Εργασία Ενσωματωμένα Συστήματα Πραγματικού Χρόνου

Υλοποιήσαμε ένα πρόγραμμα σε κώδικα C το οποίο τρέχει στο Raspberry Pi Zero 2W μέσω cross-compilation, και αποθηκεύει τα trade data και candlestick data των **Ethereum to USD**, **Amazon**, **Microsoft**, **Apple** σε αντίστοιχα αρχεία. Το πρόγραμμα υλοποιήθηκε έτσι ώστε να μπορεί να τρέχει χωρίς προβλήματα και για πολλές μέρες συνεχόμενα, όπως ζητείται στην εργασία. Φυσικά, σε περίπτωση κάποιου σφάλματος το πρόγραμμα τερματίζει. Παρ'όλα αυτά, διαπιστώθηκε μέσω testing ότι τέτοια σφάλματα είναι γενικά σπάνια και δεν επηρεάζουν τη ροή του προγράμματός μας. Παρακάτω θα εξηγήσουμε την λογική του προγράμματός μας λεπτομερώς.

Αρχή του προγράμματος αποτελεί η συνάρτηση **main**, και είναι αυτή που δημιουργεί αρχικά τα αρχεία για τα trade data, τα candlesticks και τη μέση τιμή, με ονόματα που σχετίζονται με το codeword του κάθε request στο Finnhub. Έπειτα, μέσω της main συνδέουμε το σήμα **SIGINT** (Ctr+C) με τη συνάρτηση **sigint_handler**. Αρχικοποιούμε επίσης την ουρά fifo που θα αποθηκεύει τα data του producer και από την οποία θα λαμβάνει data ο consumer όπως θα εξηγήσουμε παρακάτω. Έπειτα, αρχικοποιούνται και οι ουρές volumes και mean_prices, οι οποίες αποθηκεύουν τον όγκο ανά λεπτό και τη μέση τιμή ανα λεπτό για το κάθε trade. Οι ουρές αυτές είναι FIFO με μέγεθος 15, και αποθηκεύονται σε αυτές κάθε λεπτό τα κατάλληλα δεδομένα. Συνεπώς, οι ουρές έχουν πάντα αποθηκευμένα τα δεδομένα των τελευταίων 15 λεπτών. Τέλος, στη main δημιουργούνται 3 threads, οι **producer**, **consumer** και **counter** τα οποία τρέχουν παράλληλα. Το πρόγραμμα τερματίζει όταν τα threads επιστρέψουν NULL, οπότε και επιστρέφει η main.

Producer: Το producer thread αρχικά υλοποιεί τον κώδικα ώστε να έχουμε δημιουργία του context και σύνδεση με το Finnhub μέσω ενός websocket. Έπειτα, για όσο χρόνο δεν ζητείται έξοδος του προγράμματος, χρησιμοποιούμε την συνάρτηση **lws_service()** για να πάρουμε δεδομένα ως απάντηση από το server του Finnhub. Όταν συνδεόμαστε πρώτη φορά, στέλνουμε ένα μήνυμα εγγραφής για τα trades τα οποία θέλουμε να παρακολουθούμε. Το Finnhub μας απαντάει με μηνύματα σε μορφή JSON, τα οποία περιέχουν το trade data. Τα strings αυτά τα δίνουμε στη συνάρτηση **parse_message()** που τα κάνει parse. Παρατηρήθηκε ότι το Finnhub κάνει sign out τον χρήστη σε δεδομένες χρονικές στιγμές, ή παρουσιάζει προβλήματα σύνδεσης. Κάθε φορά που συμβαίνει αυτό, το **interrupted** flag γίνεται 1, το context καταστρέφεται και δημιουργείται καινούριο, οπότε επιδιώκεται η επανασύνδεση.

Η συνάρτηση **parse_message()** παίρνει το string που περιέχει την JSON πληροφορία και το επεξεργάζεται έτσι ώστε να εξάγει τα κατάλληλα δεδομένα για τη δημιουργία των candlestick και την αποθήκευσή τους. Στην συνάρτηση αυτή μπορούν να υπάρξουν πολλά σφάλματα αν το JSON αρχείο δεν έρθει με κατάλληλη μορφή από το Finnhub. Στην περίπτωση αυτή, η συνάρτηση επιστρέφει και δεν βάζει τίποτα στην ουρά fifo. Επίσης, υπάρχουν περιπτώσεις όπου το Finnhub αποσυνδέει αυτόματα τον χρήστη και όταν αυτός ξανασυνδέεται, του επιστρέφει μόνο pings. Αυτό το πρόβλημα λύθηκε ανιχνεύοντας τα pings, και αν έρθουν 3 στη σειρά χωρίς κάποια παρεμβολή από data, το πρόγραμμα αποσυνδέεται και ξανασυνδέεται στο Finnhub.

Αν το string έχει έρθει στην κατάλληλη μορφή, εξάγουμε την πληροφορία του χρόνου του Finnhub server, όγκου, τιμής, ονόματος μετοχής και χρόνου άφιξης και την αποθηκεύουμε σε

ένα struct τύπου **input_info** που φτιάξαμε εμείς. Το struct αυτό αποθηκεύεται στο fifo queue παίρνοντας το κλειδί του mutex και δίνοντάς το πίσω όταν τελειώσει η διαδικασία, ενώ αν αυτή είναι full, περιμένουμε μέχρι ο consumer να αφαιρέσει ένα στοιχείο ώστε να προσθέσουμε τα νέα δεδομένα.

Τέλος, όταν το **exit_requested** flag γίνει true, ο producer τρέχει κώδικα ώστε να απελευθερωθεί η μνήμη και να κλείσει το πρόγραμμα κομψά. Αρχικά κάνει update ένα flag που έχει να κάνει με την έξοδο του consumer και στέλνει σήμα που ξυπνάει τον consumer. Έπειτα, ο producer κοιμάται αφήνοντας αρκετό χρόνο στον consumer να καταναλώσει τα τελευταία δεδομένα από το queue και να τερματίσει. Τέλος, διαλύονται όλα τα mutex, κλείνουν τα αρχεία και απελευθερώνεται η μνήμη των dynamically allocated μεταβλητών μας, και το νήμα επιστρέφει NULL.

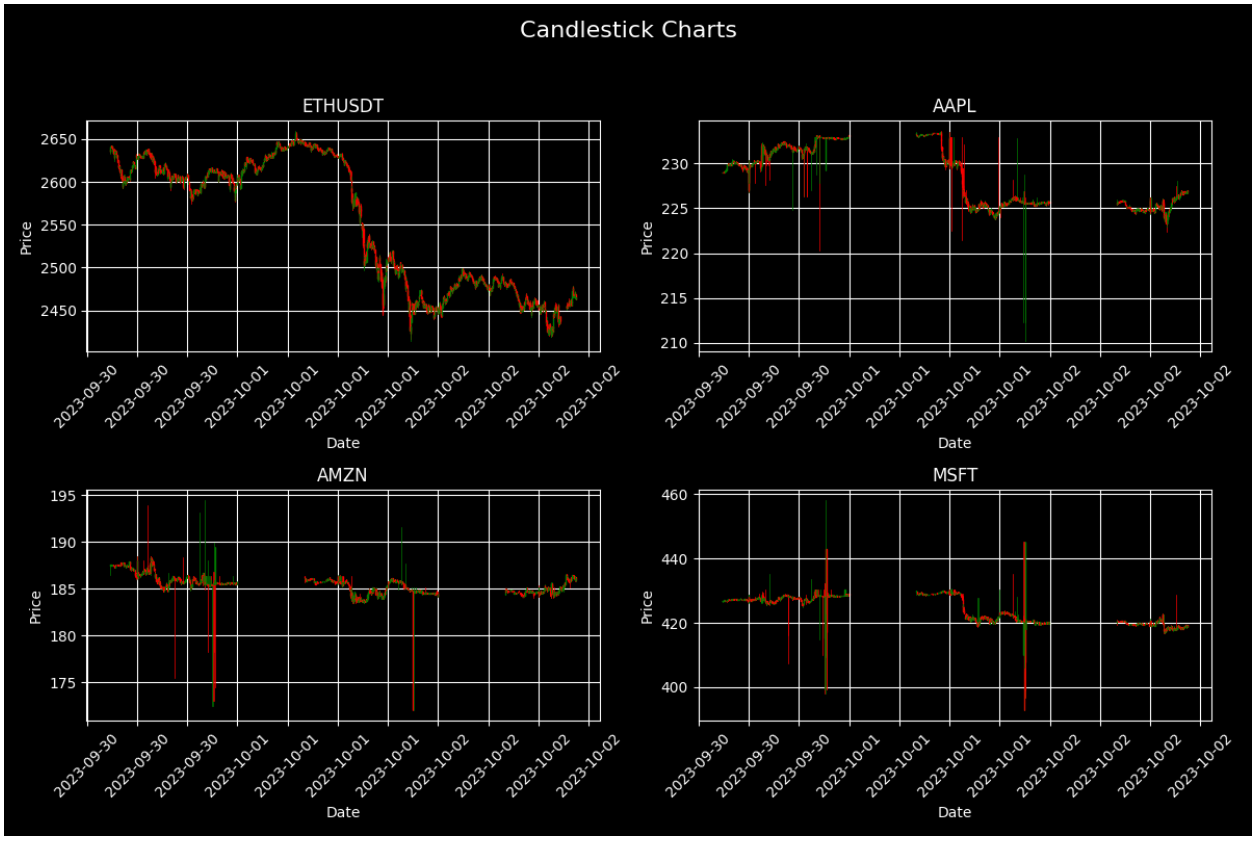
Consumer: Το thread consumer συνεχώς εξάγει items από το fifo queue και τα αποθηκεύει στο κατάλληλο αρχείο. Αν το queue είναι κενό, περιμένει να μπει τουλάχιστον ένα item στο queue ώστε να συνεχίσει. Κάθε φορά πριν διαγράψει κάτι κλειδώνει το mutex, ξεκλειδώνοντάς το όταν η διαδικασία τελειώσει. Σε περίπτωση που δοθεί το flag **consumer_can_exit** καθώς και η ουρά fifo έχει αδειάσει, δηλαδή έχουν γίνει consumed ολα τα items της, το νήμα επιστρέφει NULL.

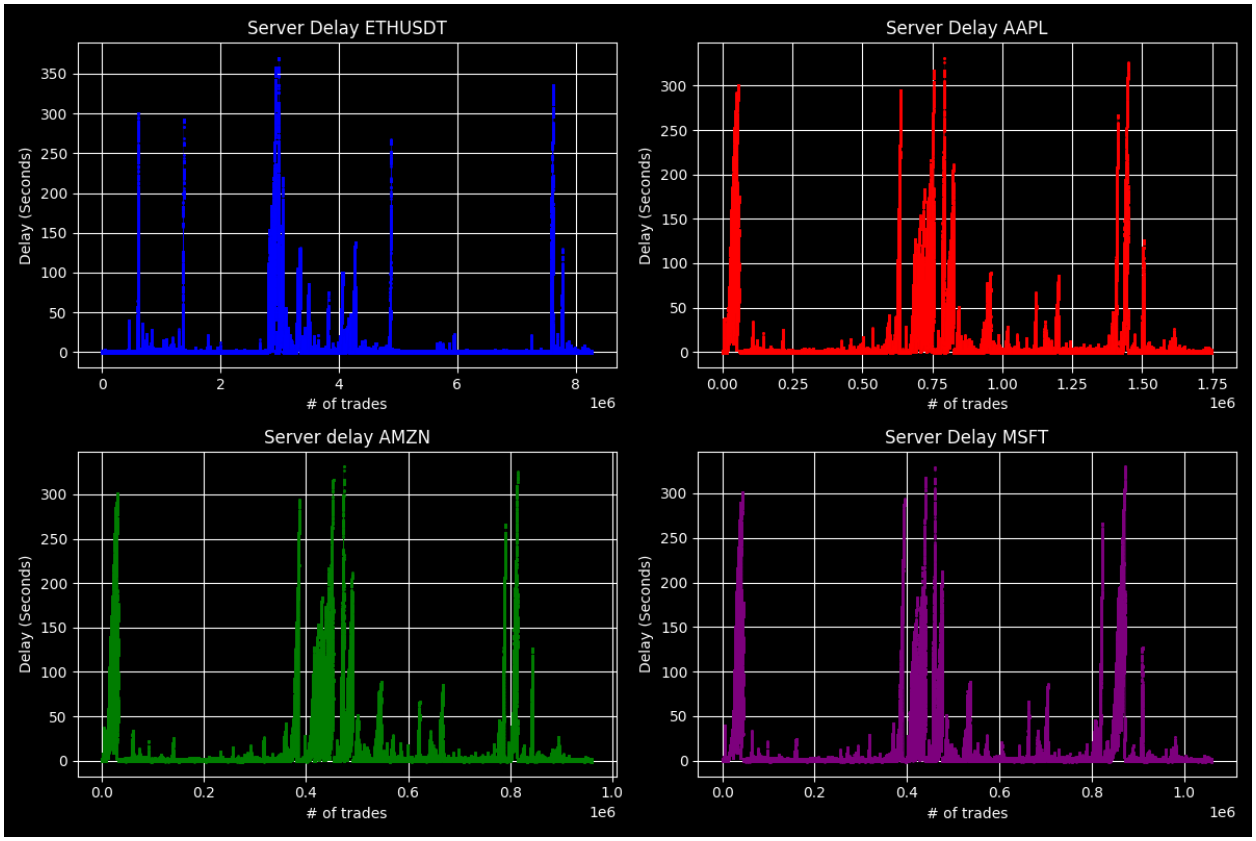
Ο τρόπος που γίνεται η αποθήκευση σε αρχεία είναι μέσω της **save_string()** συνάρτησης. Η συνάρτηση αυτή παίρνει ως είσοδο τα δεδομένα σε μορφή struct input_info και γράφει στο κατάλληλο αρχείο τα δεδομένα, καθώς και τον χρόνο εγγραφής. Επίσης, κάνει υπολογισμούς του max, min, start price, final price, και συνολικού volume που αγοράστηκε για το κάθε trade κλειδώνοντας το cntr_mut mutex όταν χειρίζεται αυτές τις μεταβλητές για ασφάλεια. Τα παραπάνω είναι χρήσιμα για τη δημιουργία candlestick και γίνονται reset κάθε λεπτό ώστε να υπολογιστούν τα καινούρια δεδομένα.

Counter: Το counter thread κοιμάται και ξυπνάει κάθε 1 λεπτό ώστε να αποθηκεύσει τα δεδομένα του candlestick. Συγκεκριμένα, χρησιμοποιήσαμε την συνάρτηση **pthread_cond_timedwait()**, η οποία μπορεί να ξυπνήσει το thread είτε μέσω του timeout ενός clk, είτε μέσω ενός σήματος που ονομάσαμε cond. Το σήμα αυτό χρειάζεται ώστε να ξυπνήσουμε το counter thread όταν στείλουμε το **SIGINT** signal και να τερματίσει το πρόγραμμα απευθείας. Αν λοιπόν ο counter έχει ξυπνήσει λόγω του cond, το νήμα επιστρέφει NULL. Αν έχει ξυπνήσει λόγω timeout, προχωράμε στην δημιουργία του candlestick και του αρχείου με τον συνολικό όγκο και την μέση τιμή. Η αποθήκευση των τιμών του candlestick είναι αρκετά εύκολη υπόθεση αφού έχουμε ήδη τα δεδομένα αυτά έτοιμα από τον consumer. Υπολογίζουμε και την μέση τιμή των τιμών για το κάθε σύμβολο και την αποθηκεύουμε στην ουρά mean_prices. Αποθηκεύουμε επίσης στην ουρά volumes το σύνολο του όγκου που αγοράστηκε για το κάθε λεπτό. Τέλος, αποθηκεύουμε τον κινούμενο μέσο όρο των τιμών των τελευταίων 15 λεπτών στο κατάλληλο αρχείο, καθώς και τον όγκο που αγοράστηκε στο ίδιο αρχείο. Πριν ξανακοιμηθεί ο counter κάνει reset όλες τις μεταβλητές που χρησιμοποιούνται για τη δημιουργία candlestick έτσι ώστε να μην έχουμε προβλήματα στους υπολογισμούς ανα λεπτό.

Κλείνοντας, όταν πατάμε το κουμπί (CTR+C), καλείται η συνάρτηση **sigint_handler()** η οποία κάνει update το exit_requested flag και στέλνει το cond σήμα ώστε να ξυπνήσει ο counter.

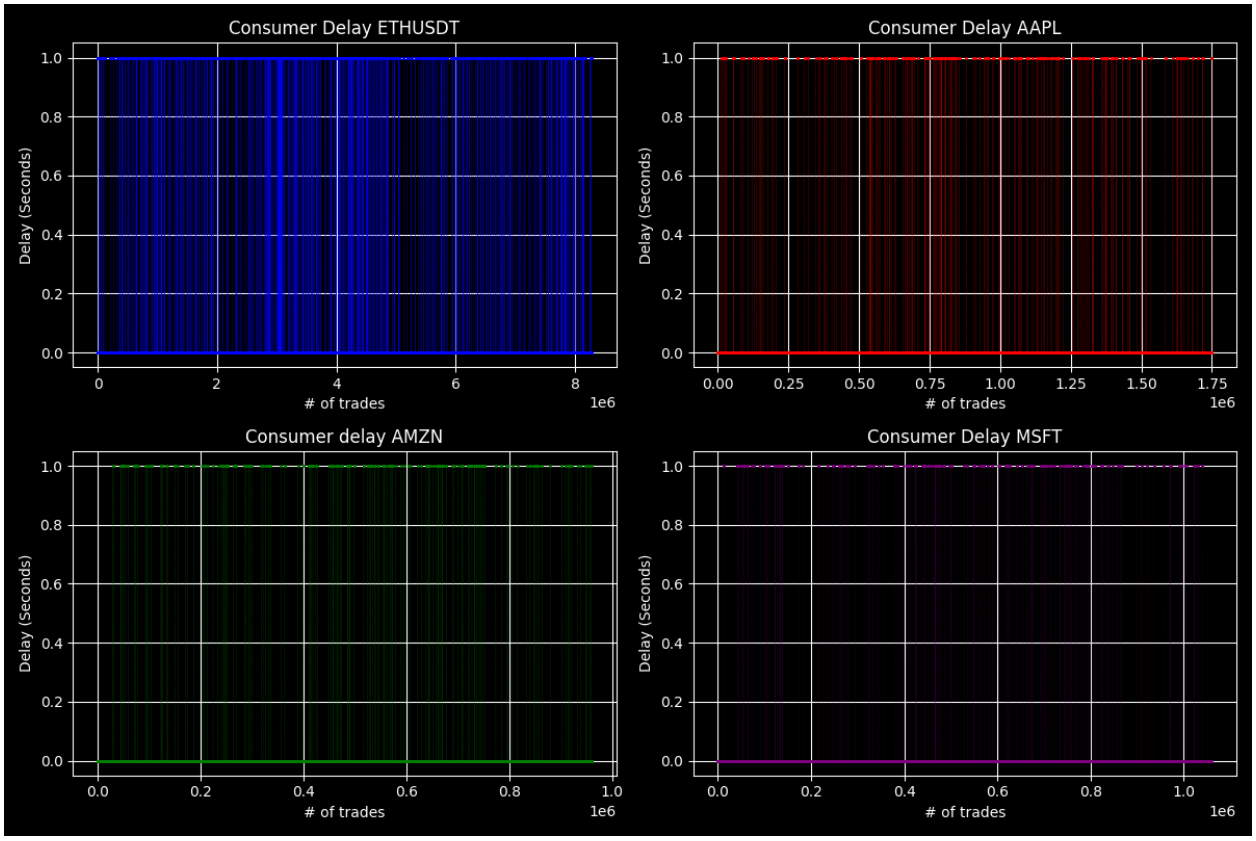
Εργασία Ενσωματωμένα Συστήματα Πραγματικού Χρόνου



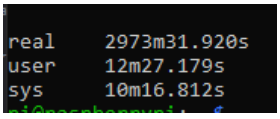


Αθανάσιος Κωνσταντής - ΑΕΜ: 10537

Εργασία Ενσωματωμένα Συστήματα Πραγματικού Χρόνου



Παραπάνω δίνονται τα γραφήματα των αποτελεσμάτων, τρέχοντας το πρόγραμμα για πάνω από 48 ώρες στο raspberry pi όπως φαίνεται στην παρακάτω εικόνα.



Το σύστημα έτρεξε για 49.55 ώρες, και η CPU ήταν απασχολημένη για 22m43s από όλον αυτόν τον χρόνο. Σε ποσοστό, μπορούμε να πούμε ότι η CPU έτρεχε κατά:

$$\frac{12\cdot 60+27.179+10\cdot 60+16.812}{2973\cdot 60+31.920}\cdot 100\% = 0.7645\%$$

Παρατηρούμε ότι η ακρίβεια στα αποτελέσματά μας είναι στην τάξη των second. Αυτό έγινε από λάθος, το οποίο παρατηρήθηκε πολύ αργά και δεν μπορούσε να διορθωθεί για να έχουμε πιο καλά αποτελέσματα. Παρ'όλα αυτά, η διόρθωσή του είναι σχετικά απλή και συνιστά την αντικατάσταση όλων των κλήσεων της time(NULL) συνάρτησης (η οποία έχει ακρίβεια second) με τη συνάρτηση gettimeofday().

Επίσης, παρατηρούμε ότι για τις μετοχές, συγκεκριμένες ώρες τα βράδια δεν έχουμε καθόλου δεδομένα, κάτι που είναι λογικό με το άνοιγμα και κλείσιμο του χρηματιστηρίου. Αντίστοιχα, με το κρυπτονόμισμα έχουμε συνεχές datastream.
Παραθέτουμε δείκτη της εργασίας στο cloud:
https://github.com/Athanasios-Konstantis/Embedded-Systems

Αθανάσιος Κωνσταντής - AEM: 10537