Ενσωματωμένα Συστήματα Πραγματικού Χρόνου

Τελική Εργασία 2022

Αντώνης Φάββας

Η παρούσα εργασία έχει ως πρωταρχικό στόχο την συλλογή και επεξεργασία πληροφοριών συ σχετιζόμενων με κάποιες μετοχές χρηματιστηρίων του εξωτερικού. Οι μετοχές που θα εξεταστούν είναι μετοχές μεγάλων εταιρειών όπως αυτών της APPLE και της AMAZON αλλά και κρυπτονομισμάτων όπως BTCUSDT. Τα δεδομένα εισαγωγής στον αλγόριθμο παρέχονται σε πραγματικό χρόνο από το finnhub και η συλλογή τους πραγματοποιείται από έναν "client", τον κώδικα του οποίου ανέπτυξε ένας συνάδελφος και μπορείτε να βρείτε τον πηγαίο κώδικα εδώ:
https://github.com/GohanDGeo/lws-finnhub-client .

1. Απαιτούμενες λειτουργίες παραγόμενου κώδικα

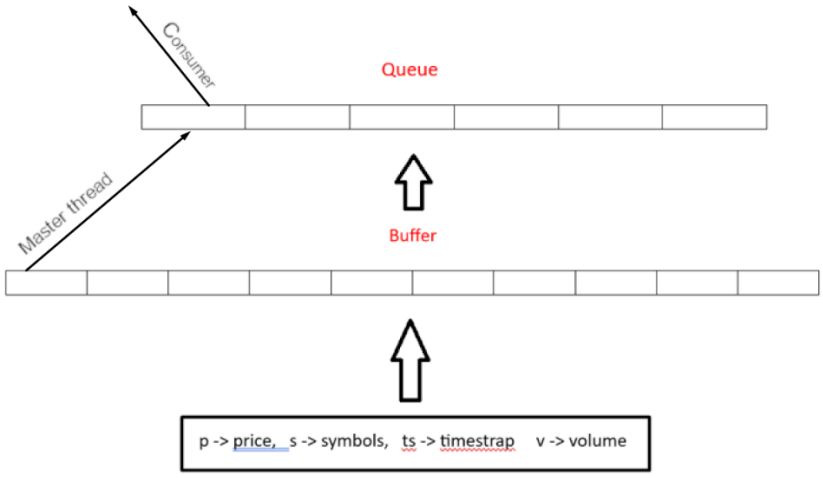
Ο τελικός κώδικας πρέπει να είναι σε θέση να φέρει εις πέρας 3 βασικές λειτουργίες:
Α) Την συλλογή και καταγραφή όλων των συναλλαγών σε πραγματικό χρόνο σε φακέλους-αρχεία (διαφορετικά για κάθε σύμβολο/μετοχή).
Β) Θα δειγματοληπτεί κάθε λεπτό τα δεδομένα που έχει συλλέξει και θα παράγει για κάθε σύμβολο το "candlestick", που αποτελείται από τις εξής μετρικές:

- Αρχική Τιμή
- Τελική Τιμή
- Μέγιστη Τιμή
- Ελάχιστη Τιμή
- Συνολικός Όγκος

C) Θα παράγει για κάθε λεπτό τον κινούμενο μέσο όρο των τιμών συναλλαγών και συνολικό όγκο των πιο πρόσφατων 15 λεπτών.
Αφού παραχθούν τα αποτελέσματα του αλγορίθμου, εκτελείται ένα κομμάτι κώδικα σε python, το οποίο υπολογίζει το ελάχιστο άθροισμα καθυστερήσεων από του χρόνους αναφοράς. Ως καθυστέρηση ορίζεται, το χρονικό διάστημα μεταξύ της λήψης της πληροφορίας και της καταγραφής της στο εκάστοτε αρχείο.
Επίσης η ανάπτυξη του τελικού κώδικα πρέπει να γίνει με τέτοιο τρόπο ώστε να ενσωματώνονται οι ιδέες και οι τεχνικές του παραλληλοποιημένου έως ένα βαθμό κώδικα producer-consumer, ο οποίος βρίσκεται εδώ: https://github.com/antonis-fav/Emdedded-Systems-Assignment-1. Αξίζει να σημειωθεί πως εξαιρετικά ενδιαφέρον, παρουσιάζει η επίδραση του αριθμού των νημάτων consumer στην προαναφερθείσα μετρική καθυστέρησης.

2.Επεξήγηση βασικών εννοιών και εντολών κώδικα

Αρχικά είναι σημαντικό να αναφερθούν κάποιες δομικής σημασίας έννοιες του παραγόμενου κώδικα. Αναλυτικά ορίζουμε 4 μεταβλητές τύπου struct. Το struct transactions_data αποτελείται από 4 μεταβλητές που σχετίζονται με τη τιμή, το σύμβολο της μετοχής, τον όγκο και τη χρονική στιγμή κάθε συναλλαγής μετοχών που πραγματοποιείται εκείνη τη στιγμή. Το struct candlestick αποτελείται από 7 μεταβλητές που σχετίζονται με την αρχική, τελική, μέγιστη, ελάχιστη τιμή, τον συνολικό όγκο, τον συνολικό αριθμό των συναλλαγών, καθώς και την μέση τιμή αυτών των συναλλαγών. Το struct queue σχετίζεται με την «ουρά» στην οποία θα αποθηκεύονται στοιχεία τύπου transactions_data και το struct workfunction ορίζει την επεξεργασία-δουλειά που θα πρέπει να γίνει για κάθε ένα από αυτά. Συγκεκριμένα το master thread που διαχειρίζεται τα εισερχόμενα μηνύματα μέσω του socket, τα αποθηκεύει ταυτόχρονα σε έναν πίνακα (buffer) μεγέθους BUF_SIZE αλλά και στην ουρά. Το κάθε στοιχείο αυτού του πίνακα και επομένως της ουράς όπως γίνεται εύκολα αντιληπτό είναι τύπου transactions_data. Σημαντική σημείωση για την ανάπτυξη του συγκεκριμένου κώδικα, θεωρήθηκε πως το μέγεθος της ουράς είναι πάντα μικρότερο από το μέγεθος του buffer, καθώς σε διαφορετική περίπτωση πρέπει να ενταχθούν επιπλέον δικλείδες ασφαλείας (mutexes), προκειμένου να αποφευχθούν περιπτώσεις race-conditions.



Εικόνα 1.

Αφού το master thread λάβει ένα μήνυμα, ελέγχει αν το σύμβολο που περιέχεται στο μήνυμα είναι ένα από αυτά που μας ενδιαφέρουν και βρίσκονται μέσα στο πίνακα char* symbols[]. Αν ο προηγούμενος έλεγχος είναι αληθής, τότε όπως προαναφέρθηκε αποθηκεύει το στοιχείο και στον buffer και στην ουρά και αυξάνει τον μετρητή του buffer κατά 1 buf_index++.
Όταν η ουρά είναι γεμάτη και δεν μπορούν να αποθηκευτούν σε αυτή άλλα στοιχεία τότε το master thread περιμένει μέχρι να λάβει το σήμα ότι αφαιρέθηκε ένα στοιχείο από αυτή p_thread_cont_wait(&fifo->notFull, fifo->mute).Κάτι το οποίο δυνητικά ισοδυναμεί με χάσιμο πληροφορίας, αφού μέσω του socket εξακολουθούν να έρχονται μηνύματα. Ωστόσο πρέπει να αναφέρουμε πως το συγκεκριμένο φαινόμενο δεν συμβαίνει συχνά, ειδικά στην περίπτωση που έχουμε

αρκετά νήματα consumers τα οποία είναι υπεύθυνα για το άδειασμα της ουράς. Η αρχικοποίηση αυτών των νημάτων consumer όσο και του νήματος που είναι υπεύθυνο για την παραγωγή και καταγραφή των candlesticks γίνεται από το master thread, όταν και εφόσον υπάρξει επιτυχημένη επικοινωνία με το Finnhub. Η εντολή αρχικοποίησης είναι pthread_create(&td[qn], NULL, &routine, NULL) , όπου με **qn** ορίζεται ο αριθμός των νημάτων και με **&routine** ορίζεται ένας double pointer στη συνάρτηση την οποία θα κληθεί να εκτελέσει το νήμα. Με παρόμοιο τόπο ορίζεται και το νήμα του candlestick: pthread_create(&candlestick_thread, NULL, &candlestickfunc, NULL).

Αφού αρχικοποιηθούν τα νήματα consumer, αμέσως προσπαθούν να λάβουν το mutex που σχετίζεται με την ουρά pthread_mutex_lock(fifo->mut) προκειμένου να αφαιρέσουν ένα στοιχείο από αυτή και να εκτελέσουν την απαραίτητη λειτουργία που τους αντιστοιχεί. Όλα αυτά με την προϋπόθεση ότι η ουρά δεν είναι άδεια και επομένως υπάρχει κάποιο στοιχείο προς εξαγωγή. Το νήμα αποθηκεύει αυτό το στοιχείο, σε ένα struct out τύπου workFunction και στη συνέχεια αποδεσμεύει τα mutexes που σχετίζονται με την ουρά, ώστε τα υπόλοιπα threads να ξεκινήσουν την εκτέλεση των λειτουργιών που τους αντιστοιχούν. Το στοιχείο τύπου workFunction που προαναφέραμε εμπεριέχει πληροφορία σχετικά με δύο μεταβλητές, Α) την τιμή του δείκτη (index), η οποία πληροφορεί το νήμα σχετικά με τη θέση του στοιχείου struct τύπου transaction_data μέσα στον buffer, Β) την διεύθυνση της συνάρτησης η οποία θα κληθεί να εκτελεστεί και θα λάβει ως όρισμα τη τιμή του δείκτη. Η ανάγνωση του δείκτη γίνεται μέσα από την εντολή int p = out.arg .Μετέπειτα ελέγχεται ποια μετοχή σχετίζεται με το στοιχείο struct τύπου workFunction με βάση το σύμβολο του, και αναλόγως κλειδώνεται από το thread το mutex που αντιστοιχεί στην συγκεκριμένη μετοχή π.χ. pthread_mutex_lock(&mutex_APPL). Εκεί ανάμεσα στο κλείδωμα και την απελευθέρωση του mutex μεσολαβεί η συνάρτηση που καλείται να εκτελέσει το thread και σχετίζεται με την καταγραφή των συναλλαγών που δέχεται ως είσοδο ο αλγόριθμος, σε διαφορετικά αρχεία για κάθε μετοχή. Πρέπει να αναφερθεί πως σε αυτό το σημείο παρατηρείται η δυνατότητα παραλληλοποίησης των διαδικασιών που πραγματοποιούνται από τα νήματα consumer, καθώς βλέπουμε ότι η καταγραφή δεδομένων συναλλαγών για διαφορετικά σύμβολα (μετοχές, κρυπτονομίσματα) μπορεί να γίνεται ταυτόχρονα-παράλληλα.

Η συνάρτηση καταγραφής των δεδομένων void* routine(int args), χρησιμοποιώντας το όρισμα της, λαμβάνει το στοιχείο μέσα από τον buffer transaction_data td = transactions[r_c], που μας παρέχει πληροφορίες σχετικά με την συναλλαγή που πρέπει να καταγραφεί μέσα στο εκάστοτε αρχείο. Έπειτα με την βοήθεια συναρτήσεων από την βιβλιοθήκη string.h, ορίζονται τα “μονοπάτια” (paths) των αρχείων στα οποία θα αποθηκευτούν όλες οι παραγόμενες πληροφορίες. Πρέπει να τονιστεί πως μετά από κάθε επιτυχημένη καταγραφή δεδομένων λαμβάνεται ο χρόνος σε milliseconds από το epoch (Ιανουάριος του 1970) μέσω της εντολής gettimeofday(&endwtime, NULL). Ο χρόνος αυτός στη συνέχεια θα αφαιρεθεί από τον χρόνο λήψης των δεδομένων της συναλλαγής timestrap, προκειμένου να υπολογιστεί ο ζητούμενος χρόνος καθυστέρησης. Συμπερασματικά η άθροιση των χρόνων καθυστέρησης για εκτελέσεις του κώδικα με διαφορετικό αριθμό νημάτων consumer είναι και αυτή που θα υποδείξει την αποτελεσματικότητα των τεχνικών παραλληλοποίησης που ενσωματώθηκαν στον τελικό κώδικα.

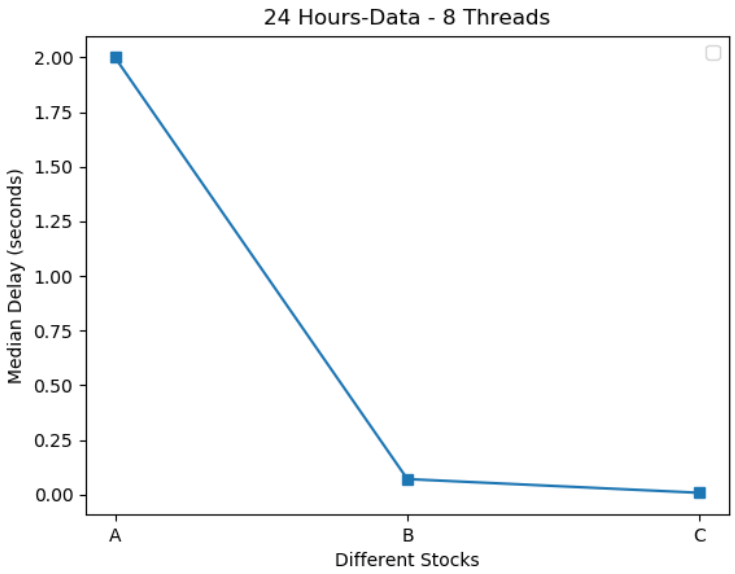
Όσον αναφορά το κομμάτι του candlestick, ο υπολογισμός του γίνεται μέσω του νήματος candlestick_thread, καλώντας την συνάρτηση void* candlestickfunc(). Στις αρχικές εντολές αυτής της συνάρτησης, αρχικοποιούνται κάποιες μεταβλητές ιδιαίτερης σημασία. Αναλυτικά ορίζεται ένας πίνακας δύο διαστάσεων candlestick_arr[15][4] στον οποίο αποθηκεύονται δεδομένα struct τύπου candlestick. Ο αριθμός των γραμμών 15 υποδηλώνει τα λεπτά για τα οποία θέλουμε να κρατήσουμε κάθε candlestick, μέχρι να το αντικαταστήσουμε από κάποιο χρονικά πιο πρόσφατο. Ο αριθμός των στηλών αντιστοιχούν στον αριθμό των συμβόλων, που επιθυμούμε να παρακολουθήσουμε. Επίσης για κάθε λεπτό αρχικοποιούνται όλες οι μεταβλητές του candlestick struct στο 0, με εξαίρεση της μεταβλητής ελάχιστης τιμής η οποία αρχικοποιείται σε έναν ικανοποιητικά μεγάλο αριθμό. Έπειτα υποβάλουμε το thread σε αδράνεια για 60 δευτερόλεπτα μέσω της συνάρτησης sleep(60), ώστε να επιτύχουμε δειγματοληψία των δεδομένων και εξαγωγή των απαραίτητων candlesticks κάθε 1 λεπτό. Επιπλέον λαμβάνουμε τον χρόνο σε milliseconds κατά τον οποίο "ξύπνησε" το thread και τον αποθηκεύουμε στην μεταβλητή milliseconds_candle . Αξίζει να σημειωθεί πως έχει προηγηθεί η αρχικοποίηση του μετρητή minute που συμβολίζει σε ποιο λεπτό από τα 15 βρισκόμαστε αλλά και του μετρητή candle_counter. Ο τελευταίος χρησιμοποιείται για την προσπέλαση δυνητικά όλων των στοιχείων του buffer που όπως αντιλαμβανόμαστε περιέχει τα πιο πρόσφατα χρονικά δεδομένα συναλλαγών. Στην περίπτωση που ο buffer έχει γεμίσει έστω και για πρώτη φορά if(buffer_full == 1), τότε θα προσπελάσουμε όλα τα στοιχεία του buffer και θα επιλέξουμε μόνο εκείνα, των οποίων ο χρόνος λήψης (timestrap) διαφέρει λιγότερο από 60000 milliseconds (1 λεπτό)από τον χρόνο milliseconds_candle if(milliseconds_candle - transactions[candle_counter].ts < 60000). Για κάθε σύμβολο-μετοχή παράγουμε τις ζητούμενες μετρικές, αρχική, τελική, μέγιστη ελάχιστη τιμή, τον συνολικό όγκο, τον αριθμό συναλλαγών που πραγματοποιήθηκαν και την μέση τιμή αυτών των συναλλαγών. Ελέγχουμε αν ο αριθμός συναλλαγών είναι διάφορος του 0 για αποφυγή σφαλμάτων και αυξάνουμε τον candle_counter μέχρι το μέγεθος του buffer. Τελικώς καταγράφουμε τα δεδομένα candlestick σε διαφορετικά αρχεία που αντιστοιχούν στα διαφορετικά σύμβολα.

Αφού παρέλθουν τα πρώτα 14 λεπτά η μεταβλητή after_15 γίνεται 1 και αρχίζει ο υπολογισμός του μέσου candlestick κάθε λεπτό. Το μέσο candlestick συλλέγει πληροφορίες σχετικά με τα candlesticks των 15 πιο πρόσφατων χρονικά λεπτών, αξιοποιώντας τον πίνακα δύο διαστάσεων που αναφέρεται παραπάνω. Πιο συγκεκριμένα το μέσο candlestick υπολογίζει και αποθηκεύει σε αρχεία την μέση τιμή των συναλλαγών και τον συνολικό όγκο των 15 πιο πρόσφατων χρονικά λεπτών. Ως τελευταίες εντολές της συνάρτησης έχουμε την αύξηση του μετρητή των λεπτών και την καταχώρηση της τιμής του 0, όταν αυτός γίνει ίσως με 15. Τέλος πρέπει να τονιστεί πως τα βασικά κομμάτια κώδικα των threads βρίσκονται σε μια while loop με όρισμα 1 προκειμένου να εκτελούνται ακατάπαυστα.

3. Παρουσίαση Πειραματικών Αποτελεσμάτων

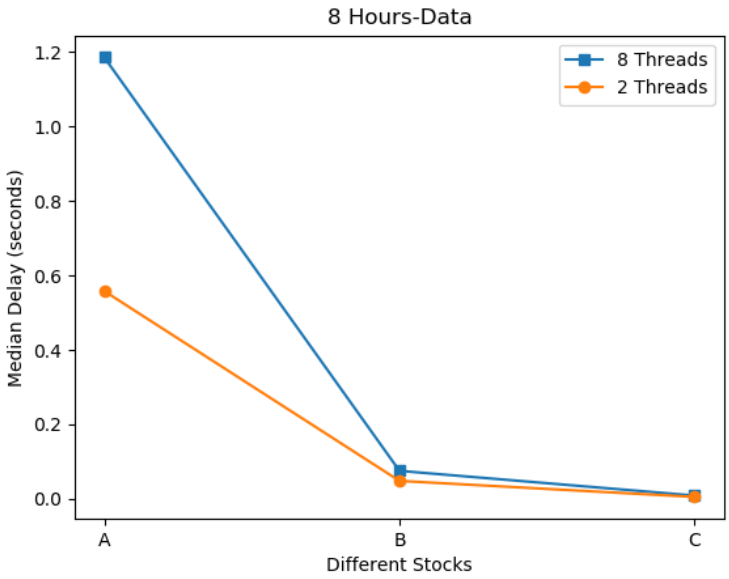
Για την παρούσα εργασία ο τελικός κώδικας εκτελέστηκε για ένα πλήθος παραμέτρων που σχετίζονται με την χρονική διάρκεια της εκτέλεσης αλλά και τον αριθμό νημάτων consumer. Ως πρώτη εκτέλεση ο κώδικας έτρεξε για ένα αρκετά μεγάλο χρονικό διάστημα μεγαλύτερο της μίας ημέρας παρακολουθώντας 4 σύμβολα (AMZN, APPL, BTCUSDT, IC MARKETS:1) και τα αποτελέσματα παρουσιάζονται στην εικόνα 2. Στον y άξονα έχουμε τον μέσο χρόνο καθυστέρησης (seconds)

αθροισμένο για κάθε διαφορετικό σύμβολο, ενώ στο χ άξονα βρίσκονται τα ονόματα των συμβόλων με την εξής αντιστοιχία A = BTCUSDT, B =AMZN, C =IC: MARKETS.



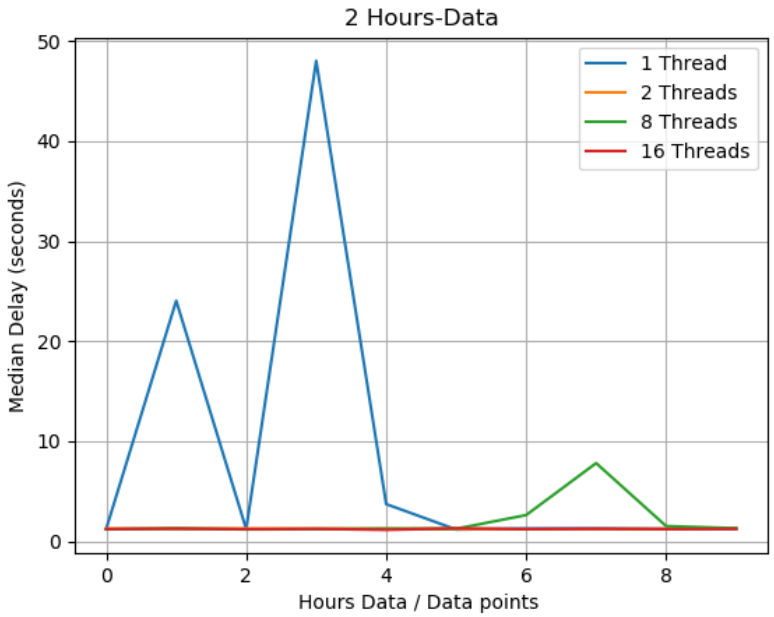
Εικόνα 2.

Στην συνέχεια ο κώδικας εκτελέστηκε για 8 ώρες με αριθμό νημάτων consumer ίσο με 8 και έπειτα ξανά για το ίδιο χρονικό διάστημα αλλά με αριθμό νημάτων consumer ίσο με 2. Τα αποτελέσματα αυτών των δυο εκτελέσεων παρουσιάζονται στην εικόνα 3.



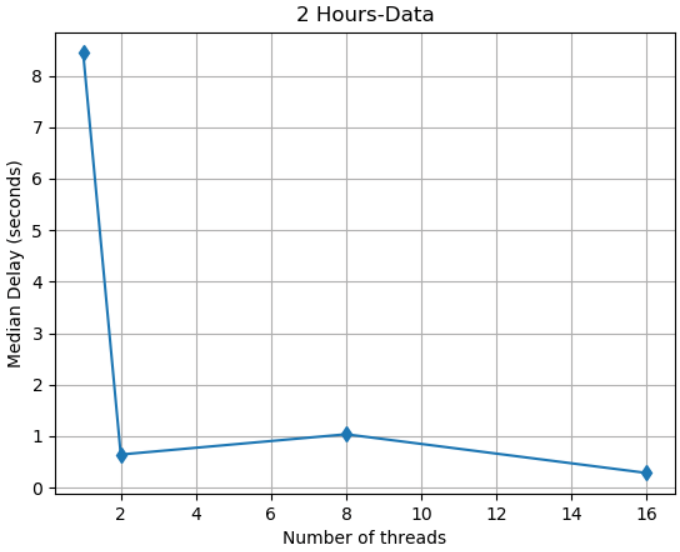
Εικόνα 3.

Τέλος προσπαθώντας πραγματικά να ελέγξουμε την επίδραση του αριθμού των νημάτων consumer στον μέσο χρόνο καθυστέρησης, αυξήσαμε τον αριθμό των μετοχών-συμβόλων που παρακολουθούμε. Πιο αναλυτικά ο κώδικας εκτελέστηκε παρακολουθώντας ταυτόχρονα τις μετοχές AMZN, APPL, BTCUSDT, IC MARKETS:1, MSFT, BYND, EXCOF, UPOW, για χρονικό διάστημα 2 ωρών. Ωστόσο δυστυχώς τα παραγόμενα αποτελέσματα του αλγορίθμου σχετίζονταν μόνο με το σύμβολο BTCUSDT, το οποίο υποδηλώνει έλλειψη μηνυμάτων εισόδου από το Finnhub που σχετίζονται με τα υπόλοιπα σύμβολα. Η συλλογή των τελικών αποτελεσμάτων που φαίνεται στην εικόνα 4, πραγματοποιήθηκε χωρίζοντας το χρονικό διάστημα των 2 ωρών σε κομμάτια ίσα με τα data points (10) και βρίσκοντας τον αθροισμένο μέσο χρόνο καθυστέρησης για κάθε ένα από αυτά τα κομμάτια και για ένα πλήθος διαφορετικών αριθμών νημάτων consumer.



Εικόνα 4.

Τα πειραματικά αποτελέσματα της εικόνας 5, προέκυψαν υπολογίζοντας τον αθροισμένο μέσο όρο καθυστέρησης για κάθε διαφορετικό αριθμό νημάτων consumer.



Εικόνα 5.

Ο τελικός κώδικας μαζί με τα παραγόμενα αρχεία εκτέλεσης βρίσκονται εδώ:

https://github.com/antonis-fav/RTES-Final-Assignement