, σε κάποιο ενδιάμεσο κόμβο ή στο σύννεφο. Με μια αποτελεσματική απόφαση κερδίζουμε σε τομείς όπως η **ταχύτητα εκτέλεσης και μεταφοράς**, καθώς και σε **εξοικονόμηση ενέργειας** που είναι λογικό να φαίνεται σε πιο μακροσκοπικό επίπεδο. Τέλος, είναι σημαντικό να εφαρμόσουμε τις κατάλληλες **τεχνικές μοντελοποίησης** ώστε **να μετρήσουμε ορθά την απόδοση** του καταμερισμού που πραγματοποιήσαμε προσφέροντας έτσι ορθά συμπεράσματα και **σταθερότητα.**

ΤΡΕΙΣ ΜΕΘΟΔΟΙ ΕΠΙΛΥΣΗΣ ΑΠΟ ΤΗΝ ΒΙΒΛΙΟΓΡΑΦΙΑ

CASE STUDY: VEHICULAR NETWORKS

Introduction

Οι τεχνολογικές εξελίξεις των τελευταίων ετών έχουν πυροδοτήσει νέες απαιτήσεις στον τομέα της αυτοκίνησης. Έννοιες όπως **αυτόνομη οδήγηση**, **αυτόματη πλοήγηση** και **φωνητική αλληλεπίδραση μεταξύ οχήματος και οδηγού** δεν βρίσκονται πλέον στη σφαίρα της φαντασίας μας. Χάρη στην πρόοδο στον τομέα των δικτύων και στο υλικό των ηλεκτρονικών υπολογιστών, είμαστε σε θέση πλέον να μπορούμε να εξυπηρετούμε delay-sensitive tasks όπως τα παραπάνω μέσω task offloading. Το paper αυτό μελετάει έναν αποδοτικό αλγόριθμο task offloading ο οποίος φροντίζει να παρέχει και load balancing για αποδοτικότερη διαχείριση πόρων.

The Architecture

Για να μπορέσουμε να έχουμε εγγύηση για το QoS μας, καταφεύγουμε στη λύση του **Mobile Edge Computing (MEC)** και δημιουργούμε **Vehicular Edge Computing Networks (VECNs)**. Η αρχιτεκτονική των δικτύων αυτών μπορεί να χωριστεί σε 4 μέρη:

1. Σε **ad-hoc δίκτυα μεταξύ των οχημάτων (Vehicular ad hoc networks η VANETs)**
2. Σε MEC δίκτυα που αποτελούνται από **Roadside Units (RSUs)** τα οποία τοποθετούνται κατά μήκος του δρόμου και τρέχουν **MEC Servers**. Τα RSUs θα χρησιμοποιούνται για να μπορούν τα οχήματα να κάνουν offload απαιτητικές διεργασίες σε edge servers με ελάχιστη καθυστέρηση.
3. Σε **Fiber-Wireless (FiWi) δίκτυα**, τα οποία με χρήση οπτικών ινών και ταχύτατου ασύρματου δικτύου μπορούν να μεταφέρουν φόρτο από τα RSUs σε ένα Remote Cloud Datacenter χιλιάδες μίλια μακριά, αν αυτό κριθεί απαραίτητο.
4. Σε ένα **Software-Defined Network (SDN)**, το οποίο είναι απαραίτητο για την **κεντροποιημένη διαχείριση του δικτύου και της υποδομής**. Ο load-balancing αλγόριθμος χρειάζεται να έχει συνολική επίγνωση της κατάστασης (φόρτου) της υποδομής μας για να πάρει τις κατάλληλες αποφάσεις, άρα χρειάζεται και πρόσβαση σε κεντροποιημένες πληροφορίες.

System Model

Η έρευνα βασίζεται στην υπόθεση ότι έχουμε έναν δρόμο διπλής κυκλοφορίας, στο οποίο **N** οχήματα κινούνται σε αυτό. Επίσης, στον δρόμο έχουν τοποθετηθεί **M** RSUs σε ισόποσες αποστάσεις ανά δύο μεταξύ τους. Σε κάθε RSU, ένας MEC Server έχει γίνει deployed.

Τα οχήματα μπορούν να επικοινωνήσουν με μια RSU με 2 τρόπους:

- Με **Vehicle-to-Vehicle(V2V)** επικοινωνία, δηλαδή να εκμεταλλευτεί τον ad-hoc χαρακτήρα του δικτύου και να μεταβιβάσει σε άλλο όχημα τα δεδομένα για να μεταδωθεί σε μία RSU εμμέσως.
- Με **Vehicle-to-Infrastructure(V2I)** επικοινωνία, δηλαδή να μεταβιβάσει το όχημα δεδομένα σε μία RSU απευθείας.

Συνεπώς υπάρχουν **M + 2** επιλογές εκτέλεσης task για κάθε όχημα:

- Η 1η επιλογή είναι το task να εκτελεστεί τοπικά στη CPU του οχήματος (optimal αν ειναι είναι intensive το task).
- Η 2η επιλογή είναι το task να εκτελεστεί σε ένα από τα M RSUs.
- Η 3η επιλογή είναι το task να εκτελεστεί στο Remote Cloud Datacenter

Η καθεμία από αυτές τις επιλογές έχει την δικιά της φόρμουλα υπολογισμού χρόνου εκτέλεσης και μετάδοσης. Χωρίς να μπούμε σε μεγάλες μαθηματικές λεπτομέρειες, ας δούμε συνοπτικά ποιοι παράγοντες επηρεάζουν κάθε επιλογή:

- Όταν ένα όχημα αποφασίζει να εκτελέσει ένα task στην CPU του, το processing delay **εξαρτάται μόνο από την υπολογιστική δυνατότητα της CPU**.
- Όταν ένα όχημα αποφασίζει να μεταβιβάσει ένα task σε ένα RSU, τότε το processing delay είναι **ο χρόνος εκτέλεσης στην CPU του RCU (tEXEC) + ο χρόνος μεταφοράς των δεδομένων στο δίκτυο (tTRANS)**. Να σημειωθεί ότι το μέγεθος του tTRANS εξαρτάται σε μεγάλο βαθμό από το αν το όχημα θα επιλέξει V2V η V2I για την επικοινωνία.
- Όταν ένα όχημα αποφασίζει να μεταβιβάσει το task στο Remote Cloud Datacenter τότε το processing delay είναι **ο χρόνος εκτέλεσης στη CPU του Datacenter + ο χρόνος μεταφοράς των δεδομένων μεταξύ οχήματος-RSU μέσω ασύρματου δικτύου + ο χρόνος μεταφοράς των δεδομένων μεταξύ RSU-Datacenter μέσω οπτικών ινών**.

Problem Solution

Το βασικό μας πρόβλημα είναι το εξής: **Θέλουμε να μειώσουμε τον συνολικό χρόνο εκτέλεσης των tasks, βρίσκοντας το κατάλληλο profile εκτέλεσης για καθένα από αυτά**. Οι παραμέτροι μας είναι οι εξής:

- Έχουμε ένα **χρονικό όριο αποδεκτής καθυστέρησης (tMAX)** για κάθε task.
- Σε κάθε RSU μπορεί να συνδεθεί **περιορισμένος αριθμός οχημάτων**.
- Με βάση τους παραπάνω περιορισμούς ο αλγόριθμος επίλυσης του πρέπει να προτείνει:
    - **Ποιο είναι το κατάλληλο μέρος για να γίνει offload το task**.
    - **Ποια μέθοδος επικοινωνίας πρέπει να επιλέξει το όχημα**.

Επίσης ο αλγόριθμος πρέπει να μην καταπονεί τους MEC Servers, δηλαδή να εφαρμόζει load balancing.

Δεν θα δοθεί μεγάλη ανάλυση στο μέρος του αλγορίθμου που βρίσκει το κατάλληλο τρόπο επικοινωνίας. **Συνοπτικά, χρησιμοποποιούνται οι κεντροποιημένες πληροφορίες από το SDN για να**

πάρθούν οι κατάλληλες αποφάσεις. Αφού ξέρουμε το μοντέλο επικοινωνίας, τότε μπορούμε να αρχίσουμε τον υπολογισμό της απόφασης του task offload. Η βηματική προσέγγιση είναι η εξής:

1. **Υπολογίζεται το processing delay για όλα τα tasks με όλους τους πιθανούς τρόπους που μπορεί να γίνει offload.**
2. **Απορρίπτονται από τα αποτελέσματα, αυτά τα οποία το processing delay τους υπερβαίνει το tMAX.**
3. **Αν το task εκτελείται στην τοπική CPU σε χρόνο μικρότερο του tMAX τότε εκτελείται εκεί, αλλιώς υπολογίζουμε βάσει των real-time συνθηκών σε δίκτυο, utilization των MEC Servers και του processing delay της κάθε μεθόδου αν το task θα εκτελεστεί σε MEC Server η στο Remote Cloud Datacenter.**

Τα αποτελέσματα της έρευνας τους έδειξαν ότι ο load-balancing αλγόριθμός τους ήταν αρκετά πιο αποτελεσματικός σε σχέση με παρόμοιους συμβατικούς αλγορίθμους που δεν είχαν load balancing.

J. Zhang, H. Guo, J. Liu and Y. Zhang, "Task Offloading in Vehicular Edge Computing Networks: A Load-Balancing Solution," in IEEE Transactions on Vehicular Technology, vol. 69, no. 2, pp. 2092-2104, Feb. 2020, doi: 10.1109/TVT.2019.2959410.

CASE STUDY: TASK OFFLOADING IN MECS

Το paper αυτό μελετάει τον «άπληστο» αλγόριθμος SMSEF [1] (Select Maximum Saved Energy First) ο οποίος, όπως αναφέρει και το όνομα του, επιλέγει κάθε φορά το offloading του εκάστοτε task ή subtask, αυστηρά με βάση την εξοικονόμηση της περισσότερης δυνατής ενέργειας. Ο συγκεκριμένος αλγόριθμος αφορά τη τεχνική MEC (Mobile edge computing) και ουσιαστικά υλοποιεί το offloading κινητών διεργασιών (με δυνατότητα διάσπασης) σε έναν MEC server. Αρχικά, κάνουμε την παραδοχή πως οι διεργασίες είναι διαιρέσιμες. Στη συνέχεια, συνθέτουμε την άπληστη απόφαση με δεδομένο το συγκεκριμένο μοντέλο [1] και τη μαθηματική φόρμουλα [1] που αναλύεται με βάση αυτό. Υλοποιώντας την απόφαση αυτή κάνουμε κάποιες βασικές δεσμεύσεις.

- Τα δεδομένα που στέλνουμε είναι ίσα με αυτά που υπολογίζονται (transmitted data = computed data)
- Τα δεδομένα πρέπει πρώτα να σταλθούν και μετά να υπολογιστούν
- Κάθε κινητό μπορεί να απασχολεί το πολύ ένα κανάλι ανά time slot (με τον κανάλι εννοούμε ένα κομμάτι του συνολικού φάσματος με εύρος ζώνης (bandwidth) B)

Επιπλέον ο αλγόριθμος μας ακολουθεί τις εξής 2 πολιτικές:

- Η απασχόληση του διαθέσιμου time slot στον MEC server ξεκινά από το deadline, δηλαδή από την αρχή του ήδη δεσμευμένου χώρου, και δεσμεύει χώρο προς τα πίσω.
    - Αυτό επιτυγχάνεται υπολογίζοντας το max latency που χρειάζεται το task/subtask που μας αφορά και κατόπιν διαιρώντας το, ώστε το 1ο διαιρεμένο μέρος να καλύπτει με

καλύτερη ακρίβεια τον διαθέσιμο χώρο. Όσο πιο κοντά στο deadline, τόσο καλύτερη η διαίρεση που σημαίνει λιγότερες πιθανές συγκρούσεις [1]

- Τα υπόλοιπα διαιρεμένα μέλη (subtasks) ακολουθούν τη λογική επαναλαμβανόμενου water filling μέχρι να σταλθούν και να υπολογιστούν όλα 1 προς 1.

Προχωρώντας στο optimization και στον αλγόριθμο:

1. θεωρούμε ως Y_i την στρατηγική Y του mobile (χρήστη) i, ώστε να επιλέξει το κατάλληλο κανάλι και το κατάλληλο occupation slot (σε χρόνο) του MEC server. Επομένως για τη στρατηγική που απασχολεί το κανάλι C σε υπολογιστικό χρόνο S έχουμε το μοντελοποιημένο σύστημα Y = [C,S].
2. Θέτουμε y* τη βέλτιστη λύση, όπου y* = [y*1,…y*N] , την επιλέγουμε (έχουμε επιλέξει δηλαδή κατάλληλα task για offloading, κατάλληλα κανάλια και σε συγκεκριμένα time slot) και τέλος την αφαιρούμε από το σύνολο των Y στοιχείων.
3. Επαναλαμβάνουμε τη διαδικασία για κάθε χρήστη.
4. Θεωρούμε F(i,Y) τη μέγιστη ενέργεια που εξοικονομήθηκε για i tasks στο σύστημα Y. Η πολυπλοκότητα του αλγορίθμου είναι $O(A*N^2)$.

Σημαντική σημείωση: Η λογική αυτή αρχικά παρουσίαζε μικρή διαφορά αποτελεσμάτων αν άλλαζε η αλληλουχία με την οποία γινόντουσαν offload τα tasks. Δηλαδή, Y_{ik} != Y_{ki} διότι κάθε φορά, ο αλγόριθμος λειτουργεί άπληστα υπέρ της πρώτης διεργασίας. Για την επίλυση του προβλήματος, υιοθετήθηκε η λογική της επαναλαμβανόμενης άπληστης απόφασης, δηλαδή κάθε φορά το υποσύστημα που προκύπτει από μια επιλογή, θεωρείται νέο σύστημα ανεπηρέαστο από το προηγούμενο.

Τέλος, για την προσομοίωση και τη μέτρηση απόδοσης, χρησιμοποιούμε άλλους 2 αλγορίθμους: τον Priority-based Algorithm (Prio) και τον Non-divisable SMSEF (N-SMSEF) του οποίου η διαφορά με τον SMSEF είναι πως οι διεργασίες δεν διαιρούνται.

Συμπέρασμα: Υλοποιώντας προσομοιώσεις και συγκρίνοντας τα αντίστοιχα figures, φαίνεται ότι για τυχαίο αριθμό χρηστών και διεργασιών ανά χρήστη, ο αλγόριθμος SMSEF υπερτερεί εμφανώς των άλλων 2 σε ένα MEC σύστημα και η υπεροχή του δεν επηρεάζεται καθόλου από τον αριθμό των χρηστών και των διεργασιών σε κάθε δυνατή περίπτωση.

F. Wei, S. Chen and W. Zou, "A greedy algorithm for task offloading in mobile edge computing system," in *China Communications*, vol. 15, no. 11, pp. 149-157, Nov. 2018, doi: 10.1109/CC.2018.8543056.

CASE STUDY: TASK OFFLOADING IN MOBILE AD-HOC NETWORKS

Το paper αυτό αφορά κυρίως μια τοπολογία δικτύου κινητών συσκευών που μεταξύ τους δημιουργούν ένα Ad-hoc δίκτυο (MANET, mobile ad-hoc network) διευρύνοντας έτσι τις επιλογές του διαμοιρασμού των διεργασιών. Πλέον πέρα από task offloading σε κάποιο cloudlet ή cloud, δεδομένων των συνθηκών που θα αναλυθούν παρακάτω, έχουμε την δυνατότητα να διαμοιράσουμε τις διεργασίες στις κινητές συσκευές που αποτελούν το MANET. Τέλος να τονίσουμε πως στην συγκεκριμένη αντιμετώπιση μεγάλο

ρόλο παίζει το μέσο ασύρματης πρόσβασης στο δίκτυο (WiFi, Bluetooth, 3G, 4G,5G) και πως ο αλγόριθμος της απόφασης έχει σχεδιαστεί γύρω από αυτό.

Αρχικά ο αλγόριθμος δέχεται ως παραμέτρους τα tasks προς εκτέλεση και το Context Monitor. Το Context Monitor αποτελεί μια συλλογή κρίσιμων ως προς την απόφαση παραμέτρων (τον συνολικό αριθμό των instruction που έχει η διεργασία, τη μέση χρήση της CPU, την μνήμη, την μπαταρία, το μέσο πρόσβασης στο δίκτυο, το bandwidth του δικτύου κ.α). Έπειτα μέσω των δοθέντων μαθηματικών σχέσεων υπολογίζονται τα κόστη για τα local execution, MANET execution, cloudlet execution και cloud execution αντίστοιχα. Στη συνέχεια ελέγχεται το μέσο ασύρματης πρόσβασης στο δίκτυο. Στην περίπτωση που η συσκευή έχει ενεργοποιημένο το WiFi, επιλέγεται η μονάδα που της αντιστοιχεί το ελάχιστο κόστος εκτέλεσης που υπολογίσαμε πριν. Στο ενδεχόμενο όπου το WiFi δεν είναι διαθέσιμο αλλά το Bluetooth interface παρέχει πρόσβαση στο Ad hoc δίκτυο τότε επιλέγεται το μικρότερο κόστος εκτέλεσης ανάμεσα σε local execution και MANET execution. Τέλος, για περιπτώσεις όπου υπάρχει πρόσβαση μόνο σε κυψελωτό δίκτυο επιλέγουμε πάλι την εκτέλεση με το μικρότερο κόστος, αυτή τη φορά ανάμεσα σε local execution και cloud execution. Να σημειώσουμε πως σε κάθε από τα παραπάνω ενδεχόμενα, ελέγχεται πάντα η διαθεσιμότητα των υποψήφιων προς task offloading μονάδων.

Κλείνοντας, αξίζει να αναφέρουμε πως η παραπάνω προσέγγιση είναι εύκολα επεκτάσιμη με την άφιξη νέων τεχνολογιών όπως για παράδειγμα ενός νέου interface που θα μπορούσε να αντικαταστήσει το Bluetooth ως μέσο πρόσβασης στο MANET.

B. Zhou, A. V. Dastjerdi, R. N. Calheiros, S. N. Srirama and R. Buyya, "A Context Sensitive Offloading Scheme for Mobile Cloud Computing Service," 2015 IEEE 8th International Conference on Cloud Computing, New York, NY, 2015, pp. 869-876, doi: 10.1109/CLOUD.2015.119.

ΣΥΜΠΕΡΑΣΜΑΤΑ ΚΑΙ ΜΕΛΛΟΝΤΙΚΕΣ ΚΑΤΕΥΘΥΝΣΕΙΣ

Με βάση τις επιστημονικές μελέτες που αναλύσαμε, αλλά παρατηρώντας παράλληλα και τις τεχνολογικές εξελίξεις της πληροφορικής, γίνεται αντιληπτό πως ο ίδιος ο κλάδος καταφέρνει να σχεδιάσει και να προτείνει λύσεις σχετικά με τη βελτιστοποίηση ήδη υπαρχουσών τεχνικών ή ακόμα και την δημιουργία νέων. Όπως φαίνεται, οι λύσεις αυτές απαντούν όλο και περισσότερο και με ποικίλους τρόπους, στα προβλήματα που συναντάμε πολύ συχνά μπροστά μας στο τομέα του cloud computing και συγκεκριμένα του task offloading. Συγκεκριμένα, στις παραπάνω μελέτες, έχουμε προτεινόμενες λύσεις ως προς την **εξοικονόμηση ενέργειας**, την **σημαντική μείωση χρόνου ασφαλούς εκτέλεσης διεργασιών** και τη **καλύτερη διαχείριση αυτών ως προς του πόρους που διαθέτουμε**. Είναι σαφές πως κάθε μία από τις προαναφερθείσες μελέτες αφορούν ένα πιο ειδικό πλαίσιο, πχ συγκεκριμένη τοπολογία δικτύου ή ειδικευμένη εφαρμογή για επικοινωνία ανθρώπου-οχήματος, όμως έχει σημασία να κοιτάξουμε την ευρύτερη εικόνα.

Ο αλγόριθμος εξοικονόμησης ενέργειας διαθέτει ιδιαίτερα μεγάλη επεκτασιμότητα. Ως αυτόνομος μηχανισμός, μπορεί να υιοθετηθεί αυτούσιος ή παραμετροποιημένος για κάθε μελλοντικά παρόμοιο σύστημα σε ένα κόσμο cloud που όσο εξελίσσεται είναι επιτακτική ανάγκη να μειώνει αναλόγως τη κατανάλωση του.

Στην περίπτωση του δικτύου οχημάτων, είναι πολύ σημαντικό να προσπαθήσουμε να εκμηδενίσουμε τα latencies, καθώς οι αποφάσεις που πρέπει να πάρει ένα έξυπνο αυτοκίνητο είναι time-critical και μια καθυστερημένη απάντηση μπορεί να έχει ως αντίκτυπο σοβαρά αυτοκινητικά ατυχήματα.

Στη περίπτωση του MANET, προκύπτει πως μια απλή προσθήκη εξελιγμένου και υποστηριζόμενου από αυτή τη τεχνολογία interface, μπορεί αποδοτικότερα να αντικαταστήσει τη χρήση Bluetooth ως μέσο σύνδεσης σε αυτό.

Οι κύριες ιδέες πίσω από τις προτεινόμενες λύσεις είναι αυτές που τους χαρίζουν μια ιδιαίτερη δυναμική, καθώς μελλοντικές παρεμφερείς τεχνολογίες που θα αντιμετωπίζουν το ίδιο πρόβλημα, μπορούν να τις θέσουν ως βάση και με τη σειρά τους καλούνται να λάβουν και να κινήσουν περαιτέρω τη σκυτάλη.

ΕΡΩΤΗΜΑΤΑ 2-3

Η ΥΠΟΔΟΜΗ

Για να διεξάγουμε τα πειράματα μας με τις διάφορες task offloading μεθόδους, πρώτα χρειάζεται να δημιουργήσουμε μια υποδομή.

Ένα **datacenter** του πραγματικού κόσμου σε πολύ απλουστευμένη μορφή αποτελείται από **hosts** (φυσικά μηχανήματα), τα οποία έχουν κάποιους **φυσικούς πόρους** (Physical CPUs ή PEs στον κόσμου του CloudSim Plus) πάνω στα οποία μπορούμε με χρήση **hypervisor** να τρέξουμε **εικονικές μηχανές** (VMs) , οι οποίες έχουν **εικονικούς πόρους** (Virtual CPUs) οι οποίοι ουσιαστικά είναι φυσικοί πόροι οι οποίοι έχουν δεσμευτεί από τον hypervisor και έχουν τεθεί στη διάθεση των VMs.

Πρέπει να σημειωθεί ότι κατά τη διάρκεια των πειραμάτων **λήφθηκε υπόψιν μόνο η CPU** ως υπολογιστικός πόρος για λόγους απλότητας και αγνοήθηκαν άλλοι σημαντικοί παράγοντες όπως η μνήμη (RAM, Disk I/O), το δίκτυο (Bandwidth, Latency) κτλ.

Σε μία cloud υπηρεσία σημαντικό ρόλο επίσης παίζει ο **πελάτης**. Είναι η οντότητα που καθορίζει:

- **Πόσα resources θέλει**, όπως VMs καθώς το cloud μοντέλο βασίζεται στην on-demand availability (τα VM δεν θα φτιαχτούν πριν γίνει το request από τον πελάτη για αυτά για παράδειγμα).
- **Τι θέλει να κάνει με τους πόρους** που δανείζεται (τι tasks θέλει να τρέξει με αυτά).

Το ρόλο του πελάτη στο CloudSim Plus έχουν οι **brokers**. Ο broker ανήκει σε ένα datacenter και σε αυτόν καταχωρούνται η λίστα με τα VMs που θέλει ο πελάτης και τι tasks επιθυμεί να τρέξουν σε αυτά.

Πηγαίνοντας λίγο πιο βαθιά στον κώδικα του CloudSim Plus, **οι brokers μέσω της συνάρτησης defaultVmMapper(), έχουν επίσης την εξουσία να καθορίσουν σε ποιο VM θα γίνει offload το task** (Cloudlet στην ορολογία του CloudSim Plus).

Για να υλοποιήσουμε την υποδομή μας κατασκευάσαμε μια κλάση CustomSimulation η οποία έχει 2 κύριες αρμοδιότητες (κανονικά προσπαθήσαμε να τηρηθεί το SRP αλλά τα circular dependencies και το tight coupling της βιβλιοθήκης δεν το επέτρεψαν):

- Δημιουργεί ένα datacenter σύμφωνα με τις παραμέτρους που δίνονται στον constructor του (Hosts,VMs,CPUs κτλ)
- Παίρνει ως παράμετρο έναν Broker και μια λίστα με Cloudlets και φροντίσει να δέσει τα VMs που ζητάει ο πελάτης και τα tasks με το simulation. Μετά μπορεί κάποιος με την **startSimulation()** να ξεκινήσει την προσωμοίωση.

Μια χρήσιμη παρατήρηση είναι ότι τα VMs χρησιμοποιούν ως scheduler τον **VmSchedulerTimeShared**.

ΟΙ ΣΥΣΚΕΥΕΣ

Μια συσκευή ορίζεται ως μία οντότητα η οποία παράγει tasks (Cloudlets). Η συμπεριφορά ενός device παρατηρείται βάσει 2 παραγόντων:

- **Από τη συχνότητα που παράγει task στη πάροδο του χρόνου**. Το πείραμα για μονάδα μέτρησης χρόνου έχει το **κβάντο χρόνου**. Το κβάντο χρόνου είναι μια διακριτή μονάδα μέτρησης και παρόλο που δεν μπορεί να μετρηθεί με μια συμβατική καθημερινή μορφή (όπως σε δευτερόλεπτα πχ), μας επιτρέπει να προσωμοιώσουμε διαφορετικές συχνότητες παραγωγής ποσότητας task κατά τη διάρκεια των κβάντων.
- **Από το μήκος των task που παράγει (σε MI).** Οι απαιτήσεις των task ως προς τα resources απεικονίζεται με **Million Instructions**. Το νούμερο αυτό δηλώνει πόσες εκατομμύρια εντολές πρέπει να εκτελεστούν στα VMs του Datacenter για να διεκπαιρωθεί το task.

Για να μοντελοποιήσουμε αυτούς τους 2 παράγοντες χρησιμοποιήθηκαν **2 κανονικές κατανομές** που ορίζουν πόσα task θα παραχθούν σε ένα συγκεριμένο κβάντο και πόσα MI θα έχει το καθένα συγκεκριμένο Cloudlet του κβάντου αυτού:

```
 public List<Cloudlet> generateTasks(int timeQuanta)
{
    double delay = 0.0;
    if(timeQuanta <= 0) throw new IllegalArgumentException("timeQuanta must
be over 0");

    List<Cloudlet> taskList = new ArrayList<>();
    for(int i = 0 ; i < timeQuanta ; i++)
    {
        for(int j = 0; j < (long)taskQuantityRNG.sample(); j++)
        {
            Cloudlet task = new
CloudletSimple((long)millionInstrRNG.sample(),2);
            task.setSubmissionDelay(delay);
            delay += 0.25;
            taskList.add(task);
        }
    }
    return taskList;
}
```

Κάθε task απαιτεί 2 πυρήνες CPU

Για να δοθεί λίγος παραπάνω ρεαλισμός, **κάθε task έχει ένα delay το οποίο σχετίζεται με την χρονική στιγμή στο οποίο κατασκευάστηκε αυτό από το Device**. Επειδή το CloudSim Plus δεν υποστηρίζει πραγματική δυναμική τοποθέτηση cloudlet κατά τη διάρκεια ενός simulation, αυτός είναι ο έμμεσος τρόπος για να του δηλώσουμε ότι δεν είναι κάθε task διαθέσιμο από την αρχή της προσομοίωσης.

ΟΙ BROKERS (+BONUS)

Όπως έχει αναφερθεί και προηγουμένως, οι brokers μπορούν να επηρεάσουν τον τρόπο με τον οποίο θα γίνει offload ένα task σε ένα VM. Για τα πειράματά μας χρησιμοποιήσαμε μια έτοιμη υλοποίηση broker (**DatatacenterBrokerSimple**) και 2 δικές μας (**DatatacenterBrokerRandom** και **DatacenterBrokerLoadBalancer**). Παρακάτω θα εξηγήσουμε τις 3 αυτές υλοποιήσεις:

SIMPLE

Ο Simple αλγόριθμος είναι υλοποιημένος από την ομάδα του CloudSim Plus και χρησιμοποιεί **Round-Robin** και επιλέγει κυκλικά VMs από τη λίστα των Waiting VMs:

```
@Override
protected Vm defaultVmMapper(final Cloudlet cloudlet) {
    if (cloudlet.isBoundToVm()) {
        return cloudlet.getVm();
    }

    if (getVmExecList().isEmpty()) {
        return Vm.NULL;
    }

    /*If the cloudlet isn't bound to a specific VM or the bound VM was not
created,
    cyclically selects the next VM on the list of created VMs.*/
    lastSelectedVmIndex = ++lastSelectedVmIndex % getVmExecList().size();
    return getVmFromCreatedList(lastSelectedVmIndex);
}
```

LOADBALANCER

Ο LoadBalancer αποτελεί **δικός μας αλγόριθμος** και ουσιαστικά φροντίζει να μοιράσει στα VMs το workload. **Επιλέγει ως VM αυτό που τη συγκεκριμένη στιγμή του έχουν ανατεθεί τα tasks με τον λιγότερο αριθμό εντολών.**

```
@Override
protected Vm defaultVmMapper(final Cloudlet cloudlet)
{
    if (cloudlet.isBoundToVm()) {
        return cloudlet.getVm();
    }

    if (getVmExecList().isEmpty()) {
        return Vm.NULL;
    }

    Map<Vm,Double> mipsPerVM = getCloudletSubmittedList()
            .stream()
            .collect(Collectors.groupingBy(
Cloudlet::getVm,
Collectors.summingDouble(Cloudlet::getLength)));

    getVmExecList().forEach(vm -> mipsPerVM.putIfAbsent(vm,0D));
    mipsPerVM.remove(Vm.NULL);

    return mipsPerVM
            .entrySet()
            .stream()
            .min(Comparator.comparingDouble(Map.Entry::getValue))
            .get().getKey();

}
```

RANDOM

Ο Random **είναι επίσης δικός μας αλγόριθμος** ο οποίος επιλέγει «χαζά» VM στη τύχη. Αλγόριθμοι σαν αυτόν σε πειράματα υποδεικνείουν συνήθως τη χειρότερη μέση περίπτωση.

```
@Override
protected Vm defaultVmMapper(Cloudlet cloudlet)
{
    if (cloudlet.isBoundToVm()) {
        return cloudlet.getVm();
    }

    if (getVmExecList().isEmpty()) {
        return Vm.NULL;
    }

    int vmSize = getVmExecList().size();
    return getVmFromCreatedList(rng.nextInt(vmSize));
}
```

Ελπίζουμε οι 2 custom task offloading μηχανισμοί που κατασκευάσαμε να καλύπτουν τις απαιτήσεις σας για το 3^{ο} ερώτημα.

ΤΟ ΠΕΙΡΑΜΑ

ΚΑΠΟΙΕΣ ΒΑΣΙΚΕΣ ΑΡΧΕΣ

Για να κριθεί ένα πείραμα ως έγκυρο, πρέπει να πληρεί κάποιους συγκεριμένους κανόνες. Ένας σημαντικός παράγοντας σύγκρισης μεταξύ των 3 task offloading μεθόδων είναι να δοθεί σε καθεμία από αυτές η **ίδια είσοδος.** Αυτό αποδείχτηκε τελικά πολύ δύσκολο και η ευθύνη πέφτει πάνω στους developers του CloudSim Plus.

ΕΝΑ (ΑΠΟ ΤΑ ΠΟΛΛΑ) ΠΡΟΒΛΗΜΑΤΑ ΤΟΥ CLOUDSIM PLUS

Το CloudletSimple υποτίθεται ότι προσφέρει μία μέθοδο reset() ή οποία θα επανεκκινούσε το Cloudlet στην αρχική του κατάσταση κάθε φορά που θα θέλαμε να ξεκινήσουμε την επόμενη προσομοίωση με άλλο configuration. Αποδείχτηκε μετά από λεπτομερές inspection στον πηγιαίο κώδικα ότι τελικά ότι το reset που κάνουν είναι ελαττωματικό και κρατούνται ακόμα πληροφορίες, τα οποία ο επόμενος broker μπορεί να διακρίνει και να μην ασχοληθεί με την επανεκτέλεση του Cloudlet. Το κερασάκι στην τούρτα ήταν ότι η μέθοδος αυτή είναι χαρακτηρισμένη ως final οπότε δεν επιτρέπεται το override για να αλλαχθεί η συμπεριφορά του χωρίς να σπάσει όλο το συμβόλαιο που υπάρχει.

MOCKITO TO THE RESCUE!

Το dataset του προβλήματος μας εξαιτίας των κατανομών βασίζεται σε ελεγχόμενη τυχαιότητα. Είναι αδύνατον χωρίς τη διατήρηση του cloudlet να μπορέσουμε σε 2 ξεχωριστές εκτελέσεις να έχουμε τα ίδια δεδομένα από τις κατανομές. Θα ήταν αδύνατο δηλαδή αν δεν υπήρχε το Mockito. Το Mockito είναι ένα mocking framework για τα unit tests της Java. Με λίγα λόγια μας επιτρέπει να φτιάξουμε Mocks (ή Test Doubles που ίσως λέει η επιστημονική βιβλιογραφία) τα οποία τους δίνουμε προσδιορισμένη συμπεριφορά στο testing όταν άλλα αντικείμενα αλληλεπιδρούν μεταξύ τους. Έτσι λοιπόν πετυχαίνουμε το κατάλληλο isolation όταν θέλουμε να τεστάρουμε ένα συγκεκριμένο unit στο λογισμικό μας ή στην περίπτωσή μας, να μπορούμε να ελέγχουμε τι επιστρέφουν οι κατανομές όταν τα Devices μας θα ζητούν samples από αυτές.

Η κλάση DeviceMock στη test suite μας ουσιαστικά αφήνει μία φορά τις κατανομές να τρέξουν κανονικά και συλλέγουν τα δεδομένα που επιστρέφουν στα arrays. Μετά κάνουν stub τα νούμερα στις mocked κατανομές τις οποίες χρησιμοποιούν τα devices μας. Όταν τελειώσει κάθε simulation γίνεται ένα reset στα mocks ώστε την επόμενη φορά που θα ξαναζητηθούν δεδομένα από αυτές, να επιστρέψουν πάλι τα ίδια ακριβώς δεδομένα.

ΜΙΑ ΣΗΜΑΝΤΙΚΗ ΠΑΡΑΤΗΡΗΣΗ

Ο κώδικας ο οποίος τρέχει το πείραμα βρίσκεται στα tests της Java λόγω της χρήσης του Mockito framework.

ΠΑΡΑΜΕΤΡΟΙ ΤΟΥ ΠΕΙΡΑΜΑΤΟΣ

Εδώ παρουσιάζουμε συνολικά τις παραμέτρους του δώσαμε κατά τη διάρκεια της διεξαγωγής του πειράματος:

Datacenter:

HOSTS	**4**
HOST_PES	**8**
HOST_PE_MIPS	**50000 (αντίστοιχος πέριπου ενός σημερινού AMD Ryzen Threadripper 3990X)**
VMS	**6**
VM_PES	**2**
VM_PE_MIPS	**50000 (100% της Physical CPU)**

Distributions:

Κατανομή	**Χαμηλή Συχνότητα**		**Μεσαία Συχνότητα**		**Υψηλή Συχνότητα**	
TaskQuantity	**μ= 25**	**σ= 5**	**μ= 50**	**σ= 5**	**μ= 100**	**σ= 10**
MillionInstructions	**μ= 20000**	**σ= 1000**	**μ= 50000**	**σ= 5000**	**μ= 100000**	**σ= 10000**

Time:

Quanta	**3**

ΜΕΤΡΙΚΕΣ ΤΟΥ ΠΕΙΡΑΜΑΤΟΣ

Για να συγκρίνουμε τις επιδόσεις των 3 αλγορίθμων χρησιμοποιήθηκαν οι εξής μετρικές:

- **Simulation Finish Time:** Πόση ώρα χρειάστηκε για να ολοκληρωθούν όλα τα tasks.
- **Average Million Instructions (MI):** Ο μέσος όρος του μεγέθους των task που στάλθηκαν στο datacenter
- **Average Execution Latency:** Ο μέσος χρόνος που χρειάζεται ένα task για να ολοκληρωθεί.
- **MI 90th Percentile:** Τα MI του 90^{ου} πιο ακριβού task.
- **Average Tail Execution Latency:** Ο μέσος χρόνος που χρειάζεται ένα task που ανήκει στο κορυφαίο 10% (από άποψη MI) για να ολοκληρωθεί.

ΑΠΟΤΕΛΕΣΜΑΤΑ ΤΟΥ ΠΕΙΡΑΜΑΤΟΣ

Στην εκτέλεση του πειράματος μας έτυχε να ισχύουν τα παρακάτω δεδομένα, τα οποία είναι κοινά και για τα 3 test instances με τους διαφορετικούς αλγορίθμους:

Total tasks executed:	**537**
Average MI:	**40930.48**
MI 90th Percentile:	**94469.80**

Παρακάτω βλέπουμε πως κάθε αλγόριθμος επηρέασε το latency των tasks:

Simple:

Simulation Finish Time:	**119.10 s**
Average Execution Latency:	**28.46 s**
Average Tail Execution Latency:	**90.66 s**

LoadBalancer:

Simulation Finish Time:	**75.31 s**
Average Execution Latency:	**23.15 s**
Average Tail Execution Latency:	**54.36 s**

Random:

Simulation Finish Time:	**86.28 s**
Average Execution Latency:	**23.63 s**
Average Tail Execution Latency:	**56.20 s**

Τα πλήρη simulation tables βρίσκονται στο φάκελο **experiment**.

ΣΧΟΛΙΑΣΜΟΣ ΑΠΟΤΕΛΕΣΜΑΤΩΝ

Θεωρούσαμε σχετικά αναμενόμενο το ότι **ο LoadBalancer αλγόριθμός μας θα συμπεριφερόταν αρκετά καλύτερα από τον Simple**,ο οποίος απλά αναθέτει κυκλικά τα tasks στα VMs χωρίς να έχει στο νου του τον φόρτο εργασίας τους. Η μεγάλη διαφορά μεταξύ των 2 απεικονίζεται στο Simulation Finish Time αλλά κυρίως στο tail latency, όπου είναι **περίπου κατά 70% μειωμένο**.

Αναπάντεχα ήταν τα αποτελέσματα **του Random αλγορίθμου σε σχέση με το Simple καθώς κατάφερε με απόλυτη τυχαιότητα να είναι αρκετά πιο ανταγωνιστικός από τον Simple**. Αυτό το φαινόμενο συνέβαινε πολύ συχνά κατά την εκτέλεση των test και όχι μόνο στο συγκεκριμένο σενάριο.