Προσομοίωση ετερογενών συμπλεγμάτων εξυπηρετητών

Τμήμα Μηχανικών Η/Υ Πανεπιστήμιο Ιωαννίνων 2021-2022

Μάθημα: Προσομοίωση και Μοντελοποίηση Υπολογιστικών Συστημάτων

Υπεύθυνος Καθηγητής: Καππές Γεώργιος

Περιγραφή του προσομοιωτή.

Η παρούσα άσκηση έχει ώς κύριο στόχο, να προσομοιώσει ένα σύμπλεγμα διακομιστών. Αρχικά, υπάρχουν εννέα κόμβοι με τον κόμβο 1 (Node 1) να λειτουργεί ως dispatcher, είναι δηλαδή ο υπεύθυνος κόμβος για την δρομολόγηση των αιτήσεων στους υπόλοιπους κόμβους του συμπλέγματος Κόμβοι 2-9.

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

Η υλοποίηση του συμπλέγματος ακολουθεί τη δομή που προκύπτει απο τον παραπάνω πίνακα. Ο κόμβος 1 λαμβάνει αιτήσεις με εκθετική κατανομή και ρυθμο ίσο με ¼=0.25 second, ενώ ολοκληρώνει(δρομολογεί) κατευθείαν την κάθε αίτηση, δηλαδή μέσα σε 0 second. Οι κόμβοι 2-6 αποτελούν τους "αργούς" κόμβους του συστήματος καθώς ακολουθούν εκθετική κατανομή με ρυθμό ⅛=0.125 second προκειμένου να εκπληρώσουν την κάθε αίτηση. Αντίστοιχα, οι "γρήγοροι" κόμβοι 7-9 ακολουθούν εκθετική κατανομή με ρυθμό ⅕=0.2 second για την εκπλήρωση μιας αίτησης. Οι κόμβοι 2-9 δεν λαμβάνουν αιτήσεις από εξωτερικούς παράγοντες παρα μόνο από τον διακομιστή. Κάθε κόμβος αποτελείται από έναν διακομιστή του οποίου το μέγεθος ουράς είναι άπειρο. Για τον πίνακα με τις πιθανότητες δρομολόγησης (routing table) μιας αίτησης από έναν κόμβο σε άλλον, δημιουργήθηκε πίνακας 9χ9 με κάθε στοιχείο του ίσο με 0 καθώς οι αιτήσεις δρομολογούνται απο τον κόμβο 1 προς τους υπόλοιπους και αφού εξυπηρετηθούν απομακρύνονται από το σύστημα. Ακολουθούν, όλα τα προηγούμενα υλοποιημένα σε κώδικα python.

```
N = ciw.create_network(
  arrival_distributions=[
      ciw.dists.Exponential(rate=0.25),ciw.dists.NoArrivals(),ciw.dists.NoArrivals(),ciw.dists.NoArrivals(),
      ciw.dists.NoArrivals(),ciw.dists.NoArrivals(),ciw.dists.NoArrivals(),ciw.dists.NoArrivals(),ciw.dists.NoArrivals()],
  service_distributions=[
      ciw.dists.Deterministic(value=0.0),ciw.dists.Exponential(rate=0.125),ciw.dists.Exponential(rate=0.125),ciw.dists.Exponential(rate=0.125),ciw.dists.Exponential(rate=0.125),
      ciw.dists.Exponential(rate=0.125),ciw.dists.Exponential(rate=0.2),ciw.dists.Exponential(rate=0.2),ciw.dists.Exponential(rate=0.2)],
   routing=[[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]],
   number_of_servers=[1,1,1,1,1,1,1,1,1],
   queue_capacities=[float('inf'),float('inf'),float('inf'),float('inf'),float('inf'),float('inf'),
                     float('inf'),float('inf'),float('inf')]
)
```

Ο κόμβος 1 δρομολογεί τις αιτήσεις σύμφωνα με δύο αλγορίθμους (RoutingDecision1,RoutingDecision2). Η επιλογή αυτή γίνεται μέσω μεταβλητής που δίνεται ως είσοδος στον προσομοιωτή(load_balancing).

Ο αλγόριθμος 1, βρίσκει κάθε φορά το πόσες αιτήσεις έχει κάθε κόμβος στην ουρά του και διαλέγει να δρομολογήσει την αίτηση στον κόμβο με τις λιγότερες αιτήσεις ανεξάρτητα απο το αν είναι "αργός" ή "γρήγορος".

Ο αλγόριθμος 2, αρχικά ταξινομεί τους κόμβους ως προς τον αριθμό των αιτήσεων που έχουν στην ουρά τους. Στην συνέχεια αν ο κόμβος με τις λιγότερες αιτήσεις ανήκει στους κόμβους 7-9 (γρήγορους) τον επιστρέφει. Εάν δεν είναι γρήγορος ελέγχει εάν ο επόμενος κόμβος είναι "γρήγορος", ο έλεγχος αυτός πραγματοποιείται μέχρι να βρεθεί ένας γρήγορος κόμβος και έπειτα ελέγχεται η συνθήκη (Αριθμός αιτήσεων γρήγορου κόμβου) - d > (Αριθμός αιτήσεων αρχικά επιλεγμένου κόμβου), όπου d ένας θετικός ακέραιος όπου δίνεται

σαν είσοδος στην αρχή της προσομοίωσης. Στην περίπτωση που κανένας από τους γρήγορους κόμβους δεν ικανοποιεί την συνθήκη επιστρέφεται ο αρχικά επιλεγμένος κόμβος. **Ιδιαίτερα σημαντικό** είναι να επισημανθεί ότι προτεραιότητα δίνεται στους γρήγορους κόμβους με τη σειρά 9 8 7 σε περίπτωση ισοβαθμίας στον αριθμό αιτήσεων του κάθε κόμβου. Η προσομοίωση αυτή γίνεται για διάρκεια sim_time_secs, μια μεταβλητή η οποία δίνεται ως είσοδος στον προσομοιωτή με μονάδα μέτρησης τα δευτερόλεπτα. Στην άσκηση αυτή κάθε πείραμα έγινε σε διάρκεια μιας μέρας ενώ παράλληλα χρησιμοποιήθηκε και 1 ώρα σαν warm up time προκειμένου να προσομοιώσει πιο ρεαλιστικές καταστάσεις στο σύμπλεγμα για εγκυρότερα αποτελέσματα.

Λειτουργικότητα/Συμπεριφορά.

Η εκτέλεση του προσομοιωτή μέσα απο το script MYE029.py ξεκινάει με την δημιουργία ενός αρχείου "results.txt", το οποίο διαχωρίζεται σε 3 ενότητες :Section 1/2/3 . Στην πρώτη ενότητα, υπάρχει μια περιγραφή του προσομοιωτή καθώς και της συνολικής συμπεριφορά του ενώ στην ενότητα 2, μπορεί κανείς να δει τα συνολικά χαρακτηριστικά του προσομοιωτή.

```
----------------------------Section 1 System---------------------------


The following simulation tries to simulate the behavior a complex network.
The system structure consists of nine nodes.
Node 1 : Dispatcher of the system.
This node is responsible for routing requests to the other eight nodes.
The first five nodes are the slow nodes.
The other three nodes are the fast nodes.
Node 1 routes the requests to nodes depending on the value of the load_balancing parameter
Algorithm RoutingDecision1 : routes the request to the least busy node.
Algorithm RoutingDecision2 : routes the request to the least busy node, giving priority to fast nodes.
Considering a input parameter d (integer).
If a fast node has >d requests from a slow node so that :
Fast_Node_Requests[i] -d > Slow_Node_Least_Busy_Requests then Dispatcher routes the request to node 'i'.
Each simulation lasts one day.
Routing Algorithm can be, RoutingDecision1 or RoutingDecision2 depending on the parameter load_balancing.
----------------------------Section 2 Network Parameters---------------------------


The following values were given during the creation of the network in the proper fields.
Arrival_Distributions:
    Dispatcher (Node 1) receives a request in exponential  distribution, with rate of 0.25 seconds
    The other eight nodes are not receiving requests from the outer world(NoArrivals)
Service_Distributions:
    Requests from Node 1 to the other nodes are being sent immediately(deterministic(0.0)).
    Nodes 2-6(slow) are serving requests, following exponential distribution with rate of 0.125 seconds.
    Nodes 7-9(fast) are serving requests, following exponential distribution with rate of 0.2 seconds.
Number_Of_Servers:
    Each node consists of one server(number_of_servers=1).
Queue_Capacities:
    Every queue on each node has infinite capacity,(Queue_Capacities=inf)
Routing Table:
    Each requests has 0% possibility to be routed to other node since routing,
    is being configured by the Dispatcher.
```

Το αρχείο έχει δημιουργηθεί με την παράμετρο "a" προκειμένου να μπορούμε να γράψουμε στο τέλος του. Αυτό θα βοηθήσει στο να δημιουργήσουμε την τρίτη ενότητα με τα αποτελέσματα κάθε πειράματος στο τέλος κάθε εκτέλεσης του simulation.

Η συνάρτηση make_simulation(sim_times_secs,load_balancing,d) αποτελεί την κύρια συνάρτηση της προσομοίωσης. Στην αρχή της, γίνεται αρχικοποίηση κάποιων καθολικών

μεταβλητών/πινάκων που θα χρησιμεύσουν στην διάρκεια της προσομοίωσης προκειμένου να αποθηκεύονται τα αποτελέσματα. Στη συνέχεια δημιουργείται ένας ατέρμων βρόχος στον οποίο γίνεται η προσομοίωση, ο οποίος τερματίζει μόνο αν ικανοποιείται μία εκ των δύο συνθηκών, το πείραμα να έχει φτάσει τις είκοσι επαναλήψεις ή να έχει βρεθεί το 95% διάστημα εμπιστοσύνης της μετρικής που επιλέξαμε δηλαδή του μέσου χρόνου αναμονής. Μόλις εισέλθουμε στο βρόχο δημιουργείται ο σπόρος (seed), για την προσομοίωση με κατάλληλο τρόπο ώστε να εξασφαλίσουμε διαφορετικότητα σε κάθε επανάληψη αλλά ταυτόχρονα και ομοιότητα μεταξύ των υπόλοιπων πειραμάτων. Στην μεταβλητή Q δημιουργείται το simulation object στο οποίο δίνεται ως παράμετρος το δίκτυο "N" που φτιάξαμε προηγουμένως και αναλήθηκε στην ενότητα "Περιγραφή του προσομοιωτή", ενώ στην παράμετρο Node_class δίνεται ένας πίνακας με τη συνάρτηση routing κάθε κόμβου.Ο πρώτος κόμβος θα πάρει συνάρτηση load_balancing δηλαδή μία εκ των RoutingDecision1-2 που έχει καταχωρηθεί στην αρχή του προγράμματος στην καθολική μεταβλητή ενώ οι υπόλοιποι κόμβοι την default συνάρτηση του ciw.

```
class RoutingDecision1(ciw.Node):
```

Η κλάση RoutingDecision1, κάνει override την εσωτερική μέθοδο που έχει το αντικείμενο Node.

```
def next_node(self, ind):
    """
    Finds the next node according the routing method:
      - if not process-based then sample from transtition matrix
      - if process-based then take the next value from the predefined route,
        removing the current node from the route
    """
    if not self.simulation.network.process_based:
        customer_class = ind.customer_class
        return random_choice(self.simulation.nodes[1:],
            self.transition_row[customer_class] + [1.0 - sum(
            self.transition_row[customer_class])])
    if ind.route == [] or ind.route[0] != self.id_number:
        raise ValueError('Individual process route sent to wrong node')
    ind.route.pop(0)
    if len(ind.route) == 0:
        next_node_number = -1
    else:
        next_node_number = ind.route[0]
    return self.simulation.nodes[next_node_number]
```

Η μέθοδος αυτή έχει τροποποιηθεί στην κλάση RoutingDecision1 προκειμένου να επιστρέφει τον λιγότερο απασχολούμενο κόμβο μεταξύ των κόμβων 2-9.

```
class RoutingDecision1(ciw.Node):
    def next_node(self,ind):

        busyness = {n: self.simulation.nodes[n].number_of_individuals for n
                    in [2, 3, 4,5,6,7,8,9]}
        chosen_n = sorted(busyness.keys(), key=lambda x: busyness[x])[0]
        return self.simulation.nodes[chosen_n]
```

Η μεταβλητή chosen_n περιέχει το id του παραπάνω κόμβου και η συνάρτηση θα επιστρέψει τον κόμβο με αυτό το id.

```
class RoutingDecision2(ciw.Node):
```

Η κλάση RoutingDecision2, κάνει επίσης override την εσωτερική μέθοδο next_node(self,ind) του αντικειμένου Node. Αρχικά, ταξινομεί τα id των κόμβων ως προς το φορτίο τους στην λίστα sortedList και μετέπειτα δημιουργούνται 2 λίστες με τους ταξινομημένους κόμβους.

```
class RoutingDecision2(ciw.Node):
    def next_node(self,ind):
        global counter2

        busyness = {n: self.simulation.nodes[n].number_of_individuals for n
                    in [2, 3, 4, 5, 6, 7, 8, 9]}
        sortedList = sorted(busyness.keys(), key=lambda x: busyness[x])
        finalSort=[]
        finalSort2=[]

        for i in range(len(sortedList)):
            finalSort.append(self.simulation.nodes[sortedList[i]])
            finalSort2.append(self.simulation.nodes[sortedList[i]])
```

Στην συνέχεια ελέγχεται η προτεραιότητα, αν κάποιος κόμβος απο τους γρήγορους έχει ίδιο πλήθος αιτήσεων με κάποιον από τους αργούς κόμβους θα πάρει την θέση του. Στον κώδικα αυτό γίνεται ως εξής, το στοιχείο 0 της ταξινομημένης λίστας καθορίζεται ως min και ελέγχεται αν ικανοποιείται η συνθήκη min==Node[i].size και Node[i].id ανηκει στο “789”.
Αν ισχύει, το στοιχείο “i” γίνεται insert στην αρχή της δεύτερης ταξινομημένης λίστας π είχαμε δημιουργήσει προηγουμένως. Στο τέλος διαγράφουμε όλα τα διπλότυπα από την δεύτερη ταξινομημένη λιστα.

```
for i in range(1,8):
    min=finalSort[0].number_of_individuals
    if min == finalSort[i].number_of_individuals and str(finalSort[i].id_number) in "789":
        finalSort2.insert(0,finalSort[i])
res=[]
for x in finalSort2:
    if x not in res:
        res.append(x)
```

Πλέον ο πίνακας res έχει τους κόμβους ταξινομημένους ως προς το φορτίο τους και παράλληλα σε περίπτωση ισοβαθμίας μεταξύ αργού και γρήγορου κόμβου δίνεται προτεραιότητα στον γρήγορο κόμβο. Έπειτα, γίνεται έλεγχος αν ο πρώτος κόμβος της λίστας ανήκει στους "γρήγορους" αν ναι τον επιστρέφει, αλλιώς ελέγχει τους κόμβους μέχρι να βρει κάποιον γρήγορο κόμβο και να διαπιστωθεί αν ο γρήγορος κόμβος που βρέθηκε, έχει d μεγαλύτερο φορτίο από τον πρώτο κόμβο στην λίστα. Σε περίπτωση που ισχύει, η αίτηση θα δρομολογηθεί στον γρήγορο κόμβο, διαφορετικά δρομολογείται στον αρχικό κόμβο.

```
if res[0].id_number >6 :
    return self.simulation.nodes[res[0].id_number]
else:
    for j in range(1,10):
        if res[j].id_number > 6:
            if (int(res[j].number_of_individuals) -d) > int(res[0].number_of_individuals):
                counter2+=1
                return self.simulation.nodes[res[j].id_number]
            else:
                break
    return self.simulation.nodes[res[0].id_number]
```

Αφού εξηγήθηκε ο κώδικας των RoutingDecision1-2 επιστρέφοντας στην κύρια συνάρτηση make_simulation(...) και έχοντας δημιουργήσει το simulation object Q γίνεται χρήση της εντολής Q.simulate_until_max_time(sim_time_secs) προκειμένου να γίνει η προσομοίωση. Στην συνέχεια δημιουργούνται πίνακες προκειμένου να υπολογίσουμε σε κάθε εκτέλεση του simulation μέσο χρόνο αναμονής(final_mean_wait), μέσο ρυθμό εξυπηρέτησης αιτήσεων(final_tput), μέση συνολική χρησιμοποίηση κόμβων(final_mean_util_all), μέση χρησιμοποίηση ανά κόμβο(total_avg_util_node). Σε κάθε run, υπολογίζονται οι παραπάνω μετρικές (χρόνος αναμονής,ρυθμός εξυπηρέτησης αιτήσεων,συνολική χρησιμοποίηση κόμβων,χρησιμοποίηση ανά κόμβο) μετά από χρόνο 3.600 seconds (warm up time) και προστίθενται στους καθολικούς πίνακες που αναφέρθηκαν παραπάνω προκειμένου να υπολογίζεται το μέσο κάθε μετρικής. Αφού γίνουν τα παραπάνω,υπολογίζεται και ελέγχεται το 95% διάστημα εμπιστοσύνης ως προς τον μέσο χρόνο αναμονής , αν ικανοποιείται η συνθήκη confidence_interval < 0.05 τότε γίνεται έξοδος από τον ατέρμων βρόχο αλλιώς έξοδος θα γίνει όταν θα γίνουν 20 εκτελέσεις του πειράματος. Τότε σχηματίζεται η τρίτη ενότητα στο αρχείο που δημιουργήθηκε κατα την εκκίνηση, γράφοντας τα αποτελέσματα των παραπάνω μετρικών καθώς και οι καθολικές παράμετροι που χρησιμοποιήθηκαν κατα την εκτέλεση του πειράματος.

Script/Αυτοματοποίηση.

Στην main του αρχείου, υπάρχει κώδικας που δίνει την επιλογή στον χρήστη να τρέξει όσα πειράματα θέλει με τις παραμέτρους που εκείνος θέλει και στο τέλος να δει όλα τα αποτελέσματα διαχωρισμένα ανάλογα με το πείραμα στο αρχείο εξόδου αλλά παρέχεται και η επιλογή να γίνουν αυτόματα ορισμένα πειράματα και να αποθηκευτούν τα αποτελέσματα στο αρχείο εξόδου για λόγους ταχύτητας και αυτοματοποίησης.

```
while True:
    print("Press 1 if you want to run many automated simulation \n")
    print("Press 0 if you want to run specific simulations of your choice \n")
    choice=int(input("Give the value of your choice "))
    if (choice==0):
        print("Give the simulation time,note 90000 seconds equal to 1 day + 1 hour warm up\n")
        print("sim_time_secs\n")
        sim_time_secs=int(input("Please give the simulation time in seconds :"))
        print("Choose the routing algorithm dispatcher is going to use.\n")
        print("load_balancing\n")
        load_balancing= int(input("Type 1 for RoutingDecision1 or 2 for RoutingDecision2"))
        if (load_balancing==1):
            make_simulation(sim_time_secs, RoutingDecision1, 0)
        else:
            print("Give a value for the 'd' parameter\n")
            d=int(input("Give a number for d :"))
            make_simulation(sim_time_secs, RoutingDecision2, d)
        x=int(input("If you want to make another simulation press 1 else press 0"))
        if (x==0):
            print("Finishing program\n")
            break
    else:
        sim_time_secs = 90000
        load_balancing = [RoutingDecision2, RoutingDecision2, RoutingDecision2, RoutingDecision2]
        make_simulation(sim_time_secs, RoutingDecision1, 0)
        for i in range(0, 4):
            make_simulation(sim_time_secs, load_balancing[i], i)
        break
```

<u>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;</u>

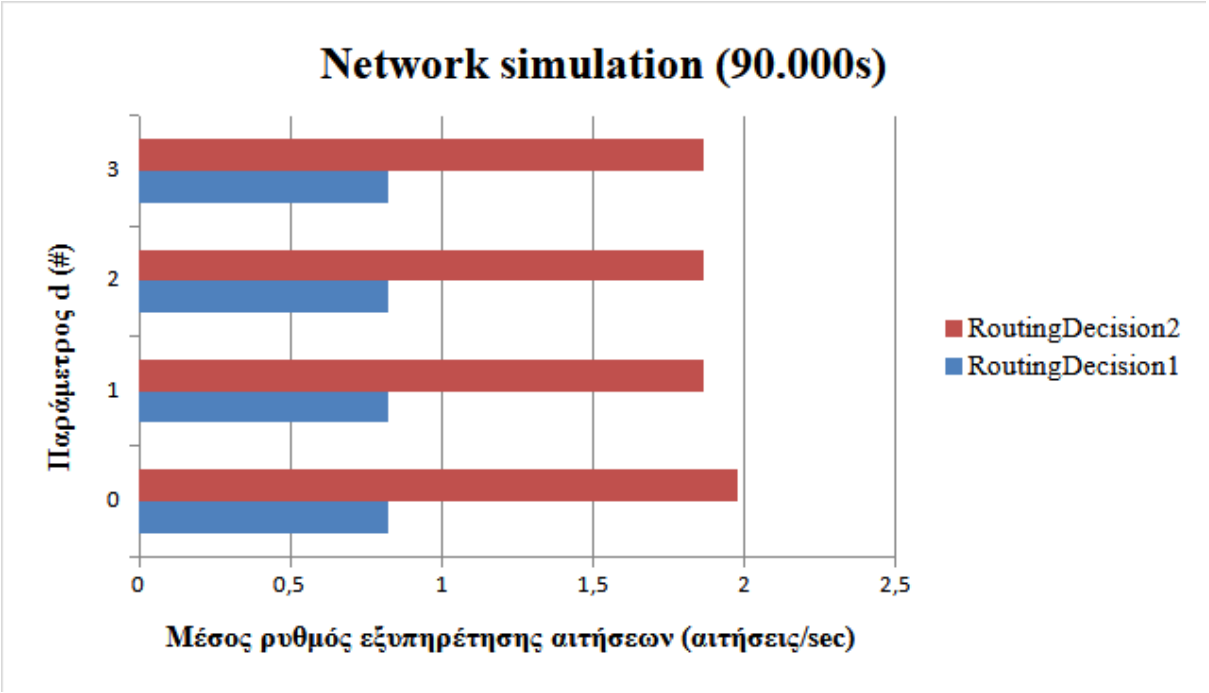
Η τιμή d που θα δώσει τον μικρότερο χρόνο αναμονής είναι οποιαδήποτε τιμή για **d>=2** αυτό συμβαίνει διότι έτσι όπως είναι σχεδιασμένο το σύμπλεγμα μας, οι ουρές δεν προλαβαίνουν να γεμίσουν με πάρα πολλές αιτήσεις . Ο μέσος χρόνος άφιξης τους είναι 4 sec ενώ ο μέσος χρόνος εξυπηρέτησης τους ειναι 5 second στους γρήγορους κόμβους και 8 στους αργούς κόμβους αντίστοιχα. Το γεγονός ότι έχουμε μόνο 1 dispatcher και 9 κόμβους workers οδηγεί στο παραπάνω φαινόμενο, καθώς οι αιτήσεις έχουν προτεραιότητα τους γρήγορους κόμβους. Η παραπάνω διαπίστωση έγινε με την χρήση του Debugger στο Pycharm και τα κατάλληλα breakpoints κοιτώντας τα περιεχόμενα κάθε κόμβου και πίνακα, αλλα και με μια μεταβλητή Counter2 η οποία αυξάνεται κατα 1 κάθε φορά που η συνθήκη για το d στον αλγόριθμο 2 ήταν αληθής. Η μέση χρησιμοποίηση ανεβαίνει ελάχιστα για τις ίδιες τιμές d δηλαδή μεγαλύτερο ή ίσο του 2. Αυτό συμβαίνει εξίσου για τον ίδιο λόγο, οι αιτήσεις για d < 2 εξυπηρετούνται κατά κύριο λόγο από τους γρήγορους κόμβους και ένα πολύ μικρό ποσοστό στους υπόλοιπους. **Το γεγονός αυτό έχει και συνέπεια μικρή τιμή στη μέση χρησιμοποίηση των κόμβων**. Όταν για το d ισχύει η συνθήκη d>=2 καμία αίτηση δεν αλλάζει κόμβο (απο αργό σε γρήγορο) καθώς κανένας κόμβος δεν είναι τόσο απασχολημένος σε κανένα διάστημα της προσομοίωσης.

<u>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;</u>

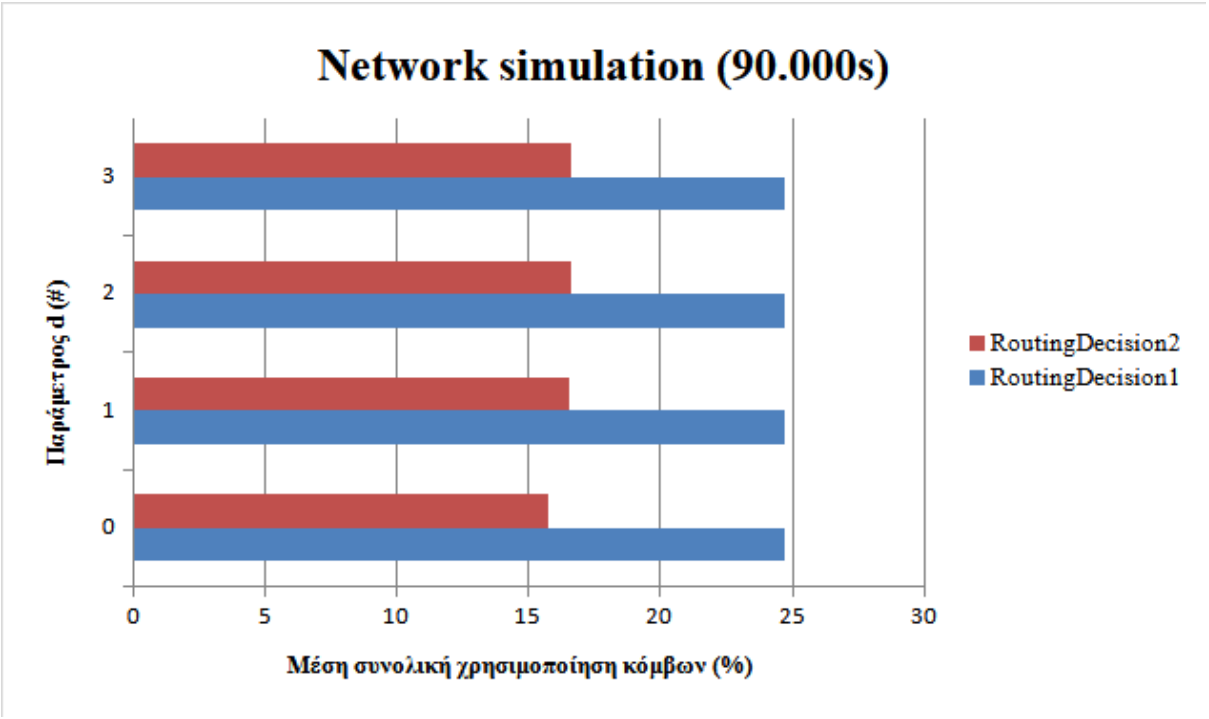
Αρχικά, ο αλγόριθμος 2 μπορεί να επιτύχει χαμηλότερο μέσο χρόνο αναμονής για d>=1. Αυτό συμβαίνει διότι καθώς δίνεται προτεραιότητα στους "γρήγορους" κόμβους, οι αιτήσεις θα εξυπηρετηθούν ταχύτερα συγκριτικά με τον αλγόριθμο 1. Παράλληλα αυτή η προτεραιότητα δεν επιβαρύνει το δίκτυο καθώς ο ρυθμός άφιξης των αιτήσεων είναι μικρός ,με αποτέλεσμα μια αίτηση αν δρομολογηθεί σε ένα γρήγορο κόμβο να έχει το μέγιστο 1 με 2 αιτήσεις μπροστά της στην ουρά να περιμένουν για εξυπηρέτηση. Απο την άλλη σε ότι αφορά την μέση χρησιμοποίηση, ο αλγόριθμος 2 μειονεκτεί σε σχέση με τον 1. Αυτό συμβαίνει γιατί με τον αλγόριθμο 2, ο μεγαλύτερος όγκος των αιτήσεων θα δρομολογηθεί στους γρήγορους κόμβους και ένα μικρό ποσοστό στους υπόλοιπους, με αποτέλεσμα οι κόμβοι 2-6 να χρησιμοποιούνται σε πολύ μικρό βαθμό και να επιδρούν αρνητικά στην μέση χρησιμοποιήση. Αντίθετα, με τον αλγόριθμο 1 οι αιτήσεις δρομολογούνται στον λιγότερο απασχολούμενο κόμβο, γεγονός που έχει ως συνέπεια να χρησιμοποιεί τους κόμβους του συμπλέγματος σε μεγαλύτερο βαθμό από τον αλγόριθμο 2.

Επίσης αν αυξήσουμε το rate στην εκθετική κατανομή στον πίνακα arrival distribution του κόμβου 1, απο 0.25 σε 1 δηλαδή σε 1 αίτηση ανα δευτερόλεπτο και τρέξουμε τον αλγόριθμο για d=10, προκειμένου να μην γίνει καμία αναδρομολόγηση διαπιστώνεται ότι ο αλγόριθμος 2 εξακολουθεί να έχει μικρότερο μέσο χρόνο αναμονής αλλα συνεχίζει να έχει μικρότερη μέση χρησιμοποίηση κόμβων.

Γραφικές Παραστάσεις/Αιτιολόγηση των Ερωτημάτων 1 και 2.

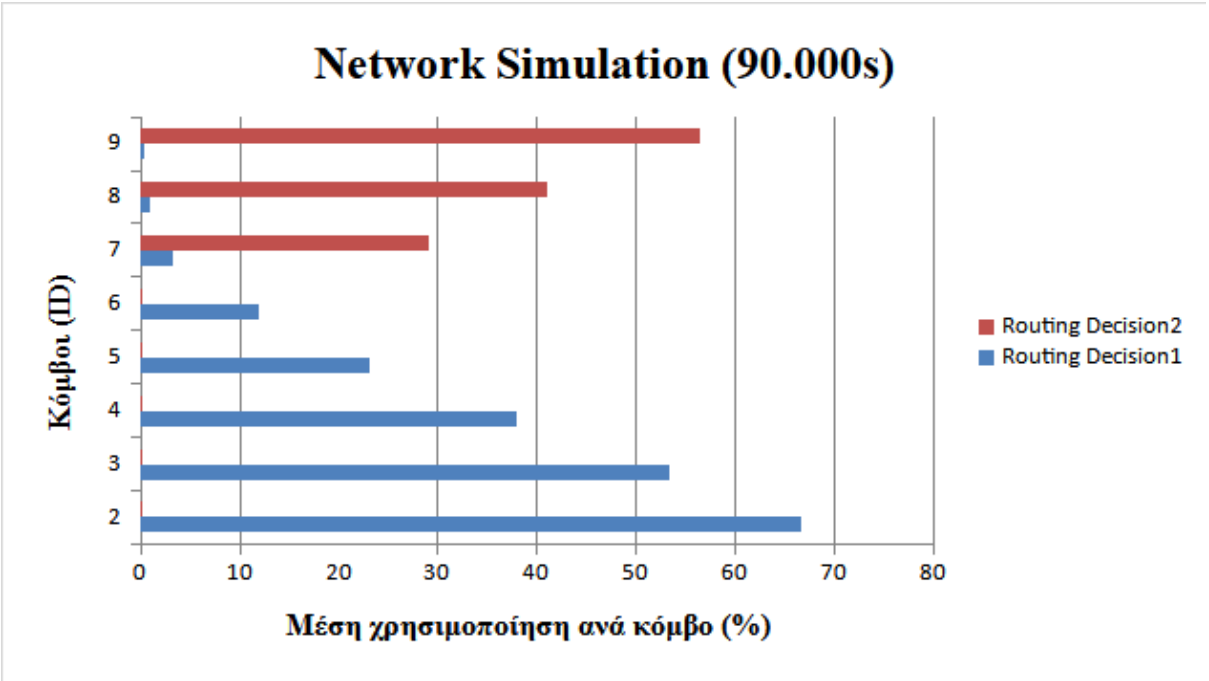


Από την παραπάνω γραφική παράσταση γίνεται ξεκάθαρο ότι η εξυπηρέτηση αιτήσεων με την χρήση του αλγόριθμου 2 είναι πολύ μεγαλύτερη, αφού διακρίνονται τα διπλάσια νούμερα εξυπηρετούμενων αιτήσεων σε αντίθεση με τον 1. Επίσης, γίνεται κατανοητό ότι τα αποτελέσματα στον αλγόριθμο 1 είναι πάντα σταθερά καθώς η παράμετρος d δεν τον επηρεάζει. Στον αλγόριθμο 2 έχουμε τον μεγαλύτερο μέσο ρυθμό εξυπηρέτησης αιτήσεων για d=0, καθώς μόνο οι γρήγοροι κόμβοι εξυπηρετούν τις αιτήσεις σε αντίθεση με τον αλγόριθμο 1 που οι αιτήσεις κατανέμονται με βάση το μικρότερο φορτίο.



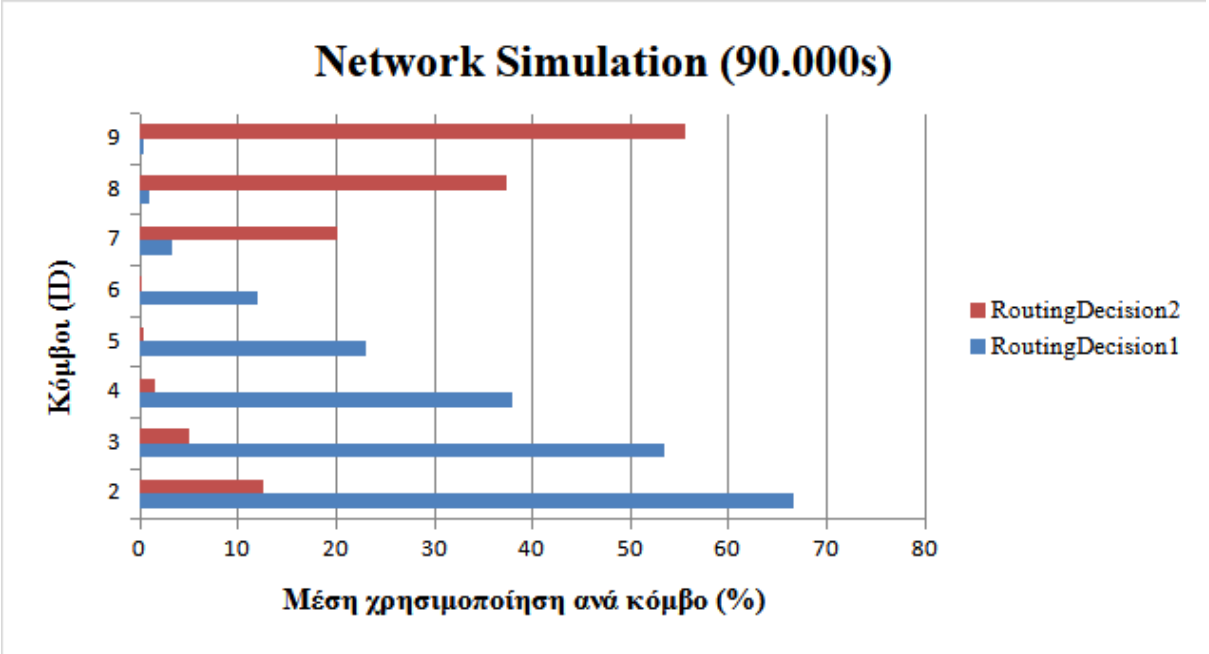
Τα συμπεράσματα της πρώτης γραφικής , μπορούν να δικαιολογηθούν και μέσα από την δεύτερη γραφική. Παρατηρείται για τον αλγόριθμο 2 ότι, η μέση συνολική χρησιμοποίηση κόμβων αυξάνεται μέχρι d=2 και μετά μένει σταθερή για κάθε d. Αυτό συμβαίνει διότι στο d=0 μόνο οι κόμβοι 9, 8 ,7 λαμβάνουν αιτήσεις ενώ για d=1 ένα μικρό κομμάτι των αιτήσεων μοιράζεται στους υπόλοιπους κόμβους. Στην τιμή d>=2 επιτυγχάνεται η μέγιστη μέση συνολική χρησιμοποίηση. Στον αλγόριθμο 1, η τιμή της συνολικής μέσης χρησιμοποίησης είναι πολύ μεγαλύτερη καθώς δεν υπάρχει καμία προτεραιότητα και οι αιτήσεις μοιράζονται σε μεγαλύτερο βαθμό στους κόμβους. Όλα τα παραπάνω αιτιολογούν παράλληλα την απάντηση που δόθηκε στο ερώτημα 1.

Μέση χρησιμοποίηση ανά κόμβο d=0.

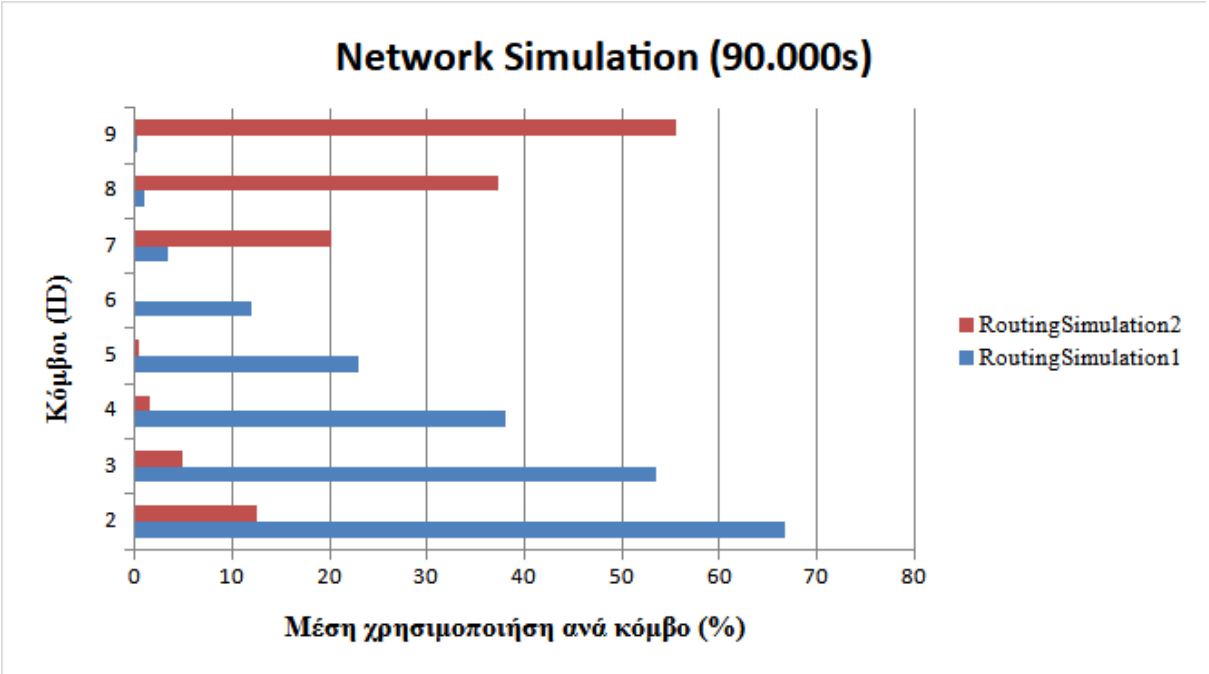


Για τον αλγόριθμο RoutingDecision2 και d=0, η προτεραιότητα των κόμβων είναι 9,8,7 γεγονός που αποδεικνύει και η παραπάνω γραφική, όλες οι αιτήσεις εξυπηρετούνται απο τους γρήγορους κόμβους και αυτό έχει ώς αποτέλεσμα μηδενική χρησιμοποίηση των υπολοίπων. Ταυτόχρονα, παρατηρείται για τον αλγόριθμο 1 ότι έχει πολύ μεγάλη μέση χρησιμοποίηση στους πρώτους κόμβους και πως το φορτίο των αιτήσεων μοιράζεται σε όλους τους κόμβους . Ωστόσο υψηλή χρησιμοποίηση έχουν και εδώ μόνο οι πρώτοι κόμβοι. Αυτό συμβαίνει διότι κατα την διάρκεια της προσομοίωσης το σύστημα δεν επιβαρύνεται σε μεγάλο βαθμό και ο μέσος χρόνος άφιξης αιτήσεων δεν ειναι ικανοποιητικος προκειμένου να στρεσάρει το σύμπλεγμα. Έτσι όλοι οι κόμβοι είναι σχεδόν πάντα, ελάχιστα απασχολημένοι με αποτέλεσμα ο αλγόριθμος να δρομολογεί την αίτηση στους πρώτους κόμβους.

Μέση χρησιμοποίηση ανά κόμβο d=1.



Μέση χρησιμοποίηση ανά κόμβο d>=2.



Στις 2 προηγούμενες γραφικές για την μέση χρησιμοποίηση ανα κόμβο στον RoutingDecision1 ισχύει ότι αναλήθηκε προηγουμένως καθώς η παράμετρος d δεν τον επηρεάζει. Η διαφορά στον RoutingDecision2 για d=1 και d>=2 είναι ότι 2 μόνο αιτήσεις άλλαξαν από γρήγορο σε αργό. Το προηγούμενο συμπέρασμα έγινε με μια μεταβλητή που μετρούσε πόσες φορές η συνθήκη if(…-d>...) έγινε αληθής κατα την διάρκεια του πειράματος. Ως προς την μέση χρησιμοποίηση σε σχέση με d=0 για τον αλγόριθμο 2, επιτυγχάνονται καλύτερα αποτελέσματα καθώς οι αιτήσεις δρομολογούνται σε περισσότερους κόμβους, ωστόσο παραμένει χαμηλότερη από αυτήν που επιτυγχάνει ο Αλγόριθμος 1.

Συμπεράσματα.

Ο αλγόριθμος 1 χρησιμοποιεί σίγουρα καλύτερα το σύμπλεγμα διακομιστών αφου πάντα πετυχαίνει μεγαλύτερη μέση χρησιμοποίηση των κόμβων, ωστόσο αυτό δεν σημαίνει ότι πετυχαίνει πάντα καλύτερα αποτελέσματα. Ο αλγόριθμος 2 απο την άλλη πετυχαίνει για τα κατάλληλα d, καλύτερο μέσο χρόνο αναμονής ως απόρροια του καλύτερου μέσου χρόνου εξυπηρέτησης που έχει ωστόσο δεν αξιοποιεί το σύμπλεγμα τόσο αποδοτικά όσο ο πρώτος. Σε πείραμα που έγινε για μέσο όρο άφιξης αιτήσεων στον Dispatcher 1 sec με στόχο να γεμίσουμε το σύμπλεγμα με αιτήσεις, και d=10 για τον αλγόριθμο 2 τα αποτελέσματα ήταν τα εξής,

```
----------------------------Section 3 Output Results----------------------------


Parameters given : sim_time_secs=90000,d=0,load_balancing=<class '__main__.RoutingDecision1'>.
I have confidence interval,0.03462520305191687
For d = 0 i changed to fast node, 0 times :
Total Mean Waiting :2.02038966962475
Total Mean Service :3.3683621285509155
Total Mean Throughput :5.427566666666666
Total Mean node 2 utilization 0.9588402262890403
Total Mean node 3 utilization 0.9446097114403141
Total Mean node 4 utilization 0.9276842392559287
Total Mean node 5 utilization 0.9055279127879213
Total Mean node 6 utilization 0.8796920893106086
Total Mean node 7 utilization 0.779583602257527
Total Mean node 8 utilization 0.7122436998159686
Total Mean node 9 utilization 0.6355132656773455
Total Mean 'all_node' utilization : 0.8429618433543319
Confidence Interval for Mean Waiting = 1.950433267069998,2.090346072179502
```

και αντίστοιχα για τον RoutingDecision2

```
Parameters given : sim_time_secs=90000,d=10,load_balancing=<class '__main__.RoutingDecision2'>.
I have confidence interval,0.04268804419161451
For d = 10 i changed to fast node, 0 times :
Total Mean Waiting :1.5678409635754857
Total Mean Service :3.2145674449421175
Total Mean Throughput :5.814472222222223
Total Mean node 2 utilization 0.8570826472846198
Total Mean node 3 utilization 0.8183841762219086
Total Mean node 4 utilization 0.7677971632774803
Total Mean node 5 utilization 0.7129228646923098
Total Mean node 6 utilization 0.6480268589334343
Total Mean node 7 utilization 0.8384755133861979
Total Mean node 8 utilization 0.8808971618307996
Total Mean node 9 utilization 0.9144265474869119
Total Mean 'all_node' utilization : 0.8047516166392077
Confidence Interval for Mean Waiting = 1.500912899236952,1.6347690279140195
```

Παρατηρείται ότι ο μέσος χρόνος αναμονής,εξυπηρέτησης ,ρυθμός εξυπηρέτησης αιτήσεων είναι καλύτερος στον Αλγόριθμο 2. Ωστόσο, μέση χρησιμοποίηση παραμένει 4% . Με βάση τα παραπάνω καταλήγουμε ότι η χρησιμοποίηση του αλγορίθμου 2 θα επιφέρει καλύτερα αποτελέσματα από τον 1.