ΜΥΕ029
Προσομοίωση και Μοντελοποίηση Υπολογιστικών Συστημάτων

Διδάσκων: Γεώργιος Καππές

Αναφορά 2ης Εργαστηριακής Άσκησης:

Προσομοίωση ετερογενών συμπλεγμάτων εξυπηρετητών

Ομάδα:

Κονδυλία Βέργου, ΑΜ 4325

Παναγιώτης Βουζαλής, ΑΜ 2653

Εαρινό Εξάμηνο 2022

Περιεχόμενα

Εισαγωγή

Σε αυτήν την εργαστηριακή άσκηση ασχοληθήκαμε με την προσομοίωση ενός (ετερογενούς) συμπλέγματος εξυπηρετητών το οποίο περιλαμβάνει εξυπηρετητές με διαφορετικούς ρυθμούς επεξεργασίας. Συγκεκριμένα ασχοληθήκαμε με την εφαρμογή 3 αλγορίθμων εξισορρόπησης φορτίου (load balancing). Για την προσομοίωση αυτή χρησιμοποιήθηκε εκτενώς η βιβλιοθήκη ciw της python.

Host System specifications

Η μελέτη έγινε στο παρακάτω σύστημα χρησιμοποιώντας τον root λογαριασμό (admin στην περίπτωση των Windows).

- CPU: AMD Ryzen 5 5600X 6core clocked @ 4.2GHz all-core
- RAM: Kingston DDR4 16GB 3600MHz
- SSD: Samsung 860 Evo 500GB
- HDD: Western Digital Black 1TB 7200rpm
- OS: Windows 10 Enterprise LTSC version 1809 build 17763.2867
- VMware Workstation Player 16

Παραδοτέα αρχεία

Η συγγραφή του κώδικα έγινε χρησιμοποιώντας το VS Code.
Τα παραδοτέα αρχεία είναι τα εξής:

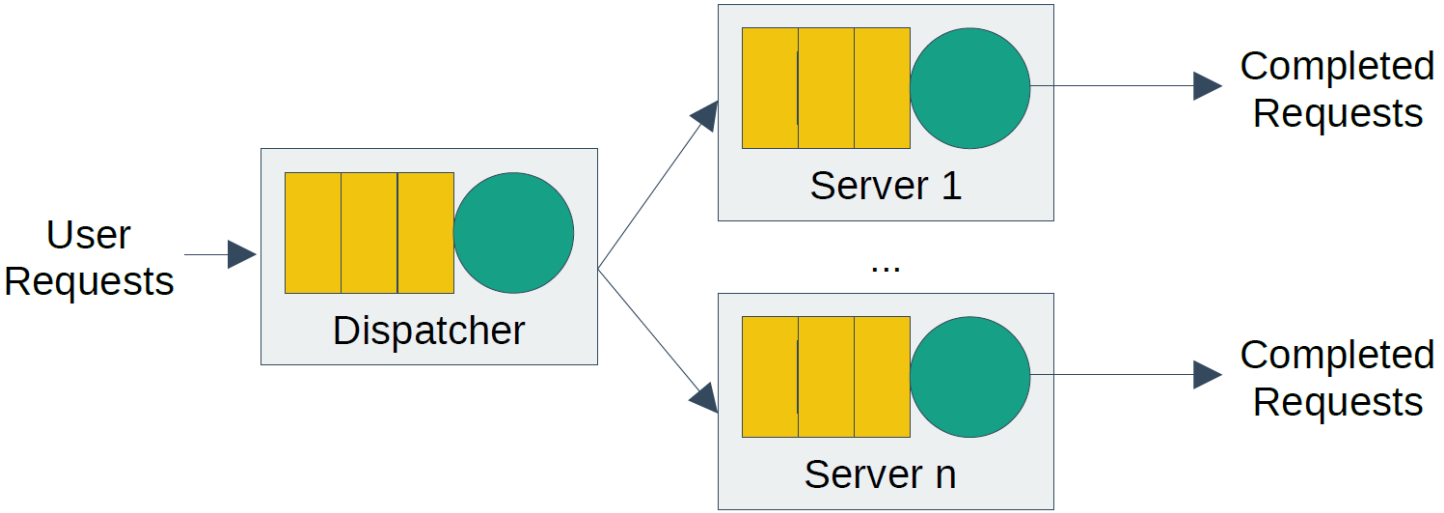
turnin.zip/

- mye029-lab2.pdf
- network_params.yml
- parsefiles.py
- simulation.py
- results.xlsx
- simulation_params.txt
- simulation_results.txt
- simulation_results_routing1.txt
- simulation_results_routing2_shuffle_OFF.txt
- simulation_results_routing2_shuffle_ON.txt

(Για την καλύτερη δυνατή εμπειρία ανάγνωσης του report.pdf συνίσταται η χρήση 27άρας οθόνης 1440p)

Δομή συμπλέγματος εξυπηρετητών προς προσομοίωση

Το σύμπλεγμα εξυπηρετητών που θα προσομοιώσουμε είναι μια τροποποιημένη μορφή αυτού που φαίνεται στην *εικόνα 1*.



Εικόνα 1: Ένα τυπικό σύμπλεγμα διακομιστών

Ο διανομέας (dispatcher) δέχεται νέες αιτήσεις και τις προωθεί στους εξυπηρετητές εργάτες για εξυπηρέτηση.
Κάθε κόμβος διαθέτει μία (1) ουρά για εργασίες που περιμένουν εξυπηρέτηση με απεριόριστο αριθμό απο θέσεις. Τόσο ο διανομέας όσο και οι εργάτες θα αποτελούνται από μία (1) μονάδα επεξεργασίας.

Σύνθεση & χαρακτηριστικά συμπλέγματος εξυπηρετητών προς προσομοίωση

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	4 sec	Εκθετική	0	Ντετερμινιστική
2-6	Εργάτης	-		8 sec	Εκθετική
7-9	Εργάτης	-		5 sec	Εκθετική

Το σύμπλεγμα θα αποτελείται απο έναν (1) κόμβο διανομέα (dispatcher node) με id = 1 και απο οκτώ (8) κόμβους εργάτες (worker nodes). Τα nodes με id 2-6 θα αναλάβουν το ρόλο των “αργών” server παλαιότερης τεχνολογίας και τα nodes με id 7-9 θα αναλάβουν το ρόλο των “γρήγορων” server.

Σημαντικό εδώ να αναφέρουμε πως οι μέσοι χρόνοι που παρουσιάζονται στον *πίνακα 1* μεταφράζονται στα αντίστοιχα rates των κατανομών, παράμετροι τις οποίες θα δώσουμε στον προσομοιωτή μας. Ειδικότερα: εκθετική κατανομή με $\mu = 4$ σημαίνει $rate = 0.25$, με $\mu = 8$ σημαίνει $rate = 0.125$, και με $\mu = 5$ σημαίνει $rate = 0.2$.

network_parameters.yml

Τα παραπάνω μεταφράζονται στο αρχείο *network_parameters.yml* το οποίο αποτελεί και την είσοδο που δίνουμε στην ciw έτσι ώστε να δημιουργήσει το δίκτυο που επιθυμούμε. Αναφέρουμε πως το routing table είναι γεμάτο μηδενικά καθώς εμείς διαλέγουμε με συγκεκριμένο τρόπο τον επόμενο node με τον ανάλογο αλγόριθμο εξισορρόπησης φορτίου.

(Το routing table είναι μεγέθους 9x9 καθώς έχουμε 9 servers που συνδέονται μεταξύ τους)

```
arrival_distributions:
- - Exponential
  - 0.25
- - NoArrivals
- - NoArrivals
- - NoArrivals
- - NoArrivals
- - NoArrivals
- - NoArrivals
- - NoArrivals
- - NoArrivals
```

```
service_distributions:
- - Deterministic
  - 0.0
- - Exponential
  - 0.125
- - Exponential
  - 0.125
- - Exponential
  - 0.125
- - Exponential
  - 0.125
- - Exponential
  - 0.125
- - Exponential
  - 0.2
- - Exponential
  - 0.2
- - Exponential
  - 0.2
```

```
routing:
- - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
  - 0.0
```

```
number_of_servers:
- 1
- 1
- 1
- 1
- 1
- 1
- 1
- 1
- 1
```

```
queue_capacities:
- "Inf"
- "Inf"
- "Inf"
- "Inf"
- "Inf"
- "Inf"
- "Inf"
- "Inf"
- "Inf"
```

simulation_parameters.txt

Οι παράμετροι της προσομοίωσής μας δίνονται ως είσοδο στον προσομοιωτή από το αρχείο *simulation_parameters.txt.*

simulation_params.txt

```
1   # sim time in seconds - final sim time will include 1hr warmup and 1 hour cooldown time
2   # RoutingDecision1 or RoutingDecision2
3   # list of thresholds d - used only for RoutingDecision2 - e.x -256, -128, -64, -32, -16,
4   # Maximum number of trials in case confidence interval fails
5
6   86400
7   RoutingDecision1
8   -1, 0, 1
9   50
```

Line 6 - sim time: Η διάρκεια της προσομοίωσης σε δευτερόλεπτα. Αυτόματα σε αυτή τη διάρκεια θα προστεθεί μία ώρα warmup και μία ώρα cooldown.
Line 7 - routing decision: Εδώ επιλέγουμε εάν θέλουμε να προσομοιώσουμε τον αλγόριθμο εξισορρόπησης φορτίου 1 ή 2 *(RoutingDecision1 και RoutingDecision2 αντίστοιχα)*
Line 8 - thresholds: Η λίστα με τις τιμές του d τις οποίες θέλουμε να χρησιμοποιήσει ο RoutingDecision2
Line 9 - max trials: Ο μέγιστος αριθμός απο trials/runs που θα εκτελέσει ο προσομοιωτής μας

Ψευδοκώδικας αλγόριθμου εξισορρόπησης φορτίου 1

Κάθε φορά που λαμβάνει μια νέα αίτηση, ο διανομέας χρησιμοποιεί έναν αλγόριθμο εξισορρόπησης φορτίου ώστε να προωθήσει την αίτηση σε έναν εξυπηρετητή εργάτη. Ο Αλγόριθμος 1 που διακρίνεται στον *πίνακα 2* επιλέγει τον λιγότερο απασχολούμενο κόμβο χωρίς να λαμβάνει υπόψιν την ετερογένεια στον ρυθμό επεξεργασίας αιτήσεων.

Πίνακας 2: Ψευδοκώδικας για τον αλγόριθμο εξισορρόπησης φορτίου 1

```
Input: Nodes[]
Output: Node ID

min = Nodes[2]

for i in 3 to 9 do
    if Nodes[i].number_of_customers < min
        min = Nodes[i]
    endif
done
return min
```

Υλοποίηση αλγόριθμου εξισορρόπησης φορτίου 1

Υλοποιήσαμε τον ίδιο αλγόριθμο με 2 διαφορετικούς τρόπους για επιβεβαίωση. Εφεξής όπου αναφερόμαστε στον αλγόριθμο εξισορρόπησης φορτίου 1 θα εννοούμε το *RoutingDecision1()*

```
class RoutingDecision0(ciw.Node):
    def next_node(self, ind):
        nodes = self.simulation.nodes

        min_node = nodes[2]
        for i in range (3, 10):
            if nodes[i].number_of_individuals < min_node.number_of_individuals: min_node = nodes[i]

        return min_node

class RoutingDecision1(ciw.Node):
    def next_node(self, ind):
        busyness = {nodeID: self.simulation.nodes[nodeID].number_of_individuals for nodeID in range(2, 10)}
        chosen_nodeID = sorted(busyness.keys(), key = lambda id: busyness[id])[0]

        return self.simulation.nodes[chosen_nodeID]
```

Ψευδοκώδικας αλγόριθμου εξισορρόπησης φορτίου 2

Ο Αλγόριθμος 2 *(πίνακας 3)* επιλέγει τον λιγότερο απασχολούμενο κόμβο λαμβάνοντας υπόψιν την ετερογένεια στο ρυθμό επεξεργασίας. Ο αλγόριθμος αρχικά ταξινομεί σε μια λίστα τους κόμβους ως προς το βαθμό απασχόλησής τους (αριθμός αιτήσεων). Στη συνέχεια, αν ο πρώτος κόμβος στην ταξινομημένη λίστα είναι γρήγορος, τότε επιστρέφει αμέσως αυτόν τον κόμβο. Διαφορετικά, εξετάζει τους επόμενους κόμβους μέχρι να βρει έναν γρήγορο κόμβο. Αν ο αριθμός εργασιών στο γρήγορο κόμβο είναι έως και d παραπάνω από τις αντίστοιχες εργασίες του πρώτου κόμβου στη λίστα, τότε ο αλγόριθμος επιστρέφει τον γρήγορο κόμβο. Διαφορετικά, επιστρέφει τον πρώτο κόμβο στην ταξινομημένη λίστα.

Πίνακας 3: Ψευδοκώδικας για τον αλγόριθμο εξισορρόπησης φορτίου 2

```
Input: Nodes[]
Output: Node

for node_id in 2 to 9 do
    Busyness[node_id-2].size = Nodes[node_id].number_of_customers
    Busyness[node_id-2].id = node_id
done
sorted_nodes = sort(Busyness by size)
if sorted_nodes[0].id > 6 then              # 7-9: Fast node
    return Nodes[sorted_nodes[0].id]
else
    for i from 1 to 19 do
        if sorted_nodes[i].id > 6 then
            if sorted_nodes[i].size – d > sorted_nodes[0].size then
                return Nodes[sorted_nodes[i].id]
            else
                break
            endif
         endif
    done
    return Nodes[sorted_nodes[0].id]
endif
```

Υλοποίηση αλγόριθμου εξισορρόπησης φορτίου 2

```
class RoutingDecision2(ciw.Node):
    def next_node(self, ind):
        nodes = self.simulation.nodes
        busyness = {nodeID: self.simulation.nodes[nodeID].number_of_individuals for nodeID in range(2, 10)}
        sorted_nodeIDs_by_busyness = sorted(busyness.keys(), key = lambda id: busyness[id])

        #random.shuffle(sorted_nodeIDs_by_busyness)

        if sorted_nodeIDs_by_busyness[0] > 6: # fast node
            return nodes[sorted_nodeIDs_by_busyness[0]]
        else:
            for i in range (1, 8):
                if sorted_nodeIDs_by_busyness[i] > 6:
                    if nodes[sorted_nodeIDs_by_busyness[i]].number_of_individuals - threshold >
nodes[sorted_nodeIDs_by_busyness[0]].number_of_individuals:
                        return nodes[sorted_nodeIDs_by_busyness[i]]
                    else:
                        break
            return nodes[sorted_nodeIDs_by_busyness[0]]
```

Η χρησιμότητα της μεθόδου random.shuffle(sorted_nodeIDs_by_busyness) βρίσκεται στη σελίδα ανάλυσης των αποτελεσμάτων της προσομοίωσης.

Overview κώδικα προσομοιωτή & script αυτοματοποίησης

```
 9   global min_wait_time
10   min_wait_time = float("inf")
11   global min_wait_time_threshold
12   min_wait_time_threshold = -1
13
14   global max_util
15   max_util = float("-inf")
16   global max_util_threshold
17   max_util_threshold = -1
18
19 > class RoutingDecision0(ciw.Node):···
28
29 > class RoutingDecision1(ciw.Node):···
35
36 > class RoutingDecision2(ciw.Node):···
64
65 > def getClass(classname):···
67
68 > def calculate_standard_deviation(elapsedTrials, avg_waits, avg_wait_time):
79
80 > def calculate_conf_interval(ci, elapsedTrials, sd_wait, avg_wait_time):···
109
110 > def simulation():···
213
214 > def finalizeSimulationResults():···
```

simulation.py > ...

```
1   import ciw
2   import parsefiles
3   import sys
4   import numpy
5   import math
6   from scipy import stats
7   import random
```

```
240  print('Simulation started...')
241
242  try:
243      original_stdout = sys.stdout # Save a reference to the original standard output
244      sys.stdout = open('simulation_results.txt', 'w')
245  except FileNotFoundError as e:
246      print('Cannot create simulation_results.txt. Quitting...')
247      sys.exit()
248
249  sim_time, algo, thresholds, max_trials =
250  parsefiles.parseSimulationParameters('simulation_params.txt')
251
252  #print(sim_time, algo, thresholds, max_trials)
253  print('\nSimulation Parameters:')
254  print('Simulation time incl. 1hr warmup and  1hr cooldown (seconds):', sim_time)
255  print('Load balance algo:', algo)
256  print('Threshold values:', thresholds)
257  print('Maximum number of trials:', max_trials)
258
259  print('\nSimulation results:')
260  if algo == 'RoutingDecision1': simulation()
261
262  if algo == 'RoutingDecision2':
263      for d in thresholds:
264          global threshold
265          threshold = d
266          #ciw.seed(threshold) # this is wrong here
267
268          simulation()
269
270      print('\nMin wait time (ms):', "{:.3f}".format(min_wait_time * 1000))
271      print('Threshold with min wait time: d =', min_wait_time_threshold)
272
273      print('\nMax util (percentage):', "{:.3f}".format(max_util * 100))
274      print('Threshold with max util: d =', max_util_threshold)
275
276  sys.stdout = original_stdout # Reset the standard output to its original value
277  finalizeSimulationResults()
278  print('Simulation completed successfully.')
```

Seed προσομοίωσης

Η συνάρτηση *ciw.seed()* ορίζει ένα σπόρο για την αρχικοποίηση της γεννήτριας τυχαίων αριθμών και είναι ζωτικής σημασίας για την επαναληψιμότητα και την επιβεβαίωση των αποτελεσμάτων.
Είναι σημαντικό σε κάθε επανάληψη της προσομοίωσης να χρησιμοποιείται διαφορετικός σπόρος.
Η υλοποίηση της seed βρίσκεται μέσα στη μέθοδο *simulation() στη γραμμή 110 του simulation.py.*
Ακολουθώντας τις οδηγίες απο το tutorial της ciw υλοποιήσαμε με τον ίδιο τρόπο τη δική μας *ciw.seed()*.

```
def simulation():
    global moreRunsNeeded
    moreRunsNeeded = True

    for trial in range(max_trials):
        ciw.seed(trial)

        #print('\nTrial %i' % trial)

        if not moreRunsNeeded == True: break

        global Q
        Q = ciw.Simulation(
            ciw.create_network_from_yml('network_params.yml'), tracker = ciw.trackers.NodePopulation(),
            node_class = [getClass(algo), ciw.Node, ciw.Node, ciw.Node, ciw.Node, ciw.Node, ciw.Node, ciw.Node, ciw.Node])
        Q.simulate_until_max_time(sim_time) # 1hr warmup, sim_time hr data capture, 1hr cooldown
```

Προσομοίωση

Ο προσομοιωτής που υλοποιήσαμε δέχεται ως είσοδο το είδος του αλγορίθμου εξισορρόπησης φορτίου, την παράμετρο d, και τη διάρκεια της προσομοίωσης σε δευτερόλεπτα *(αρχείο simulation_parameters.txt)*. Επιπλέον, δέχεται τα χαρακτηριστικά του συμπλέγματος εξυπηρετητών, όπως περιγράφονται στον *πίνακα 1 (αρχείο network_parameters.yml)*.

Η προσομοίωση του συστήματος εκτελείται για χρόνο ίσο με 24 ώρες (86400 sec). Αυτόματα σε αυτή τη διάρκεια προστίθεται μία ώρα warmup και μία ώρα cooldown.

Χρησιμοποιώντας τη μέθοδο των ανεξάρτητων επαναλήψεων βρίσκουμε το διάστημα εμπιστοσύνης για το μέσο χρονο αναμονής με επίπεδο εμπιστοσύνης 90%, 95% ή 99% και διακόπτουμε την προσομοίωση αναλόγως. Ως μέγιστος αριθμός επαναλήψεων ορίζεται ο αριθμός 50 *(max_trials)*.

Έξοδος προσομοίωσης

Ο προσομοιωτής μας υπολογίζει τις ακόλουθες μετρικές:

- μέσο χρόνο αναμονής *(wait time)*
- μέσο ρυθμό εξυπηρέτησης αιτήσεων *(service rate)*
- μέση συνολική χρησιμοποίηση κόμβων *(total node util)*
- μέση χρησιμοποίηση για κάθε κόμβο ξεχωριστά *(node util)*

Επιπλέον δημιουργεί ένα αρχείο στο δίσκο ονόματι *simulation_results.txt* στο οποίο παρουσιάζονται τα αποτελέσματα της προσομοίωσης. Το αρχείο αυτό χωρίζεται σε 3 ενότητες:

- περιγραφή του συστήματος *(system description)*
- παράμετροι εισόδου *(simulation parameters)*
- αποτελέσματα εξόδου *(simulation results)*

Στις επόμενες σελίδες παρουσιάζεται αναλυτικά το αρχείο εξόδου.

simulation_results.txt

Παρακάτω παρουσιάζονται αντιπροσωπευτικά αρχεία εξόδου για μια προσομοίωση του συμπλέγματος εξυπηρετητών μας στην οποία χρησιμοποιείται ο *RoutingDecision1* και ο *RoutingDecision2*.

```
System Description:

Node network: [Arrival Node, Node 1, Node 2, Node 3, Node 4, Node 5, Node 6, Node 7, Node 8, Node 9, Exit Node]

Transitive node network: [Node 1, Node 2, Node 3, Node 4, Node 5, Node 6, Node 7, Node 8, Node 9]

Arrival distributions and rates for each transitive node and customer class:
{1: {0: Exponential: 0.25},
 2: {0: NoArrivals}, 3: {0: NoArrivals}, 4: {0: NoArrivals}, 5: {0: NoArrivals}, 6: {0: NoArrivals},
 7: {0: NoArrivals}, 8: {0: NoArrivals}, 9: {0: NoArrivals}}

Service distributions and rates for each transitive node and customer class:
{1: {0: Deterministic: 0.0},
 2: {0: Exponential: 0.125}, 3: {0: Exponential: 0.125}, 4: {0: Exponential: 0.125}, 5: {0: Exponential: 0.125}, 6: {0: Exponential: 0.125},
 7: {0: Exponential: 0.2}, 8: {0: Exponential: 0.2}, 9: {0: Exponential: 0.2}}

Number of servers per transitive node:
[Node 1: 1] [Node 2: 1] [Node 3: 1] [Node 4: 1] [Node 5: 1] [Node 6: 1] [Node 7: 1] [Node 8: 1] [Node 9: 1]

Queueing capacity per transitive node:
[Node 1: inf] [Node 2: inf] [Node 3: inf] [Node 4: inf] [Node 5: inf] [Node 6: inf] [Node 7: inf] [Node 8: inf] [Node 9: inf]

Routing table:
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
[[0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]]
```

```
Simulation Parameters:

Simulation time incl. 1hr warmup and  1hr cooldown (seconds): 93600
Load balance algo: RoutingDecision1
Threshold values: [-256, -128, -64, -32, -16, -8, -4, -2, -1, 0, 1, 2, 4, 8, 16, 32, 64, 128, 256]
Maximum number of trials: 50

Simulation results:

////////////////////
Algo: RoutingDecision1
Avg wait time(ms): 2.208
Avg service rate(services/second): 25.353
Avg total node util(percentage): 21.922
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '66.697', Node 3: '53.407', Node 4: '37.921', Node 5: '22.946', Node 6: '11.778',
                            Node 7: '3.271', Node 8: '1.016', Node 9: '0.264'}
////////////////////
```

RoutingDesicion1

```
Simulation Parameters:
Simulation time incl. 1hr warmup and  1hr cooldown (seconds): 93600
Load balance algo: RoutingDecision2
Threshold values: [0, 1]
Maximum number of trials: 50

Simulation results:

////////////////////
Algo: RoutingDecision2
Threshold value: 0
Avg wait time(ms): 3.520
Avg service rate(services/second): 25.358
Avg total node util(percentage): 21.916
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '66.677', Node 3: '53.383', Node 4: '37.898', Node 5: '22.925', Node 6: '11.757',
                            Node 7: '3.306', Node 8: '1.034', Node 9: '0.269'}
////////////////////

////////////////////
Algo: RoutingDecision2
Threshold value: 1
Avg wait time(ms): 2.208
Avg service rate(services/second): 25.353
Avg total node util(percentage): 21.922
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '66.697', Node 3: '53.407', Node 4: '37.921', Node 5: '22.946', Node 6: '11.778',
                            Node 7: '3.271', Node 8: '1.016', Node 9: '0.264'}
////////////////////
Min wait time (ms): 2.208
Threshold with min wait time: d = 1

Max util (percentage): 21.922
Threshold with max util: d = 1
```

RoutingDecision2

random.shuffle*(sorted_nodeIDs_by_busyness*)

Στη διάρκεια των πειραμάτων μας αναρωτηθήκαμε τι επίδραση θα υπήρχε στα wait times, στο service rate, καθώς και στη χρησιμοποίηση των κόμβων εαν στον αλγόριθμο δρομολόγησης 2 *(RoutingDecision2)* δεν λαμβάνονταν υπ' όψιν η ταξινομημένη λίστα του busyness των κόμβων αλλά ένα τυχαίο permutation αυτής.

Έτσι προσθέσαμε τη γραμμή 47 στο αρχείο *simulation.py*.

Συνεπώς στις επόμενες σελίδες θα παρουσιάσουμε τόσο τα αποτελέσματα για την κανονική υλοποίηση του αλγορίθμου *(random.shuffle = OFF)* όσο και αυτά για την τροποποιημένη μορφή του *(random.shuffle = ON)*

Ερώτημα 1

Για τον Αλγόριθμο εξισορρόπησης φορτίου 2, ποια τιμή του d δίνει τον μικρότερο μέσο χρόνο αναμονής; Ποια τιμή του d δίνει την υψηλότερη μέση χρησιμοποίηση;

Για να απαντήσουμε σε αυτό το ερώτημα, τρέχουμε το σενάριο αυτοματοποίησής μας με την εξής είσοδο:

```
Simulation Parameters:
Simulation time incl. 1hr warmup and  1hr cooldown (seconds): 93600
Load balance algo: RoutingDecision2
Threshold values: [-256, -128, -64, -32, -16, -8, -4, -2, -1, 0, 1, 2, 4, 8, 16, 32, 64, 128, 256]
Maximum number of trials: 50
```

Η έξοδος για αυτό το ερώτημα βρίσκεται στα αρχεία *simulation_results_routing2_shuffle_OFF.txt* και *simulation_results_routing2_shuffle_ON*.txt

simulation_results_routing2_shuffle_OFF.txt

```
////////////////////
Algo: RoutingDecision2
Threshold value: -1
Avg wait time(ms): 372.693
Avg service rate(services/second): 40.045
Avg total node util(percentage): 13.877
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '0.000', Node 3: '0.000', Node 4: '0.000', Node 5: '0.000', Node 6: '0.000',
                            Node 7: '60.203', Node 8: '41.265', Node 9: '23.429'}
////////////////////

////////////////////
Algo: RoutingDecision2
Threshold value: 0
Avg wait time(ms): 3.520
Avg service rate(services/second): 25.358
Avg total node util(percentage): 21.916
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '66.677', Node 3: '53.383', Node 4: '37.898', Node 5: '22.925', Node 6: '11.757',
                            Node 7: '3.306', Node 8: '1.034', Node 9: '0.269'}
////////////////////

////////////////////
Algo: RoutingDecision2
Threshold value: 1
Avg wait time(ms): 2.208
Avg service rate(services/second): 25.353
Avg total node util(percentage): 21.922
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '66.697', Node 3: '53.407', Node 4: '37.921', Node 5: '22.946', Node 6: '11.778',
                            Node 7: '3.271', Node 8: '1.016', Node 9: '0.264'}
////////////////////

Min wait time (ms): 2.208
Threshold with min wait time: d = 1

Max util (percentage): 21.922
Threshold with max util: d = 1
```

Στο αρχείο εξόδου με shuffle = OFF παρατηρούμε τα εξής:

- Για οποιαδήποτε τιμή d <= 1 τα αποτελέσματα είναι ακριβώς τα ίδια μεταξύ τους.
- Για οποιαδήποτε τιμή d >= 1 τα αποτελέσματα είναι ακριβώς τα ίδια μεταξύ τους.

Για d αρνητικό παρατηρείται συνολικά μεγαλύτερο wait time *(με διαφορά τάξης μεγέθους από αυτό που παρατηρείται σε d >= 0)* το οποίο συνδυάζεται όμως με αισθητά αυξημένο service rate αλλά και με μικρότερη συνολική χρησιμοποίηση κόμβων. Κοιτάζοντας τον κάθε κόμβο ξεχωριστά παρατηρούμε πως ολόκληρη η χρησιμοποίηση περιορίζεται στους κόμβους 7,8,9 (οι γρήγοροι σέρβερ στο σύμπλεγμα) ενώ οι υπόλοιποι δεν χρησιμοποιούνται καθόλου.

Για d μεγαλύτερο ίσο του μηδενός παρατηρείται αισθητά μειωμένο wait time *(με διαφορά τάξης μεγέθους από αυτό που παρατηρείται σε d < 0)* το οποίο συνδυάζεται με μειωμένο service rate αλλά και με μεγαλύτερη συνολική χρησιμοποίηση κόμβων. Κοιτάζοντας τον κάθε κόμβο ξεχωριστά παρατηρούμε πως οι κόμβοι 2-6 είναι αυτοί που χρησιμοποιούνται περισσότερο και οι κόμβοι 7,8,9 δε χρησιμοποιούνται σχεδόν καθόλου.

Συνολικά για τις τιμές του d = [-256, -128, -64, -32, -16, -8, -4, -2, -1, 0, 1, 2, 4, 8, 16, 32, 64, 128, 256] παρατηρούμε πως η τιμή του d που δίνει το μικρότερο μέσο χρόνο αναμονής και ταυτόχρονα την υψηλότερη μέση χρησιμοποίηση είναι μια τιμή d >= 1 *(αρχεία εξόδου εδώ).*

simulation_results_routing2_shuffle_ON.txt

```
////////////////////
Algo: RoutingDecision2
Threshold value: -1
Avg wait time(ms): 1775.190
Avg service rate(services/second): 40.027
Avg total node util(percentage): 13.911
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '0.000', Node 3: '0.000', Node 4: '0.000', Node 5: '0.000', Node 6: '0.000',
                            Node 7: '41.630', Node 8: '41.707', Node 9: '41.862'}
////////////////////

////////////////////
Algo: RoutingDecision2
Threshold value: 0
Avg wait time(ms): 1436.310
Avg service rate(services/second): 30.598
Avg total node util(percentage): 18.144
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '20.480', Node 3: '20.630', Node 4: '20.628', Node 5: '20.595', Node 6: '20.493',
                            Node 7: '20.141', Node 8: '20.210', Node 9: '20.118'}
////////////////////

////////////////////
Algo: RoutingDecision2
Threshold value: 1
Avg wait time(ms): 1111.522
Avg service rate(services/second): 29.368
Avg total node util(percentage): 18.898
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '24.289', Node 3: '24.185', Node 4: '24.283', Node 5: '24.277', Node 6: '24.112',
                            Node 7: '16.386', Node 8: '16.262', Node 9: '16.291'}
////////////////////
```

```
Min wait time (ms): 998.323
Threshold with min wait time: d = 8

Max util (percentage): 19.047
Threshold with max util: d = 4
```

```
////////////////////
Algo: RoutingDecision2
Threshold value: 4
Avg wait time(ms): 999.396
Avg service rate(services/second): 29.138
Avg total node util(percentage): 19.047
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '24.974', Node 3: '24.886', Node 4: '24.953', Node 5: '24.844', Node 6: '24.902',
                            Node 7: '15.703', Node 8: '15.552', Node 9: '15.612'}
////////////////////

////////////////////
Algo: RoutingDecision2
Threshold value: 8
Avg wait time(ms): 998.323
Avg service rate(services/second): 29.138
Avg total node util(percentage): 19.047
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '24.973', Node 3: '24.892', Node 4: '24.952', Node 5: '24.846', Node 6: '24.904',
                            Node 7: '15.700', Node 8: '15.550', Node 9: '15.604'}
////////////////////

Min wait time (ms): 998.323
Threshold with min wait time: d = 8

Max util (percentage): 19.047
Threshold with max util: d = 4
```

Στο αρχείο εξόδου με shuffle = ON παρατηρούμε τα εξής:

- Για οποιαδήποτε τιμή d <= 1 τα αποτελέσματα είναι ακριβώς τα ίδια μεταξύ τους.
- Για οποιαδήποτε τιμή d >= 1 τα αποτελέσματα είναι σχεδόν τα ίδια μεταξύ τους.

Για d αρνητικό παρατηρείται συνολικά μεγαλύτερο wait time το οποίο συνδυάζεται όμως με αυξημένο service rate αλλά και μικρότερη συνολική χρησιμοποίηση κόμβων. Κοιτάζοντας τον κάθε κόμβο ξεχωριστά παρατηρούμε πως ολόκληρη η χρησιμοποίηση περιορίζεται στους κόμβους 7,8,9 (οι γρήγοροι σέρβερ στο σύμπλεγμα) ενώ οι υπόλοιποι δεν χρησιμοποιούνται καθόλου.

Για d μεγαλύτερο ίσο του μηδενός παρατηρείται μειωμένο wait time το οποίο συνδυάζεται με μειωμένο service rate αλλά και με μεγαλύτερη συνολική χρησιμοποίηση κόμβων. Κοιτάζοντας τον κάθε κόμβο ξεχωριστά παρατηρούμε πως όλοι οι κόμβοι έχουν σχεδόν την ίδια χρησιμοποίηση. Ο φόρτος φαίνεται να είναι ισομερώς κατανεμημένος, κάτι το οποίο πιστεύουμε πως είναι αναμενόμενο μιας και ο αλγόριθμος σε κάθε του “σκέψη” για επιλογή επόμενου κόμβου παίρνει ένα τυχαίο permutation της λίστας όλων των κόμβων, άρα για μεγάλο αριθμό επαναλήψεων αυτής της σκέψης είναι λογικό να παρατηρηθεί σχεδόν η ίδια χρησιμοποίηση σε όλους τους κόμβους, γιατί έτσι λειτουργεί και η μέθοδος shuffle() της python.

Συνολικά για τις τιμές του d = [-256, -128, -64, -32, -16, -8, -4, -2, -1, 0, 1, 2, 4, 8, 16, 32, 64, 128, 256] παρατηρούμε πως η τιμή του d που δίνει το μικρότερο μέσο χρόνο αναμονής είναι η τιμή d = 8 και η τιμή που δίνει την υψηλότερη μέση χρησιμοποίηση είναι η τιμή d = 4 *(αρχεία εξόδου εδώ).*

Ερώτημα 2

Συγκρίνοντας στατιστικά τους αλγορίθμους 1 και 2, μπορεί ο αλγόριθμος 2 να πετύχει χαμηλότερο μέσο χρόνο αναμονής και υψηλότερη χρησιμοποίηση των εξυπηρετητών από τον αλγόριθμο 1;

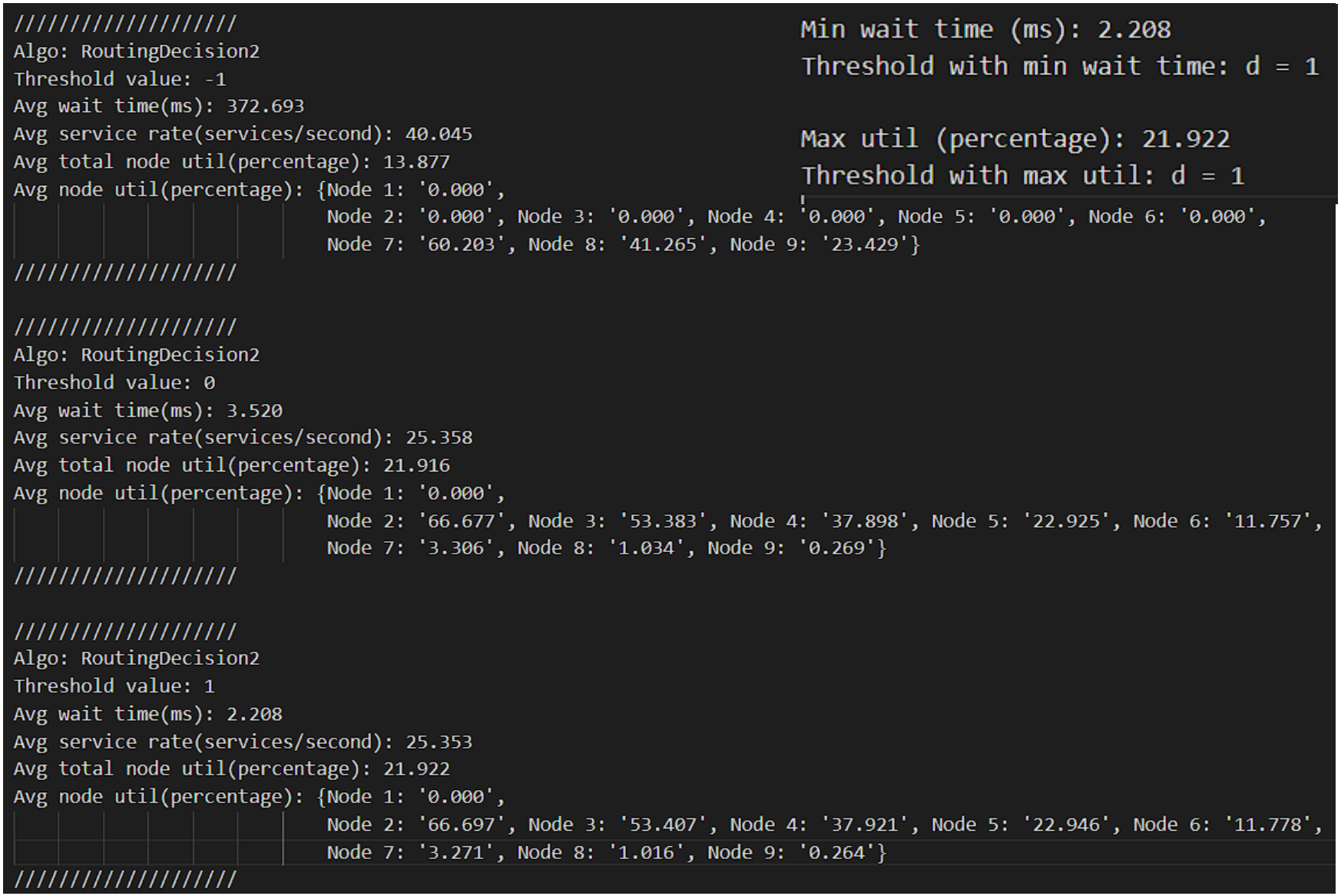
```
Simulation Parameters:

Simulation time incl. 1hr warmup and  1hr cooldown (seconds): 93600
Load balance algo: RoutingDecision1
Threshold values: [-256, -128, -64, -32, -16, -8, -4, -2, -1, 0, 1, 2, 4, 8, 16, 32, 64, 128, 256]
Maximum number of trials: 50

Simulation results:

////////////////////
Algo: RoutingDecision1
Avg wait time(ms): 2.208
Avg service rate(services/second): 25.353
Avg total node util(percentage): 21.922
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '66.697', Node 3: '53.407', Node 4: '37.921', Node 5: '22.946', Node 6: '11.778',
                            Node 7: '3.271', Node 8: '1.016', Node 9: '0.264'}
////////////////////
```

RoutingDecision1



RoutingDecision2 - shuffle OFF

```
////////////////////
Algo: RoutingDecision2
Threshold value: 4
Avg wait time(ms): 999.396
Avg service rate(services/second): 29.138
Avg total node util(percentage): 19.047
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '24.974', Node 3: '24.886', Node 4: '24.953', Node 5: '24.844', Node 6: '24.902',
                            Node 7: '15.703', Node 8: '15.552', Node 9: '15.612'}
////////////////////

////////////////////
Algo: RoutingDecision2
Threshold value: 8
Avg wait time(ms): 998.323
Avg service rate(services/second): 29.138
Avg total node util(percentage): 19.047
Avg node util(percentage): {Node 1: '0.000',
                            Node 2: '24.973', Node 3: '24.892', Node 4: '24.952', Node 5: '24.846', Node 6: '24.904',
                            Node 7: '15.700', Node 8: '15.550', Node 9: '15.604'}
////////////////////

Min wait time (ms): 998.323
Threshold with min wait time: d = 8

Max util (percentage): 19.047
Threshold with max util: d = 4
```

RoutingDecision2 - shuffle ON

Συγκρίνοντας στατιστικά τους αλγορίθμους 1 και 2, μπορεί ο αλγόριθμος 2 να πετύχει χαμηλότερο μέσο χρόνο αναμονής και υψηλότερη χρησιμοποίηση των εξυπηρετητών από τον αλγόριθμο 1;

```
Min wait time (ms): 998.323
Threshold with min wait time: d = 8

Max util (percentage): 19.047
Threshold with max util: d = 4
```

RoutingDecision2 - shuffle ON

Συγκρίνοντας στατιστικά τους αλγορίθμους 1 και 2 παρατηρούμε πως ο αλγόριθμος 2 **δεν μπορεί** να πετύχει χαμηλότερο χρόνο αναμονής ή υψηλότερη χρησιμοποίηση των εξυπηρετητών απο τον αλγόριθμο 1. Αυτό ισχύει είτε shuffle = OFF είτε shuffle = ON.

```
Min wait time (ms): 2.208
Threshold with min wait time: d = 1

Max util (percentage): 21.922
Threshold with max util: d = 1
```

RoutingDecision2 - shuffle OFF

```
////////////////////
Algo: RoutingDecision1
Avg wait time(ms): 2.208
Avg service rate(services/second): 25.353
Avg total node util(percentage): 21.922
```

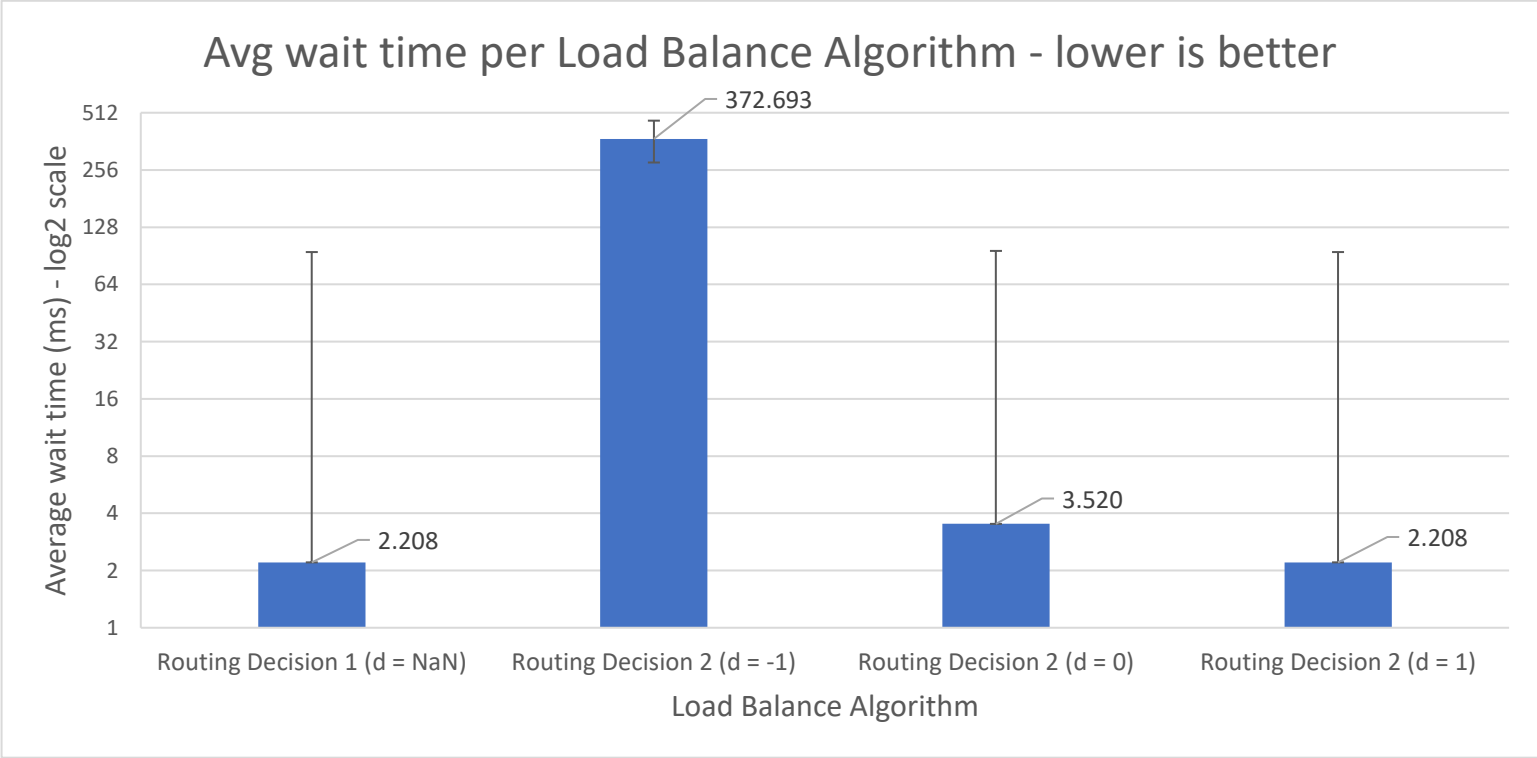
RoutingDecision1

Γραφικές παραστάσεις

Σύγκριση RoutingDecision1 & RoutingDecision2 (shuffle = OFF)

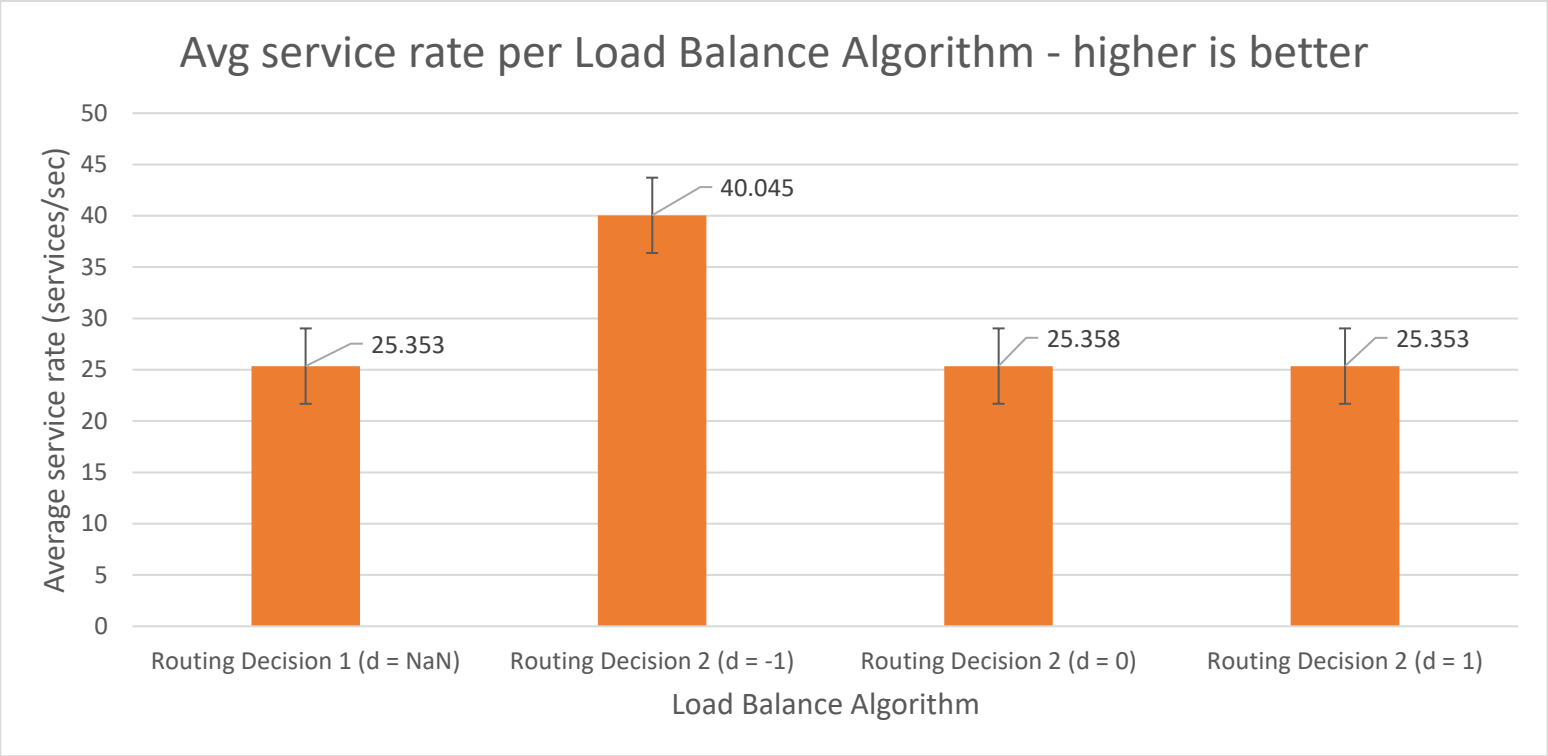
Μικρότερο μέσο χρόνο αναμονής παρουσιάζει ο αλγόριθμος 1 και ο αλγόριθμος 2 με d = 1.

Σε αυτήν την μετρική ο αλγόριθμος 2 με d = -1 έχει την χειρότερη απόδοση.



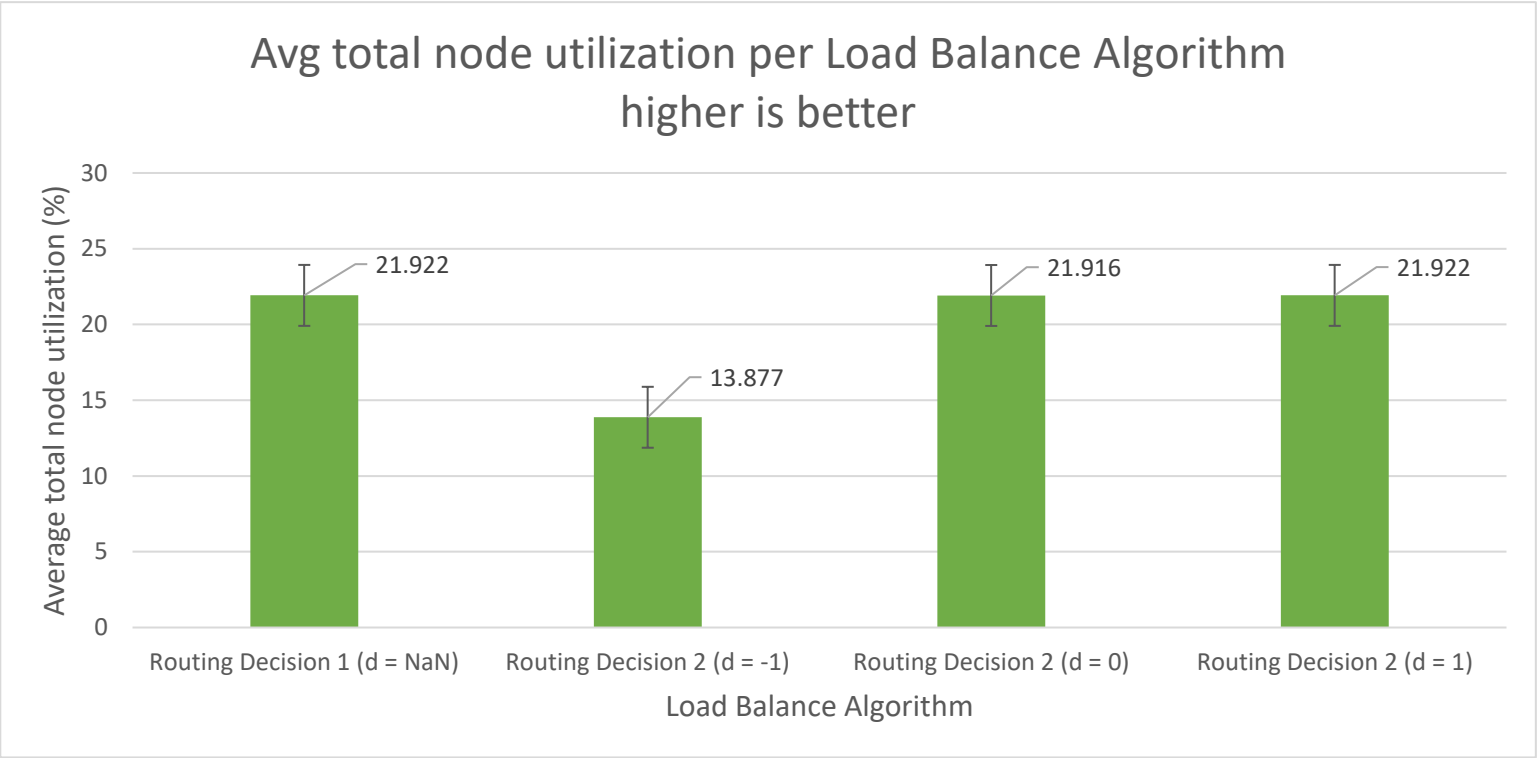
Υψηλότερο μέσο ρυθμό εξυπηρέτησης παρουσιάζει ο αλγόριθμος 2 με d = -1.

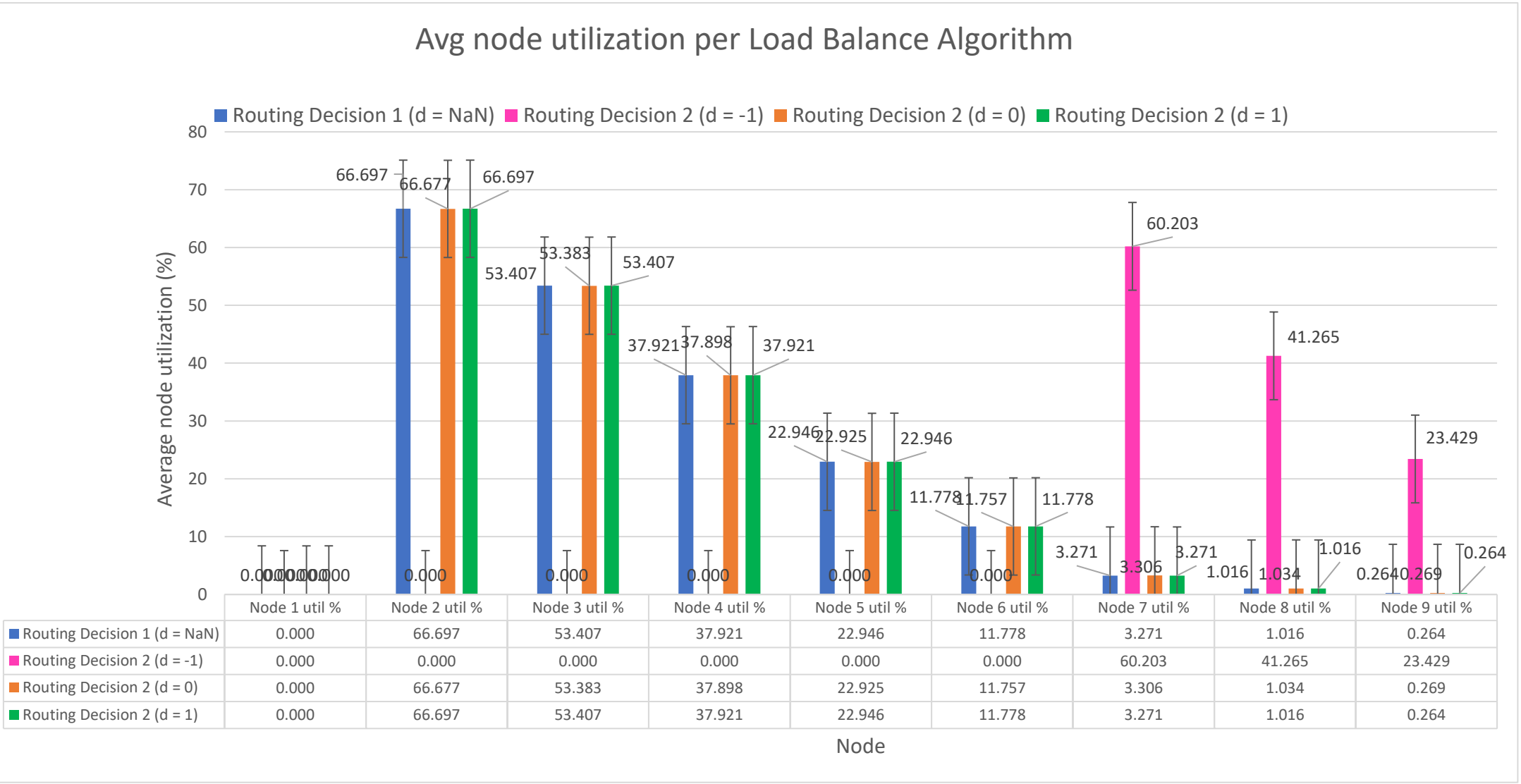
Οι υπόλοιποι αλγόριθμοι έχουν χαμηλότερες και παρόμοιες τιμές.



Υψηλότερη μέση συνολική χρησιμοποίηση κόμβων παρουσιάζει ο αλγόριθμος 1 και ο αλγόριθμος 2 με d = -1.

Ωστόσο και ο αλγόριθμος 2 με d = 0 έχει αρκετά υψηλή μέση συνολική χρησιμοποίηση κόμβων, σχεδόν ίδια με τους δυο προαναφερθέντες αλγόριθμους.





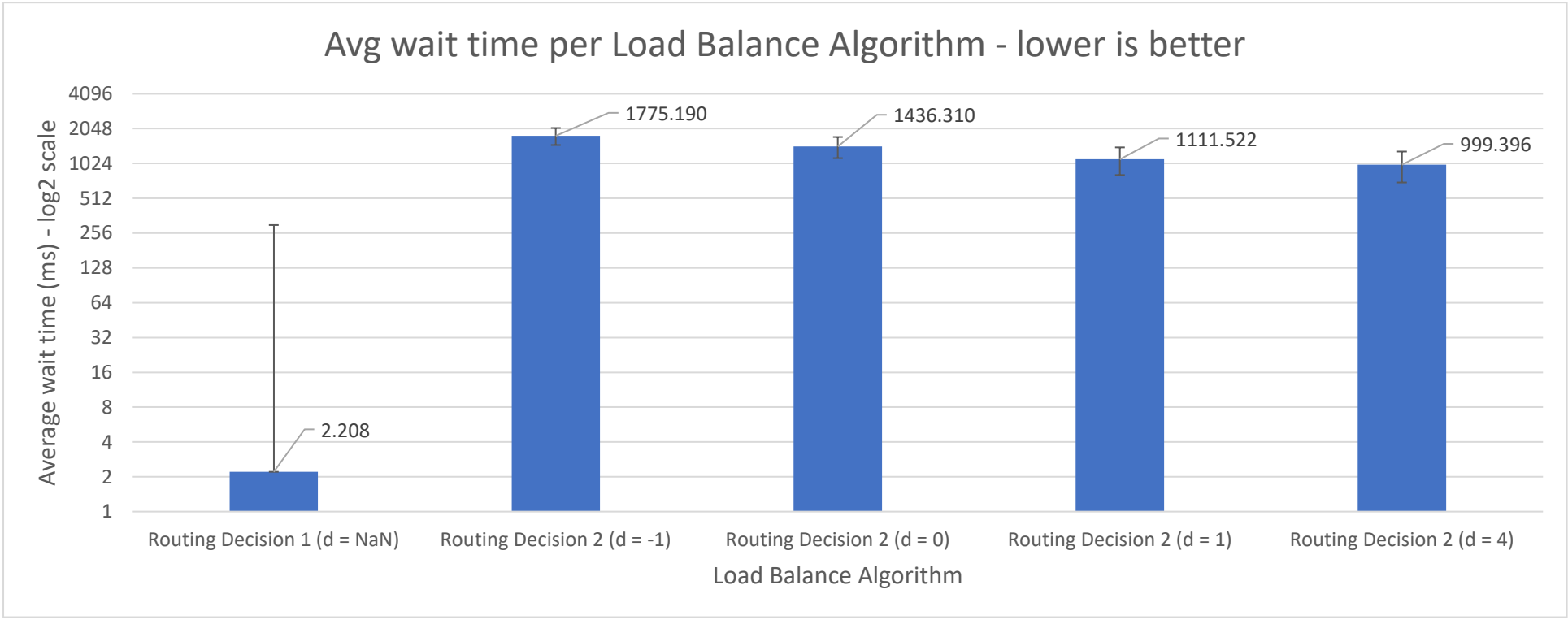
	Node 1 util %	Node 2 util %	Node 3 util %	Node 4 util %	Node 5 util %	Node 6 util %	Node 7 util %	Node 8 util %	Node 9 util %
Routing Decision 1 (d = NaN)	0.000	66.697	53.407	37.921	22.946	11.778	3.271	1.016	0.264
Routing Decision 2 (d = -1)	0.000	0.000	0.000	0.000	0.000	0.000	60.203	41.265	23.429
Routing Decision 2 (d = 0)	0.000	66.677	53.383	37.898	22.925	11.757	3.306	1.034	0.269
Routing Decision 2 (d = 1)	0.000	66.697	53.407	37.921	22.946	11.778	3.271	1.016	0.264

Παρατηρούμε ότι ο αλγόριθμος 2 με d = -1 έχει την υψηλότερη μέση χρησιμοποίηση των κόμβων 7, 8 και 9 που είναι οι γρήγοροι εργάτες. Στους υπόλοιπους κόμβους η χρησιμοποίηση είναι μηδενική.

Αντιθέτως, οι υπόλοιποι αλγόριθμοι παρουσιάζουν υψηλότερη μέση χρησιμοποίηση στον κόμβο 2, ενώ από τον κόμβο 3 έως τον 9 η χρησιμοποίηση μειώνεται. Ιδιαίτερα στους γρήγορους κόμβους η χρησιμοποίηση είναι ελάχιστη.

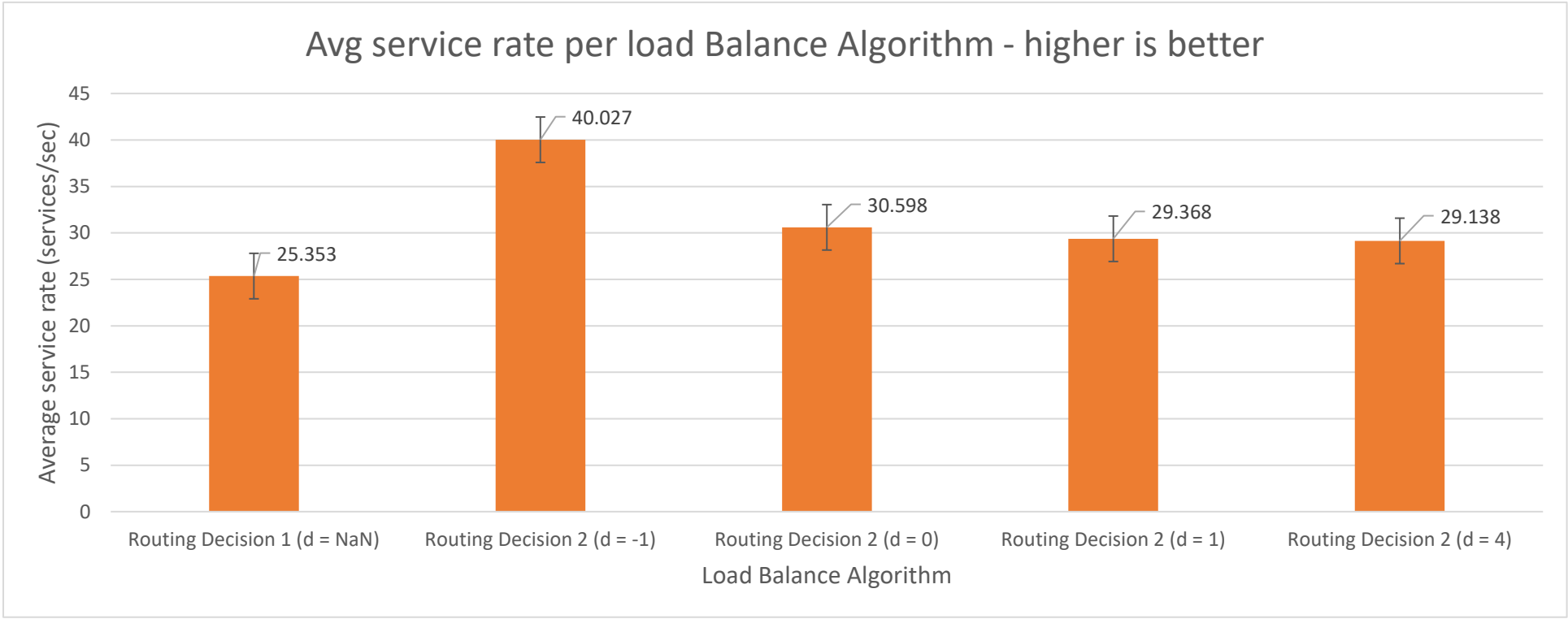
Ο κόμβος 1 είναι ο dispatcher, για αυτό είναι μηδενική η χρησιμοποίησή του από τους αλγορίθμους.

Σύγκριση RoutingDecision1 & RoutingDecision2 (shuffle = ON)

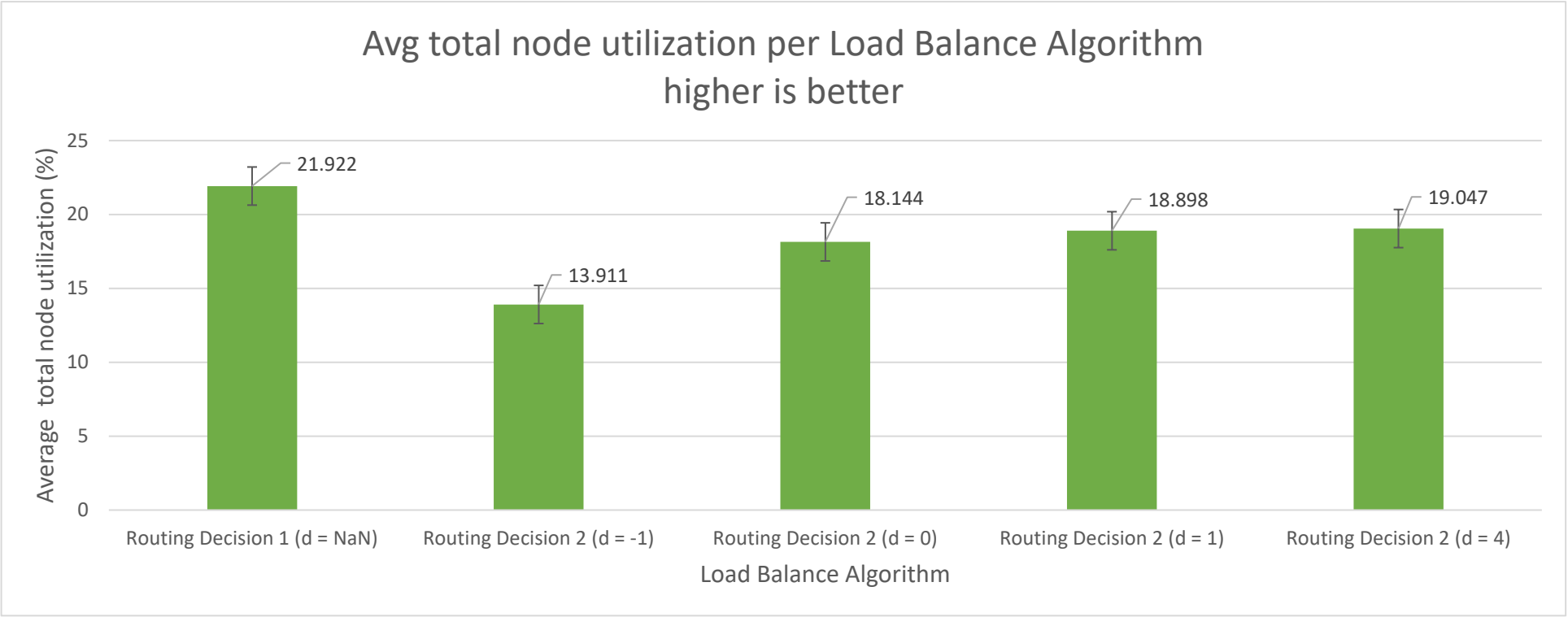


Μικρότερο μέσο χρόνο αναμονής παρουσιάζει ο αλγόριθμος 1.

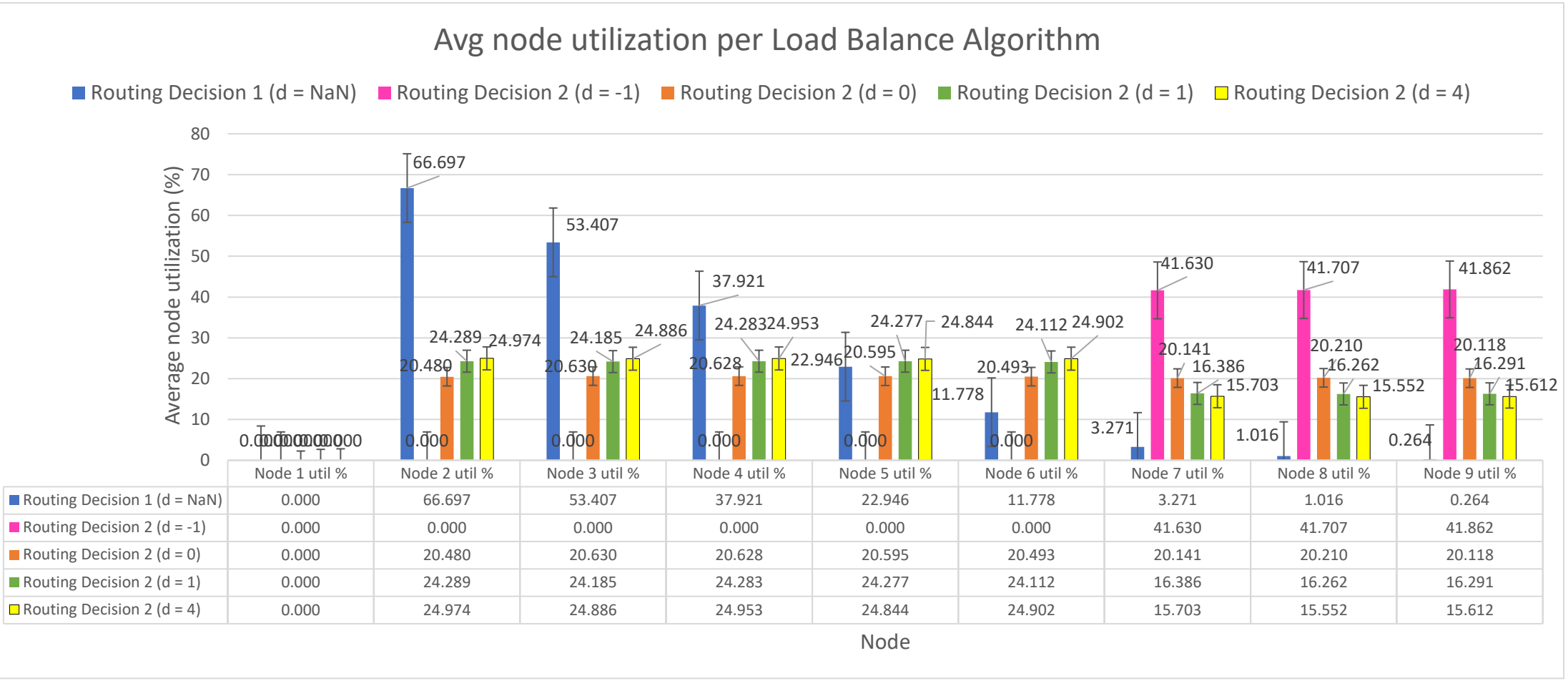
Οι αλγόριθμοι 2 με shuffle = ON και οποιοδήποτε threshold έχουν τις χειρότερες αποδόσεις.



Υψηλότερο μέσο ρυθμό εξυπηρέτησης παρουσιάζει ο αλγόριθμος 2 με d = -1.
Οι υπόλοιποι αλγόριθμοι έχουν χαμηλότερες και παρόμοιες τιμές.



Υψηλότερη μέση συνολική χρησιμοποίηση κόμβων παρουσιάζει ο αλγόριθμος 1.
Ωστόσο ο αλγόριθμος 2 με d = 0, 1 και 4 έχουν υψηλότερη μέση συνολική χρησιμοποίηση κόμβων από αυτόν με d = -1, αλλά όχι όση έχει ο αλγόριθμος 1.



	Node 1 util %	Node 2 util %	Node 3 util %	Node 4 util %	Node 5 util %	Node 6 util %	Node 7 util %	Node 8 util %	Node 9 util %
Routing Decision 1 (d = NaN)	0.000	66.697	53.407	37.921	22.946	11.778	3.271	1.016	0.264
Routing Decision 2 (d = -1)	0.000	0.000	0.000	0.000	0.000	0.000	41.630	41.707	41.862
Routing Decision 2 (d = 0)	0.000	20.480	20.630	20.628	20.595	20.493	20.141	20.210	20.118
Routing Decision 2 (d = 1)	0.000	24.289	24.185	24.283	24.277	24.112	16.386	16.262	16.291
Routing Decision 2 (d = 4)	0.000	24.974	24.886	24.953	24.844	24.902	15.703	15.552	15.612

Ο αλγόριθμος 1 έχει την υψηλότερη μέση χρησιμοποίηση στον κόμβο 2, ενώ από τον κόμβο 3 έως τον 9 η χρησιμοποίηση μειώνεται.

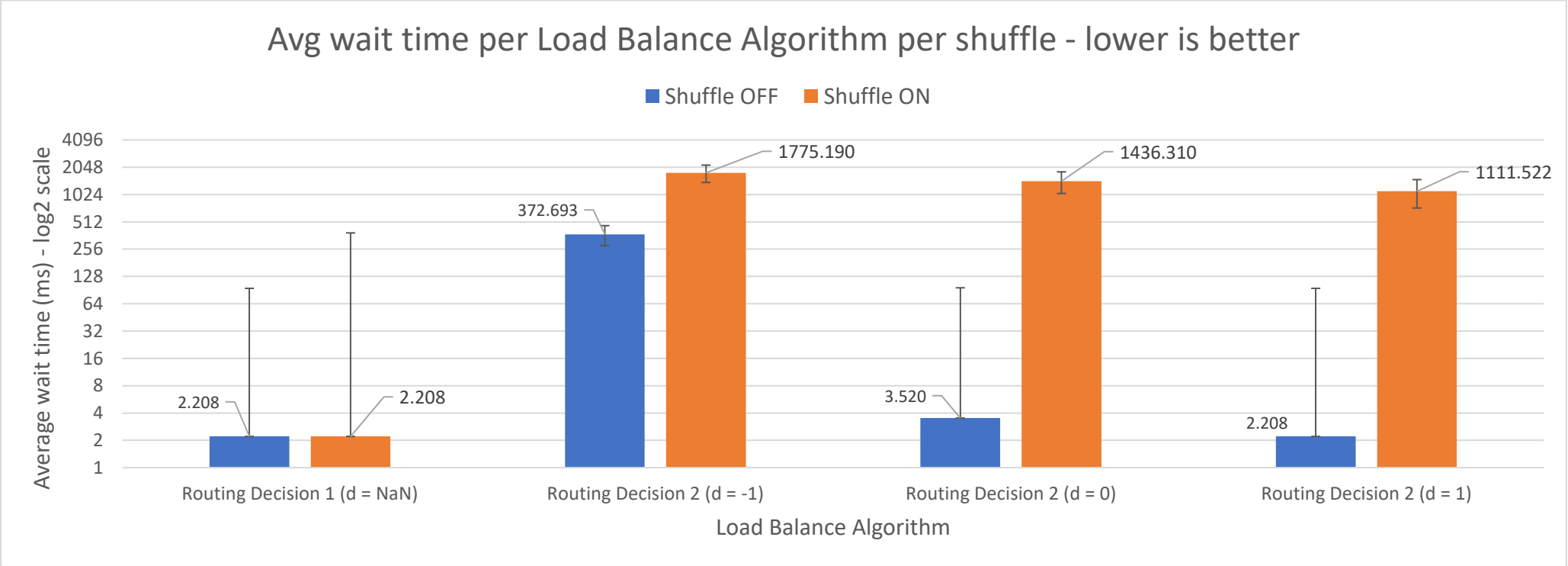
Ιδιαίτερα στους γρήγορους κόμβους η χρησιμοποίηση είναι ελάχιστη.

Ο αλγόριθμος 2 με d = -1 έχει την υψηλότερη μέση χρησιμοποίηση των κόμβων 7, 8 και 9 που είναι οι γρήγοροι εργάτες και δεν αξιοποιεί καθόλου τους υπόλοιπους κόμβους.

Όταν ο αλγόριθμος 2 έχει d = 0, 1, 4 χρησιμοποιούνται εξίσου όλοι οι κόμβοι.

Ο κόμβος 1 είναι ο dispatcher, για αυτό είναι μηδενική η χρησιμοποίησή του από τους αλγορίθμους.

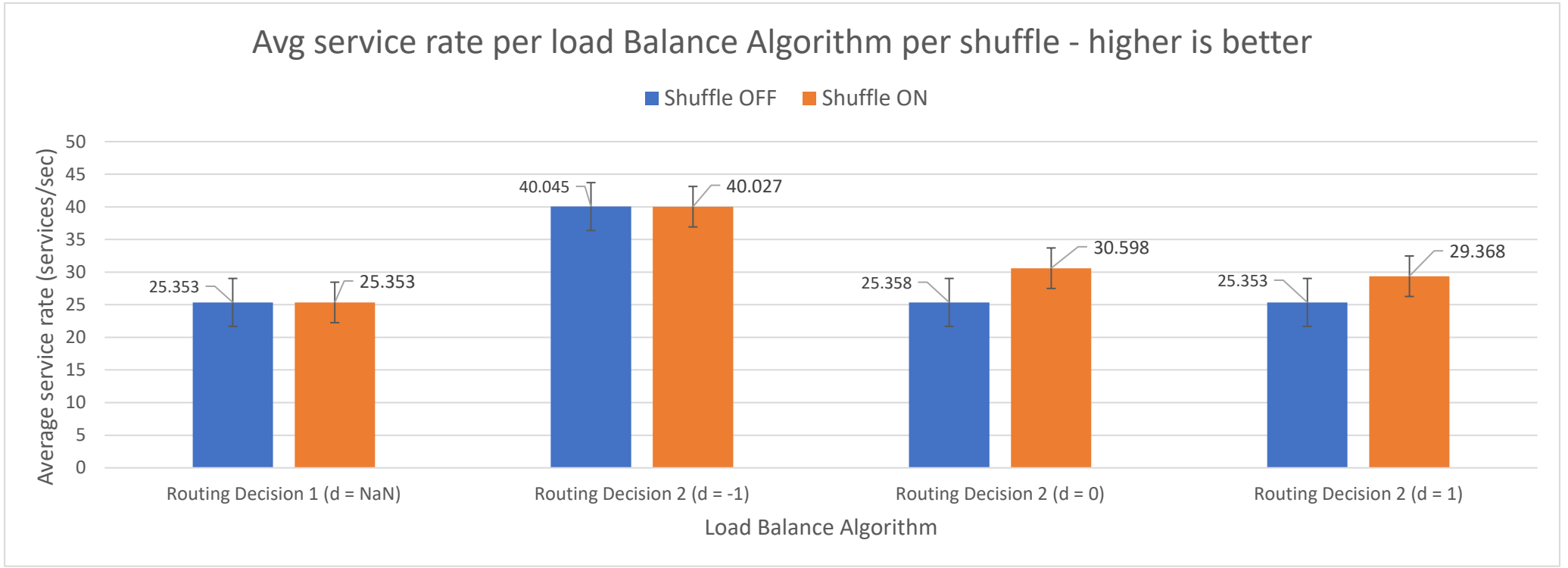
Σύγκριση RoutingDecision1 & RoutingDecision2 (shuffle = OFF) & RoutingDecision2 (shuffle = ON)



Ο μέσος χρόνος αναμονής αυξάνεται ραγδαία στον αλγόριθμο 2 με d = 0 ή 1 όταν εισάγουμε το shuffle = ON.

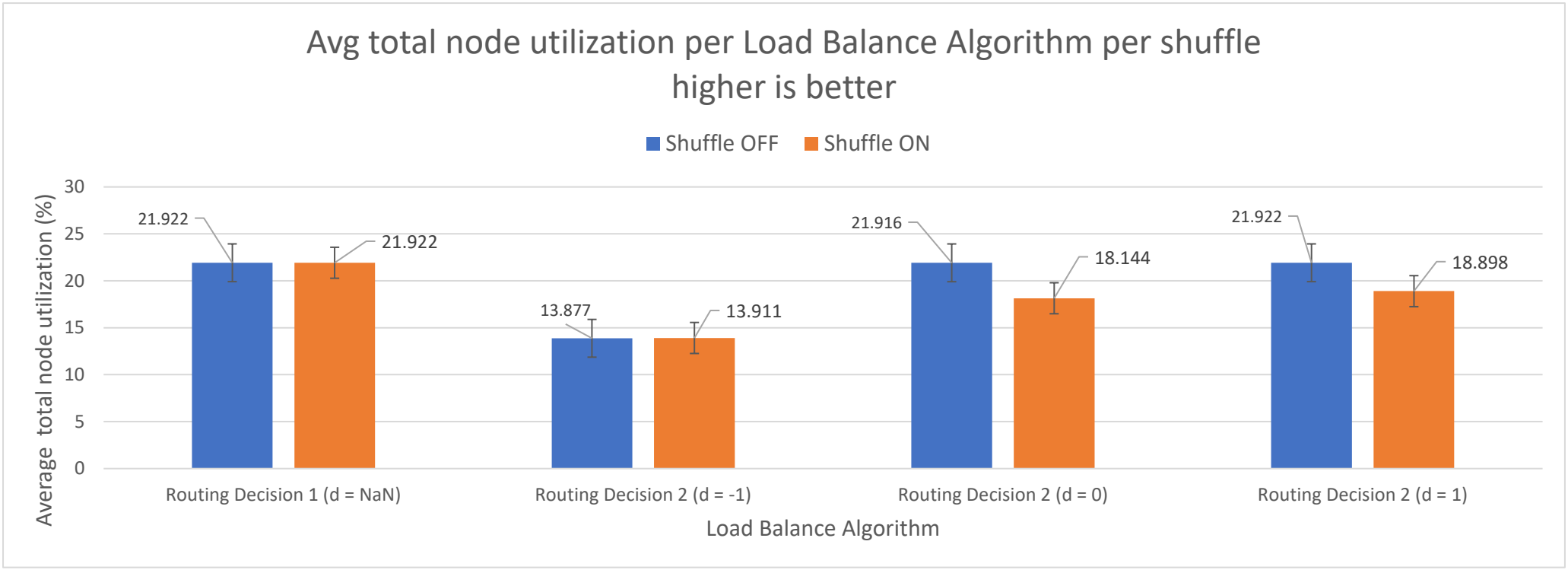
Μικρή αύξηση προκαλεί το shuffle = ON για d = -1.

Σε αυτήν την περίπτωση το shuffle = ON προκαλεί χειρότερη απόδοση.

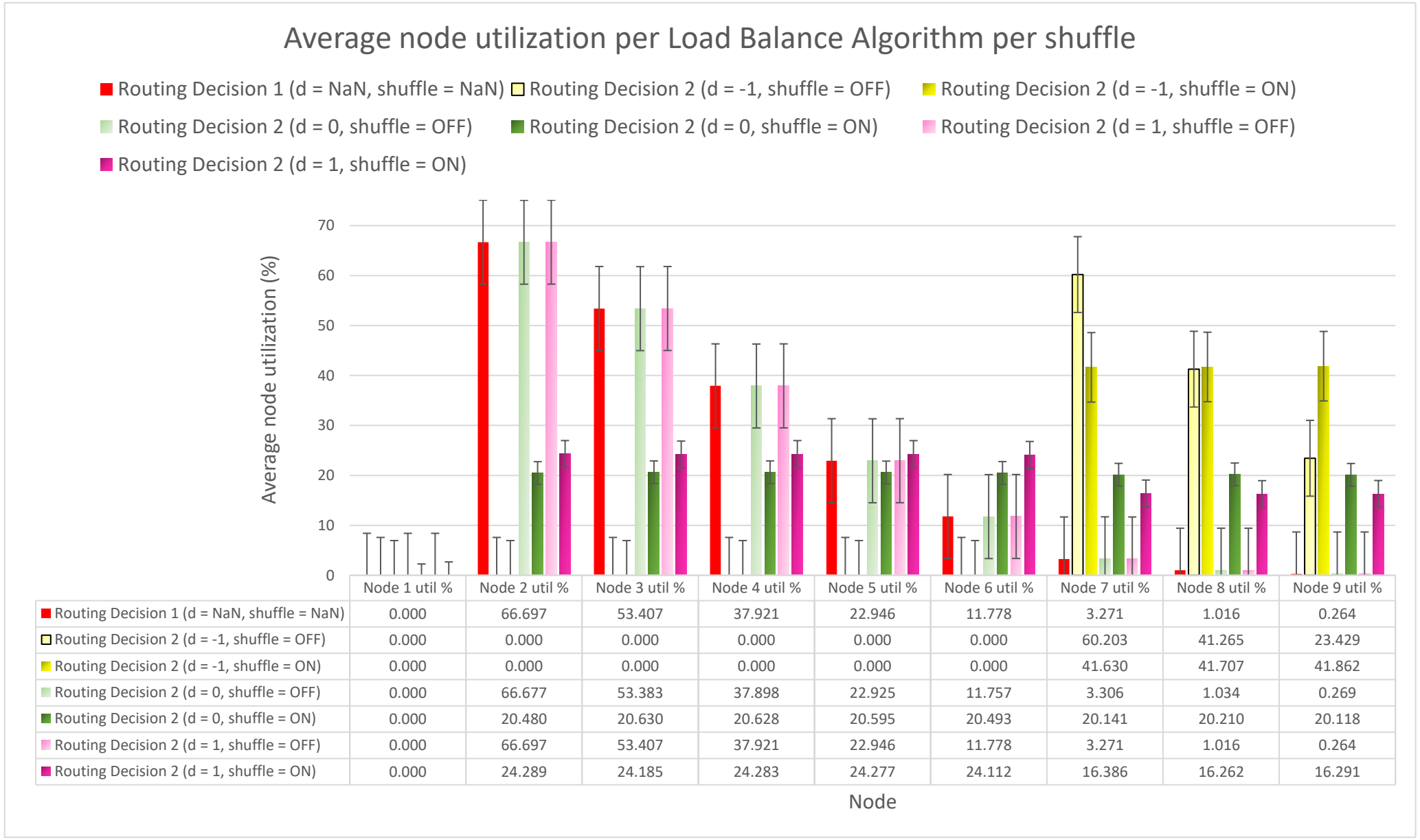


Για d = 0 ή 1 ο μέσος ρυθμός εξυπηρέτησης αυξάνεται, όταν υπάρχει shuffle = ON στον αλγόριθμο 2.

Άρα, έχουμε και καλύτερη απόδοση service rate - wise.



Για shuffle = ON η μέση συνολική χρησιμοποίηση κόμβων μειώνεται στον αλγόριθμο 2 με d = 0 ή 1, ενώ έχουμε οριακά μικρή αύξηση για d = -1.



	Node 1 util %	Node 2 util %	Node 3 util %	Node 4 util %	Node 5 util %	Node 6 util %	Node 7 util %	Node 8 util %	Node 9 util %
Routing Decision 1 (d = NaN, shuffle = NaN)	0.000	66.697	53.407	37.921	22.946	11.778	3.271	1.016	0.264
Routing Decision 2 (d = -1, shuffle = OFF)	0.000	0.000	0.000	0.000	0.000	0.000	60.203	41.265	23.429
Routing Decision 2 (d = -1, shuffle = ON)	0.000	0.000	0.000	0.000	0.000	0.000	41.630	41.707	41.862
Routing Decision 2 (d = 0, shuffle = OFF)	0.000	66.677	53.383	37.898	22.925	11.757	3.306	1.034	0.269
Routing Decision 2 (d = 0, shuffle = ON)	0.000	20.480	20.630	20.628	20.595	20.493	20.141	20.210	20.118
Routing Decision 2 (d = 1, shuffle = OFF)	0.000	66.697	53.407	37.921	22.946	11.778	3.271	1.016	0.264
Routing Decision 2 (d = 1, shuffle = ON)	0.000	24.289	24.185	24.283	24.277	24.112	16.386	16.262	16.291

Η ύπαρξη του shuffle = ON αλλάζει δραστικά την μέση χρησιμοποίηση κόμβων
για τον αλγόριθμο 2 με d = 0 ή 1.

Όταν δεν χρησιμοποιούμε shuffle = ON, η υψηλότερη χρησιμοποίηση παρουσιάζεται στον κόμβο 2 και μετά τον κόμβο 3 έχουμε φθίνουσα χρησιμοποίηση με σχεδόν ελάχιστη στους γρήγορους εργάτες.

Όταν ενεργοποιούμε το shuffle = ON το φορτίο κατανέμεται ισόποσα στους κόμβους και έχουμε μετριασμένες τιμές χρησιμοποίησης.

Βιβλιογραφία

Ciw documentation: https://ciw.readthedocs.io/en/latest/