ΜΥΕ029 – Προσομοίωση και Μοντελοποίηση Υπολογιστικών Συστημάτων

2η Εργαστηριακή Άσκηση

Προσομοίωση ετερογενών συμπλεγμάτων εξυπηρετητών

Ομάδα
Παναγιώτης Παρασκευόπουλος 2905
Άγγελος Μπεζαΐτης 4432

Περιγραφή Συστήματος και Προσομοιωτή

Στην παρούσα εργασία μελετήσαμε την απόδοση ενός ετερογενούς συμπλέγματος εξυπηρετητών. Η αρχιτεκτονική του συστήματος που μελετήσαμε βασίζεται σε 12 κόμβους:

- **Κόμβος 1 (Dispatcher):** Η λειτουργία αυτού του κόμβου είναι να διαμοιράζει τις εργασίες στους υπόλοιπους κόμβους. Δέχεται αιτήσεις με εκθετική κατανομή και τις προωθεί ακαριαία στους “εργάτες”.

- **Κόμβοι 2-12 (Workers):** Σε αυτούς τους κόμβους έχουμε εξυπηρετητές που αναλαμβάνουν τις εργασίες που προωθεί ο Dispatcher. Το σύμπλεγμα αυτό είναι ετερογενές ως προς τον ρυθμό εξυπηρέτησης και χωρίζεται σε κόμβους από 3 κατηγορίες. Οι πρώτοι κόμβοι 2-6 είναι οι αργοί και έχουν μέσο χρόνο εξυπηρέτησης 12 sec. Οι επόμενοι κόμβοι 7-9 έχουν μέσο χρόνο εξυπηρέτησης 8 sec και τέλος οι γρήγοροι κόμβοι 10-12 έχουν χρόνο εξυπηρέτησης 4 sec.

Πάνω σε αυτό το σύστημα υλοποιήσαμε τον προσομοιωτή μας σε Python με χρήση της βιβλιοθήκης Ciw όπως μας είχε ζητηθεί. Κάθε πείραμα του προσομοιωτή μας δοκιμάζει την λειτουργία του συστήματος για 24 ώρες με μία περίοδο προθέρμανσης (warm-up) διάρκειας 1 ώρας για την αποφυγή της επίδρασης της αρχικής (κενής) κατάστασης. Εφαρμόστηκε και η μέθοδος ανεξάρτητων επαναλήψεων κατά την οποία ο προσομοιωτής εκτελεί επαναλήψεις μέχρι να επιτευχθεί διάστημα εμπιστοσύνης 95% με σχετικό σφάλμα μικρότερο του 5% για τον μέσο χρόνο αναμονής.

Για την εξισορρόπηση του φορτίου χρησιμοποιήθηκαν 2 στρατηγικές με την μορφή των δύο αλγορίθμων που μας δόθηκαν.

- **Αλγόριθμος 1:** Ο αλγόριθμος αυτός επιλέγει τον λιγότερο απασχολούμενο κόμβο χωρίς να λαμβάνει υπόψη την ετερογένεια στον ρυθμό επεξεργασίας αιτήσεων. Με λίγα λόγια αναθέτει την εργασία στον κόμβο που έχει τον μικρότερο φόρτο εργασίας αγνοώντας όμως την ταχύτητα του εξυπηρετητή.
- **Αλγόριθμος 2:** Ο αλγόριθμος 2 επιλέγει και αυτός όπως ο 1 τον λιγότερο απασχολούμενο κόμβο, όμως σε αντίθεση με τον 1 λαμβάνει υπόψη την ετερογένεια στο ρυθμό επεξεργασίας. Αυτό σημαίνει πως στον αλγόριθμο αυτόν υπάρχει μία τιμή d όπου ορίζεται από εμάς και αν ο ρυθμός εργασιών στον γρήγορο κόμβο είναι έως και d παραπάνω από τις αντίστοιχες εργασίες του πρώτου κόμβου στην λίστα τότε ο αλγόριθμος επιστρέφει τον γρήγορο κόμβο. Διαφορετικά επιστρέφει τον πρώτο κόμβο της ταξινομημένης λίστας.

Ανάλυση κώδικα και οδηγίες εκτέλεσης

Ο κώδικας μας έχει οργανωθεί σε δύο αρχεία python.

- **simulation.py:** Περιέχει μία κλάση ClusterSimulation και την υλοποίηση των αλγορίθμων δρομολόγησης. Μπορεί να τρέχει αυτόνομα στην περίπτωση που θελήσουμε να τρέξουμε ένα μεμονωμένο πείραμα. (Δεν παράγει κάποιο αρχείο αποτελέσματος). Για παράδειγμα αν θέλουμε να τρέξουμε τον αλγόριθμο 2 με d = 3 για 24 ώρες θα γράψουμε την εντολή: **python3 simulation.py 86400 2 3**

- **run_experiments.py:** Αυτό είναι πρακτικά ένα script αυτοματοποίησης που εκτελεί την σειρά τον πειραμάτων για τον αλγόριθμο 1 και 2 και αποθηκεύει τα αποτελέσματα σε txt μορφή. Τρέχει με την εντολή: **python3 run_experiments.py**

Σε αυτό το σημείο θα κάνουμε ανάλυση του κώδικα και πως υλοποιήσαμε κάθε απαίτηση του προσομοιωτή μας.

Αρχικά κάνουμε εισαγωγή των απαραίτητων βιβλιοθηκών και ορίζουμε τις βασικές παραμέτρους που θα υλοποιήσουμε. Όπως φαίνεται και παρακάτω.

```
import ciw
import sys
import math
```

```
# --- ΠΑΡΑΜΕΤΡΟΙ ---
D_PARAMETER = 0
WARMUP_TIME = 3600  # 1 ώρα Warm-up
CONFIDENCE_LEVEL = 0.95
DESIRED_REL_ERROR = 0.05

# Πίνακας t-distribution (από Παράρτημα 1 εκφώνησης, στήλη two-tails
0.05)
T_TABLE = {
    1: 12.71, 2: 4.303, 3: 3.182, 4: 2.776, 5: 2.571,
    6: 2.447, 7: 2.365, 8: 2.306, 9: 2.262, 10: 2.228,
    11: 2.201, 12: 2.179, 13: 2.160, 14: 2.145, 15: 2.131,
    16: 2.120, 17: 2.110, 18: 2.101, 19: 2.093, 20: 2.086,
    21: 2.080, 22: 2.074, 23: 2.069, 24: 2.064, 25: 2.060,
    26: 2.056, 27: 2.052, 28: 2.048, 29: 2.045, 30: 2.042,
    40: 2.021, 60: 2.000, 80: 1.990, 100: 1.984, 1000: 1.962
}
```

Το πιό σημαντικό κομμάτι του κώδικα μας είναι η υλοποίηση των αλγορίθμων εξισορρόπησης φορτίου. Αυτό γίνεται με την δημιουργία δύο κλάσεων RoutingDecision1 και RoutingDecision2 μία για τον κάθε αλγόριθμο

```
class RoutingDecision1(ciw.Node):
    def next_node(self, ind):
        # Οι Workers είναι από το index 2 έως 12 (Nodes 2-12)
        # Το index 0 είναι ArrivalNode, το index 1 είναι Dispatcher
(Self)
        workers = [self.simulation.nodes[i] for i in range(2, 13)]

        # Επιλογή του worker με τους λιγότερους πελάτες
        best_worker = min(workers, key=lambda x:
x.number_of_individuals)
        return best_worker
```

Στην κλάση RoutingDesicion1 όπως βλέπουμε από πάνω παίρνουμε ως όρισμα την αρχική κατάσταση της προσομοίωσης και κάνουμε προσπέλαση όλων τον κόμβων αποθηκεύοντας τους σε μία μεταβλητή workers. Στην συνέχεια ελέγχουμε το πλήθος των ενεργών ατόμων για κάθε κόμβο-εργάτη και εντοπίζουμε τον κόμβο με τους λιγότερους πελάτες το οποίο αποθηκεύεται στην μεταβλητή best_worker όπου είναι και το node που στην τελική επιστρέφεται. Σε περίπτωση ισοπαλίας διατηρείται ο κόμβος που βρέθηκε πρώτος δηλαδή αυτός με το μικρότερο id.

```
class RoutingDecision2(ciw.Node):
    def next_node(self, ind):
        # Εύρος 2 έως 13 για να πιάσουμε τους Workers (Nodes 2-12)
        workers = [self.simulation.nodes[i] for i in range(2, 13)]

        # Ταξινόμηση με βάση το πλήθος πελατών
        sorted_workers = sorted(workers, key=lambda x:
x.number_of_individuals)
        best_node = sorted_workers[0]

        # Το Node ID στην Ciw είναι το index στη λίστα nodes.
        # Nodes 10, 11, 12 (Fast) αντιστοιχούν στα indices 10, 11, 12.
        best_node_idx = self.simulation.nodes.index(best_node)

        # Αν ο καλύτερος είναι ήδη Fast (Indices 10-12), τον επιλέγουμε
        if best_node_idx >= 10:
            return best_node
        else:
            # Ψάχνουμε στη λίστα για Fast κόμβο που να ικανοποιεί τη
συνθήκη
            for node in sorted_workers:
                idx = self.simulation.nodes.index(node)
                if idx >= 10: # Είναι Fast (Nodes 10-12)
                    # Συνθήκη: load <= min_load + d
                    if node.number_of_individuals <=
best_node.number_of_individuals + D_PARAMETER:
                        return node
                    else:
                        break
            return best_node
```

Εδώ βλέπουμε την υλοποίηση του αλγορίθμου 2 (κλάση RoutingDecision2). Όπως και ο αλγόριθμος 1 κάνει προσπέλαση όλων των κόμβων και αποθηκεύονται στην μεταβλητή workers. Ταξινομούμε τους κόμβους κατά αύξουσα σειρά φόρτου και αποθηκεύουμε την λίστα στην μεταβλητή sorted_workers. Στην συνέχεια παίρνουμε τον πρώτο κόμβο στην λίστα (best_node) και ελέγχουμε αν το ID του είναι >= 10. Αν ναι τότε απλά επιστρέφουμε το (best_node) καθώς αυτό σημαίνει ότι είναι γρήγορος εργάτης (ID 10-12). Αν όχι σαρώνουμε την λίστα και ελέγχουμε εάν κάποιος από τους γρήγορους κόμβους ικανοποιεί την συνθήκη $N_{fast} <= N_{best} + d$. Αν ναι επιλέγεται ο γρήγορος κόμβος και όχι ο καλύτερος παρόλο που έχει περισσότερο φόρτο για να

εκμεταλλευτούμε τον ταχύτερο ρυθμό εξυπηρέτησης. Αν δεν βρεθεί γρήγορος κόμβος που να εξυπηρετεί την συνθήκη τότε απλά επιστρέφεται το best_node.

Τώρα θα δούμε τον κεντρικό πυρήνα της προσομοίωσης. Με την συνάρτηση run_single_replication υλοποιούμε την εκτέλεση ενός μεμονωμένου και αυτοτελούς πειράματος για να γίνει στατιστική επεξεργασία.

```
def run_single_replication(sim_duration, algorithm, seed):
    print(f"   -> Running replication {seed}...", end='\r', flush=True)
    ciw.seed(seed)

    node_classes = [ciw.Node] * 12
    if algorithm == 1:
        node_classes[0] = RoutingDecision1
    elif algorithm == 2:
        node_classes[0] = RoutingDecision2

    params = get_network_params()
    N = ciw.create_network(**params)
    Q = ciw.Simulation(N, node_class=node_classes)

    Q.simulate_until_max_time(WARMUP_TIME + sim_duration)

    records = Q.get_all_records()

    # --- ΥΠΟΛΟΓΙΣΜΟΣ ΣΤΑΤΙΣΤΙΚΩΝ ---
    valid_records = [r for r in records if r.arrival_date >
WARMUP_TIME]

    # 1. Μέσος Χρόνος Αναμονής
    # Node 1 is Dispatcher, ignore wait time there (should be 0 anyway)
    waiting_times = [r.waiting_time for r in valid_records if r.node >
1]
    mean_wait = sum(waiting_times) / len(waiting_times) if
waiting_times else 0.0

    # 2. Χρησιμοποίηση ανά κόμβο
    node_busy_time = {i: 0.0 for i in range(2, 13)}

    for r in records:
        if r.node == 1: continue

        start = r.service_start_date
```

```
        end = r.service_end_date

        effective_start = max(start, WARMUP_TIME)
        effective_end = min(end, WARMUP_TIME + sim_duration)

        if effective_end > effective_start:
            node_busy_time[r.node] += (effective_end - effective_start)

    utilizations = {i: node_busy_time[i] / sim_duration for i in
range(2, 13)}

    # 3. Ρυθμός εξυπηρέτησης
    completed_in_window = [r for r in records if r.service_end_date >
WARMUP_TIME and r.service_end_date <= WARMUP_TIME + sim_duration]
    throughput = len(completed_in_window) / sim_duration

    return mean_wait, utilizations, throughput
```

Αρχικά ορίζουμε ένα seed με την βοήθεια τις βιβλιοθήκες ciw ώστε κάθε φορά που θα καλεστεί η συνάρτηση να είναι στατιστικά ανεξάρτητη από τις άλλες επαναλήψεις. Δημιουργούμε μία λίστα όπου για τους κόμβους εργάτες κρατάμε την τυπική συμπεριφορά και για τον Dispatcher αντικαθιστούμε την κλάση με μία από τις δικές μας RoutingDesicion1 ή RoutingDesicion2. Στην συνέχεια δημιουργούμε το αντικείμενο της προσομοίωσης με την εντολή ciw.create_network και με παράμετρο ένα λεξικό για την αντιστοίχιση των ρυθμών εξυπηρέτησης. Το λεξικό αυτό δημιουργείται με την συνάρτηση get_network_params() που εμείς ορίσαμε με βάση τον πίνακα που μας δόθηκε.

Πίνακας 1: Σύνθεση και χαρακτηριστικά συμπλέγματος εξυπηρετητών

Αναγνωριστικό κόμβου	Είδος	Μέσος χρόνος μεταξύ αφίξεων	Κατανομή αφίξεων	Μέσος χρόνος ολοκλήρωσης αίτησης	Κατανομή χρόνου εξυπηρέτησης
1	Διανομέας	1 sec	Εκθετική	0	Ντετερμινιστική
2-6	Εργάτης	-	-	12 sec	Εκθετική
7-9	Εργάτης	-	-	8 sec	Εκθετική
10-12	Εργάτης	-	-	4 sec	Εκθετική

Συνεχίζοντας στην ανάλυση της συνάρτησης run_single_replication εκτελούμε την προσομοίωση με συνολικό χρόνο WARMUP_TIME + sim_duration όπως μας ζητήθηκε

```
N = ciw.create_network(**params)
    Q = ciw.Simulation(N, node_class=node_classes)
```

Επόμενο βήμα ήταν ο υπολογισμός του μέσου χρόνου αναμονής. Φιλτράρουμε τις εγγραφές records κρατώντας μόνο όσες εργασίες έφτασαν στο σύστημα μετά το πέρας τους warm-up και κατά τον υπολογισμό του μέσου όρου, ελέγχουμε μόνο τους κόμβους εκτός του Dispatcher καθώς δεν θέλουμε να τον συμπεριλάβουμε κατα τον υπολογισμό του μέσου όρου.

```
 # --- ΥΠΟΛΟΓΙΣΜΟΣ ΣΤΑΤΙΣΤΙΚΩΝ ---
    valid_records = [r for r in records if r.arrival_date >
WARMUP_TIME]

    # 1. Μέσος Χρόνος Αναμονής
    # Node 1 is Dispatcher, ignore wait time there (should be 0 anyway)
    waiting_times = [r.waiting_time for r in valid_records if r.node >
1]
    mean_wait = sum(waiting_times) / len(waiting_times) if
waiting_times else 0.0
```

Αμέσως μετά υπολογίζουμε την χρησιμοποίηση ανά κόμβο

```
 # 2. Χρησιμοποίηση ανά κόμβο
    node_busy_time = {i: 0.0 for i in range(2, 13)}

    for r in records:
        if r.node == 1: continue

        start = r.service_start_date
        end = r.service_end_date

        effective_start = max(start, WARMUP_TIME)
        effective_end = min(end, WARMUP_TIME + sim_duration)

        if effective_end > effective_start:
            node_busy_time[r.node] += (effective_end - effective_start)

    utilizations = {i: node_busy_time[i] / sim_duration for i in
range(2, 13)}
```

Σε αυτό το σημείο του κώδικα είχαμε το πρόβλημα ότι μια εργασία μπορεί να ξεκινήσει κατά τη διάρκεια του warm-up και να τελειώσει κατά τη διάρκεια της κανονικής προσομοίωσης (ή το αντίστροφο) και αν απλά μετρούσαμε αν μια εργασία ανήκει στο διάστημα, θα χάναμε ακρίβεια. Για αυτό τον λόγο βρίσκουμε το effective_start και το effective_end κάθε εργασίας για να φιλτράρουμε σωστά τους

χρόνους για τις εργασίες που ξεκινούν πριν το τέλος του warm-up και τις εργασίες που τελειώνουν μετά το τέλος της προσομοίωσης. Άρα με την node_busy_time παίρνουμε τον ακριβή χρόνο που ένας κόμβος ήταν απασχολημένος μέσα στα χρονικά πλαίσια της προσομοίωσης και διαιρώντας με το sim_duration παίρνουμε την ακριβή χρησιμοποίηση κάθε κόμβου.

Τέλος υπολογίζουμε και τον ρυθμό εξυπηρέτησης και επιστρέφουμε τις τρεις αυτές τιμές στο τέλος του κάθε πειράματος.

```
    # 3. Ρυθμός εξυπηρέτησης
    completed_in_window = [r for r in records if r.service_end_date >
WARMUP_TIME and r.service_end_date <= WARMUP_TIME + sim_duration]
    throughput = len(completed_in_window) / sim_duration

    return mean_wait, utilizations, throughput
```

Στο τελευταίο μέρος του simulation.py έχουμε την main συνάρτηση όπου υλοποιεί τον αλγόριθμο ελέγχου των ανεξάρτητων επαναλήψεων.

```
# --- MAIN ---
def main():
    if len(sys.argv) < 4:
        print("Usage: python simulation.py <sim_time> <algo> <d>")
        sys.exit(1)

    sim_time = float(sys.argv[1])
    algo = int(sys.argv[2])
    d_val = int(sys.argv[3])

    global D_PARAMETER
    D_PARAMETER = d_val

    print(f"--- Starting Simulation (Algo: {algo}, d: {d_val}, Time:
{sim_time}) ---")

    # Γρήγορο διαγνωστικό check
    print("Running diagnostic check...", end=' ', flush=True)
    try:
        run_single_replication(100, algo, 999)
        print("OK!")
    except Exception as e:
        print(f"\nERROR in Diagnostic: {e}")
```

```
        sys.exit(1)

    replications = 0
    mean_waits = []
    all_utilizations = {i: [] for i in range(2, 13)}
    all_throughputs = []

    min_reps = 10
    max_reps = 50

    while replications < max_reps:
        replications += 1
        mw, utils, th = run_single_replication(sim_time, algo,
replications)

        mean_waits.append(mw)
        for i in utils: all_utilizations[i].append(utils[i])
        all_throughputs.append(th)

        if replications >= min_reps:
            mean_X = sum(mean_waits) / replications
            variance = sum([(x - mean_X)**2 for x in mean_waits]) /
(replications - 1)
            S = math.sqrt(variance)
            t_crit = get_t_value(replications - 1)
            hw = t_crit * (S / math.sqrt(replications))

            rel_error = (hw / mean_X) if mean_X > 0 else 1.0

            # Καθαρίζουμε την γραμμή προόδου και τυπώνουμε το
αποτέλεσμα
            print(f"\rRep {replications}: Mean Wait = {mean_X:.4f} s,
Rel Error = {rel_error:.4f}   ")

            if rel_error <= DESIRED_REL_ERROR:
                print("--> Precision Met!")
                break

    final_mean_wait = sum(mean_waits) / replications

    # Υπολογισμός διασποράς και διαστήματος εμπιστοσύνης
    if replications > 1:
```

```
        variance = sum([(x - final_mean_wait)**2 for x in mean_waits])
/ (replications - 1)
        S = math.sqrt(variance)
        t_crit = get_t_value(replications - 1)
        hw = t_crit * (S / math.sqrt(replications))
        rel_error = (hw / final_mean_wait) if final_mean_wait > 0 else
1.0
    else:
        hw = 0.0
        rel_error = 0.0

    final_throughput = sum(all_throughputs) / replications
    avg_node_util = {i: sum(all_utilizations[i])/replications for i in
range(2, 13)}
    final_total_util = sum(avg_node_util.values()) / 11

    # --- ΕΓΓΡΑΦΗ ΣΕ ΑΡΧΕΙΟ ---
    filename = f"results_algo{algo}_d{d_val}.txt"
    with open(filename, "w", encoding='utf-8') as f:
        f.write("--- ΑΠΟΤΕΛΕΣΜΑΤΑ ΠΡΟΣΟΜΟΙΩΣΗΣ ---\n")
        f.write(f"Αλγόριθμος: {algo}, D: {d_val}\n")
        f.write(f"Χρόνος Προσομοίωσης: {sim_time} sec (+{WARMUP_TIME}
warmup)\n")
        f.write(f"Επαναλήψεις: {replications}\n\n")
        f.write(f"Μέσος Χρόνος Αναμονής: {final_mean_wait:.6f} sec\n")
        f.write(f"(95% CI Half-width: {hw:.6f}, Rel. Error:
{rel_error:.4f})\n")
        f.write(f"Μέσος Ρυθμός Εξυπηρέτησης: {final_throughput:.6f}
jobs/sec\n")
        f.write(f"Μέση Συνολική Χρησιμοποίηση:
{final_total_util:.4f}\n")
        f.write("Μέση Χρησιμοποίηση ανά Κόμβο:\n")
        for i in range(2, 13):
            desc = "Slow" if i <= 6 else ("Medium" if i <= 9 else
"Fast")
            f.write(f"  Node {i} ({desc}): {avg_node_util[i]:.4f}\n")

    print(f"\nResults written to {filename}")

if __name__ == "__main__":
    main()
```

Με συνοπτικά λόγια η main εκτελεί ένα βρόχο while ο οποίος τρέχει προσομοιώσεις μέχρι να ικανοποιηθούν τα κριτήρια τερματισμού. Μετά από κάθε επανάληψη υπολογίζεται το διάστημα εμπιστοσύνης για τον μέσο χρόνο αναμονής. Λαμβάνοντας υπόψη την μέση τιμή την διασπορά καθώς και την κρίσιμη τιμή t από την κατανομή που μας δόθηκε (μέσω της συνάρτησης που υλοποιήσαμε get_t_value), υπολογίζεται το εύρος ημι-διαστήματος και το σχετικό σφάλμα. Όταν το σχετικό σφάλμα γίνει μικρότερο ή ίσο του σχετικού στόχου τερματίζεται ο βρόχος και στην περίπτωση που έχουμε εξαιρετικά υψηλή διασπορά έχουμε βάλει επίσης ένα maximum αριθμό επαναλήψεων που δεν μπορεί να ξεπεραστεί. Τέλος εξάγονται τα αποτελέσματα σε ένα αρχείο κειμένου results_algoX_dY.txt.

Στο δεύτερο κομμάτι υλοποιούμε το δεύτερο αρχείο python, run_experiments.py. Αυτό λειτουργεί ως ο κύριος οδηγός της πειραματικής μας διαδικασίας καθώς εκτελεί μαζικά τα σενάρια χρήσης και συγκεντρώνει αποτελέσματα πολλών πειραμάτων ταυτόχρονα.

```
import os
import re

# Ρυθμίσεις
SIM_TIME = 86400  # 24 ώρες
PYTHON_CMD = "python"

# Λίστα πειραμάτων: (Αλγόριθμος, d)
experiments = [
    (1, 0),        # Αλγόριθμος 1
    (2, 0),        # Αλγόριθμος 2 με d=0
    (2, 1),        # Αλγόριθμος 2 με d=1
    (2, 2),        # Αλγόριθμος 2 με d=2
    (2, 3),        # Αλγόριθμος 2 με d=3
    (2, 4),        # Αλγόριθμος 2 με d=4
    (2, 5)         # Αλγόριθμος 2 με d=5
]
```

Αρχικά εισάγουμε τις απαραίτητες βιβλιοθήκες και μεταβλητές και ορίζουμε την λίστα πειραμάτων όπως φαίνεται και από πάνω. Κάθε tuple αντιστοιχεί σε μία ξεχωριστή εκτέλεση του προσομοιωτή για τους δύο αλγόριθμους καθώς και για τις διάφορες τιμές του d για τον δεύτερο αλγόριθμο. Με μία for για κάθε πείραμα στην λίστα που ορίσαμε εκτελούμε την simulation.py με:

```
# Εκτέλεση του simulation.py
    cmd = f"{PYTHON_CMD} simulation.py {SIM_TIME} {algo} {d}"
    exit_code = os.system(cmd)
```

Στην συνέχεια κάνουμε ανάγνωση των αποτελεσμάτων με την χρήση μίας συνάρτησης parse_results(filename) η οποία ορίστηκε από εμάς η οποία πρακτικά διαβάζει το αρχείο αποτελεσμάτων που παράγεται και εξάγει τα αποτελέσματα κάνοντας αναζήτηση με κανονικές εκφράσεις.

```
# Ανάγνωση αποτελεσμάτων
    filename = f"results_algo{algo}_d{d}.txt"
    if os.path.exists(filename):
        wait, util = parse_results(filename)
        results_summary.append({
            'algo': algo,
            'd': d,
            'wait': wait,
            'util': util
        })
    else:
        print(f"Warning: Output file {filename} not found."


def parse_results(filename):
    """Διαβάζει το αρχείο αποτελεσμάτων και εξάγει τα βασικά
νούμερα."""
    try:
        with open(filename, 'r', encoding='utf-8') as f:
            content = f.read()

        # Αναζήτηση με Regular Expressions
        wait_match = re.search(r"Μέσος Χρόνος Αναμονής:\s+([0-9.]+)",
content)
        util_match = re.search(r"Μέση Συνολική
Χρησιμοποίηση.*:\s+([0-9.]+)", content)

        mean_wait = float(wait_match.group(1)) if wait_match else 0.0
        total_util = float(util_match.group(1)) if util_match else 0.0

        return mean_wait, total_util
    except Exception as e:
        print(f"Error parsing {filename}: {e}")
        return 0.0, 0.0
```

Στο τέλος της εκτέλεσης, το script τυπώνει έναν μορφοποιημένο πίνακα (Summary Report) στο τερματικό. Αυτός ο πίνακας παρέχει μια άμεση, συγκριτική

εικόνα όλων των πειραμάτων, διευκολύνοντας την επαλήθευση των αποτελεσμάτων και την εισαγωγή των δεδομένων στα γραφήματα της αναφοράς.

Παρουσίαση και ανάλυση αποτελεσμάτων προσομοίωσης

Με την εκτέλεση του run_experiments.py αρχείου μας εκτελείται η προσομοίωση για τους δύο αλγορίθμους και τις τιμές του d. Παρακάτω παρουσιάζονται τα αποτελέσματα της προσομοίωσης μας.

--- ΑΠΟΤΕΛΕΣΜΑΤΑ ΠΡΟΣΟΜΟΙΩΣΗΣ ---

Αλγόριθμος: 1, D: 0

Χρόνος Προσομοίωσης: 86400.0 sec (+3600 warmup)

Επαναλήψεις: 10

Μέσος Χρόνος Αναμονής: 1.526850 sec

(95% CI Half-width: 0.033422, Rel. Error: 0.0219)

Μέσος Ρυθμός Εξυπηρέτησης: 1.999303 jobs/sec

Μέση Συνολική Χρησιμοποίηση: 0.7503

Μέση Χρησιμοποίηση ανά Κόμβο:

Node 2 (Slow): 0.9534

Node 3 (Slow): 0.9420

Node 4 (Slow): 0.9282

Node 5 (Slow): 0.9129

Node 6 (Slow): 0.8957

Node 7 (Medium): 0.8250

Node 8 (Medium): 0.7836

Node 9 (Medium): 0.7347

Node 10 (Fast): 0.5314

Node 11 (Fast): 0.4271

Node 12 (Fast): 0.3196

--- ΑΠΟΤΕΛΕΣΜΑΤΑ ΠΡΟΣΟΜΟΙΩΣΗΣ ---

Αλγόριθμος: 2, D: 0

Χρόνος Προσομοίωσης: 86400.0 sec (+3600 warmup)

Επαναλήψεις: 10

Μέσος Χρόνος Αναμονής: 0.514695 sec

(95% CI Half-width: 0.009764, Rel. Error: 0.0190)

Μέσος Ρυθμός Εξυπηρέτησης: 2.002178 jobs/sec

Μέση Συνολική Χρησιμοποίηση: 0.6287

Μέση Χρησιμοποίηση ανά Κόμβο:

Node 2 (Slow): 0.8092

Node 3 (Slow): 0.7676

Node 4 (Slow): 0.7206

Node 5 (Slow): 0.6629

Node 6 (Slow): 0.5990

Node 7 (Medium): 0.4415

Node 8 (Medium): 0.3589

Node 9 (Medium): 0.2757

Node 10 (Fast): 0.8266

Node 11 (Fast): 0.7658

Node 12 (Fast): 0.6880

--- ΑΠΟΤΕΛΕΣΜΑΤΑ ΠΡΟΣΟΜΟΙΩΣΗΣ ---

Αλγόριθμος: 2, D: 1

Χρόνος Προσομοίωσης: 86400.0 sec (+3600 warmup)

Επαναλήψεις: 10

Μέσος Χρόνος Αναμονής: 2.111877 sec

(95% CI Half-width: 0.021526, Rel. Error: 0.0102)

Μέσος Ρυθμός Εξυπηρέτησης: 2.002042 jobs/sec

Μέση Συνολική Χρησιμοποίηση: 0.5667

Μέση Χρησιμοποίηση ανά Κόμβο:

Node 2 (Slow): 0.6859

Node 3 (Slow): 0.6326

Node 4 (Slow): 0.5703

Node 5 (Slow): 0.5054

Node 6 (Slow): 0.4368

Node 7 (Medium): 0.2929

Node 8 (Medium): 0.2251

Node 9 (Medium): 0.1662

Node 10 (Fast): 0.9405

Node 11 (Fast): 0.9097

Node 12 (Fast): 0.8689

--- ΑΠΟΤΕΛΕΣΜΑΤΑ ΠΡΟΣΟΜΟΙΩΣΗΣ ---

Αλγόριθμος: 2, D: 2

Χρόνος Προσομοίωσης: 86400.0 sec (+3600 warmup)

Επαναλήψεις: 10

Μέσος Χρόνος Αναμονής: 4.347787 sec

(95% CI Half-width: 0.019405, Rel. Error: 0.0045)

Μέσος Ρυθμός Εξυπηρέτησης: 1.999564 jobs/sec

Μέση Συνολική Χρησιμοποίηση: 0.5399

Μέση Χρησιμοποίηση ανά Κόμβο:

Node 2 (Slow): 0.6151

Node 3 (Slow): 0.5627

Node 4 (Slow): 0.5006

Node 5 (Slow): 0.4389

Node 6 (Slow): 0.3735

Node 7 (Medium): 0.2447

Node 8 (Medium): 0.1859

Node 9 (Medium): 0.1358

Node 10 (Fast): 0.9761

Node 11 (Fast): 0.9626

Node 12 (Fast): 0.9426

--- ΑΠΟΤΕΛΕΣΜΑΤΑ ΠΡΟΣΟΜΟΙΩΣΗΣ ---

Αλγόριθμος: 2, D: 3

Χρόνος Προσομοίωσης: 86400.0 sec (+3600 warmup)

Επαναλήψεις: 10

Μέσος Χρόνος Αναμονής: 6.885982 sec

(95% CI Half-width: 0.022471, Rel. Error: 0.0033)

Μέσος Ρυθμός Εξυπηρέτησης: 1.998660 jobs/sec

Μέση Συνολική Χρησιμοποίηση: 0.5284

Μέση Χρησιμοποίηση ανά Κόμβο:

  Node 2 (Slow): 0.5822

  Node 3 (Slow): 0.5298

  Node 4 (Slow): 0.4709

  Node 5 (Slow): 0.4083

  Node 6 (Slow): 0.3465

  Node 7 (Medium): 0.2285

  Node 8 (Medium): 0.1713

  Node 9 (Medium): 0.1256

  Node 10 (Fast): 0.9904

  Node 11 (Fast): 0.9840

  Node 12 (Fast): 0.9749

--- ΑΠΟΤΕΛΕΣΜΑΤΑ ΠΡΟΣΟΜΟΙΩΣΗΣ ---

Αλγόριθμος: 2, D: 4

Χρόνος Προσομοίωσης: 86400.0 sec (+3600 warmup)

Επαναλήψεις: 10

Μέσος Χρόνος Αναμονής: 9.620479 sec

(95% CI Half-width: 0.035361, Rel. Error: 0.0037)

Μέσος Ρυθμός Εξυπηρέτησης: 2.001288 jobs/sec

Μέση Συνολική Χρησιμοποίηση: 0.5264

Μέση Χρησιμοποίηση ανά Κόμβο:

Node 2 (Slow): 0.5720

Node 3 (Slow): 0.5183

Node 4 (Slow): 0.4620

Node 5 (Slow): 0.4006

Node 6 (Slow): 0.3406

Node 7 (Medium): 0.2231

Node 8 (Medium): 0.1708

Node 9 (Medium): 0.1251

Node 10 (Fast): 0.9957

Node 11 (Fast): 0.9931

Node 12 (Fast): 0.9891

--- ΑΠΟΤΕΛΕΣΜΑΤΑ ΠΡΟΣΟΜΟΙΩΣΗΣ ---

Αλγόριθμος: 2, D: 5

Χρόνος Προσομοίωσης: 86400.0 sec (+3600 warmup)

Επαναλήψεις: 10

Μέσος Χρόνος Αναμονής: 12.481504 sec

(95% CI Half-width: 0.035259, Rel. Error: 0.0028)

Μέσος Ρυθμός Εξυπηρέτησης: 2.001545 jobs/sec

Μέση Συνολική Χρησιμοποίηση: 0.5247

Μέση Χρησιμοποίηση ανά Κόμβο:

Node 2 (Slow): 0.5644

Node 3 (Slow): 0.5135

Node 4 (Slow): 0.4571

Node 5 (Slow): 0.3943

Node 6 (Slow): 0.3387

Node 7 (Medium): 0.2214

Node 8 (Medium): 0.1676

Node 9 (Medium): 0.1241

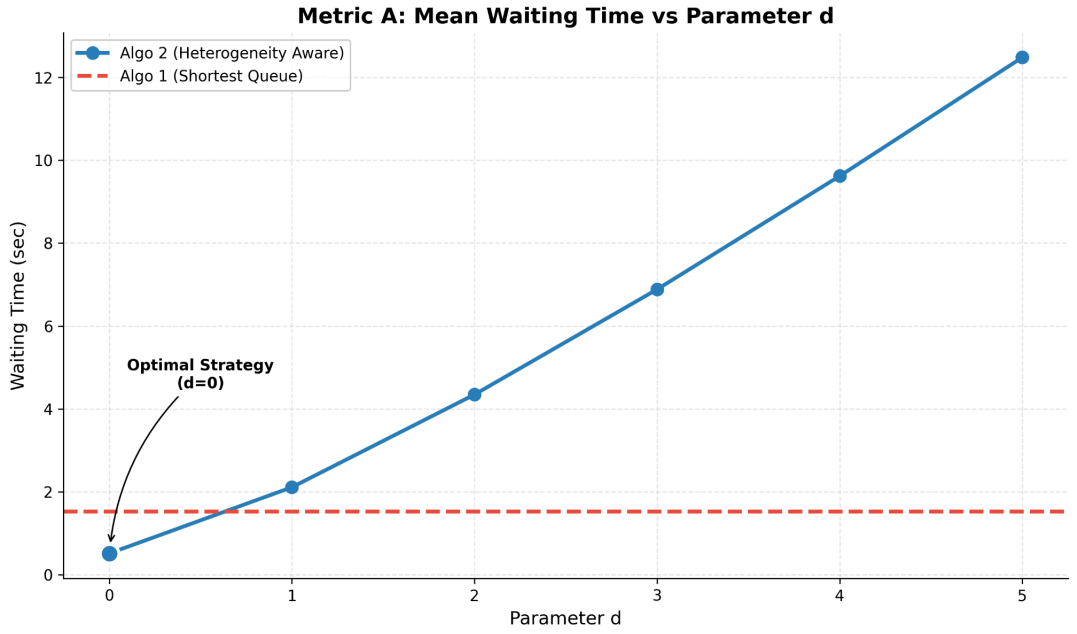
Node 10 (Fast): 0.9981

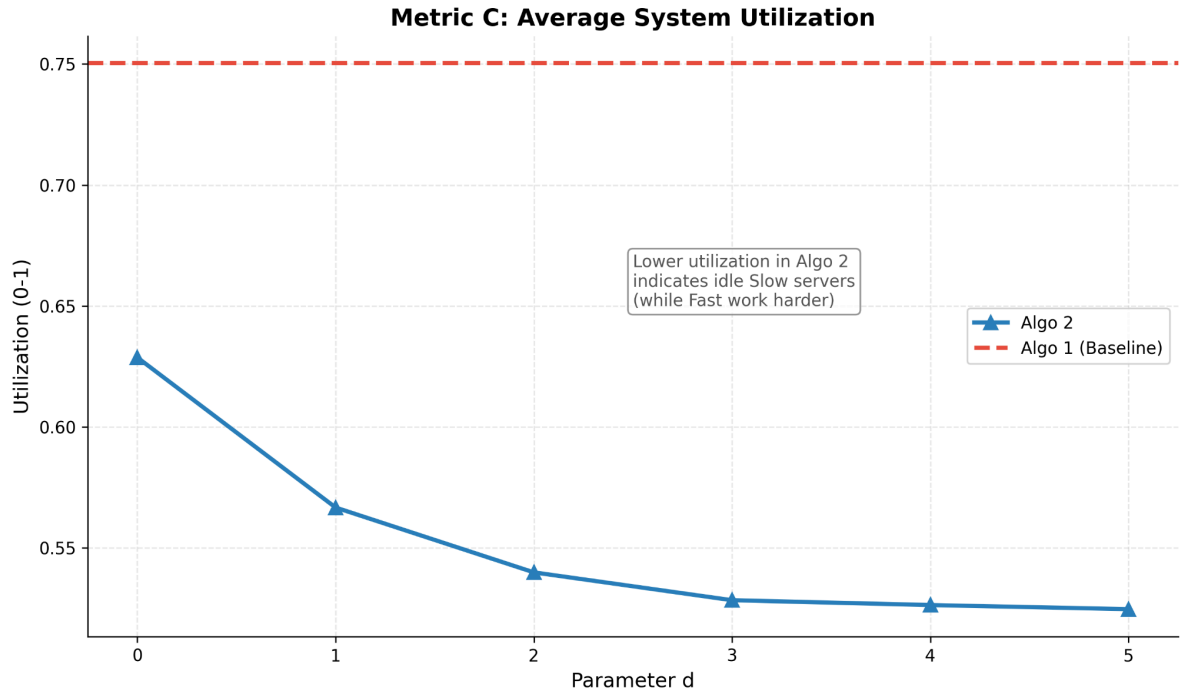
Node 11 (Fast): 0.9970

Node 12 (Fast): 0.9951

Με βάση αυτά τα δεδομένα που εξάγαμε από τα πειράματα φτιάχνουμε τον παρακάτω πίνακα για να μελετήσουμε την συμπεριφορά του αλγορίθμου 2 για τις διάφορες τιμές του d.

Παράμετρος d	Μέσος Χρόνος Αναμονής (sec)	Μέση Συνολική Χρησιμοποίηση
0	0.5147	0.6287
1	2.1119	0.5667
2	4.3478	0.5399
3	6.8860	0.5284
4	9.6205	0.5264
5	12.4815	0.5247





Με βάση αυτά τα δεδομένα που εξάγαμε παρατηρούμε ότι η τιμή του d που ελαχιστοποιεί τον χρόνο αναμονής είναι η d = 0 (0.5147 sec). Παρατηρούμε επίσης ότι η αύξηση του d προκαλεί και αύξηση του χρόνου αναμονής αρκετά καθώς οι μεγαλύτερες τιμές του d σημαίνει ότι ο αλγόριθμος θα στείλει εργασίες στους γρήγορους εξυπηρετητές ακόμα και αν έχουν ήδη μεγάλο φόρτο εργασίας στην ουρά. Άρα η αναμονή στην ουρά φαίνεται να είναι πιο κοστοβόρα από την άμεση εξυπηρέτηση από πιο αργούς κόμβους.

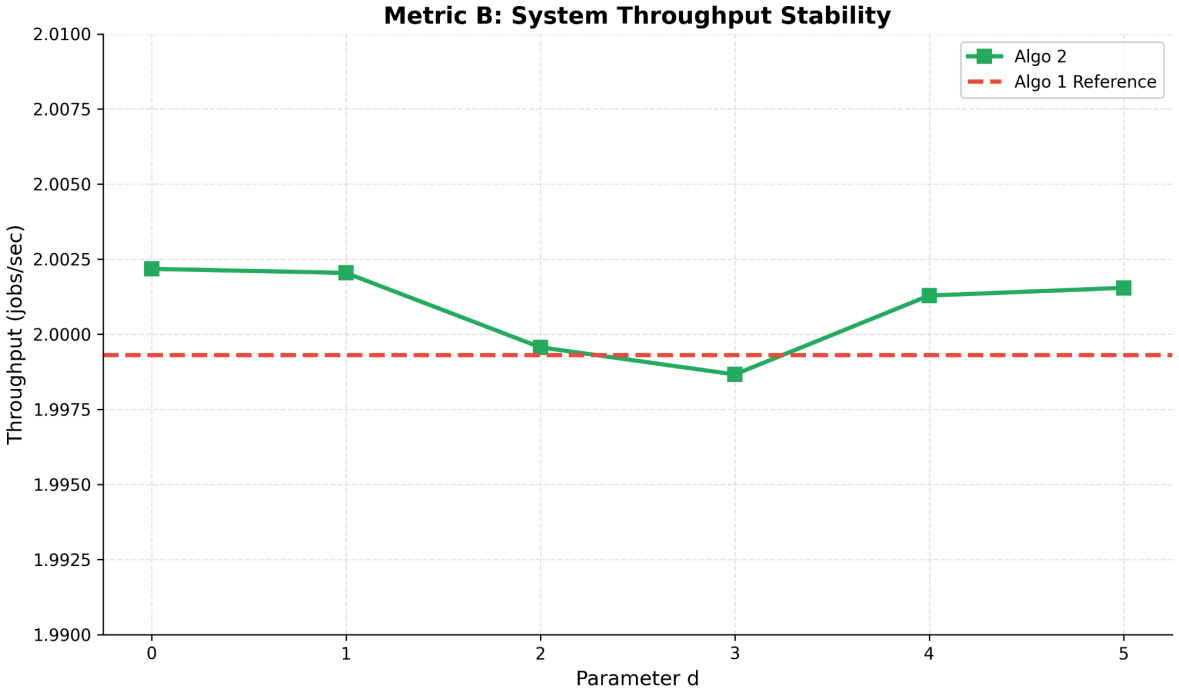
Από την άλλη παρατηρούμε ότι η τιμή που δίνει την μέγιστη μέση χρησιμοποίηση είναι επίσης για d = 0 καθώς αυτό σημαίνει ότι πολλές εργασίες θα δρομολογηθούν σε πιο αργούς εξυπηρετητές που χρησιμοποιούνται για περισσότερη ώρα. Όσο μεγαλώνει το d τόσο και μικραίνει η χρησιμοποίηση καθώς οι γρήγοροι κόμβοι ολοκληρώνουν τις εργασίες πιο γρήγορα.

Καθώς δείξαμε ότι ο για d = 0 έχουμε χαμηλότερο μέσο χρόνο αναμονής και υψηλότερη χρησιμοποίηση, για να δούμε αν ο αλγόριθμος 2 μπορεί να πετύχει καλύτερες τιμές τον συγκρίνουμε μόνο με την περίπτωση που έχουμε τον αλγόριθμο 1 τσεκάρουμε τις τιμές του αλγορίθμου 2 μόνο για d = 0. Τα αποτελέσματα φαίνονται στον παρακάτω πίνακα.

Μετρική	Αλγόριθμος 1	Αλγόριθμος 2 (d=0)	Ποσοστιαία Μεταβολή
Μέσος Χρόνος Αναμονής	1.5269 sec	0.5147 sec	-66.29% (Βελτίωση)
Μέση Χρησιμοποίηση	0.7503	0.6287	-16.21%

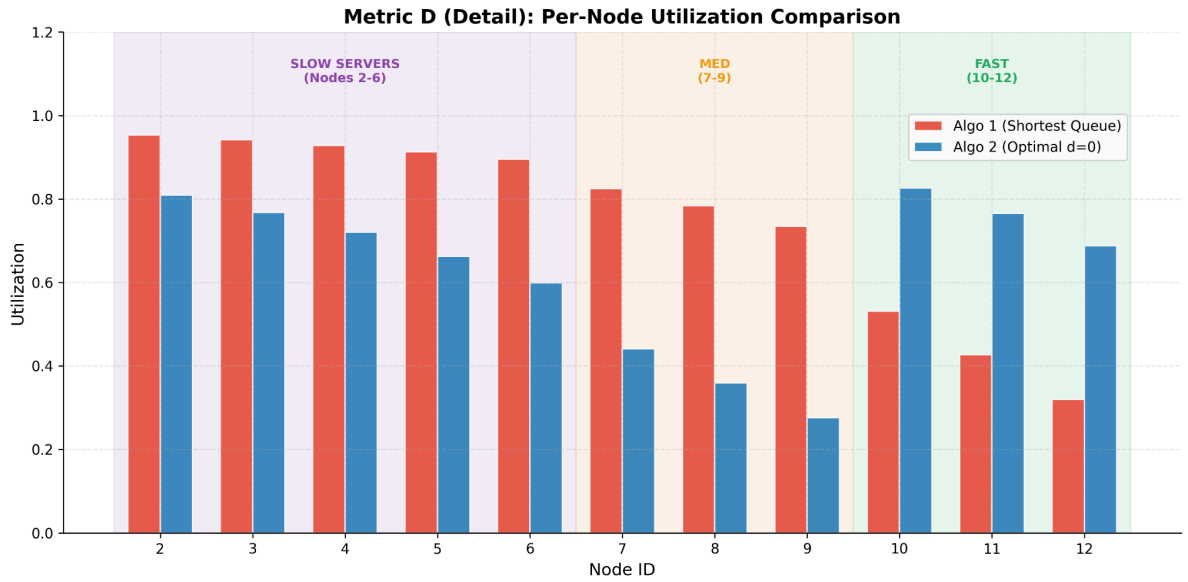
Άρα απαντώντας στο 2ο ερώτημα ο αλγόριθμος 2 παρουσιάζει σημαντικά χαμηλότερο χρόνο αναμονής από τον 1 (κατά -66%) καθώς εκμεταλλεύονται την πληροφορία για την ταχύτητα των κόμβων έχοντας έτσι αποδοτικότερη δρομολόγηση. Όμως ο αλγόριθμος 2 δεν μπορεί να έχει υψηλότερη χρησιμοποίηση συγκριτικά με τον αλγόριθμο 1 κάτι που δείχνει ξεκάθαρα την αποδοτικότητα του αλγορίθμου 2.

Σε αυτό το σημείο αφού απαντήθηκαν τα δύο ζητούμενα της άσκησης παρουσιάζουμε περαιτέρω δεδομένα και παρατηρήσεις που εντοπίσαμε.

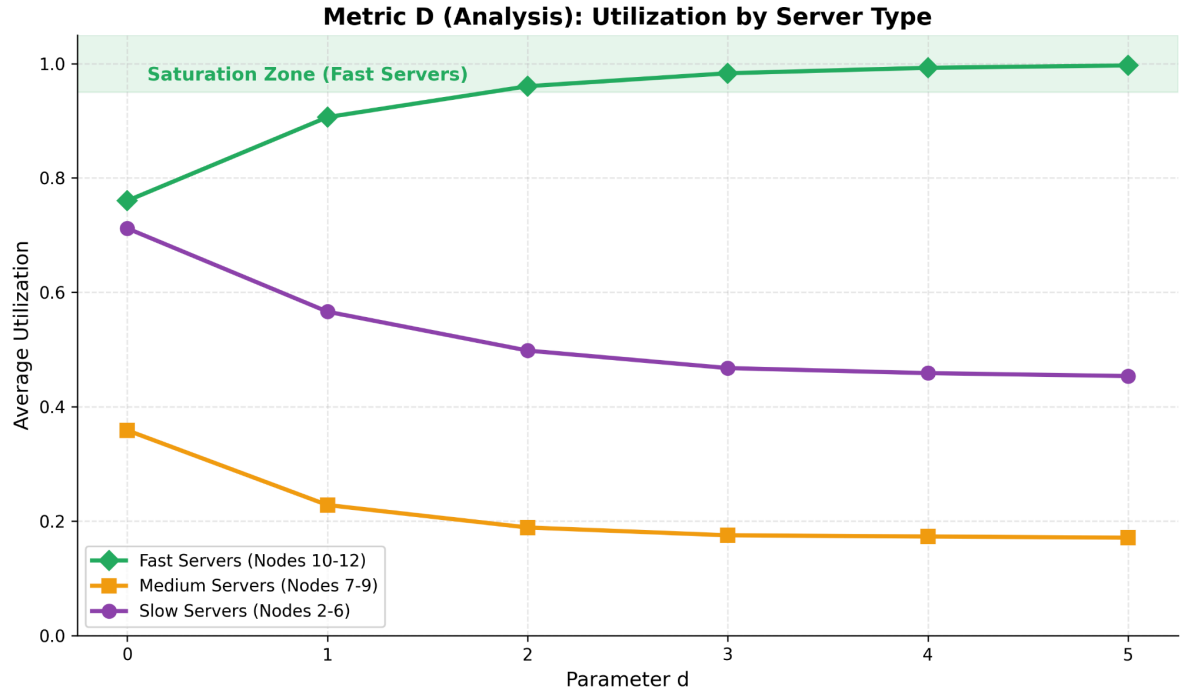


Για τον μέσο ρυθμό εξυπηρέτησης όπως βλέπουμε δεν υπάρχουν πολύ μεγάλες διακυμάνσεις στις τιμές κάτι που είναι απόλυτα λογικό για ένα ευσταθές σύστημα. Το Throughput παραμένει σταθερό κοντά στις 2 εργασίες/δευτερόλεπτο και δεν απορρίπτει καμία εργασία. Πρακτικά αυτό μας αποδεικνύει πως η αύξηση της αποδοτικότητας του συστήματος που εντοπίσαμε παραπάνω ισχύει καθώς το σύστημα όντως εξυπηρετεί όλο τον φόρτο εργασίας σε όλες τις περιπτώσεις.

Παρακάτω παρουσιάζεται και ένα γράφημα για ανάλυση κατανομής των εργασιών ανά κόμβο.



Παρατηρούμε ότι στον αλγόριθμο 1 (κόκκινο χρώμα) έχουμε πολύ υψηλή χρησιμοποίηση στους αργούς κόμβους και πολύ χαμηλή στους υψηλούς. Αυτό γίνεται γιατί ο αλγόριθμος 1 δρομολογεί τις εργασίες ανεξάρτητα από το πόσο γρήγοροι είναι οι κόμβοι με αποτέλεσμα οι γρήγοροι εξυπηρετητες να περιμένουν καθώς τελειώνουν την εργασία τους πιο γρήγορα. Αντίθετα ο αλγόριθμος 2 (μπλε χρώμα) αναγνωρίζει την ταχύτητα των γρήγορων κόμβων και αναθέτει περισσότερη δουλειά σε αυτούς αυξάνοντας την χρησιμοποίηση και την αποδοτικότητα του συστήματος. Παρακάτω βλέπουμε παρόμοια αποτελέσματα και για τις διάφορες τιμές d του αλγορίθμου 2 για την χρησιμοποίηση ομαδοποιημένα ανά κατηγορία κόμβων.



Συμπεράσματα

Ολοκληρώνοντας αυτή την άσκηση μπορέσαμε να βγάλουμε και κάποια γενικά συμπεράσματα. Αρχικά τα αποτελέσματα μας έδειξαν ξεκάθαρα την ανάγκη οι αλγόριθμοι να γνωρίζουν τα χαρακτηριστικά του υλικού του συστήματος, καθώς στην περίπτωση που αγνοούνταν οι διάφορες ταχύτητες επεξεργασίας με τον αλγόριθμο 1 παρατηρούμε μείωση στην απόδοση του συστήματος. Αυτό μας δείχνει την μεγάλη σημασία που έχει η αναγνώριση της ετερογένειας από τους αλγορίθμους στον ρυθμό επεξεργασίας δεδομένων. Τέλος παρατηρήσαμε ότι η αύξηση του d αποδείχθηκε λανθασμένη για την αποδοτικότητα του συστήματος και η βέλτιστη στρατηγική ήταν η προτίμηση σε μικρό βαθμό των γρήγορων κόμβων.

Bonus ερώτημα

Στις προσομοιώσεις ουράς αναμονής, το σύστημα εκκινεί συνήθως σε κατάσταση κενή και αδρανή. Συνεπώς, οι πρώτοι πελάτες απολαμβάνουν μηδενικούς ή πολύ μικρούς χρόνους αναμονής, οι οποίοι δεν είναι αντιπροσωπευτικοί της μακροχρόνιας λειτουργίας του συστήματος.

Στόχος της παρούσας ανάλυσης είναι ο ακριβής προσδιορισμός του μήκους της μεταβατικής κατάστασης (l), ώστε να εφαρμοσθεί η μέθοδος της Διαγραφής Αρχικών Δεδομένων. Σε αντίθεση με την αυθαίρετη επιλογή χρονικού διαστήματος (π.χ. 1 ώρα), η μέθοδος αυτή βασίζεται σε στατιστική ανάλυση χρονοσειρών για να εντοπίσει πότε το σύστημα εισέρχεται πραγματικά σε Κατάσταση Ισορροπίας.

Μεθοδολογία Υλοποίησης
Για τον εντοπισμό του σημείου τομής (l), ακολουθήθηκε η παρακάτω διαδικασία και υλοποιήθηκε στο αρχείο bonus_analysis.py:

Εκτελέστηκαν R = 30 ανεξάρτητες επαναλήψεις της προσομοίωσης. Για κάθε επανάληψη r, καταγράφηκε η ακολουθία των χρόνων αναμονής για τους πρώτους N=15.000 πελάτες.
Στη συνέχεια, υπολογίστηκε ο Συλλογικός Μέσος Όρος για κάθε n-οστό πελάτη ξεχωριστά, ώστε να εξαλειφθεί η τυχαιότητα της κάθε μίας εκτέλεσης.
Επειδή η καμπύλη εξακολουθεί να παρουσιάζει υψηλή διακύμανση, εφαρμόστηκε φίλτρο Κυλιόμενου Μέσου Όρου με παράθυρο εύρους w=500.

Τεκμηρίωση των Παραμετρων

Σενάριο Αναφοράς:
Θεωρήσαμε ότι το να φτιάξουμε γραφήματα μεταβατικής κατάστασης για 7 διαφορετικά πειράματα και να κάνουμε οπτική επιθεώρηση στο καθένα είναι υπερβολικό για αυτό το ερώτημα. Το ζητούμενο του Bonus είναι να δείξουμε ότι ξέρουμε τη μέθοδο.

Επιλέξαμε το βέλτιστο σενάριο (Αλγόριθμος 2, d=0) ως σενάριο αναφοράς. Καθώς αυτός ο αλγόριθμος είναι ο πιο αποδοτικός για την μέθοδο δρομολόγησης των εργασιών και επιτυγχάνει τη μεγαλύτερη ροή, αποτελεί ένα ασφαλές κριτήριο για την εκτίμηση της ισορροπίας όλου του συστήματος.

Επαναλήψεις:
Σε αυτήν την ανάλυση, ο στόχος μας είναι διαφορετικός από πριν. Δεν ψάχνουμε απλά ένα νούμερο (μέσο όρο), αλλά προσπαθούμε να ζωγραφίσουμε μια καμπύλη (τον χρόνο αναμονής ανά πελάτη) για να δούμε με το μάτι πότε "ισιώνει".
Τα δεδομένα μιας μόνο επανάληψης είναι γεμάτα "θόρυβο" (τυχαία σκαμπανεβάσματα).
Για να καθαρίσει η εικόνα και να δούμε την την καμπύλη, πρέπει να πάρουμε τον μέσο όρο πολλών επαναλήψεων για κάθε πελάτη ξεχωριστά.
Για αυτό κάνουμε 30 επαναλήψεις ώστε η καμπύλη να είναι αρκετά "λεία" για να βγάλουμε συμπέρασμα. Με 10 επαναλήψεις, η γραμμή στο γράφημα θα είχε πολλά σκαμπανεβάσματα και θα δυσκολευόμασταν να βρούμε το σημείο τομής.

Κυλιόμενος Μέσος Όρος:
Ο Κυλιόμενος Μέσος Όρος με παράθυρο w=500 είναι ουσιαστικά ένα smoothing filter.

Η προσομοίωση είναι τυχαία. Ακόμα κι αν πάρουμε τον μέσο όρο 30 επαναλήψεων, η τιμή του χρόνου αναμονής από τον έναν πελάτη στον άλλον θα έχει τεράστιες διακυμάνσεις.
π.χ.
Ο πελάτης #500 μπορεί να περιμένει 2 δευτερόλεπτα.
Ο πελάτης #501 μπορεί να βρει άδειο server και να περιμένει 0.
Ο πελάτης #502 μπορεί να πέσει σε ουρά και να περιμένει 5 δευτερόλεπτα.

Αν σχεδιάζαμε αυτές τις τιμές (οι γκρίζες γραμμες στο διάγραμμα που βγάλαμε), το γράφημα θα ήταν μια γραμμή γεμάτη spikes. Θα ήταν αδύνατο να πούμε πού ακριβώς σταματάει να ανεβαίνει και πού αρχίζει να ισιώνει, γιατί τα σκαμπανεβάσματα θα έκρυβαν την πραγματική τάση.

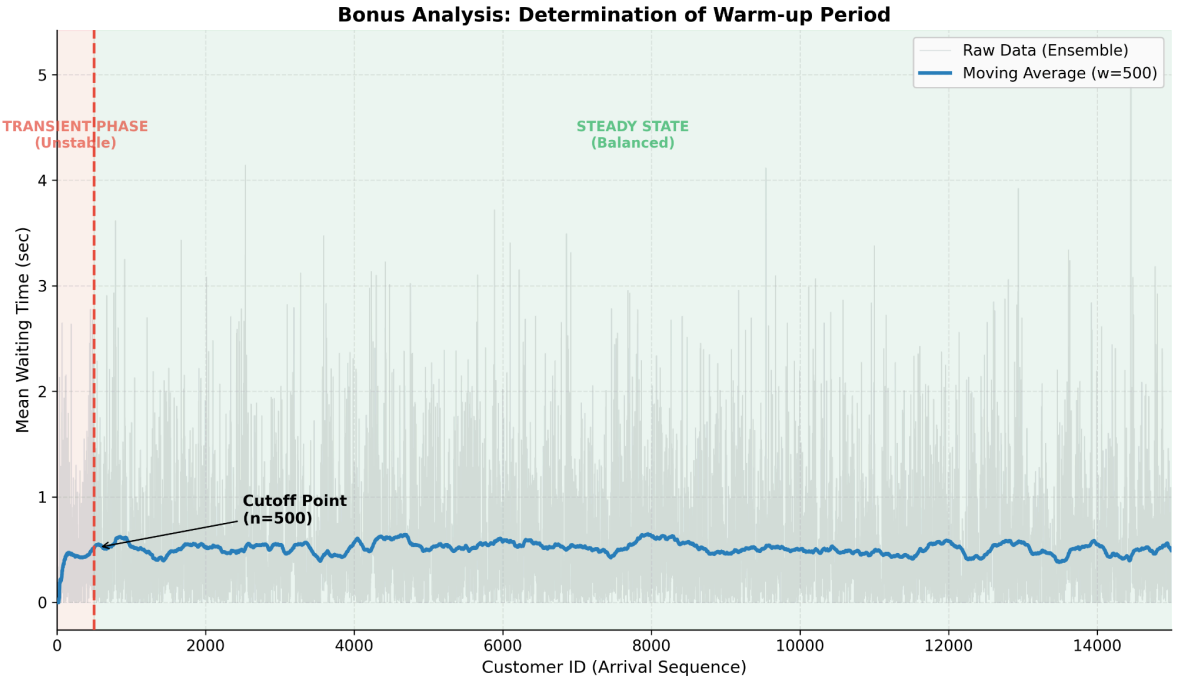
Ο Κυλιόμενος Μέσος Όρος λειτουργεί ως εξής:
Για να υπολογίσουμε την τιμή στο σημείο X (π.χ. στον πελάτη 1000), δεν παίρνουμε την τιμή του πελάτη 1000.
Παίρνουμε τον μέσο όρο των τιμών των 500 προηγούμενων πελατών (από τον 501 έως τον 1000).
Με αυτό πετυχαίνουμε την εξαφάνιση των ακραίων τιμών, δηλαδή αν ένας τυχαίος πελάτης είχε τεράστια αναμονή, αυτή η τιμή "αραιώνει" μέσα στο δείγμα των 500 και δεν χαλάει το γράφημα και μας επιτρέπει να δούμε τη μεγάλη εικόνα. Βλέπουμε ξεκάθαρα αν η ουρά κατά μέσο όρο μεγαλώνει (μεταβατική κατάσταση) ή αν έχει σταθεροποιηθεί (μόνιμη κατάσταση).

Γιατί διαλέξαμε το 500;
Με ένα πολύ μικρό παράθυρο (π.χ. w=10), το γράφημα θα παρέμενε πολύ "νευρικό" και κατσαρό. Δεν θα ξεχωρίζαμε πότε ισιώνει.
Με ένα πολύ μεγάλο παράθυρο (π.χ. w=5000), η καμπύλη θα γινόταν πολύ αργή. Θα αργούσε να δείξει ότι το σύστημα σταθεροποιήθηκε και θα νομίζαμε ότι το warm-up είναι μεγαλύτερο από το πραγματικό.
Το 500 για ένα δείγμα 15.000 πελατών είναι μια χρυσή τομή που καθαρίζει τον θόρυβο χωρίς να αλλοιώνει υπερβολικά την πληροφορία για το πότε συμβαίνει η σταθεροποίηση.

Πελάτες:
Επειδή ο ρυθμός αφίξεων είναι λ = 1 πελάτης/δευτερόλεπτο, οι 15.000 πελάτες αντιστοιχούν χοντρικά σε 15.000 δευτερόλεπτα (περίπου 4 ώρες) λειτουργίας του συστήματος. Στα προηγούμενα πειράματα είχαμε ορίσει αυθαίρετα το Warm-up στα 3.600 δευτερόλεπτα (1 ώρα). Άρα, τρέχοντας 15.000 πελάτες, καλύπτουμε χρόνο 4πλάσιο από το αναμενόμενο warm-up. Αυτό μας εξασφαλίζει ότι σίγουρα θα συμπεριλάβουμε το σημείο που το σύστημα ισορροπεί.

Για να πειστούμε με το μάτι ότι η καμπύλη "ίσιωσε", δεν αρκεί να δούμε το σημείο που σταματάει να ανεβαίνει. Πρέπει να δούμε και μια μεγάλη ευθεία γραμμή μετά από αυτό.
Αν τρέχαμε μόνο 2.000 πελάτες, η καμπύλη θα σταματούσε μόλις έφτανε στην κορυφή. Δεν θα ήμασταν σίγουροι αν όντως σταθεροποιήθηκε ή αν ήταν απλώς μια προσωρινή παύση πριν ανέβει κι άλλο.
Με τους 15.000 πελάτες, βλέπουμε την άνοδο (π.χ. στους πρώτους 1.000) και μετά μια τεράστια ευθεία (από το 1.000 έως το 15.000) που επιβεβαιώνει πέραν πάσης αμφιβολίας ότι είμαστε σε Steady State.

Αποτελέσματα Ανάλυσης



Όπως δείχνει το διάγραμμα, ο μέσος χρόνος αναμονής αυξάνεται γρήγορα για τους πρώτους πελάτες, επειδή εκείνη την ώρα γεμίζουν οι ουρές.

Κοιτάζοντας το γράφημα, παρατηρούμε ότι η καμπύλη σταθεροποιείται (εισέρχεται σε οριζόντια τροχιά με φυσιολογικές διακυμάνσεις) μετά τον πελάτη με αύξοντα αριθμό 500.

Αφαιρώντας τα δεδομένα των πρώτων 500 πελατών από όλες τις επαναλήψεις, υπολογίστηκαν εκ νέου τα στατιστικά στοιχεία (για τον Αλγόριθμο 2, d=0):

Μέσος Χρόνος Αναμονής: 0.513823 sec
Throughput: 2.000558 jobs/sec
Μέση Συνολική Χρησιμοποίηση: 0.6277

Εκτύπωση Αποτελεσμάτων Bonus Ανάλυσης

```
--- Running Transient Analysis (Bonus) ---
Algorithm 2 (d=0), Replications: 30, Customers: 15000
Running replication 30/30...
Processing data...

--- ΑΠΟΤΕΛΕΣΜΑΤΑ BONUS ---
Εντοπίστηκε σταθεροποίηση μετά τον πελάτη: 500
(Αυτό είναι το μήκος της μεταβατικής κατάστασης σε πλήθος πελατών)
Γράφημα αποθηκεύτηκε: bonus_analysis_plot.png

Recalculating statistics excluding initial data...

--- ΤΕΛΙΚΑ ΣΤΑΤΙΣΤΙΚΑ (STEADY STATE) ---
Μέσος Χρόνος Αναμονής: 0.513823 sec
Throughput: 2.000558 jobs/sec
Μέση Συνολική Χρησιμοποίηση: 0.6277
Done.
```

Η σύγκριση με την αρχική μέθοδο (Warm-up χρόνου) επιβεβαιώνει την εγκυρότητα των αποτελεσμάτων, καθώς οι αποκλίσεις είναι αμελητέες, υποδεικνύοντας ότι και οι δύο μέθοδοι απέκλεισαν επιτυχώς τη μεταβατική φάση.