ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ

Ιεράρχηση προτεραιοτήτων μηνυμάτων σε ένα gRPC κανάλι

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΡΕΤΣΑ Κ. ΜΑΡΙΑ

Επιβλέπων : Εμμανουήλ Βαρβαρίγος
Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ
ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ, ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ
ΠΛΗΡΟΦΟΡΙΚΗΣ

Ιεράρχηση προτεραιοτήτων μηνυμάτων σε ένα gRPC κανάλι

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

ΡΕΤΣΑ Κ. ΜΑΡΙΑ

Επιβλέπων : Εμμανουήλ Βαρβαρίγος
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την 31η Μαΐου 2022

.....	.....	.....
Εμμανουήλ Βαρβαρίγος	Θεοδώρα Βαρβαρίγου	Συμεών Παπαβασιλείου
Καθηγητής Ε.Μ.Π.	Καθηγήτρια Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Ιούνιος 2022

.....

Μαρία Κ. Ρέτσα

Διπλωματούχα Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.

Η παρούσα διπλωματική εργασία εκπονήθηκε σε συνεργασία με τη Nokia Solutions and Networks Hellas Single Member S.A.

ΠΕΡΙΛΗΨΗ

Το RPC (Remote Procedure Call) ή κλήση απομακρυσμένης διαδικασίας, είναι ένα πρωτόκολλο που χρησιμοποείται για την επικοινωνία μεταξύ διεργασιών – προγραμμάτων που βρίσκονται σε διαφορετικά συστήματα, τα οποία επικοινωνούν μεταξύ τους μέσω κάποιου δικτύου, χωρίς να χρειάζεται οι διεργασίες να γνωρίζουν τις λεπτομέρειες του δικτύου αυτού. Το RPC καθιστά το μοντέλο προγραμματισμού διακομιστή – πελάτη ιδιαίτερα εύκολο στην υλοποίησή του, γι' αυτό και είναι ιδιαίτερα δημοφιλές στα σύγχρονα κατανεμημένα συστήματα και σε αρχιτεκτονικές μικροϋπηρεσιών.

Το gRPC (google RPC) είναι ένα ανοιχτού κώδικα framework για την υποστήριξη του RPC πρωτοκόλλου που βασίζεται στα Google Protocol Buffers και λειτουργεί πάνω από HTTP/2 συνδέσεις. Παρουσιάζει πολλά πλεονεκτήματα τόσο στο κομμάτι της υλοποίησης και συντήρησης ενός συστήματος, όσο και σε επίπεδο απόδοσης. Το gRPC είναι ανεξάρτητο της πλατφόρμας και της γλώσσας προγραμματισμού που χρησιμοποιείται καθώς έχει τη δυνατότητα να παράγει αυτόματα κώδικα σε πληθώρα γλωσσών, ο οποίος μπορεί να χρησιμοποιηθεί από τη βασική εφαρμογή. Επίσης με τη χρήση του HTTP/2 πρωτοκόλλου επιτυγχάνεται πολυπλεξία των διαφόρων streams μεταξύ πελάτη και διακομιστή πάνω από την ίδια TCP σύνδεση χωρίς να απαιτείται να δημιουργηθεί, εδραιωθεί και τέλος να τερματιστεί μία νέα σύνδεση για κάθε νέα κλήση απομακρυσμένης διαδικασίας.

Στο gRPC η κάθε ροή που εξυπηρετεί μία κλήση απομακρυσμένης διαδικασίας είναι ισότιμη, και επομένως δεν υπάρχει κάποιος τρόπος διαφοροποίησης των rpc με βάση την προτεραιότητά τους. Υπάρχουν περιπτώσεις όμως στις οποίες κάτι τέτοιο θα ήταν επιθυμητό. Ο διαχωρισμός των rpc θα μπορούσε να επιτευχθεί με τη δημιουργία νέου καναλιού με ξεχωριστή σύνδεση για κάθε ένα από αυτά, ωστόσο τότε θα υπήρχε χρήση περισσότερων πόρων, και θα χανόταν το πλεονέκτημα της ύπαρξης μόνο μίας TCP σύνδεσης στο gRPC.

Στόχος της διπλωματικής αυτής ήταν μια προσέγγιση επίλυσης του προαναφερθέντος προβλήματος. Συγκεκριμένα, αναπτύχθηκε ένα κομμάτι λογισμικού στην πλευρά του πελάτη, πριν από το gRPC επίπεδο, το οποίο έχει το ρόλο ενός χρονοδρομολογητή: δέχεται από τον πελάτη αιτήματα να κάνει κλήσεις απομακρυσμένης διαδικασίας με διαφορετικές προτεραιότητες, και βάσει αυτών επιτρέπει στην κλήση με την μεγαλύτερη προτεραιότητα κάθε φορά να καταλαμβάνει το κανάλι, χωρίς να πολυπλέκεται, ώστε να στέλνεται και να εξυπηρετείται πρώτη. Η λύση αυτή προσθέτει μία χρονική επιβάρυνση λόγω των καθυστερήσεων στη διαδικασία της χρονοδρομολόγησης, ωστόσο επιτυγχάνει καλύτερο QoS (ποιότητα υπηρεσίας) σε σχέση με την δημιουργία νέας σύνδεση για κάθε προτεραιότητα, καθώς δεν υπάρχει σπατάλη περισσότερων πόρων από όσους χρειάζονται.

Λέξεις – κλειδιά: κλήση απομακρυσμένης διαδικασίας, πρωτόκολλο, gRPC, HTTP/2, πελάτης, διακομιστής, χρονοδρομολογητής, ποιότητα υπηρεσίας, TCP σύνδεση, προτεραιότητα κλήσης, ιεράρχηση προτεραιοτήτων, εξοικονόμηση πόρων, πολυπλεξία πακέτων

ABSTRACT

RPC (Remote Procedure Call) is a software protocol used for the communication between processes that live in different systems, which are connected through a network, without the need for the processes to understand the network details. RPC makes the implementation of a client – server model easy, and that is the reason why it is extremely popular in modern distributed systems and microservices architectures.

GRPC (google RPC) is an open-source RPC framework based on Google Protocol Buffers that uses HTTP/2 connections. It has many advantages concerning the implementation and maintenance, as well as the performance of a system. GRPC is a cross-platform and cross-language framework, as it provides auto-generated code in many languages which can be used by the main application. Additionally, because of the use of the HTTP/2 protocol, it is possible to achieve multiplexing of the streams between the client and the server over the same TCP connection, without the need of the creation, establishment and termination of a new TCP connection for each rpc.

In gRPC, each stream serving a rpc is coequal, so there is not a way to distinct the rpcs based on their priority. There are cases in which this would be really useful and desirable, though. The discrimination of the rpcs could be achieved with the creation of a new channel with a different connection for each one of them, although then there would be use of more resources, and the advantage from the existance of a sole TCP connection in gRPC would be lost.

Purpose of this thesis is to approach a solution for the above problem. Specifically, a software component was developed on the client side, before the gRPC level, that serves the purpose of a scheduler: it receives requests from the client to send rpcs with different priorities, and based on them it allows only the rpc with the greatest priority to occupy the channel, without multiplexing, so that it is served first among the rest. This solution adds some time overhead due to delays in the scheduling process. Nevertheless, greater QoS (quality of service) is accomplished contrary to the creation of a new TCP connection for each priority, which leads to waste of resources.

Keywords: remote procedure call, rpc, protocol, gRPC, HTTP/2, protocol buffers, client, server, scheduler, quality of service, TCP connection, rpc priority, prioritization, resource saving, packet multiplexing

ΕΥΧΑΡΙΣΤΙΕΣ

Η παρούσα διπλωματική εργασία ολοκληρώθηκε με τη συμβολή πολλών ανθρώπων τους οποίους στο σημείο αυτό θα ήθελα να ευχαριστήσω.

Καταρχάς, θα ήθελα να ευχαριστήσω τον επιβλέποντα καθηγητή μου κ. Εμμανουήλ Βαρβαρίγο που μου έδωσε την ευκαιρία να ασχοληθώ με ένα θέμα μέσω του οποίου μπόρεσα να εμπλουτίσω τις γνώσεις και τις ικανότητές μου, καθώς και τον κ. Παναγιώτη Κόκκινο και κ. Αριστοτέλη Κρέτση για τη συμβολή τους. Θα ήθελα επίσης να ευχαριστήσω τον κ. Ηλία Γράβαλο, του οποίου ιδέα ήταν το θέμα της παρούσας διπλωματικής, καθώς και όλους τους συνεργάτες του από την NOKIA Hellas οι οποίοι με καθοδηγούσαν συνεχώς στην πορεία της εργασίας αυτής.

Τέλος, θα ήθελα να ευχαριστήσω τους φίλους, τις φίλες και την οικογένειά μου για τη στήριξή τους μέχρι και σήμερα.

ΠΕΡΙΕΧΟΜΕΝΑ

ΠΕΡΙΛΗΨΗ 5
ABSTRACT 8
ΕΥΧΑΡΙΣΤΙΕΣ 10
ΠΕΡΙΕΧΟΜΕΝΑ 12
ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ 14
ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ 16
ΔΟΜΗ ΤΗΣ ΔΙΠΛΩΜΑΤΙΚΗΣ 17
ΒΑΣΙΚΕΣ ΑΡΧΕΣ CLIENT-SERVER ΕΠΙΚΟΙΝΩΝΙΑΣ ΜΕΣΩ RPC 18
Εισαγωγικές έννοιες 18
Σχετικά με το HTTP/2 20
Εισαγωγή 20
Streams 21
Frames 21
Είδη frames 22
Σχετικά με το gRPC 24
Πρωτόκολλο RPC 24
Είδη rpc μεθόδων στο gRPC 26
Protocol Buffers 26
Πλεονεκτήματα gRPC 28
Το πρόβλημα των προτεραιοτήτων στο gRPC 29
Επιπρόσθετα εργαλεία 30
Wireshark 30
Ghz 31
ΙΕΡΑΡΧΗΣΗ ΠΡΟΤΕΡΑΙΟΤΗΤΩΝ ΣΤΟ gRPC 34
Υλοποίηση χρονοδρομολογητή 35
Εισαγωγή 35
Αρχιτεκτονική υλοποίησης 35
Ανάπτυξη διακομιστή συστήματος 36
Υλοποίηση σε C++ με processes 36
Πρώτη υλοποίηση σε C++ με threads 38
Δεύτερη υλοποίηση σε C++ με threads 41
Πρώτη υλοποίηση σε Go με goroutines 43

Δεύτερη υλοποίηση σε Go με goroutines ... 45
Επεκτάσεις βασικού αλγορίθμου ... 47
Εισαγωγή ... 47
Το πρόβλημα του starvation ... 47
Επίλυση με aging ... 48
Επίλυση με time-out ... 49
Υλοποίηση χρονοδρομολογητή με time-out ... 49
ΑΠΟΤΕΛΕΣΜΑΤΑ ... 51
Κλήσεις απομακρυσμένης διαδικασίας ... 52
Αποτελέσματα μετρήσεων ... 55
Wireshark ... 55
Ghz ... 59
Επίδοση scheduler ... 69
Εισαγωγή ... 69
Επίδοση υλοποίησης σε C++ με processes ... 69
Επίδοση πρώτης υλοποίησης σε C++ με threads ... 71
Επίδοση δεύτερης υλοποίησης σε C++ με threads ... 72
Επίδοση πρώτης υλοποίησης σε Go με goroutines ... 74
Επίδοση δεύτερης υλοποίησης σε Go με goroutines ... 75
Επίδοση υλοποίησης χρονοδρομολογητή με time-out ... 77
ΣΥΜΠΕΡΑΣΜΑΤΑ ... 79
Συμπεράσματα ... 80
Μελλοντικές επεκτάσεις ... 83
Πηγαίος κώδικας ... 84
ΒΙΒΛΙΟΓΡΑΦΙΑ ... 85

ΚΑΤΑΛΟΓΟΣ ΕΙΚΟΝΩΝ

Εικόνα 1: Αφαιρετική απεικόνιση πρωτοκόλλου HTTP/2 20
Εικόνα 2: HTTP/2 πλαίσια σε TCP σύνδεση 21
Εικόνα 3: Δομή ενός HTTP/2 frame 22
Εικόνα 4: Σχηματική αναπαράσταση μιας κλήσης απομακρυσμένης διαδικασίας 24
Εικόνα 5: Σχηματική αναπαράσταση gRPC για έναν server και δύο clients 25
Εικόνα 6: Ροή εργασιών των Protocol Buffers 27
Εικόνα 7: Παράδειγμα .proto file 27
Εικόνα 8: Χειρισμός control και data από το gRPC 279
Εικόνα 9: Επιθυμητός χειρισμός control και data 279
Εικόνα 10: gRPC πακέτο αιτήματος 30
Εικόνα 11: gRPC πακέτο απάντησης 31
Εικόνα 12: Παράδειγμα χρήσης του εργαλείου ghz 32
Εικόνα 13: Αποτελέσματα εκτέλεσης πειράματος σε μορφή html 33
Εικόνα 14: Αρχιτεκτονική του συστήματος 36
Εικόνα 15: Αρχιτεκτονική συστήματος με C++ processes 37
Εικόνα 16: Πρώτη αρχιτεκτονική συστήματος με C++ threads 39
Εικόνα 17: Δεύτερη αρχιτεκτονική συστήματος με C++ threads 41
Εικόνα 18: Πρώτη αρχιτεκτονική συστήματος με Go threads 43
Εικόνα 19: Δεύτερη αρχιτεκτονική συστήματος με Go threads 45
Εικόνα 20: Priority based δρομολόγηση 47
Εικόνα 21: Starvation στην priority based δρομολόγηση 48
Εικόνα 22: Ορισμός μεθόδου SayHello 52
Εικόνα 23: Ορισμός μηνυμάτων HelloRequest και HelloReply 52
Εικόνα 24: Ορισμός μεθόδου ComputeMean 52
Εικόνα 25: Ορισμός μηνυμάτων IntNumber και FloatNumber 52
Εικόνα 26: Ορισμός μεθόδου ComputeMeanRepeated 53
Εικόνα 27: Ορισμός μηνύματος FloatNumberList 53
Εικόνα 28: Ορισμός μεθόδου SendLongString 53
Εικόνα 29: Ορισμός μηνύματος LongString 53
Εικόνα 30: Ορισμός μεθόδου ComputeMeanRepeatedOrSendLongString 54
Εικόνα 31: Ορισμός μηνυμάτων FloatOrLongString και FloatNumberListOrLongString .. 54
Εικόνα 32: Αρχική αρχιτεκτονική client – server 55
Εικόνα 33: Παρακολούθηση της κλήσης SayHello στο Wireshark 55

Εικόνα 34: Αρχιτεκτονική client multithreading 56
Εικόνα 35: Παρακολούθηση των κλήσεων από τον multithreading client στο Wireshark .. 57
Εικόνα 36: Ορισμός maxConcurrentStreams = 1 στον Go client 58
Εικόνα 37: Παρακολούθηση των κλήσεων από τον multithreading client με maxConcurrentStreams = 1 στο Wireshark..... 58
Εικόνα 38: Average latency, rps, total time και total time/calls συναρτήσει του concurrency 61
Εικόνα 39: Average latency, rps, total time και total time/calls συναρτήσει του αριθμού των cpu cores..... 63
Εικόνα 40: Average latency, rps, total time και total time/calls συναρτήσει των δεδομένων 64
Εικόνα 41: Average latency, rps, total time και total time/calls συναρτήσει του συνολικού αριθμού των κλήσεων 66
Εικόνα 42: Requests per second συναρτήσει παραμέτρων για δύο είδη rpc και multiplexing 67
Εικόνα 43: Total time/calls συναρτήσει παραμέτρων για δύο είδη rpc και multiplexing 68
Εικόνα 44: Παράδειγμα input file 69
Εικόνα 45: Μέσος χρόνος εκτέλεσης rpc για την C++ processes υλοποίηση..... 70
Εικόνα 46: Συνολικός χρόνος εκτέλεσης rpc για την C++ processes υλοποίηση..... 71
Εικόνα 47: Μέσος χρόνος εκτέλεσης rpc για την πρώτη C++ threads υλοποίηση..... 72
Εικόνα 48: Συνολικός χρόνος εκτέλεσης rpc για την πρώτη C++ threads υλοποίηση 72
Εικόνα 49: Μέσος χρόνος εκτέλεσης rpc για την δεύτερη C++ threads υλοποίηση 73
Εικόνα 50: Συνολικός χρόνος εκτέλεσης rpc για την δεύτερη C++ threads υλοποίηση..... 73
Εικόνα 51: Μέσος χρόνος εκτέλεσης rpc για την πρώτη Go υλοποίηση..... 74
Εικόνα 52: Συνολικός χρόνος εκτέλεσης rpc για την πρώτη Go υλοποίηση 75
Εικόνα 53: Μέσος χρόνος εκτέλεσης rpc για την δεύτερη Go υλοποίηση 76
Εικόνα 54: Συνολικός χρόνος εκτέλεσης rpc για την δεύτερη Go υλοποίηση 76
Εικόνα 55: Μέσος χρόνος εκτέλεσης rpc για την υλοποίηση με time-out..... 77

ΚΑΤΑΛΟΓΟΣ ΠΙΝΑΚΩΝ

Πίνακας 1: Ψευδοκώδικας scheduler στην C++ processes υλοποίηση 38
Πίνακας 2: Ψευδοκώδικας processes στην C++ processes υλοποίηση 38
Πίνακας 3: Ψευδοκώδικας scheduler στην πρώτη C++ threads υλοποίηση 40
Πίνακας 4: Ψευδοκώδικας threads στην πρώτη C++ threads υλοποίηση 41
Πίνακας 5: Ψευδοκώδικας scheduler στην δεύτερη C++ threads υλοποίηση 42
Πίνακας 6: Ψευδοκώδικας threads στην δεύτερη C++ threads υλοποίηση 43
Πίνακας 7: Ψευδοκώδικας scheduler στην πρώτη Go υλοποίηση 44
Πίνακας 8: Ψευδοκώδικας goroutines στην πρώτη Go υλοποίηση 45
Πίνακας 9: Ψευδοκώδικας scheduler στην δεύτερη Go υλοποίηση 46
Πίνακας 10: Ψευδοκώδικας goroutines στην δεύτερη Go υλοποίηση 46
Πίνακας 11: Time-out χρόνοι για κάθε προτεραιότητα 49
Πίνακας 12: Ψευδοκώδικας threads με προσθήκη time-out μηχανισμού 50

ΔΟΜΗ ΤΗΣ ΔΙΠΛΩΜΑΤΙΚΗΣ

Η παρούσα διπλωματική είναι οργανωμένη σε τρία βασικά κεφάλαια:

- Βασικές αρχές client-server επικοινωνίας μέσω RPC: γίνεται μία μικρή εισαγωγή στην επιστήμη των δικτύων υπολογιστών και στη δομή του διαδικτύου σύμφωνα με το μοντέλο TCP/IP. Στη συνέχεια γίνεται μία σύντομη περιγραφή του πρωτοκόλλου HTTP/2 και του gRPC framework, καθώς και κάποιων εργαλείων που χρησιμοποιηθήκαν στο πλαίσιο της εργασίας.
- Ιεράρχηση προτεραιοτήτων στο gRPC: παρουσιάζονται αναλυτικά οι διάφορες υλοποιήσεις που πραγματοποιήθηκαν για την προσθήκη χρονοδρομολογητή σε έναν gRPC client καθως και για την επέκταση του χρονοδρομολογητή με μηχανισμό time-out για την αποφυγή φαινομένων starvation.
- Αποτελέσματα: παρουσιάζονται τα αποτελέσματα της μελέτης και των μετρήσεων επίδοσης του gRPC framework, καθώς και των πειραμάτων που πραγματοποιήθηκαν πάνω στις υλοποιήσεις του πρακτικού μέρους.
- Συμπεράσματα: σχολιάζονται τα αποτελέσματα που προέκυψαν στο προηγούμενο κεφάλαιο και περιγράφονται πιθανές μελλοντικές επεκτάσεις της παρούσας διπλωματικής.

ΒΑΣΙΚΕΣ ΑΡΧΕΣ CLIENT-SERVER ΕΠΙΚΟΙΝΩΝΙΑΣ ΜΕΣΩ RPC

Εισαγωγικές έννοιες

Η χρήση του διαδικτύου, είναι πλέον ένα αναπόσπαστο κομμάτι της καθημερινότητας σχεδόν όλων των ατόμων, τόσο για την ενημέρωσή τους, όσο και για την μεταξύ τους επικοινωνία. Αποτελεί το μοναδικό μέσο συνέχισης της κανονικής λειτουργίας της κοινωνίας σε όλους τους τομείς της, και αποφυγής της αποξένωσης των ανθρώπων σε καταστάσεις ακραίας απομόνωσης, όπως έγινε φανερό κατά την περίοδο της πρόσφατης πανδημίας. Εκτός αυτού, οι συσκευές της σύγχρονης τεχνολογίας ολοένα και περισσότερο ενσωματώνουν λειτουργίες που απαιτούν τη μεταξύ τους σύνδεση και επικοινωνία. Γίνεται, επομένως, κατανοητό ότι τα δίκτυα υπολογιστών είναι μία από τις σημαντικότερες επιστήμες της σημερινής εποχής.

Τα δίκτυα υπολογιστών ανήκουν στην ευρύτερη κατηγορία των τηλεπικοινωνιακών δικτύων, και αντιπροσωπεύουν σύνολα από αυτόνομους διασυνδεδεμένους υπολογιστές. Σκοπός της ύπαρξης των δικτύων είναι η κατανομή των πόρων ενός συστήματος και η ανταλλαγη πληροφοριών μεταξύ των συμμετεχόντων συσκευών. Μία σημαντική μέθοδος ανάπτυξης και λειτουργίας τέτοιων κατανεμημένων συστημάτων είναι το μοντέλο πελάτη – διακομιστή.

Το μοντέλο πελάτη – διακομιστή (client – server) αποτελεί ένα από τα πιο συνηθισμένα μοντέλα αρχιτεκτονικής λογισμικού στην επιστήμη των υπολογιστών. Κατά τη συγκεκριμένη μέθοδο, ο πελάτης θέτει ένα αίτημα και ο διακομιστής επιστρέφει μία απάντηση ή εκτελεί μία σειρά από ενέργειες. Πιο συγκεκριμένα, ο πελάτης και ο διακομιστής είναι τμήματα λογισμικού που εκτελούνται σε διαφορετικές διεργασίες, οι οποίες μπορούν να εκτελούνται σε διαφορετικούς υπολογιστές, συνδεδεμένους όμως μέσω ενός δικτύου.

Μία σύνδεση μεταξύ δύο τέτοιων εφαρμογών επιτυγχάνεται μέσω της χρήσης μιας καθιερωμένης ομάδας διαδικτυακών πρωτοκόλλων, τα οποία είναι οργανωμένα σε επίπεδα, σύμφωνα με το μοντέλο TCP/IP. Στη βάση ενός απλουστευμένου τέτοιου μοντέλου βρίσκεται το πρωτόκολλο συνδέσμου (πχ. Ethernet), ενώ ακριβώς πάνω του λειτουργεί το πρωτόκολλο δικτύου (πχ. IP). Τα δύο ανώτερα επίπεδα είναι εκείνο της μεταφοράς και της εφαρμογής.

Το στρώμα της μεταφοράς είναι αυτό που αφορά την μεταφορά των δεδομένων, καθώς και λειτουργίες όπως είναι ο έλεγχος σφαλμάτων, η κατάτμηση πακέτων και η ρύθμιση ροής. Ένα από τα σημαντικότερα πρωτόκολλα μεταφοράς είναι το Πρωτόκολλο Ελέγχου Μεταφοράς (TCP), κατά το οποίο η μεταφορά δεδομένων γίνεται μέσω μιας εδραιωμένης σύνδεσης. Μία TCP σύνδεση χρειάζεται την ανταλλαγή τριών πακέτων για να εδραιωθεί, διαδικασία γνωστή ως τριμερής χειραψία (3-way handshake), ενώ για να τερματιστεί απαιτεί την ανταλλαγή τεσσάρων πακέτων (4-way handshake).

Όσον αφορά στο επίπεδο της εφαρμογής, αυτό είναι υπεύθυνο για την παράδοση μηνυμάτων στη μορφή που απαιτεί η κάθε εφαρμογή. Το Πρωτόκολλο Μεταφοράς Υπερκειμένου (HyperText Transfer Protocol, HTTP) είναι ένα πρωτόκολλο επιπέδου εφαρμογής, το οποίο χρησιμοποιείται για την επικοινωνία μεταξύ πελάτη και διακομιστή. Το HTTP/1.1 είναι το μοντέλο του πρωτοκόλλου HTTP που έχει καθιερωθεί ευρέως για την επικοινωνία μεταξύ εφαρμογών και χρησιμοποιεί TCP συνδέσεις.

Εισαγωγή

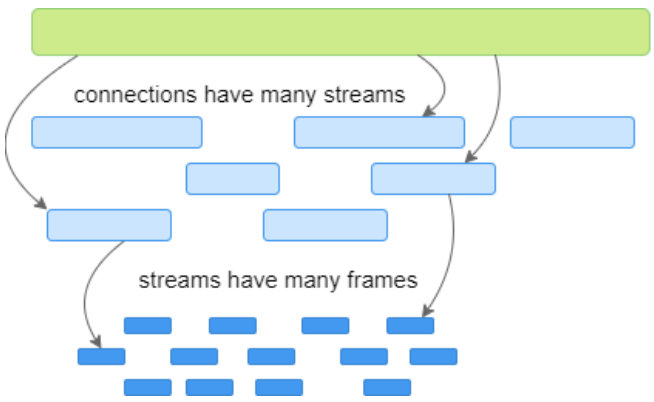
Το HTTP/2 είναι η επόμενη έκδοση του πρωτοκόλλου HTTP/1.1, γεγονός που την καθιστά την πιο πρόσφατη έκδοση του Hypertext Transfer Protocol, δηλαδή του διαδικτυακού πρωτοκόλλου μεταφοράς δεδομένων ανάμεσα σε ένα πελάτη και ένα διακομιστή στον Παγκόσμιο Ιστό.

Το HTTP/1.1 εκδόθηκε τον Ιούνιο του 1999, ενώ το μεγαλύτερο μέρος του web σήμερα συνεχίζει να τρέχει πάνω από την έκδοση αυτή, εδώ και 23 χρόνια. Δεδομένου ότι πολλά έχουν αλλάξει στο διαδίκτυο από τότε, είναι αξιοσημείωτο το γεγονός ότι το HTTP/1.1 έχει παραμείνει αλλά και ακμάσει για τόσο μεγάλο χρονικό διάστημα. Τα τελευταία χρόνια, ωστόσο, λόγω της αύξησης του όγκου των πληροφοριών που κυκλοφορούν στο διαδίκτυο, παρατηρείται επέκταση της κλίμακας των διαδικτυακών εφαρμογών και της κίνησης των δεδομένων μεταξύ αυτών.

Το HTTP/2, που εκδόθηκε το 2015, διατηρεί τα πλεονεκτήματα που έχουν καταστήσει την προηγούμενη έκδοσή του τόσο επιτυχημένη και ευρέως χρησιμοποιούμενη, ενώ παράλληλα προσπαθεί να λύσει το πρόβλημα της κλιμακωσιμότητας που υφίσταται στο web development, εισάγοντας την έννοια των streams.

Κατά τη δημιουργία μίας HTTP σύνδεσης, χρειάζεται πρώτα να εδραιωθεί μία TCP σύνδεση, να ασφαλιστεί με χρήση του πρωτοκόλλου TLS και να γίνει μία αρχική ανταλλαγή μηνυμάτων σχετικά με τις απαιτούμενες ρυθμίσεις, διεργασίες οι οποίες προσθέτουν σημαντική χρονική επιβάρυνση. Η έκδοση HTTP/1.1 προσπάθησε να απλοποιήσει την παραπάνω διαδικασία μετατρέποντας τις συνδέσεις σε μακρόβια, επαναχρησιμοποιούμενα αντικείμενα τα οποία μένουν αδρανή όταν δεν χρησιμοποιούνται, ώστε μηνύματα προς τον ίδιο προορισμό να μπορούν να στέλνονται μέσω της ίδιας υπάρχουσας σύνδεσης, χωρίς να απαιτείται η δημιουργία κάθε φορά νέας με επιπλέον χρονική καθυστέρηση. Το πρόβλημα με αυτή τη προσέγγιση είναι ότι μία σύνδεση μπορεί να χειριστεί ένα request τη φορά, με αποτέλεσμα εάν ένα μήνυμα είναι μεγάλο, τα νέα request που προκύπτουν ενώ αυτό στέλνεται, να αναγκάζομαι να περιμένουν ή πιο συχνά να δημιουργούν νέα σύνδεση, επιλογές που σίγουρα κοστίζουν σε χρόνικη καθυστέρηση.

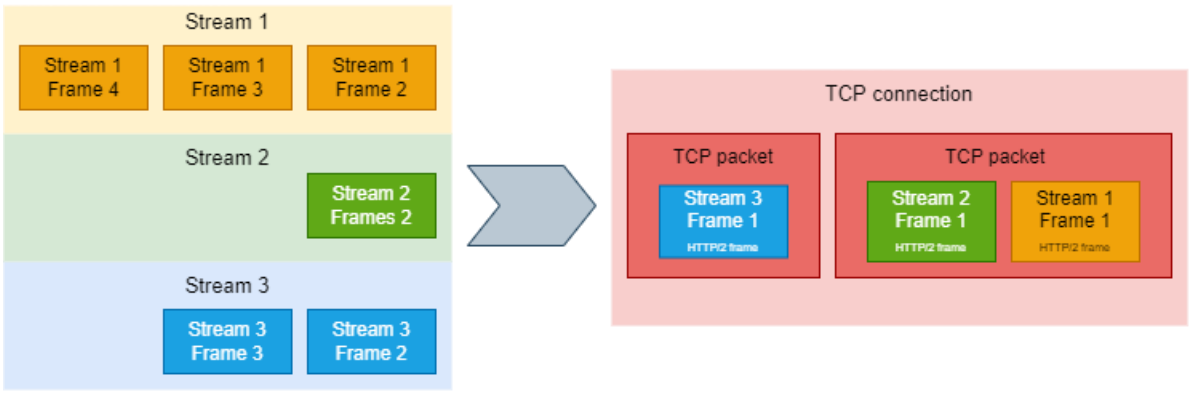
Το HTTP/2 επεκτείνει την ιδέα μιας μόνιμης και μακρόβιας σύνδεσης εισάγοντας ένα επίπεδο πάνω από τις συνδέσεις, το επίπεδο των streams. Stream θεωρείται μία σειρά από σημασιολογικά συνδεδεμένα μηνύματα, τα οποία ονομάζονται πλαίσια (frames). Μία αφαιρετική απεικόνιση της δομής αυτής φαίνεται στην Εικόνα 1.



Εικόνα 1: Αφαιρετική απεικόνιση πρωτοκόλλου HTTP/2

Το βασικό πλεονέκτημα των streams είναι ότι μπορούν να τρέχουν ταυτόχρονα, εμπλέκοντας τα μηνύματά τους στην ίδια TCP σύνδεση. Ταυτόχρονα ωστόσο, δεν σημαίνει παράλληλα, καθώς μόνο ένα TCP πακέτο μπορεί να υπάρχει σε μία σύνδεση κάθε χρονική στιγμή. Επομένως ο αποστολέας μπορεί να αποστείλει τα πλαίσια των διαφόρων streams στη σύνδεση είτε με τρόπο round – robin, είτε δίνοντας προτεραιότητα σε κάποιο stream ή με οποιονδήποτε άλλον αλγόριθμο επιλέξει.

Ένα παράδειγμα φαίνεται στην Εικόνα 2. Έστω ότι υπάρχει ένα σύστημα για παραγγελίες φαγητού σε ένα εστιατόριο. Ως client θεωρείται μία σελίδα στην οποία οι πελάτες μπορούν να δημιουργήσουν την παραγγελία ενώ server είναι το backend της εφαρμογής. Ο client θέλει να στείλει τρία διαφορετικά είδη πληροφοριών στον server: μία νέα παραγγελία που έγινε (stream 1), τον αριθμό των ατόμων που είναι συνδεδεμένοι στη σελίδα μια χρονική στιγμή (stream 2), καθώς και τον αριθμό των ατόμων που βρίσκονται στη σελίδα αλλά δεν είναι συνδεδεμένοι σε κάποιο λογαριασμό (stream 3). Το stream 1 έρχεται πρώτο από το stream 2, ωστόσο αποτελείται από πολλά πλαίσια, λόγω του μεγάλου όγκου μιας παραγγελίας. Το stream 2 που αποτελείται από ένα μοναδικό μήνυμα, δεν χρειάζεται να περιμένει τα frames της παραγγελίας να ολοκληρώσουν την αποστολή τους στη σύνδεση, αλλά μπορεί να στείλει το μήνυμά του ταυτόχρονα με το stream 1.



Εικόνα 2: HTTP/2 πλαίσια σε TCP σύνδεση

Streams

Ένα stream, λοιπόν, είναι μία ανεξάρτητη και αμφίδρομη ακολουθία από πλαίσια που ανταλλάζονται ανάμεσα σε έναν πελάτη και έναν διακομιστή μέσα σε μία HTTP/2 σύνδεση. Μία HTTP/2 σύνδεση μπορεί δηλαδή να περιέχει πολλά ταυτόχρονα streams με κάθε μία από τις δύο πλευρές να μπορούν να εισάγουν πλαίσια από διάφορα streams. Ένα stream μπορεί να χρησιμοποιείται μονόπλευρα ή να μοιράζεται ανάμεσα στον client και τον server. Σε κάθε περίπτωση, το κάθε stream συνοδεύεται από ένα μοναδικό αναγνωριστικό, με τα stream που δημιουργούνται από τον client να έχουν υποχρεωτικά περιττό αριθμό, και αυτά που δημιουργούνται από τον server να έχουν άρτιο αριθμό ως αναγνωριστικό. Η τιμή 0 χρησιμοποιείται από το root stream το οποίο μεταφέρει πλαίσια που αφορούν ολόκληρη τη σύνδεση και όχι κάποιο συγκεκριμένο stream.

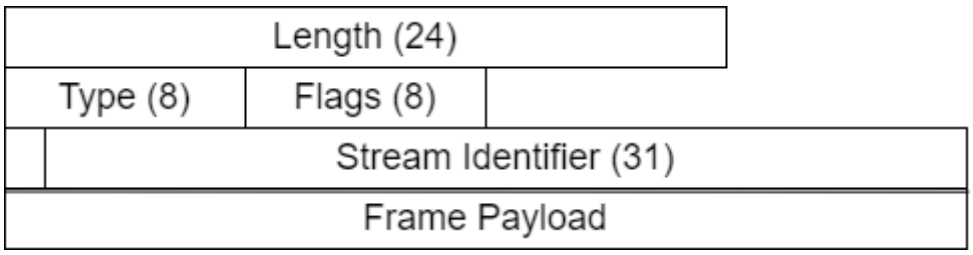
Frames

Όπως αναφέρθηκε προηγουμένως, μονάδα του HTTP/2 πρωτοκόλλου είναι το πλαίσιο (frame). Τα βασικά είδη πλαισίων που αφορούν τα αιτήματα και τις απαντήσεις στο πρωτόκολλο είναι τα HEADERS και τα DATA πλαίσια, ενώ μερικά από τα συμπληρωματικά είδη που εξυπηρετούν επιπλέον ιδιότητες του HTTP/2 είναι τα

SETTINGS, WINDOW_UPDATE και PING πλαίσια. Η δομή ενός HTTP/2 frame φαίνεται στην Εικόνα 3. Κάθε HTTP/2 πλαίσιο ξεκινάει με μία επικεφαλίδα των 72 bits (9 bytes), ακολουθούμενη από το μεταβλητού μεγέθους ωφέλιμο φορτίο. Τα πεδία της επικεφαλίδας είναι τα εξής:

- Length (24 bits): Το μήκος του ωφέλιμου φορτίου το οποίο δεν θα πρέπει να ξεπερνάει ένα μέγιστο μήκος που θέτει ο παραλήπτης.
- Type (8 bits): Το είδος του πλαισίου.
- Flags (8 bits): Boolean σημαίες σχετικές με το είδος του πλαισίου.
- Stream Identifier (31 bits): Το αναγνωριστικό του stream στο οποίο ανήκει το συγκεκριμένο πλαίσιο.

Ανάμεσα στα πεδία Flags και Stream Identifier βρίσκεται ένα πεδίο μήκους 1 bit, του οποίου η σημασία είναι μη καθορισμένη και το οποίο αγνοείται.



Εικόνα 3: Δομή ενός HTTP/2 frame

Είδη frames

Το πιο συνηθισμένο είδος πλαισίου είναι το DATA frame, το οποίο μεταφέρει μεταβλητού μήκους πληροφορίες που σχετίζονται με ένα stream (διάφορο του root stream). Ένα ή και περισσότερα πλαίσια DATA μπορούν να χρησιμοποιηθούν, για παράδειγμα, για τη μετάδοση ωφέλιμου φορτίου ενός HTTP αιτήματος ή απάντησης. Στην επικεφαλίδα ενός τέτοιου πλαισίου, υπάρχει η σημαία END_STREAM, η οποία όταν είναι ίση με 1, σηματοδοτεί ότι το συγκεκριμένο πλαίσιο είναι το τελευταίο που θα στείλει η πηγή για αυτό το stream.

Ένα άλλο είδος πλαισίου, το οποίο πρέπει και αυτό να σχετίζεται με κάποιο stream διάφορο του root, είναι το HEADERS πλαίσιο, το οποίο στέλνεται κατά την αρχικοποίηση του stream. Το συγκεκριμένο είδος πλαισίου μεταφέρει πληροφορίες αναφορικά με χαρακτηριστικά του stream, όπως το εάν έχει κάποια προτεραιότητα, εάν εξαρτάται από κάποιο άλλο stream κ.α.

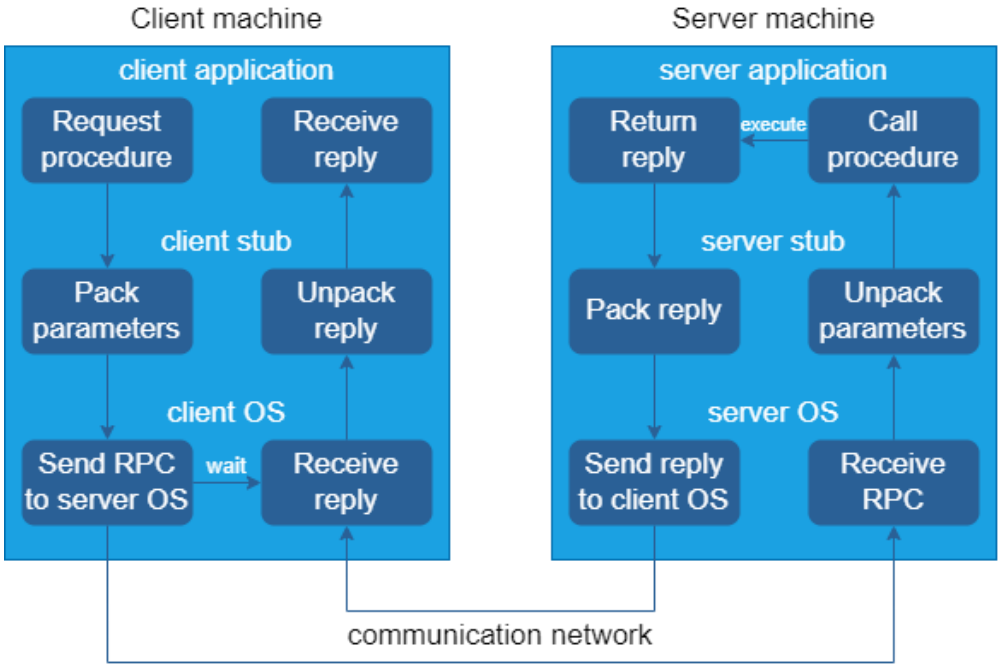
Σε αντίθεση με τα δύο προηγούμενα, το SETTINGS πλαίσιο μεταφέρει πληροφορίες για ολόκληρη τη σύνδεση, και δεν συνδέεται με κάποιο stream. Τα SETTINGS πλαίσια είναι υποχρεωτικό να στέλνονται κατά τη δημιουργία μίας σύνδεσης καθώς περιέχουν χαρακτηριστικά του αποστολέα, ενώ μπορούν να σταλούν ξανά και σε επόμενη χρονική στιγμή, για παράδειγμα ως επιβεβαίωση ότι ο παραλήπτης έλαβε το αρχικό SETTINGS πλαίσιο του αποστολέα. Στο ωφέλιμο φορτίο του πλαισίου μεταξύ άλλων καθορίζονται οι παράμετροι SETTINGS_MAX_CONCURRENT_STREAMS, το οποίο θέτει το μέγιστο αριθμό παράλληλων streams που επιτρέπει ο αποστολέας να δημιουργηθούν από τον παραλήπτη, και SETTINGS_INITIAL_WINDOW_SIZE, που ρυθμίζει το αρχικό μέγεθος παραθύρου (window size) του αποστολέα το οποίο χρησιμοποιείται για τον έλεγο ροής.

Τα WINDOW_UPDATE πλαίσια χρησιμοποιούνται για την υλοποίηση ελέγχου ροής (flow control) τόσο σε επίπεδο ολόκληρης της σύνδεσης, όταν σχετίζονται με το stream 0, όσο και σε επίπεδο stream. Ο έλεγχος ροής επιτυγχάνεται με την ύπαρξη ενός παραθύρου για κάθε αποστολέα σε κάθε stream, που ισούται με τον αριθμό των bytes από data που επιτρέπεται να στείλει ο αποστολέας σε κάθε στιγμή. Με κάθε αποστολή ενός data πλαισίου, ο αποστολέας μειώνει το παράθυρό του, σε επίπεδο σύνδεσης και σε επίπεδο stream, κατά το μήκος του ωφέλιμου φορτίου που έστειλε σε bytes. Ο παραλήπτης, όταν λαμβάνει το data πλαίσιο, απαντάει με WINDOW_UPDATE πλαίσια, ένα σε επίπεδο σύνδεσης και ένα σε επίπεδο stream, ώστε ο αποστολέας να ενημερώσει το παράθυρό του αυξάνοντας το κατά τον αριθμό των bytes που υποδυκνείονται στο ωφέλιμο φορτίο του WINDOW_UPDATE πλαισίου. Με αυτόν τον τρόπο εξασφαλίζεται η ισορροπία μεταξύ του ρυθμού παραγωγής και κατανάλωσης δεδομένων στο κανάλι.

Δύο τελευταία είδη πλαισίων είναι το RST_STREAM, το οποίο στέλνεται για τον άμεσο τερματισμό ενός stream και το PING, που είναι ένας τρόπος να μετρηθεί ο round – trip χρόνοw αλλά και να καθοριστεί εάν η σύνδεση είναι ακόμα ενεργή. Πέραν αυτών, υπάρχουν και άλλα είδη frames τα οποία όμως δεν θα αναλυθούν στο πλαίσιο αυτής της διπλωματικής.

Πρωτόκολλο RPC

Το RPC (Remote Process Call) ή κλήση απομακρυσμένης διαδικασίας, είναι ένα πρωτόκολλο επικοινωνίας που μπορεί να χρησιμοποιηθεί από ένα πρόγραμμα όταν θέλει να ζητήσει μία υπηρεσία από ένα άλλο πρόγραμμα που βρίσκεται σε διαφορετικό μηχάνημα, χωρίς να χρειάζεται να γνωρίζει τις λεπτομέρειες του δικτύου μέσω του οποίου τα δύο μηχανήματα είναι συνδεδεμένα. Το RPC χρησιμοποιεί το μοντέλο του πελάτη – διακομιστή: το αιτούμενο πρόγραμμα έχει το ρόλο του πελάτη, ενώ το πρόγραμμα που παρέχει την υπηρεσία είναι ο διακομιστής. Ο πελάτης καλεί μια διεργασία σαν αυτή να βρίσκεται στο τοπικό του περιβάλλον, δηλαδή σαν μια σύγχρονη λειτουργία που απαιτεί από το το thread που εκτελεί την κλήση να ανασταλλεί μέχρι να επιστραφεί το αποτέλεσμα.



Εικόνα 4: Σχηματική αναπαράσταση μιας κλήσης απομακρυσμένης διαδικασίας

Όταν ένα πρόγραμμα που χρησιμοποιεί το RPC πρωτόκολλο μετατρέπεται σε εκτελέσιμο, ένα stub προστίθεται αυτόματα στον κώδικά του, το οποίο λειτουργεί σαν εκπρόσωπος του απομακρυσμένου κώδικα. Όταν ένας client καλεί ένα rpc πραγματοποιείται η εξής διαδικασία:

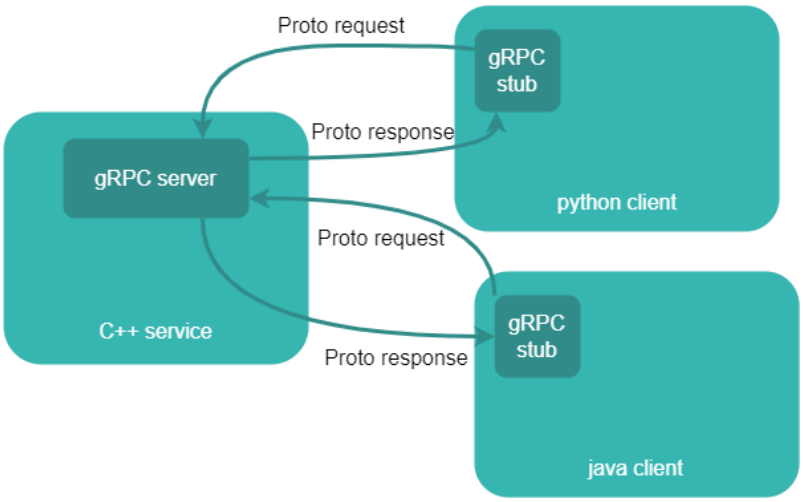
1. Ο πελάτης καλεί το client stub δίνοντάς του τις παραμέτρους της συνάρτησης που θέλει να εκτελέσει.
2. Το stub «πακετάρει» τις παραμέτρους αυτές σε ένα συγκεκριμενής δομής, σύμφωνα με το πρωτόκολλο, μήνυμα και ύστερα εκτελεί μία κλήση συστήματος για να το στείλει.
3. Το λειτουργικό σύστημα του πελάτη στέλνει το μήνυμα μέσω του δικτύου από το μηχάνημα του πελάτη στο απομακρυσμένο μηχάνημα του διακομιστή.
4. Το λειτουργικό σύστημα του διακομιστή προωθεί το εισερχόμενο μήνυμα στο server stub.
5. Το server stub εξάγει από το μήνυμα που λαμβάνει τις παραμέτρους και με αυτές καλεί την κατάλληλη συνάρτηση.

6. Όταν η συνάρτηση ετελεστεί στον server, το αποτέλεσμα επιστρέφεται στο server stub, που το μετατρέπει στο κατάλληλο μήνυμα, και το στέλνει πίσω στο μηχάνημα του πελάτη.
7. Το client stub λαμβάνει το μήνυμα, εξάγει από αυτό το ζητούμενο αποτέλεσμα, και η εκτέλεση του προγράμματος του πελάτη συνεχίζεται.

Το RPC πρωτόκολλο είναι ιδιαίτερα δημοφιλές στα σύγχρονα κατανεμημένα συστήματα και σε αρχιτεκτονικές μικροϋπηρεσιών καθώς παρουσιάζει πλήθος πλεονεκτημάτων. Οι εφαρμογές που επικοινωνούν μεταξύ τους μέσω του πρωτοκόλλου δεν χρειάζεται να γνωρίζουν τα στοιχεία του δικτύου που συνδέει τα μηχανήματά τους, ούτε λεπτομέρειες σχετικά με την υλοποίηση των συναρτήσεων και την εσωτερική δομή της κάθε μίας. Αντιθέτως, το κάθε πρόγραμμα καλεί μια συνάρτηση σαν αυτή να είναι μία τοπική διαδικασία του συστήματός του, και το RPC αναλαμβάνει να διεκπεραιώσει την απαιτούμενη επικοινωνία. Επιπροσθέτως, εξασφαλίζεται αφαιρετικότητα και ιδιαίτερη ευκολία ανάπτυξης ενός συστήματος, καθώς το RPC κρύβει από τον προγραμματιστή τις διαδικασίες ανταλλαγής μηνυμάτων τόσο τις εσωτερικές (από το πρόγραμμα στο stub και αντιστρόφως) όσο και τις εξωτερικές (μέσω του δικτύου).

gRPC framework

Το gRPC (google RPC) είναι ένα open-source framework για την υποστήριξη του RPC πρωτοκόλλου, που αναπτύχθηκε από την Google το 2015 και μπορεί να τρέξει σε οποιοδήποτε περιβάλλον. Βασίζεται στα Google Protocol Buffers, που είναι μηχανισμοί με τους οποίους μπορούν να οριστούν πρότυπα templates για τη δομή των συναρτήσεων και των μηνυμάτων που θα ανταλλάζονται μέσω του RPC. Το gRPC είναι ανεξάρτητο της γλώσσας ή της πλατφόρμας στην οποία είναι ανεπτυγμένη μία εφαρμογή, γεγονός που το καθιστά ιδιαίτερα εύχρηστο.



Εικόνα 5: Σχηματική αναπαράσταση gRPC για έναν server και δύο clients

Στο κομμάτι του δικτύου, το gRPC λειτουργεί πάνω από το HTTP/2 πρωτόκολλο, στο οποίο σε ένα connection μπορούν να συνυπάρχουν πολλά streams τα οποία αντιστοιχούν σε services. Εάν δηλαδή κατά την επικοινωνία μεταξύ δύο εφαρμογών υπάρχουν πολλές διαφορετικές υπηρεσίες, η κάθε μία από αυτές θα στέλνει τα πακέτα της στο δικό της stream, ενώ τα πακέτα των διαφορετικών streams θα πολυπλέκονται πάνω από την ίδια HTTP/2 σύνδεση. Με αυτόν τον τρόπο και αναφορικά με το gRPC, κάθε rpc αντιστοιχεί σε ένα HTTP/2 stream και κάθε rpc μήνυμα αντιστοιχεί σε ένα HTTP/2 πακέτο.

Είδη rpc μεθόδων στο gRPC

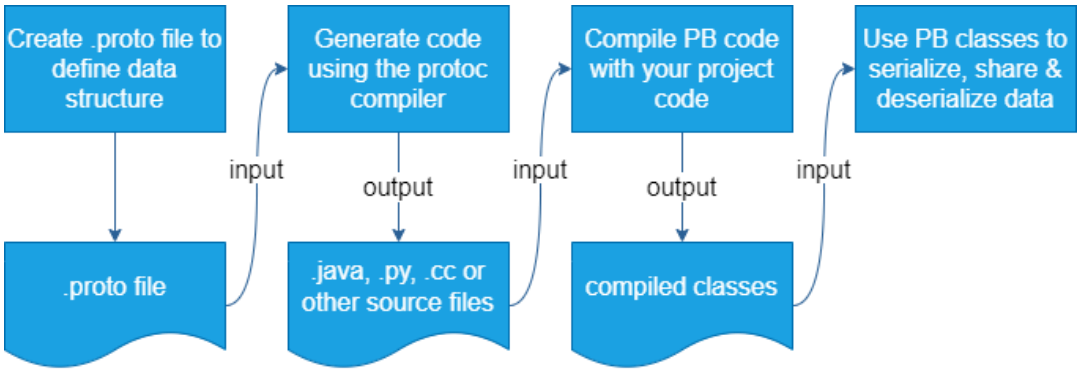
Το gRPC framework ορίζει την ύπαρξη τεσσάρων ειδών κλήσεων απομακρυσμένης διαδικασίας:

- Ένα unary RPC κατά το οποίο ο πελάτης στέλνει ένα αίτημα στο διακομιστή, με τη χρήση του stub, και στη συνέχεια αναμένει την απάντηση, όπως συμβαίνει σε μια συνηθισμένη κλήση συνάρτησης.
- Ένα server-side streaming RPC κατά το οποίο ο πελάτης στέλνει ένα αίτημα στο διακομιστή και του επιστρέφεται μία ροή από την οποία μπορεί να διαβάσει ακολουθία μηνυμάτων – απαντήσεων. Ο πελάτης διαβάζει από τη ροή μέχρι να μην υπάρχουν άλλα μηνύματα.
- Ένα client-side streaming RPC, κατά το οποίο ο πελάτης γράφει μία ακολουθία από μηνύματα και τη στέλνει στον διακομιστή. Μόλις ολοκληρώσει την αποστολή των μηνυμάτων, περιμένει ο διακομιστής να τα διαβάσει και να απαντήσει.
- Ένα bidirectional streaming RPC κατά το οποίο και οι δύο πλευρές στέλνουν μία ακολουθία από μηνύματα χρησιμοποιώντας δύο ροές. Οι δύο ροές είναι ανεξάρτητες μεταξύ τους, επομένως ο πελάτης και ο διακομιστής μπορούν να γράφουν και να διαβάζουν μηνύματα με όποια σειρά θέλουν. Για παράδειγμα, ο διακομιστής θα μπορούσε να διαβάσει όλα τα μηνύματα του client πρωτού αρχίσει να στέλνει τις απαντήσεις του ή να εκτελέσει οποιονδήποτε άλλο συνδυασμό ανάγνωσης και εγγραφής μηνυμάτων. Η σειρά των μηνυμάτων σε κάθε ροή διατηρείται.

Protocol Buffers

Τα Protocol Buffers παρέχουν έναν ανεξάρτητο γλώσσας, ανεξάρτητο πλατφόρμας και επεκτάσιμο μηχανισμό για τη σειριοποίηση δεδομένων. Είναι ένας συνδυασμός της γλώσσας ορισμού των προτύπων που πρέπει να ακολουθούν τα δεδομένα, του κώδικα που παράγεται από τον proto μεταγλωττιστή και διαφόρων runtime βιβλιοθηκών. Τα protocol buffers είναι backward αλλά και forward-compatible, που σημαίνει πως ό,τι γράφεται σε παρούσα έκδοση θα είναι συμβατό με protocol buffers γραμμένα σε προηγούμενες αλλά και επόμενες εκδόσεις, αντίστοιχα. Επίσης, μπορούν να επεκταθούν με την προσθήκη νέας πληροφορίας, χωρίς αυτό να ακυρώνει τα υπάρχοντα δεδομένα ή να απαιτεί ανανέωση κώδικα.

Τα protocol buffers περιγράφουν τη δομή μηνυμάτων και υπηρεσιών σε ειδικά αρχεία .proto. Ο proto μεταλγωττιστής καλείται κατά το build time στα .proto αρχεία, για να παράξει κώδικα σε διάφορες γλώσσες προγραμματισμού. Ο παραγόμενος κώδικας παρέχει κλάσεις για τα δηλωμένα στα .proto files μηνύματα και υπηρεσίες, καθώς και ειδικές συναρτήσεις πρόσβασης σε κάθε πεδίο και κάθε συνάρτηση αυτών, τα οποία μπορούν να χρησιμοποιηθούν από τη βασική εφαρμογή. Με αυτόν τον τρόπο παρέχεται ένας εύκολος τρόπος επικοινωνίας ανάμεσα σε εφαρμογές γραμμένες σε διαφορετική γλώσσα, αφού αυτές μέσω του αυτόματα παραγόμενου κώδικα στη γλώσσα τους, μπορούν να έχουν πρόσβαση στα κοινής μορφής δεδομένα που ανταλλάζουν. Μία σχηματική αναπαράσταση του workflow των protocol buffers φαίνεται στην Εικόνα 5.



Εικόνα 6: Ροή εργασιών των Protocol Buffers

Στην Εικόνα 7 φαίνεται ένα παράδειγμα ενός .proto file, όπως υπάρχει στην επίσημη σελίδα του gRPC. Αρχικά ορίζεται η υπηρεσία Greeter. Μία υπηρεσία ουσιαστικά αποτελεί την ομαδοποίηση σημασιολογικά συνδεδεμένων rpc. Μέσα στην υπηρεσία Greeter ορίζεται η rpc μέθοδος SayHello, η οποία δέχεται ως παράμετρο ένα HelloRequest μήνυμα και επιστρέφει ένα HelloReply μήνυμα. Ο ορισμός των δύο αυτών μηνυμάτων φαίνεται παρακάτω στο αρχείο. Το HelloRequest μήνυμα περιέχει ένα πεδίο name, τύπου string, ενώ το HelloReply μήνυμα περιέχει ένα πεδίο message, τύπου string. Ο τύπος ενός πεδίου μπορεί να είναι ένα μήνυμα που έχει οριστεί παραπάνω. Το κάθε πεδίο ενός message συνοδεύεται από έναν αριθμό, το οποίο είναι σαν μοναδικό αναγνωριστικό και δεν μπορεί να χρησιμοποιηθεί από άλλο πεδίο μέσα στο ίδιο μήνυμα. Κατά τον μεταγλωττισμό του συγκεκριμένου αρχείου, παράγονται συναρτήσεις όπως constructors και accessors (πχ. GetName(), SetName(), GetMessage(), SetMessage()) στην επιθυμητή γλώσσα, και οι οποίες μπορούν να χρησιμοποιηθούν από τις εφαρμογές για τη δημιουργία μηνυμάτων, την πρόσβαση σε πεδία τους, και την κλήση rpc.

```
// The greeting service definition.
service Greeter {
  // Sends a greeting
  rpc SayHello (HelloRequest) returns (HelloReply) {}
}

// The request message containing the user's name.
message HelloRequest {
  string name = 1;
}

// The response message containing the greetings
message HelloReply {
  string message = 1;
}
```

Εικόνα 7: Παράδειγμα .proto file

Πλεονεκτήματα gRPC

Το gRPC χαρακτηρίζεται από πλήθος θετικών στοιχείων, γεγονός που το καθιστά τόσο δημοφιλές στα σύγχρονα συστήματα. Καταρχάς, παρουσιάζει τα οφέλη που ήδη προσφέρει το πρωτόκολλο RPC. Ένα πρόγραμμα μπορεί να καλέσει εύκολα κάποια διαδικασία που βρίσκεται σε ένα απομακρυσμένο σύστημα, με τρόπο παρόμοιο με το να καλούσε κάποια τοπική διαδικασία και χωρίς να το απασχολούν λεπτομέρειες του δικτύου με το οποίο τα δύο συστήματα συνδέονται. Επιτυγχάνεται έτσι αφαιρετικότητα και ευκολία στην υλοποίηση της επικοινωνίας και αλληλεπίδρασης μεταξύ εφαρμογών.

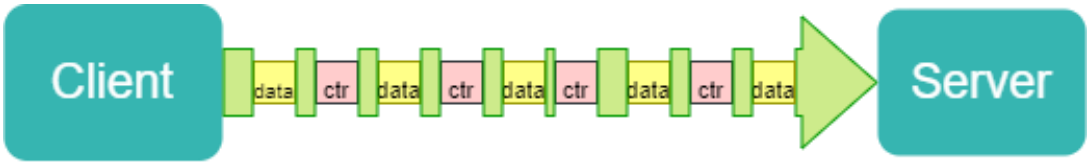
Το gRPC, ακόμα, έχει τη δυνατότητα να διευκολύνει σε μεγάλο βαθμό το κομμάτι της υλοποίησης και συντήρησης ενός συστήματος. Στο gRPC χρειάζεται απλά να δηλωθούν οι υπηρεσίες και τα μηνύματα που αυτές δέχονται και στέλνουν σε μορφή protocol buffers, και στη συνέχεια δημιουργείται αυτόματα κώδικας στην επιθυμητή γλώσσα, ο οποίος μπορεί να χρησιμοποιηθεί από την βασική εφαρμογή. Είναι επομένως ένα cross-platform και cross-language framework ώστε να μην απαιτείται από τον προγραμματιστή να πρέπει να εξοικειωθεί σε μια νέα γλώσσα, ούτε από μία εφαρμογή γραμμένη σε συγκεκριμένη τεχνολογία να ξαναγραφτεί από την αρχή. Τα protocol buffers είναι επίσης backward αλλά και forward-compatible, που σημαίνει πως νέος κώδικας μπορεί να προστεθεί ή να αφαιρεθεί χωρίς αυτό να προκαλεί πρόβλημα σε εφαρμογές που αναπτύχθηκαν γύρω από προηγούμενες εκδοχές .proto αρχείων.

Το gRPC παρουσιάζει σημαντικά πλεονεκτήματα και σε επίπεδο απόδοσης. Συγκεκριμένα, λόγω της χρήσης protocol buffers, τα ανταλλασόμενα μηνύματα σειριοποιούνται και συμπιέζονται γρήγορα και αποτελεσματικά για να μεταδοθούν. Επίσης λόγω της χρήσης του HTTP/2 πρωτοκόλλου εξασφαλίζεται πολυπλεξία των streams ανάμεσα σε client και server πάνω από την ίδια TCP σύνδεση. Με αυτόν τον τρόπο δεν απαιτείται να δημιουργηθεί, να εδραιωθεί και τέλος να τερματιστεί μία νέα σύνδεση για κάθε rpc που πρέπει να σταλεί και επομένως πετυχαίνεται μεγάλη εξοικονόμηση χρόνου και πόρων.

Το πρόβλημα των προτεραιοτήτων στο gRPC

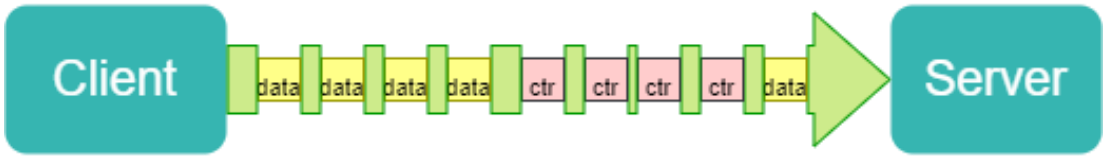
Το gRPC framework, παρά τα πολύ σημαντικά οφέλη που παρέχει, παρουσιάζει κάποιες αδυναμίες που αφορούν την ιεράρχηση προτεραιοτήτων των κλήσεων απομακρυσμένης διαδικασίας. Παρόλο που το HTTP/2 πρωτόκολλο υποστηρίζει ανάθεση προτεραιοτήτων στα streams, στο gRPC όλα τα streams που εξυπηρετούν την εκτέλεση rpc είναι ισότιμα, και ως αποτέλεσμα δεν υπάρχει τρόπος διαφοροποίησης των rpc με βάση την προτεραιότητά τους.

Η ιεράρχηση προτεραιοτήτων, ωστόσο, θα ήταν επιθυμητή σε ορισμένες περιπτώσεις. Για παράδειγμα, έστω ένα σύστημα αποτελούμενο από έναν πελάτη και έναν διακομιστή. Οι δύο πλευρές επικοινωνούν μέσω δύο rpc: το ένα αφορά τα δεδομένα ανταλλαγής μεταξύ client και server, και το άλλο αφορά ρυθμίσεις σχετικά με την επεξεργασία ή τον τρόπο αποθήκευσης αυτών των δεδομένων στον server. Σύμφωνα με το gRPC, αν τα δύο αυτά rpc καλούνται ταυτόχρονα, τα αντίστοιχα πακέτα τους στέλνονται με τρόπο round-robin στο κανάλι, όπως φαίνεται στην Εικόνα 12.



Εικόνα 8: Χειρισμός control και data από το gRPC

Αυτή η πολυπλεξία των δύο ειδών πακέτων ωστόσο, πιθανώς να μην είναι επιθυμητή, καθώς τα control πακέτα θα μπορούσαν να μεταφέρουν σημαντικές πληροφορίες σχετικά με τον χειρισμό των πακέτων δεδομένων που στέλνονται ταυτόχρονα με αυτά. Εάν, για παράδειγμα τα control πακέτα ενημερώνουν τον server ότι πρέπει να αλλάξει το σημείο που αποθηκεύει τα δεδομένα του, τα δεδομένα που μεταφέρονται στα data πακέτα που διανύουν το κανάλι ανάμεσα στα control πακέτα θα αποθηκευτούν στο προηγούμενο σημείο, αφού ο server δεν έχει ακόμα ενημερωθεί για την αλλαγή αυτή. Θα ήταν κατάλληλο λοιπόν, όταν ο πελάτης επιθυμεί να στείλει control πακέτα, να διακόπτει την αποστολή data πακέτων, μέχρι να ολοκληρωθεί το rpc που αφορά τον έλεγχο, όπως φαίνεται σχηματικά στην Εικόνα 13. Σε αυτή την περίπτωση το rpc του ελέγχου θα είχε μεγαλύτερη προτεραιότητα από το rpc των δεδομένων.



Εικόνα 9: Επιθυμητός χειρισμός control και data

Από μετρήσεις που έγιναν στο πλαίσιο της παρούσας διπλωματικής, παρατηρήθηκε ότι η πολυπλεξία rpc κλήσεων θεωρητικά διαφορετικών προτεραιοτήτων, προκαλεί καθυστέρηση, σε σχέση με την ξεχωριστή εκτέλεσή τους, γεγονός που δεν είναι επιθυμητό, εφόσον αιτήματα υψηλότερης προτεραιότητας δεν θα έπρεπε να καθυστερούν από την εκτέλεση λιγότερο σημαντικών αιτημάτων.

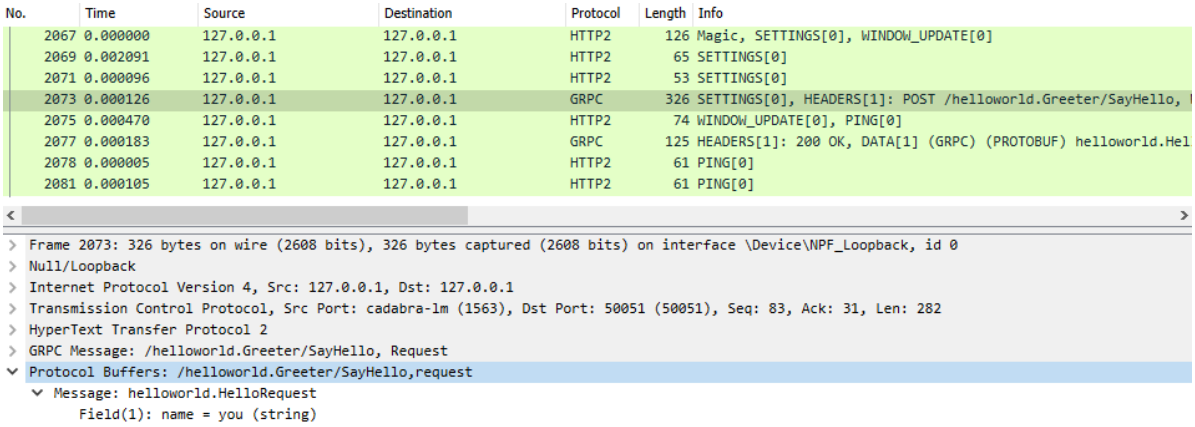
Wireshark

Το Wireshark είναι το πιο δημοφιλές, ανοιχτού κώδικα, λογισμικό ανάλυσης διαδικτυακών πρωτοκόλλων. Είναι ένα πρόγραμμα ανεξάρτητο της πλατφόρμας του μηχανήματος στο οποίο τρέχει, που χρησιμοποιεί το pcap API για την παρακολούθηση της κίνησης ενός δικτύου, ενώ παράλληλα παρέχει το κατάλληλο γραφικό front-end για την απεικόνισή των πακέτων. Το Wireshark δεν προσθέτει κίνηση, αλλά συλλαμβάνει τα πακέτα, χωρίς να τα τροποποιεί, και επιτρέπει στο χρήστη να εξετάσει το περιεχόμενό τους, σε ένα αρκετά user-friendly περιβάλλον.

Το Wireshark είναι ένα κατάλληλο εργαλείο για την ανάλυση των gRPC μηνυμάτων που αταλλάσσονται σε ένα δίκτυο και την καλύτερη κατανόηση της μορφής τους. Παρέχει ειδικά, εξειδικευμένα σε πρωτόκολλα components, για την απεικόνιση των gRPC πακέτων, συγκεκριμένα τα gRPC dissector και Protocol Buffers dissector. Τα components αυτά παρέχουν λειτουργίες όπως:

- Υποστηρίζουν την αποκωδικοποίηση gRPC μηνυμάτων που έχουν σειριοποιηθεί σε protocol buffer μορφή.
- Υποστηρίζουν την ανάλυση gRPC μηνυμάτων που προέρχονται από unary, server-streaming, client-streaming και bidirectional streaming RPC κλήσεις.
- Παρέχουν ενισχυμένη ανάλυση σειριοποιημένων protocol buffer δεδομένων καθώς επιτρέπουν την φόρτωση .proto αρχείων.

Ένα παράδειγμα ανάλυσης gRPC μηνυμάτων φαίνεται στις παρακάτω εικόνες. Καταρχάς στις δύο εικόνες φαίνεται η ανταλλαγή πακέτων ανάμεσα στις διευθύνσεις 127.0.0.1 (Source) και 127.0.0.1 (Destination), γεγονός που υποδεικνύει ότι ο πελάτης και ο διακομιστής βρίσκονται και οι δύο localhost. Υπάρχουν επίσης στήλες στις οποίες φαίνεται ο χρόνος στον οποίο εντοπίστηκε το κάθε πακέτο σε σχέση με τον εντοπισμό του πρώτου πακέτου (Time), ο αύξων αριθμός του πακέτου (No.), το πρωτόκολλο στο οποίο ανήκει το πακέτο (Protocol), το μήκος του πακέτου (Length) και μερικές βασικές πληροφορίες για αυτό (Info).

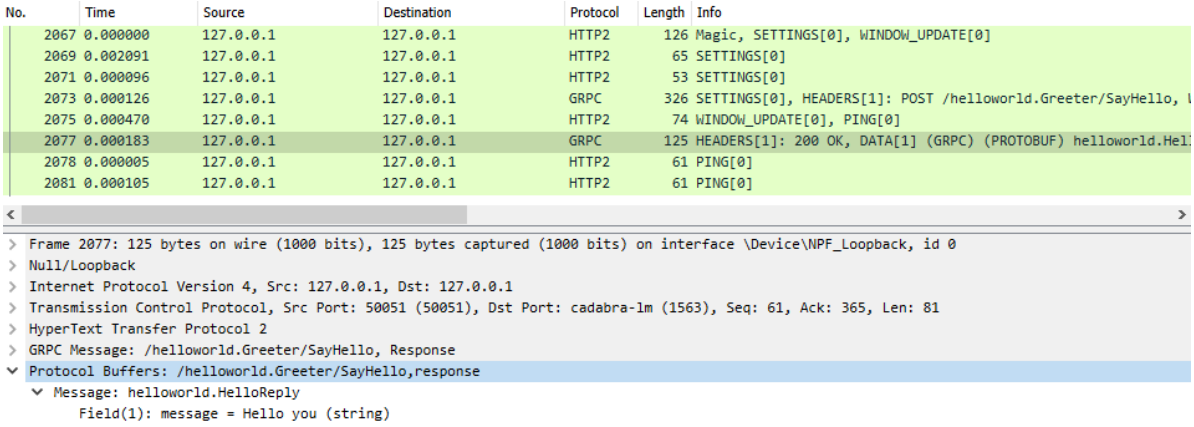


Εικόνα 10: gRPC πακέτο αιτήματος

Παρατηρείται η ύπαρξη δύο stream: του root stream με τον αριθμό 0, και του stream με τον αριθμό 1. Το περιττό αναγνωριστικό του συγκεκριμένου stream υποδεικνύει ότι δημιουργήθηκε από τον πελάτη. Ο αριθμός του stream φαίνεται ανάμεσα στις αγκύλες μετά

το είδος του κάθε HTTP/2 πλαισίου, στην στήλη Info. Για παράδειγμα υπάρχουν SETTINGS, WINDOW_UPDATE, PING πλαίσια, τα οποία ανήκουν στο stream 0 ([0]), καθώς και HEADERS, DATA πλαίσια τα οποία ανήκουν στο stream 1 ([1]). Να σημειωθεί ότι τα πακέτα που περιέχουν πλαίσια μόνο του stream 0 εμφανίζονται να είναι του πρωτοκόλλου HTTP/2, ενώ εκείνα που περιέχουν και πλαίσια του stream 1 εμφανίζονται να είναι του πρωτοκόλλου GRPC (στήλη Protocol).

Στην Εικόνα 8, φαίνεται η ανάλυση ενός πακέτου που περιέχει ένα gRPC αίτημα, της rpc κλήσης SayHello. Το μήνυμα αυτό είναι ένα HelloRequest και περιέχει ένα πεδίο, με αναγνωριστικό 1, που ονομάζεται name, έχει την τιμή you και τύπο string. Στην Εικόνα 9, φαίνεται αντίστοιχα η ανάλυση της απάντησης του διακομιστή. Το gRPC μήνυμα της απάντησης, είναι ένα HelloReply μήνυμα και περιέχει ένα πεδίο, με αναγνωριστικό 1, το οποίο ονομάζεται message και έχει τιμή Hello you, και τύπο string. Κατά το συγκεκριμένο rpc SayHello, ο client στέλνει ένα HelloRequest με το όνομά του (στη συγκεκριμένη περίπτωση το όνομά του είναι you), και ο server απαντάει με ένα HelloReply με το μήνυμα Hello, {client's name} (στη συγκεκριμένη περίπτωση Hello, you).



Εικόνα 11: gRPC πακέτο απάντησης

Ghz

Το ghz είναι ένα command line εργαλείο για την πραγματοποίηση benchmarking ελέγχων των gRPC υπηρεσιών, υλοποιημένο ως βιβλιοθήκη της γλώσσας Go. Χρησιμοποιείται για την εκτέλεση δοκιμών και διαδικασιών αποσφαλμάτωσης συστημάτων που έχουν αναπτυχθεί με τη χρήση του gRPC framework σε τοπικό περιβάλλον.

Η βασική χρήση της CLI εντολής του ghz είναι:

```
ghz [<flags>] [<host>]
```

όπου `[<host>]` είναι η διεύθυνση και η θύρα του διακομιστή που υλοποιεί τις rpc συναρτήσεις. Μερικές επιλογές `[<flags>]` που μπορούν να χρησιμοποιηθούν είναι οι εξής:

- `--config=` Το path προς ένα JSON αρχείο που περιέχει ρυθμίσεις και παραμέτρους σχετικά με την εκτέλεση της εντολής.
- `--proto=` Το path προς το .proto αρχείο στο οποίο περιγράφονται οι διάφορες υπηρεσίες, μέθοδοι και μηνύματα που χρησιμοποιούνται.
- `--call=` Το όνομα της rpc μεθόδου που θα κληθεί κατά την εκτέλεση της εντολής, σε μορφή 'package.Service.Method'.

- `--insecure` Η σημαία αυτή υποδεικνύει τη χρήση plaintext και μη ασφαλούς σύνδεσης με τον διακομιστή (όχι TLS).
- `--concurrency=` Ο αριθμός των νημάτων που θα εκτελέσουν τις rpc κλήσεις ταυτόχρονα. Σε περίπτωση που δεν καθοριστεί ο αριθμός αυτός, η default τιμή είναι 50.
- `--total=` Ο αριθμός των rpc κλήσεων που θα εκτελεστούν συνολικά. Σε περίπτωση που δεν καθοριστεί ο αριθμός αυτός, η default τιμή είναι 200.
- `--data=` Τα δεδομένα που θα σταλούν ως payload για κάθε rpc κλήση σε μορφή JSON.
- `--output=` Το path του αρχείου στο οποίο θα γραφεί το αποτέλεσμα της εντολής. Εάν δεν καθοριστεί χρησιμοποιείται το stdout.
- `--format=` Η μορφή του αποτελέσματος της εντολής. Μπορεί να είναι csv, json, html κ.α.
- `--cpus=` Ο αριθμός των cpu cores που θα χρησιμοποιήσει η εντολή για την αποστολή των rpc.

Ένα παράδειγμα της εντολής ghz φαίνεται στην Εικόνα 10.

```
~ ./ghz --insecure --proto ./helloworld.proto --call helloworld.Greeter.SayHello --data
'{"name": "you"}' --total 100 --concurrency 2 --cpus 2 --format "html" 127.0.0.1:50051
```

Εικόνα 12: Παράδειγμα χρήσης του εργαλείου ghz

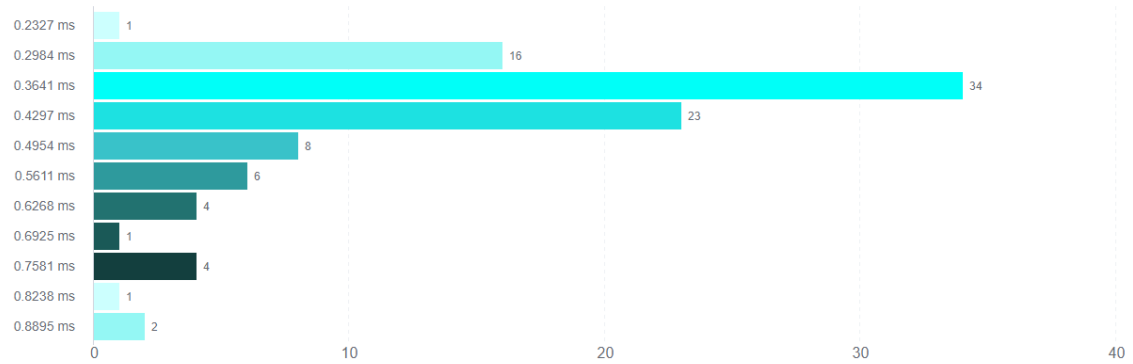
Κατά την εκτέλεση της παραπάνω εντολής, δημιουργείται ένας προσωρινός Go client που θα επικοινωνήσει με τον server που ακούει στη διεύθυνση 127.0.0.1 και θύρα 50051. Ο client θα δημιουργήσει 2 νήματα (concurrency) κάθε ένα από τα οποία θα εκτελέσει 50 rpc, συνολικά δηλαδή θα εκτελεστούν 100 κλήσεις (total). Το .proto αρχείο που θα χρησιμοποιηθεί είναι το ./helloworld.proto (proto), ενώ οι rpc κλήσεις που θα γίνουν θα είναι της μεθόδου SayHello (call) με HelloRequest μηνύματα με τιμή you στο πεδίο name (data). Για την εκτέλεση του πειράματος θα χρησιμοποιηθούν 2 cpu cores του τοπικού μηχανήματος (cpus), και μη ασφαλής σύνδεση (insecure). Τα αποτελέσματα του πειράματος θα επιστραφούν σε μορφή html (format).

Στην Εικόνα 11 φαίνονται τα αποτελέσματα που προκύπτουν από το ghz για την εκτέλεση της προηγούμενης εντολής. Στις πληροφορίες των αποτελεσμάτων, σημειώνεται ότι εκτελέστηκαν 100 rpc κλήσεις σε 27.10 milliseconds, με μέση διάρκεια 0.40 milliseconds η κάθε μία, οι οποίες εκτελέστηκαν όλες σωστά (OK status 100%). Φαίνεται επίσης ένα ιστόγραμμα του χρόνου εκτέλεσης των rpc, καθώς και ένα διάγραμμα για το latency distribution.

Summary

Count	100
Total	27.10 ms
Slowest	0.89 ms
Fastest	0.23 ms
Average	0.40 ms
Requests / sec	3689.98

Histogram



Latency distribution

10 %	25 %	50 %	75 %	90 %	95 %	99 %
0.28 ms	0.31 ms	0.36 ms	0.44 ms	0.59 ms	0.70 ms	0.83 ms

Status distribution

Status	Count	% of Total
OK	100	100.00 %

Εικόνα 13: Αποτελέσματα εκτέλεσης πειράματος σε μορφή html

ΙΕΡΑΡΧΗΣΗ ΠΡΟΤΕΡΑΙΟΤΗΤΩΝ ΣΤΟ gRPC

Υλοποίηση χρονοδρομολογητή

Εισαγωγή

Όπως περιγράφηκε προηγουμένως, το gRPC framework, παρά τα οφέλη που παρουσιάζει, δεν παρέχει κάποιο τρόπο διαφοροποίησης των rpc κλήσεων με βάση την προτεραιότητα που θέλει να τους αποδώσει ο πελάτης.

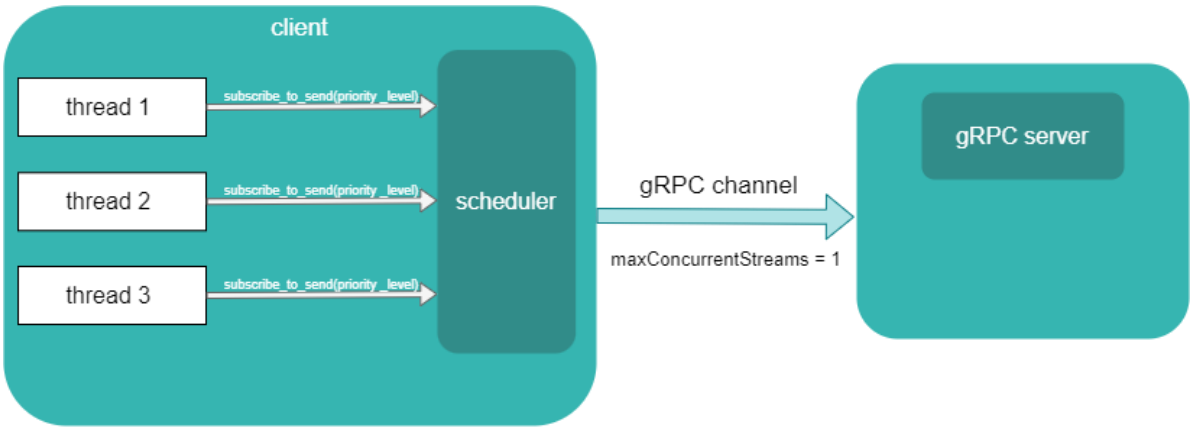
Μία πρόταση επίλυσης του συγκεκριμένου προβλήματος, είναι η ανάπτυξη μιας αρχιτεκτονικής στην οποία τα rpc requests του πελάτη επαυξάνονται με ένα πεδίο που σηματοδοτεί την προτεραιότητά τους. Στη συνέχεια, ένας scheduler ακούει τα rpcs και με βάση την προτεραιότητά τους, πραγματοποιεί δρομολόγηση αυτών πανώ στο gRPC κανάλι ώστε όσο μεγαλύτερη είναι η προτεραιότητα μιας κλήσης τόσο πιο γρήγορα να ολοκληρώνεται.

Αρχιτεκτονική υλοποίησης

Στην υλοποίηση που αναπτύχθηκε στο πλαίσιο της παρούσας διπλωματικής, η οποία αποτελεί μία ενδεικτική προσέγγιση της παραπάνω πρότασης, προστέθηκε ένα κομμάτι λογισμικού μέσα στην εφαρμογή του πελάτη το οποίο αναλαμβάνει το ρόλο του χρονοδρομολογητή: δέχεται από τον πελάτη αιτήματα αποστολής rpc με συγκεκριμένη προτεραιότητα, και με βάση αυτήν δρομολογεί τα rpc με τρόπο τέτοιον ώστε οι μεγαλύτερες προτεραιότητες να εξυπηρετούνται και να ολοκληρώνονται πρώτες.

Κατά τη συγκεκριμένη υλοποίηση το σύστημα που αναπτύχθηκε αποτελείται από έναν πελάτη και έναν διακομιστή. Θεωρήθηκε ότι η κάθε κλήση απομακρυσμένης διαδικασίας που πραγματοποιείται συνοδεύεται από έναν θετικό αριθμό που συμβολίζει την προτεραιότητα αυτής: όσο πιο μεγάλος είναι ο αριθμός αυτός, τόσο μεγαλύτερη είναι και η προτεραιότητα, δηλαδή τόσο πιο σημαντικό είναι να ολοκληρωθεί γρήγορα η κλήση. Τα rpc που πρέπει να γίνουν μαζί με τις συνοδευτικές προτεραιότητες μπορούν είτε να δημιουργούνται από τον πελάτη, ως αποτέλεσμα εσωτερικών διεργασιών ή εξωτερικών triggers, είτε να δίνονται στον πελάτη από κάποιο εξωτερικό σύστημα.

Θεωρήθηκε, επίσης, ότι το κάθε rpc εκτελείται από διαφορετική οντότητα, για παράδειγμα μία διεργασία ή ένα νήμα, ώστε να υπάρχει δυνατότητα ταυτόχρονης εκτέλεσης rpc. Η κάθε μία τέτοια οντότητα στέλνει η ίδια το rpc μέσω του καναλιού. Ο χρονοδρομολογητής εκτελείται και εκείνος από μία τέτοια οντότητα και διατηρεί μία δομή στην οποία αποθηκεύει τα rpc που πρέπει να σταλούν και με βάση την προτεραιότητά αυτών είτε δίνει άδεια είτε απαγορεύει στις υπόλοιπες οντότητες να στείλουν τα rpc τους. Στόχος της χρονοδρομολόγησης που εκτελεί ο scheduler είναι τα rpc με υψηλότερο priority να εξυπηρετούνται πρώτα, δηλαδή να δέχονται άδεια από τον scheduler να στείλουν σχετικά άμεσα, ενώ αυτά με χαμηλότερη προτεραιότητα να περιμένουν να ολοκληρωθούν τα μεγαλύτερων προτεραιοτήτων rpc, δηλαδή ο scheduler να τους απαγορεύει να στείλουν όσο υπάρχουν rpc με μεγαλύτερες προτεραιότητες. Μία αφαιρετική απεικόνιση της αρχιτεκτονικής του συστήματος φαίνεται στην Εικόνα 14. Οι οντότητες στη συγκεκριμένη απεικόνιση είναι threads, και κατά τη δημιουργία τους ενημερώνουν τον χρονοδρομολογητή για την ύπαρξη τους καθώς και την προτεραιότητα της κλήσης που έχουν αναλάβει να διεκπεραιώσουν (subscribe_to_send(priority_level)).



Εικόνα 14: Αρχιτεκτονική του συστήματος

Να σημειωθεί ότι οι δύο βασικές γλώσσες προγραμματισμού που χρησιμοποιηθήκαν στις υοποιήσεις ήταν η Go και η C++, προκειμένου να καθοριστεί η καταλληλότητα τόσο διαδικαστικών όσο και αντικειμενοστρεφών γλωσσών στην προτεινόμενη αρχιτεκτονική. Η Go είναι μία από τις δημοφιλέστερες σύγχρονες γλώσσες προγραμματισμού, που διευκολύνει την κλιμακωσιμότητα των software applications και είναι ιδιαίτερα διαδεδομένη σε cloud native εφαρμογές. Από την άλλη, η C++ είναι μία παλαιότερη γλώσσα που παραμένει όμως μέχρι και σήμερα ευρέως χρησιμοποιούμενη σε embedded συστήματα, λειτουργικά συστήματα, βάσεις δεδομένων, web browsers κ.α.

Ανάπτυξη διακομιστή συστήματος

Στο συγκεκριμένο σύστημα αναπτύχθηκε ένας server, ο οποίος υλοποιήθηκε σε γλώσσα Go, και ο οποίος υλοποιεί όλες τις απαραίτητες συναρτήσεις για την εξυπηρέτηση των κλήσεων απομακρυσμένης διαδικασίας που εκτελεί ο πελάτης.

Θεωρήθηκε ότι ο server θέτει έναν περιορισμό maxConcurrentStreams = 1, το οποίο σημαίνει ότι σε κάθε TCP πακέτο, υπάρχουν HTTP/2 πλαίσια από ένα μόνο stream, και όχι από περισσότερα.

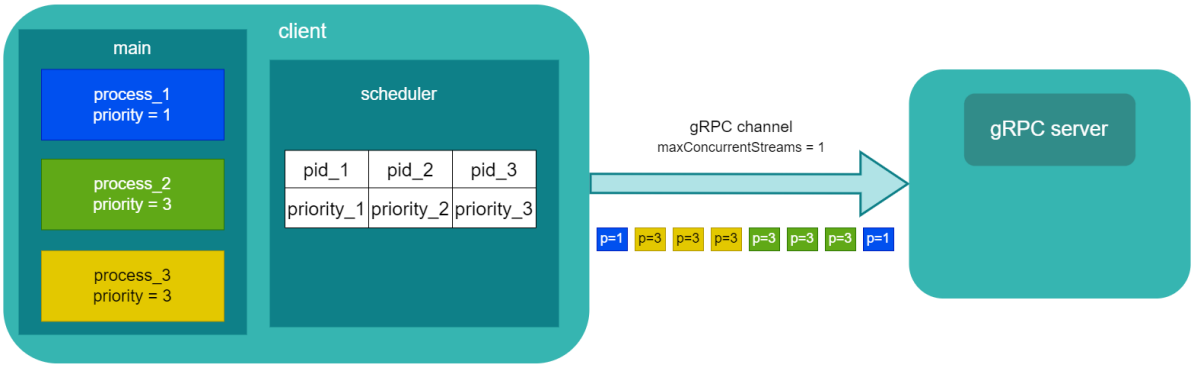
Υλοποίηση σε C++ με processes

Η πρώτη υλοποίηση του client σύμφωνα με την παραπάνω αρχιτεκτονική έγινε σε γλώσσα C++ και με την χρήση διεργασιών. Ο client σε αυτή την περίπτωση, θεωρείται ότι αποτελείται από δύο προγράμματα – διεργασίες, οι οποίες δεν έχουν συγγενική σχέση μεταξύ τους. Το πρώτο είναι το main πρόγραμμα, που υλοποιεί το βασικό functionality του client και τη σύνδεση με τον server, ενώ το δεύτερο είναι το πρόγραμμα του χρονοδρομολογητή, το οποίο αναλαμβάνει τη δρομολόγηση των rpc στο gRPC κανάλι. Το main πρόγραμμα για κάθε rpc που θέλει να στείλει ο client στον server δημιουργεί ένα process, με το οποίο έχει σχέση γονέα – παιδιού. Η επικοινωνία μεταξύ των processes γίνεται με kernel signals καθώς και μέσω ενός socket object.

Όταν ένα process έχει κάποιο rpc να στείλει, προκειμένου να εγγραφεί, δηλαδή να ενημερώσει τον χρονοδρομολογητή σχετικά με την κλήση που θέλει να κάνει, γράφει στο socket το pid (process id) του μαζί με το priority του rpc που έχει αναλάβει και στέλνει ένα SIGUSR1 signal στο process του scheduler ώστε αυτό να διαβάσει το socket. Αντίστοιχη διαδικασία εκτελεί όταν διεκπεραιώνει το rpc του για να απεγγραφεί.

Ο scheduler αποθηκεύει σε έναν πίνακα ένα ζεύγος (pid, priority) για κάθε process που εγγράφεται. Με κάθε SIGUSR1 σήμα διαβάζει από το socket και προσθέτει ή αφαιρεί από

τον πίνακα του ένα ζεύγος, εάν πρόκειται για εγγραφή ή απεγγραφή αντίστοιχα. Παράλληλα αναζητάει κάθε φορά το μεγαλύτερο priority και στέλνει σήματα SIGINT (κοιμίζει ένα process) και SIGCONT (ξυπνάει ένα process) ώστε σε κάθε στιγμή να τρέχει μόνο το process με τη μεγαλύτερη προτεραιότητα. Σε περίπτωση ισοτιμίας priority, προτεραιότητα αναλαμβάνει το παλαιότερο process.



Εικόνα 15: Αρχιτεκτονική συστήματος με C++ processes

Μια σχηματική αναπαράσταση του συστήματος αυτού φαίνεται στην Εικόνα 15. Έστω ότι αρχικά δημιουργείται από το main process το process_1 ώστε να στείλει ένα rpc με προτεραιότητα 1 (μπλε χρώμα). Το process_1 γράφει στο socket επικοινωνίας με τον scheduler την τιμή (“subscribe”, pid_1, priority_1), προκειμένου να τον ενημερώσει σχετικά με το pid του και την προτεραιότητα του rpc που θέλει να στείλει, και έπειτα του στέλνει ένα σήμα SIGUSR1. Αφού στείλει το μήνυμα, η διεργασία κοιμάται μέχρι να λάβει σήμα από τον χρονοδρομολογητή ότι μπορεί να στείλει το rpc της στο κανάλι. Ο scheduler μόλις λάβει το σήμα, διαβάζει από το socket την τιμή που έχει γραφεί, προσθέτει στον πίνακά του το ζεύγος (pid_1, priority_1), και αναζητάει το μεγαλύτερο priority ανάμεσα στις διεργασίες που έχουν εγγραφεί. Εφόσον μόνο η process_1 έχει εγγραφεί μέχρι στιγμής, το priority_1 είναι το μεγαλύτερο και ο scheduler στέλνει σήμα SIGCONT στο process_1, ώστε να το ξυπνήσει και αυτό να ξεκινήσει την αποστολή του rpc του μέσω του καναλιού.

Εάν ο client αποφασίσει να στείλει ένα rpc με προτεραιότητα ίση με 3, τότε δημιουργεί ένα process_2 (πράσινο χρώμα). Το process_2, γράφει στο socket, στέλνει κατάλληλο μήνυμα στον scheduler και κοιμάται. Ο scheduler εγγράφει το ζεύγος (pid_2, priority_2) στον πίνακά του, και αναζητάει το μεγαλύτερο priority. Μεγαλύτερη προτεραιότητα, τώρα έχει το process_2, επομένως ο scheduler στέλνει σήμα SIGINT στο process_1 για να το κοιμίσει, προκειμένου να πάψει να στέλνει τα rpc πακέτα του, και σήμα SIGCONT στο process_2 για να το ξυπνήσει, ώστε να ξεκινήσει να στέλνει να rpc πακέτα του, χωρίς αυτά να πολυπλέκονται με τα πακέτα του process_1 στο κανάλι και να καθυστερούν, εφόσον έχει τη μεγαλύτερη προτεραιότητα.

Ο client, στην περίπτωση του σχήματος της Εικόνας 15, αποφασίζει να στείλει ένα τρίτο rpc, πρωτού τελειώσουν τα δύο προηγούμενα, με προτεραιότητα ίση με 3 (κίτρινο χρώμα). Δημιουργεί λοιπόν το process_3, το οποίο με τη σειρά του ενημερώνει τον χρονοδρομολογητή για την εγγραφή του και έπειτα κοιμάται. Ο scheduler, μετά την εγγραφή του process_3 στον πίνακά του, αναζητάει ξανά το μεγαλύτερο priority. Προτεραιότητα συνεχίζει να έχει το process_2, καθώς ενώ οι διεργασίες 2 και 3 έχουν ίσο priority, το process_2 προηγείται εφόσον εγγράφηκε πρώτο. Ο scheduler, λοιπόν, συνεχίζει να αφήνει το process_2 να στέλνει και τις δύο άλλες διεργασίες να παραμένουν αδρανείς.

Όταν το process_2 τελειώσει την εκτέλεση του rpc, γράφει στο socket την τιμή (“remove”, pid_2), προκειμένου να απεγγραφεί, ενημερώνει τον χρονοδρομολογητή με ένα σήμα

SIGUSR1, και έπειτα τερματίζεται. Ο χρονοδρομολογητής, λαμβάνοντας το σήμα και διαβάζοντας το socket, αφαιρεί το ζεύγος (pid_2, priority_2) από τον πίνακά του και αναζητάει τη μεγαλύτερη προτεραιότητα. Μεγαλύτερο priority τώρα έχει το process_3, γι' αυτό και του στέλνει σήμα SIGCONT για να το ξυπνήσει, ενώ το process_1 συνεχίζει να κοιμάται. Το process_3 στέλνει τώρα μόνο του τα πακέτα μέσα από το κανάλι, χωρίς να καθυστερείται από κάποιο άλλο process μικρότερης προτεραιότητας. Όταν ολοκληρώσει την εκτέλεση του rpc του, ενημερώνει τον scheduler και με αντίστοιχη διαδικασία που περιγράφηκε προηγουμένως, ο χρονοδρομολογητής δίνει άδεια στο process_1 να συνεχίσει την αποστολή των πακέτων του. Η εναλλαγή των πακέτων των διαφόρων διεργασιών στο gRPC κανάλι φαίνεται στο σχήμα της Εικόνας 15.

Στους παρακάτω πίνακες φαίνεται ο ψευδοκώδικας για τον scheduler καθώς και τα υπόλοιπα processes.

	Ψευδοκώδικας scheduler

	`while true:`
	`  max_priority_process = get_max_priority_process()`
	`  if max_priority_process != running_process:`
	`  send(running_process, SIGINT)`
	`  running_process = max_priority_process`
	`  send(max_priority_process, SIGCONT)`

SIGUSR1	`input = read_from_socket()`
	`if input == (“subscribe”, pid, priority):`
	`  subscribe(pid, priority)`
	`if input == (“remove”, pid):`
	`  remove_priority(pid)`

Πίνακας 1: Ψευδοκώδικας scheduler στην C++ processes υλοποίηση

	Ψευδοκώδικας processes

	`write_to_socket((“subscribe”, pid, priority))`
	`send(scheduler, SIGUSR1)`
	`sleep()`

SIGCONT	`send_rpc()`
	`write_to_socket((“remove”, pid))`
	`send(scheduler, SIGUSR1)`

SIGINT	`sleep()`

Πίνακας 2: Ψευδοκώδικας processes στην C++ processes υλοποίηση

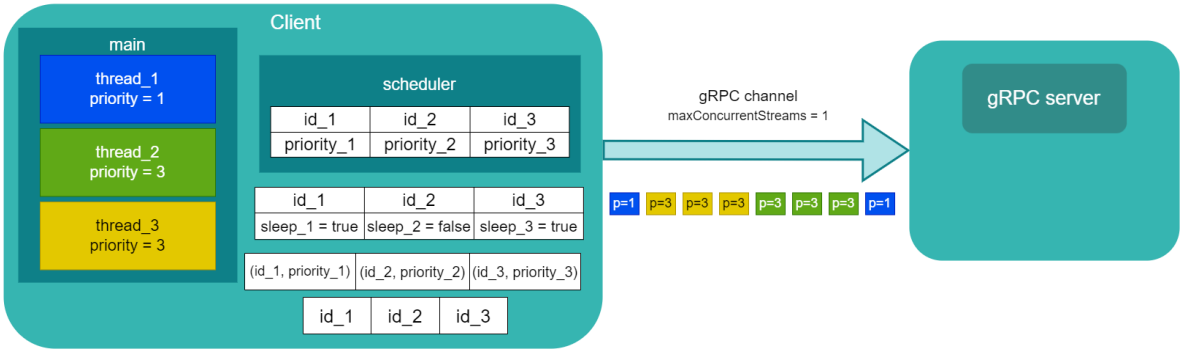
Πρώτη υλοποίηση σε C++ με threads

Η δεύτερη υλοποίηση του client έγινε σε γλώσσα C++ και με την χρήση νημάτων. Σε αυτήν την περίπτωση, όπως και στην πρώτη υλοποίηση, ο client αποτελείται από δύο βασικά νήματα, του ίδιου όμως προγράμματος, εκείνο που υλοποιεί τις βασικές λειτουργίες του πελάτη και τη σύνδεσή του με τον διακομιστή, και εκείνο που έχει το ρόλο του χρονοδρομολογητή. Κάθε φορά που ο πελάτης θέλει να εκτελέσει κάποιο rpc, το βασικό νήμα δημιουργεί ένα thread το οποίο αναλαμβάνει να διεκπεραιώσει το συγκεκριμένο rpc.

Η επικοινωνία μεταξύ των διαφόρων νημάτων γίνεται μέσω των πόρων της κοινής τους μνήμης. Για την πρόσβαση στους κοινούς πόρους χρησιμοποιούνται κλειδώματα, προκειμένου η ανάγνωση και η εγγραφή σε αυτούς να είναι ατομικές.

Στην κοινή μνήμη των threads υπάρχει μία subscribe queue, στην οποία προσθέτει ένα ζεύγος (id, priority) κάθε thread που δημιουργείται για να στείλει ένα rpc, καθως και μία unsubscribe queue, στην οποία προσθέτει το id του όποιο thread ολοκληρώνει το rpc του. Υπάρχει επίσης ένας πίνακας με ένα ζεύγος (id, sleep) για κάθε thread, όπου sleep = true υποδηλώνει ότι το thread με id δεν μπορεί να στείλει το rpc του και θα πρέπει να είναι αδρανές, ενώ sleep = false σημαίνει ότι μπορεί να στείλει. Όταν, λοιπόν, ένα thread δημιουργείται για να διεκπεραιώσει κάποια κλήση, τότε προσθέτει το id του στην ουρά εγγραφής και έπειτα συνεχώς διαβάζει το ζεύγος (id, sleep) από τον κατάλληλο πίνακα μέχρι να γίνει sleep = false. Τότε ξεκινάει να στέλνει το rpc ενώ παράλληλα ελέγχει αν παραμένει false η τιμή του sleep. Σε περίπτωση που αυτό αλλάξει, το νήμα σταματάει να στέλνει και παραμένει αδρανές μέχρι να γίνει ξανά η τιμή του sleep ίση με false. Όταν ολοκληρωθεί το rpc, πριν τερματιστεί το νήμα πρέπει να προσθέσει το id του στην ουρά απεγγραφής.

Στην πλευρά του χρονοδρομολογητή, εκείνος διατηρεί έναν πίνακα προτεραιοτήτων με ένα ζεύγος (id, priority) για κάθε νήμα που είναι εγγεγραμμένο, ενώ ελέγχει συνεχώς τις ουρές εγγραφής και απεγγραφής. Σε περίπτωση που η ουρά εγγραφής δεν είναι κενή, τότε προσθέτει στον πίνακά του το νήμα που έχει εγγραφεί. Αντίστοιχα εάν η ουρά απεγγραφής δεν είναι κενή, τότε αφαιρεί από τον πίνακα το ζεύγος που αντιστοιχεί στο νήμα που έχει απεγγραφεί. Κάθε φορά που γίνεται μία εγγραφή ή απεγγραφή, ο scheduler αναζητάει το νέο μέγιστο priority, και ανανεώνει τον πίνακα των μεταβλητών sleep, με τρόπο τέτοιον ώστε να είναι false μόνο η μεταβλητή του νήματος που έχει την μεγαλύτερη προτεραιότητα. Εάν δύο νήματα έχουν ίσο priority, προτεραιότητα αναλαμβάνει το παλαιότερο νήμα.



Εικόνα 86: Πρώτη αρχιτεκτονική συστήματος με C++ threads

Μία σχηματική αναπαράσταση της συγκεκριμένης υλοποίησης φαίνεται στην Εικόνα 16. Έστω ότι αρχικά δημιουργείται από το main thread το thread_1 ώστε να στείλει ένα rpc με προτεραιότητα 1 (μπλε χρώμα). Το thread_1 προσθέτει στο πίσω μέρος της ουράς εγγραφής το ζεύγος (id_1, priority_1), καθώς και στον πίνακα με τις sleep μεταβλητές το ζεύγος (id_1, true) και περιμένει μέχρι η μεταβλητή αυτή να γίνει false από τον scheduler. Ο scheduler αντιλαμβάνεται ότι η ουρά εγγραφής δεν είναι άδεια και την διαβάζει. Προσθέτει το ζεύγος (id_1, priority_1) στον πίνακά του και αναζητάει την μεγαλύτερη προτεραιότητα. Εφόσον μόνο το thread_1 υπάρχει στον πίνακα προτεραιοτήτων, το priority_1 είναι το μεγαλύτερο και ο scheduler θέτει τη μεταβλητή sleep_1 ίση με false και το thread_1 ξεκινάει να στέλνει τα πακέτα του.

Πρωτού ολοκληρωθεί το rpc του thread_1, ο πελάτης αποφασίζει να στείλει ένα rpc με προτεραιότητα 3, και το main thread δημιουργεί το thread_2 για να το εκτελέσει (πράσινο χρώμα). Το thread_2 προσθέτει το ζεύγος (id_2, priority_2) στην ουρά εγγραφής, εισάγει το id_2 στον πίνακα sleep και περιμένει την μεταβλητή sleep_2 να γίνει false. Ο scheduler διαβάζει το ζεύγος (id_2, priority_2), το προσθέτει στον πίνακα προτεραιοτήτων και έπειτα αναζητάει σε αυτόν τη μέγιστη προτεραιότητα. Η μέγιστη προτεραιότητα πλέον είναι το priority_2, οπότε θέτει sleep_1 = true, ώστε το thread_1 να απελευθερώσει το κανάλι και sleep_2 = false, ώστε το thread_2 να αρχίσει να στέλνει τα πακέτα του.

Όσο το thread_2 εκτελεί το rpc του, ο πελάτης θέλει να στείλει ένα τρίτο rpc με προτεραιότητα 3 (κίτρινο χρώμα), επομένως δημιουργείται ένα thread_3 που μέσω της ουράς εγγραφής ενημερώνει τον χρονοδρομολογητή ότι θέλει να χρησιμοποιήσει το κανάλι και ύστερα περιμένει να γίνει sleep_3 = false. Ο scheduler προσθέτει το κατάλληλο ζεύγος στον πίνακα, και αναζητάει την πεγαλύτερη προτεραιότητα η οποία παραμένει η priority_2, εφόσον το thread_2 είναι παλαιότερο. Όταν ολοκληρωθεί το rpc του thread_2, εκείνο προσθέτει το id_2 στην ουρά απεγγραφής. Όταν ο χρονοδρομολογητής το διαβάσει, αφαιρεί το ζεύγος (id_2, priority_2) από τον πίνακά του και αφού βρει ότι τώρα η μεγαλύτερη προτεραιότητα είναι η priority_3, ενημερώνει κατάλληλα την μεταβλητή sleep_3. Το thread_3 ξεκινάει να στέλνει, ενώ μόλις ολοκληρώσει το rpc του, ο scheduler ενημερώνεται μέσω της ουράς απεγγραφής και με αντίστοιχη με προηγουμένως διαδικασία επιτρέπει στο thread_1 να συνεχίσει να στέλνει τα πακέτα του. Η εναλλαγή των πακέτων των διαφόρων νημάτων στο κανάλι φαίνεται στο αντίστοιχο σημείο του σχήματος της Εικόνας 16.

Στους πίνακες 3 και 4 φαίνεται ο ψευδοκώδικας για τον scheduler καθώς και τα υπόλοιπα threads.

Ψευδοκώδικας scheduler

```
while true:
  if subscribe_queue.size > 0:
    id, priority = subscribe_queue.pop()
    if priority > max_priority:
      max_priority = priority
      max_id = id

  if unsubscribe_queue.size > 0:
    unsubscrive_id = unsubscribe_queue.pop()
    max_id, max_priority = get_max_priority()
    running_id = -1

  if max_id != running_id:
    sleep[running_id] = true
    sleep[max_id] = false
    running_id = max_id
```

Πίνακας 3: Ψευδοκώδικας scheduler στην πρώτη C++ threads υλοποίηση

Ψευδοκώδικας threads

```
for rpc_packets_to_send:
  while(sleep == true)
  send_rpc_packet()

unsubscribe_queue.push(id)
```

Πίνακας 4: Ψευδοκώδικας threads στην πρώτη C++ threads υλοποίηση

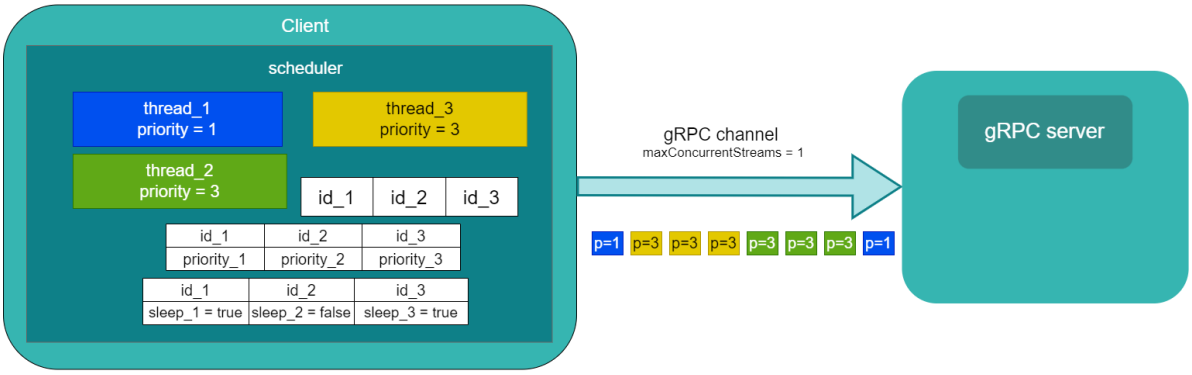
Δεύτερη υλοποίηση σε C++ με threads

Η συγκεκριμένη υλοποίηση μοιάζει αρκετά με την προηγούμενη, με τη διαφορά ότι αυτή τη φορά τα νήματα που εκτελούν τα διάφορα rpc δημιουργούνται από το νήμα του scheduler και όχι από το main thread. Η επικοινωνία μεταξύ των νημάτων γίνεται και πάλι μέσω της κοινής μνήμης τους, ενώ για την πρόσβαση σε κοινούς πόρους χρησιμοποιούνται κλειδώματα.

Οι δομές που υπάρχουν στην κοινή μνήμη των νημάτων είναι μία unsubscribe queue, ο πίνακας των μεταβλητών sleep και ο πίνακας προτεραιοτήτων, οι οποίες όλες έχουν ίδια λειτουργία με την προηγούμενη υλοποίηση. Η ουρά εγγραφής, η οποία απουσιάζει, δεν είναι απαραίτητη εφόσον ο scheduler δημιουργεί τα νήματα των rpc και επομένως δεν χρειάζεται να ενημερωθεί από κάποιο άλλο νήμα μέσω κάποιας δομής.

Όταν, λοιπόν, ο πελάτης θέλει να εκτελέσει μία κλήση, ο scheduler δημιουργεί ένα thread και προσθέτει το ζεύγος (id, priority) στον πίνακα προτεραιοτήτων του. Παράλληλα, ελέγχει συνεχώς τη ουρά απεγγραφής, και αν αυτή δεν είναι κενή, αφαιρεί το ζεύγος που αντιστοιχεί στο νήμα που έχει απεγγραφεί. Κάθε φορά που γίνεται μία εγγραφή ή απεγγραφή, ο χρονοδρομολογητής αναζητάει τη μέγιστη προτεραιότητα, και ανανεώνει κατάλληλα τον πίνακα με τις sleep μεταβλητές, ώστε κάθε στιγμή να στέλνει πακέτα στο κανάλι μόνο το thread που εκτελεί το rpc με το μέγιστο priority, ενώ τα υπόλοιπα παραμένουν αδρανή. Σε περίπτωση ισοτιμίας προτεραιοτήτων, προηγείται το νήμα που δημιουργήθηκε νωρίτερα.

Τα υπόλοιπα νήματα, αφού δημιουργηθούν, περιμένουν να τεθεί η τιμή της sleep μεταβλητής τους ίση με false. Μόλις γίνει αυτό, ξεκινούν να στέλνουν τα απαραίτητα για την εκτέλεση των rpc τους πακέτα, ενώ παράλληλα ελέγχουν την sleep μεταβλητή τους, για την περίπτωση που χρειάζεται να διακόψουν την αποστολή των πακέτων. Όταν ολοκληρώσουν την rpc κλήση τους, προσθέτουν το id τους στο πίσω μέρος της ουράς απεγγραφής.



Εικόνα 97: Δεύτερη αρχιτεκτονική συστήματος με C++ threads

Μια σχηματική αναπαράσταση του παραπάνω συστήματος για την ύπαρξη τριών νημάτων φαίνεται στην Εικόνα 17. Αρχικά ο πελάτης θέλει να στείλει ένα rpc με προτεραιότητα 1.

Τότε ο χρονοδρομολογητής δημιουργεί το κατάλληλο νήμα (μπλε χρώμα) και εισάγει το ζεύγος (id_1, priority_1) στον πίνακα προτεραιοτήτων. Αναζητώντας τη μέγιστη προτεραιότητα, βρίσκει ότι αυτή είναι η priority_1, εφόσον αυτή είναι η μοναδική προτεραιότητα στον πίνακα, και θέτει sleep_1 = false. Το thread_1 τότε ξεκινάει να στέλνει πακέτα.

Κατά τη δημιουργία του thread_2 (πράσινο χρώμα), ο χρονοδρομολογητής εισάγει τα κατάλληλα ζεύγη στους πίνακες προτεραιοτήτων και μεταβλητών sleep, και έπειτα βρίσκει ότι η priority_2, που ισούται με 3, είναι η μαγαλύτερη προτεραιότητα. Επομένως, ανανεώνει κατάλληλα τον πίνακα sleep θέτοντας sleep_1 = true και sleep_2 = false. Τότε το thread_1 σταματάει να στέλνει πακέτα, απελευθερώνοντας το κανάλι, ενώ το thread_2 ξεκινάει την εκτέλεση του rpc του.

Με τη δημιουργία του thread_3 με προτεραιότητα 3 (κίτρινο χρώμα), ο χρονοδρομολογητής εξετάζει τη μέγιστη προτεραιότητα, η οποία παραμένει η priority_2 εφόσον το thread_2 δημιουργήθηκε νωρίτερα από το thread_3, και άρα θέτει sleep_3 = true. Όταν το thread_2 διεκπεραιώσει την εκτέλεση του rpc του, πριν τερματιστεί, προσθέτει το id_2 στην ουρά απεγγραφής. Ο scheduler, ο οποίος αντιλαμβάνεται ότι η ουρά απεγγραφής δεν είναι κενή, αφαιρεί το ζεύγος (id_2, priority_2) από τον πίνακα προτεραιοτήτων και βρίσκει ότι πλέον η μέγιστη προτεραιότητα είναι η priority_3, επομένως ανανεώνει κατάλληλα τον πίνακα των sleep μεταβλητών. Μόλις το rpc του thread_3 ολοκληρωθεί, ο scheduler ακολουθεί αντίστοιχη διαδικασία και το thread_1 συνεχίζει την αποστολή των πακέτων του.

Ο ψευδοκώδικας του χρονοδρομολογητή, καθώς και των υπόλοιπων νημάτων φαίνεται στους πίνακες 5 και 6.

Ψευδοκώδικας scheduler

```
while true:
  id, rpc, priority = read_rpc;
  if priority > max_priority:
    sleep[max_id] = true
    sleep[id] = false
    max_id = id
    max_priority = priority
  thread(id, rpc)

  if unsubscribe_queue.size > 0:
    unsubscrive_id = unsubscribe_queue.pop()
    max_id, max_priority = get_max_priority()
    sleep[max_id] = false
```

Πίνακας 5: Ψευδοκώδικας scheduler στην δεύτερη C++ threads υλοποίηση

Ψευδοκώδικας threads

```
for rpc_packets_to_send:
  while(sleep == true)
  send_rpc_packet()

unsubscribe_queue.push(id)
```

Πίνακας 6: Ψευδοκώδικας threads στην δεύτερη C++ threads υλοποίηση

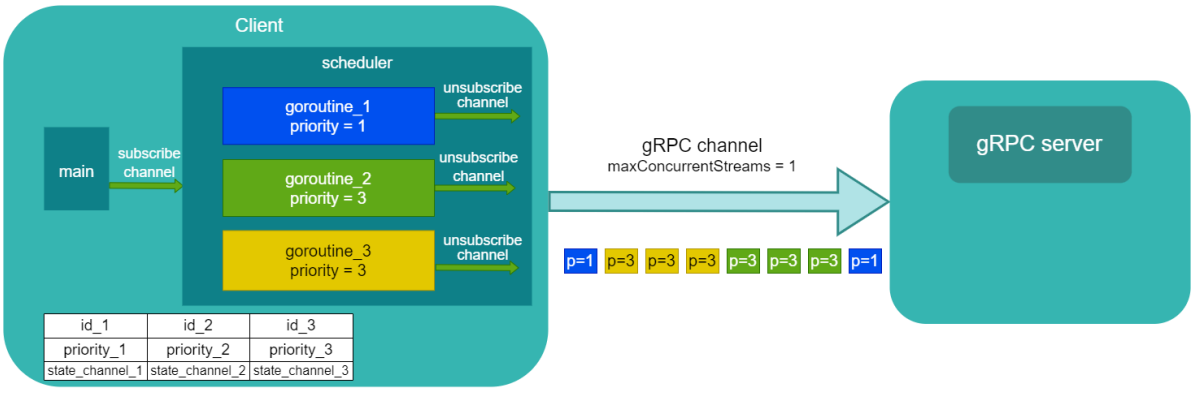
Πρώτη υλοποίηση σε Go με goroutines

Η επόμενη υλοποίηση του πελάτη έγινε σε γλώσσα Go και με τη χρήση goroutines. Μία goroutine στην Go είναι μία συνάρτηση που τρέχει παράλληλα με το υπόλοιπο πρόγραμμα, και ουσιαστικά υλοποιείται με ένα ξεχωριστό thread. Ο client αποτελείται από το main πρόγραμμα που υλοποιεί το βασικό functionality του πελάτη και τη σύνδεσή του με τον διακομιστή, και από το goroutine του χρονοδρομολογητή. Όταν ο πελάτης θέλει να εκτελέσει ένα rpc ενημερώνει τον χρονοδρομολογητή σχετικά με το rpc και την προτεραιότητά του, και ο χρονοδρομολογητής δημιουργεί ένα νέο goroutine για κάθε τέτοιο rpc. Η επικοινωνία μεταξύ των goroutines επιτυγχάνεται με τη χρήση των channels, τα οποία είναι ειδικά bidirectional pipes που παρέχει η Go για την επικοινωνία ανάμεσα σε goroutines.

Στην κοινή μνήμη όλων των threads βρίσκεται ένα subscribe και ένα unsubscribe channel. Όταν ο πελάτης θέλει να εκτελέσει ένα rpc, τότε το main πρόγραμμα γράφει στο subscribe channel πληροφορίες σχετικά με αυτό το rpc καθώς και με την προτεραιότητά του. Αντίστοιχα, όταν ένα goroutine έχει ολοκληρώσει το rpc του, γράφει στο unsubscribe channel το id του.

Στο κάθε goroutine δίνεται ένα state channel στο οποίο ο scheduler στέλνει την κατάσταση στην οποία πρέπει να είναι το thread κάθε φορά, αν θα πρέπει δηλαδή να στέλνει στο κανάλι (κατάσταση true) ή όχι (κατάσταση false). Το goroutine ελέγχει το channel αυτό περιοδικά και αναλόγως στέλνει ή περιμένει άδεια να στείλει. Όταν ολοκληρώσει το rpc του ενημερώνει τον scheduler ότι τελείωσε γράφοντας στο unsubscribe channel.

Ο scheduler διαβάζει από το subscribe channel τι rpc πρέπει να εκτελεστεί και με ποιά προτεραιότητα και δημιουργεί ένα goroutine για να το διεκπεραιώσει. Διατηρεί μία map δομή στην οποία κλειδί είναι το id του κάθε goroutine και τιμή είναι η προτεραιότητα και το state channel του συγκεκριμένου goroutine. Κάθε φορά που γίνεται κάποιο subscribe ή unsubscribe αναζητάει την μεγαλύτερη προτεραιότητα και στέλνει στο κατάλληλο goroutine μέσω του state channel του να σταματήσει, να ξεκινήσει ή να συνεχίσει την εκτέλεση του rpc του, ώστε κάθε φορά στο κανάλι να υπάρχουν πακέτα μόνο του goroutine που εκτελεί το rpc με τη μεγαλύτερη προτεραιότητα. Σε περίπτωση ίσων προτεραιοτήτων, προηγείται το goroutine που δημιουργήθηκε νωρίτερα.



Εικόνα 108: Πρώτη αρχιτεκτονική συστήματος με Go threads

Στο σχήμα της Εικόνας 18, φαίνεται μια σχηματική αναπαράσταση της παραπάνω αρχιτεκτονικής. Στην αρχή, ο πελάτης θέλει να στείλει ένα rpc με προτεραιότητα 1, επομένως το main πρόγραμμα γράφει στο subscribe channel πληροφορίες σχετικά με αυτό

το rpc. Ο χρονοδρομολογητής αντιλαμβάνεται ότι κάτι έχει γραφεί στο κανάλι εγγραφής και το διαβάζει. Δημιουργεί το state_channel_1 και προσθέτει στο map του την κατάλληλη εγγραφή (id_1, priority_1, state_channel_1). Ύστερα, δημιουργεί το goroutine_1 (μπλε χρώμα) για να εκτελέσει το συγκεκριμένο rpc δίνοντας του το state_channel_1 μέσω του οποίου θα το ενημερώνει σχετικά με την κατάσταση στην οποία θα πρέπει να βρίσκεται κάθε στιγμή. Το goroutine_1 περιμένει αδρανές μέχρι να του δοθεί άδεια να ξεκινήσει την αποστολή των πακέτων του. Ο scheduler αναζητάει τη μεγαλύτερη προτεραιότητα, η οποία βρίσκει ότι είναι η priority_1 και γράφει στο state_channel_1 την τιμή true, ώστε το goroutine_1 να ξεκινήσει την εκτέλεση του rpc του.

Όταν δημιουργηθεί η ανάγκη για την εκτέλεση ενός δεύτερου rpc με προτεραιότητα 3, το βασικό πρόγραμμα ενημερώνει αντίστοιχα τον χρονοδρομολογητή, ο οποίος εκτελεί τις απαραίτητες ενέργειες και έπειτα δημιουργεί το goroutine_2 (πράσινο χρώμα). Η μεγαλύτερη προτεριαότητα πλέον είναι η priority_2 και επομένως ο scheduler γράφει την τιμή false στο κανάλι του goroutine_1, ώστε αυτό να διακόψει την εκτέλεσή του, και την τιμή true στο κανάλι του goroutine_2 , το οποίο ξεκινάει την αποστολή των πακέτων του.

Στη συνέχεια, το βασικό πρόγραμμα ενημερώνει τον χρονοδρομολογητή μέσω του subscribe channel για την ανάγκη εκτέλεσης ενός τρίτου rpc με προτεραιότητα ίση με 3. Ο χρονοδρομολογητής τότε, αφού προσθέσει τις κατάλληλες τιμές στο map του, δημιουργεί το goroutine_3 (κίτρινο χρώμα). Επειδή όμως το goroutine_2 είναι παλαιότερο από το goroutine_3, παρόλο που οι προτεραιότητές τους είναι ίσες, ο scheduler δεν γράφει κάποια τιμή στα κανάλια τους, αφήνοντας το goroutine_2 να συνεχίσει την εκτέλεσή του και το goroutine_3 να είναι αδρανές. Όταν το goroutine_2 ολοκληρώσει το rpc του, ενημερώνει τον χρονοδρομολογητή μέσω του καναλιού απεγγραφής, και αυτός με τη σειρά του διαγράφει τη θέση του map που αντιστοιχεί στο goroutine_2 και αναζητάει την επόμενη μεγαλύτερη προτεραιότητα. Αφού βρει ότι το μέγιστο priority είναι εκείνο του goroutine_3, γράφει στο κανάλι του την τιμή true, ώστε αυτό να ξεκινήσει την αποστολή των πακέτων του, ενώ το goroutine_1 παραμένει αδρανές. Κατά την ολοκλήρωση και του goroutine_3, ο χρονοδρομολογητής επιτρέπει στο goroutine_1 να συνεχίσει την αποστολή των πακέτων του, ενημερώνοντάς του μέσω του state_channel_1.

Ο ψευδοκώδικας του χρονοδρομολογητή, καθώς και των υπόλοιπων goroutine φαίνεται στους πίνακες 7 και 8.

	Ψευδοκώδικας scheduler
	`wait for input from channel`

input from subscribe channel	`rpc, priority = read_from_subscribe_channel()`
	`goroutine(rpc, state_channel, id)`
	
	`if priority > running_id.priority:`
	`  running_id.channel <- false`
	`  state_channel <- true`
	`  running_id = id`

input from unsubscribe channel	`id = read_from_unsubscribe_channel()`
	`max_id = get_maximum_priority()`
	`max_id.channel <- true`

Πίνακας 7: Ψευδοκώδικας scheduler στην πρώτη Go υλοποίηση

	Ψευδοκώδικας goroutines
	`wait for input from state channel`
true in state channel	`run_rpc()` `unsubscribe_channel <- id`
false in state channel	`wait for input from state channel`

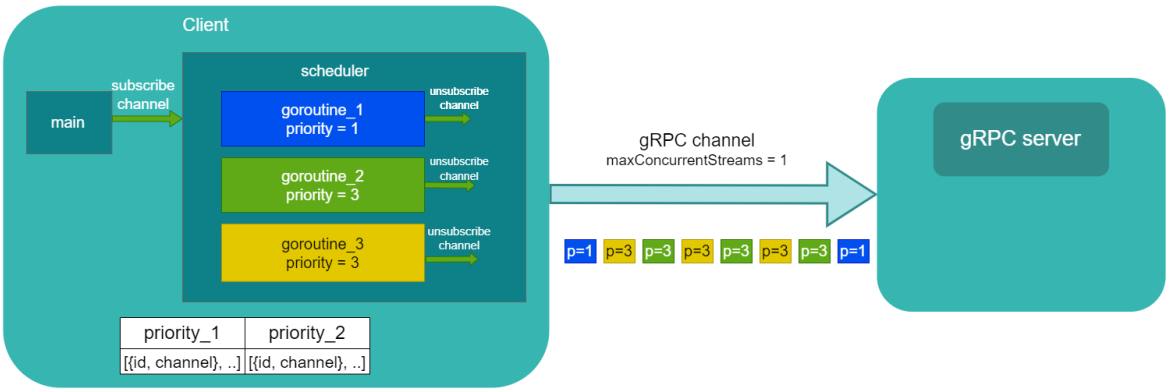
Πίνακας 8: Ψευδοκώδικας goroutines στην πρώτη Go υλοποίηση

Δεύτερη υλοποίηση σε Go με goroutines

Η τελευταία υλοποίηση του client έγινε σε γλώσσα Go με χρήση goroutines, και μοιάζει αρκετά με την προηγούμενη υλοποίηση που περιγράφηκε. Χρησιμοποιούνται και πάλι channels για την επικοινωνία μεταξύ των goroutines. Το βασικό πρόγραμμα ενημερώνει τον scheduler σχετικά με τα rpc που πρέπει να εκτελεστούν μέσω του καναλιού εγγραφής, και ο εκείνος δημιουργεί τα κατάλληλα goroutines. Τα goroutines ενημερώνονται μέσω των state channels τους σχετικά με την κατάσταση στην οποία πρέπει να βρίσκονται και όταν ολοκληρώσουν την εκτέλεσή τους γράφουν το id τους στο unsubscribe channel.

Η διαφορά αυτής της υλοποίησης με τις προηγούμενες, είναι ότι τα rpc που έχουν την ίδια προτεραιότητα μεταξύ τους τρέχουν ταυτόχρονα, με τα πακέτα τους να πολυπλέκονται στο κανάλι, σύμφωνα με τον τρόπο που το gRPC ήδη πολυπλέκει τα ταυτόχρονα streams. Η χρονοδρομολόγηση, δηλαδή, γίνεται ανά προτεραιότητα και όχι ανά ξεχωριστό rpc.

Ο scheduler σε αυτήν την περίπτωση, διατηρεί μία map δομή στην οποία κλειδί είναι η προτεραιότητα και τιμή είναι ένας πίνακας με το id και το state channel του κάθε goroutine που έχει τη συγκεκριμένη προτεραιότητα. Κάθε φορά που γίνεται κάποιο subscribe ή unsubscribe ο χρονοδρομολογητής αναζητάει την μεγαλύτερη προτεραιότητα και στέλνει στα κατάλληλα goroutines μέσω των state channels τους να σταματήσουν, να ξεκινήσουν ή να συνεχίσουν να στέλνουν.



Εικόνα 119: Δεύτερη αρχιτεκτονική συστήματος με Go threads

Στο παράδειγμα που φαίνεται στο σχήμα της Εικόνας 19, αρχικά ο πελάτης θέλει να εκτελέσει ένα rpc με προτεραιότητα ίση με 1. Ο scheduler ενημερώνεται μέσω του καναλιού εγγραφής και δημιουργεί την goroutine_1 (μπλε χρώμα), τα στοιχεία της οποίας προσθέτει στο map του. Μέγιστη προτεραιότητα είναι η 1, επομένως ο χρονοδρομολογητής

γράφει την τιμή true στα κανάλια όσων goroutines έχουν προτεραιότητα ίση με 1, στη συγκεκριμένη περίπτωση μόνο στο κανάλι του goroutine_1.

Όταν ο χρονοδρομολογητής ενημερώνεται για την εκτέλεση ενός δεύτερου rpc με προτεραιότητα 3, προσθέτει την τιμή 3 στα κλειδιά του map του και δημιουργεί το goroutine_2 (πράσινο χρώμα). Εφόσον η 3 είναι πλέον η μέγιστη προτεραιότητα, ο scheduler στέλνει false στα κανάλια των goroutines με την προηγούμενη μέγιστη προτεραιότητα, και true στα κανάλια των goroutines με τη νέα μέγιστη προτεραιότητα. Επομένως το goroutine_1 παύει να εκτελεί το rpc του, ενώ το goroutine_2 ξεκινάει την αποστολή των πακέτων του.

Όταν δημιουργείται η ανάγκη εκτέλεσης ενός τρίτου rpc με προτεραιότητα 3, ο scheduler δημιουργεί το goroutine_3 (κίτρινο χρώμα) και προσθέτει το id και το state channel του στον πίνακα που αντιστοιχεί στην προτεραιότητα 3 στο map του. Εφόσον η προτεραιότητα 3 είναι η μέγιστη εκείνη τη στιγμή, ο χρονοδρομολογητής γράφει την τιμή true στο κανάλι του goroutine_3, και επομένως τώρα τα πακέτα των goroutine 2 και 3, που εκτελούν rpc ίσων προτεραιοτήτων, πολυπλέκονται στο gRPC κανάλι. Κάθε goroutine που διεκπεραιώνει το rpc του ενημερώνει τον χρονοδρομολογητή μέσω του unsubscribe channel και όταν ολοκληρωθούν όλα τα goroutines που αντιστοιχούν στη μέγιστη προτεραιότητα, τότε εκείνος αναζητάει τη νέα μέγιστη προτεραιότητα. Το μεγαλύτερο priority τώρα είναι το 1, και άρα ο scheduler δίνει άδεια στο goroutine_1 να συνεχίσει να στέλνει τα πακέτα του.

Ο ψευδοκώδικας του χρονοδρομολογητή, καθώς και των υπόλοιπων goroutine φαίνεται στους πίνακες 9 και 10.

	Ψευδοκώδικας scheduler
	`wait for input from channel`

input from subscribe channel	`rpc, priority = read_from_subscribe_channel()`
	`goroutine(rpc, state_channel, id)`
	`if priority > max_priority:`
	`  set_asleep(max_priority)`
	`  state_channel <- true`
	`  max_priority = priority`

input from unsubscribe channel	`id, priority = read_from_unsubscribe_channel()`
	`if len(array[priority]) == 0:`
	`  max_priority = get_maximum_priority()`
	`  set_active(max_priority)`

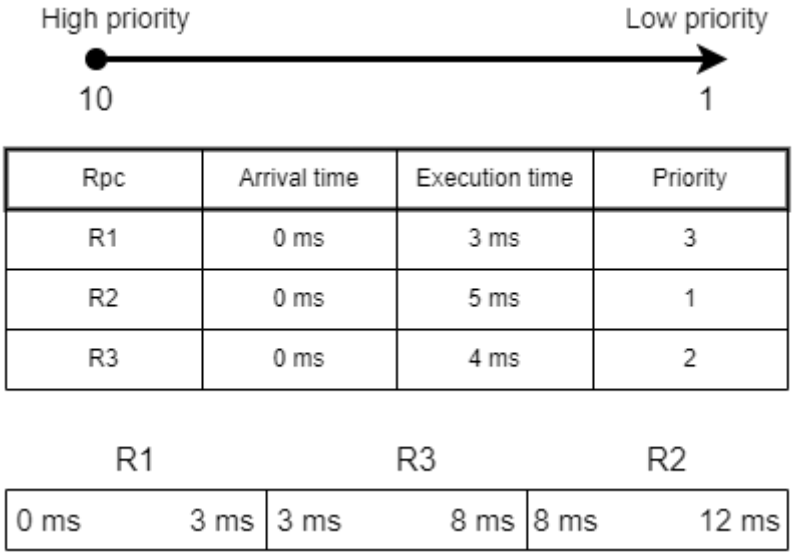
Πίνακας 9: Ψευδοκώδικας scheduler στην δεύτερη Go υλοποίηση

	Ψευδοκώδικας goroutines
	`wait for input from state channel`
true in state channel	`run_rpc()` `unsubscribe_channel <- id`
false in state channel	`wait for input from state channel`

Πίνακας 10: Ψευδοκώδικας goroutines στην δεύτερη Go υλοποίηση

Εισαγωγή

Οι υλοποιήσεις που περιγράφηκαν παραπάνω χρησιμοποιούν όλες έναν priority based scheduling αλγόριθμο. Ο πελάτης, δηλαδή, παραχωρεί μία προτεραιότητα σε κάθε rpc κλήση που θέλει να εκτελέσει, αναλόγως του πόσο σημαντική είναι αυτή, και ο χρονοδρομολογητής επιλέγει ποιό rpc θα εκτελεστεί κάθε φορά με βάση την προτεραιότητά του. Σε περίπτωση ισοτιμίας προτεραιοτήτων σε κάποιες υλοποιήσεις επιλέγεται το rpc που δημιουργήθηκε νωρίτερα, ενώ στην τελευταία υλοποίηση υπάρχει πολυπλεξία των κλήσεων που έχουν ίσο priority. Μια σχηματική αναπαράσταση του priority based αλγορίθμου δρομολόγησης φαίνεται στην Εικόνα 20.



Εικόνα 2012: Priority based δρομολόγηση

Στην παραπάνω εικόνα, φαίνεται ότι το rpc R1 έχει τη μεγαλύτερη προτεραιότητα, επομένως εκτελείται πρώτο. Μόλις ολοκληρωθεί, σειρά έχει το R3 το οποίο έχει την αμέσως επόμενη μεγαλύτερη προτεραιότητα, ενώ τελευταίο εκτελείται το R2, που έχει και τη μικρότερη προτεραιότητα. Να σημειωθεί ότι όσο μεγαλύτερος είναι ο αριθμός προτεραιότητας ενός rpc, τόσο μεγαλύτερη είναι και η προτεραιότητά του.

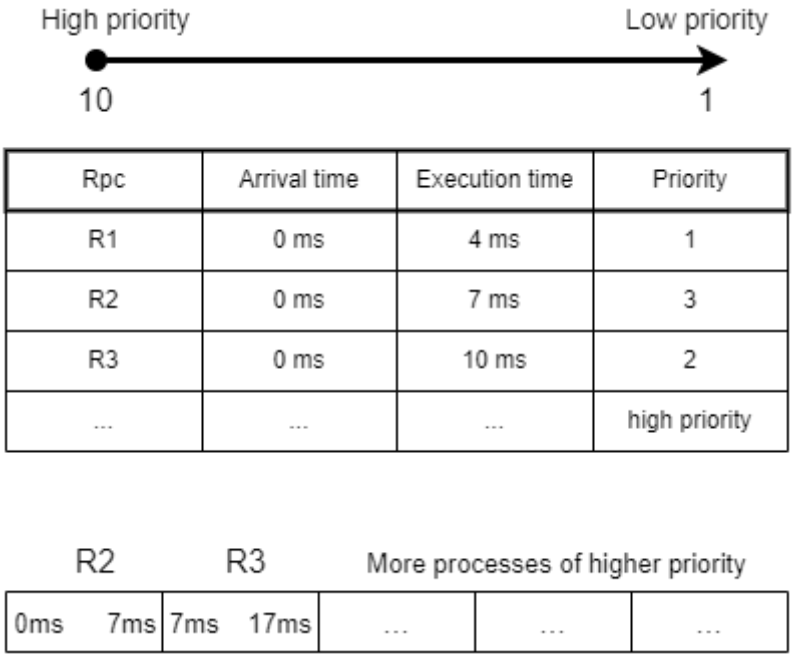
Το πρόβλημα του starvation

Κατά τον priority based αλγόριθμο χρονοδρομολόγησης, ωστόσο, μπορεί να δημιουργηθεί πρόβλημα εάν η προτεραιότητα ενός rpc είναι πολύ μικρή και συνεχώς προκύπτουν rpc κλήσεις μεγαλύτερων προτεραιοτήτων. Σε αυτή την περίπτωση ο χρονοδρομολογητής θα επιτρέπει στις κλήσεις με υψηλότερο priority να εκτελούνται, ενώ το rpc με το χαμηλό priority θα συνεχίσει να περιμένει τη σειρά του.

Στο παράδειγμα της Εικόνας 21, το rpc R2 έχει τη μεγαλύτερη προτεραιότητα, ενώ το R1 έχει τη μικρότερη προτεραιότητα. Εάν πριν προλάβει το R1 να ολοκληρωθεί, συνεχίσουν να έρχονται rpc με προτεραιότητες μεγαλύτερες του 1, τότε το R1 θα παραμείνει κολλημένο, καθώς ο χρονοδρομολογητής δεν θα του δώσει ποτέ την άδεια να χρησιμοποιείσει το κανάλι. Το πρόβλημα που δημιουργείται τότε ονομάζεται λιμοκτονία (starvation).

Συγκεκριμένα, starvation ονομάζεται το φαινόμενο κατά το οποίο μία διεργασία, στην περίπτωση της παρούσας διπλωματικής μία rpc κλήση, που είναι έτοιμη να εκτελεστεί και

έχει χαμηλή προτεραιότητα, περιμένει συνεχώς τη σειρά της να εκτελεστεί λόγω της συνεχούς άφιξης διεργασιών μεγαλύτερων προτεραιοτήτων.



Εικόνα 21: Starvation στην priority based δρομολόγηση

Η λιμοκτονία είναι ένα σοβαρό πρόβλημα, καθώς σε ένα σύστημα θα έπρεπε να ικανοποιούνται όλα τα αιτήματα, και να μην υπάρχουν rpc calls που πρέπει να εκτελεστούν, αλλά παραμένουν συνεχώς αδρανή περιμένοντας τη σειρά τους. Θα πρέπει λοιπόν να υπάρχει κάποιος μηχανισμός που να εξασφαλίζει ότι ακόμα και αν ένα rpc έχει χαμηλή προτεραιότητα, κάποια στιγμη θα ικανοποιηθεί.

Επίλυση με aging

Ένας μηχανισμός επίλυσης του προβλήματος της λιμοκτονίας είναι το aging. Το aging είναι μία τεχνική κατά την οποία η προτεραιότητα ενός rpc αυξάνεται σταδιακά όσο το rpc αυτό περιμένει αδρανές χωρίς να ικανοποιείται. Με αυτόν τον τρόπο, μία κλήση χαμηλής προτεραιότητας, όσο περνάει ο χρόνος μετατρέπεται σε κλήση υψηλής προτεραιότητας, αυξάνοντας έτσι την πιθανότητά της να επιλαγεί από το scheduler και να εκτελεστεί.

Για παράδειγμα, έστω ότι μία κλήση έχει προτεραιότητα ίση με 1 τη στιγμή που δημιουργείται, και επομένως μάλλον υπάρχουν πολλές κλήσεις μεγαλύτερων προτεραιοτήτων που θα εκτελεστούν πρώτα, με κίνδυνο λιμοκτονίας. Έστω τώρα, ότι η προτεραιότητα της κλήσης αυτής αυξάνεται κατά μία μονάδα είτε κάθε κάποιο καθορισμένο χρονικό διάστημα (πχ. 5 ms) είτε κάθε κάποιο αριθμό rpc κλήσεων που ολοκληρώνονται (πχ 2 rpc calls). Όσο η κλήση αυτή δεν ολοκληρώνεται, τόσο θα αυξάνεται η προτεραιότητά της, με αποτέλεσμα κάποια στιγμή να αποκτήσει την υψηλότερη προτεραιότητα και ο χρονοδρομολογητής να αναγκαστεί να της δώσει άδεια να εκτελεστεί. Με αυτόν τον τρόπο, κάθε κλήση χαμηλής προτεραιότητας κάποια στιγμή θα ολοκληρωθεί, και θα αποφευχθούν έτσι φαινόμενα λιμοκτονίας.

Κάτι το οποίο πρέπει να μελετηθεί κατά τη χρήση του μηχανισμού aging, είναι ο καθορισμός του χρονικού διαστήματος που θα μεσολαβεί ανάμεσα σε κάθε αύξηση της προτεραιότητας μιας κλήσης. Το χρονικό αυτό διάστημα θα μπορούσε είτε να είναι σταθερό, είτε να αλλάζει δυναμικά αναλόγως του χρόνου που έχει ήδη παραμείνει αδρανής

η κλήση. Για παράδειγμα, θα μπορούσε να υπάρχει αύξηση της προτεραιότητας μιας κλήσης είτε κάθε 5 ms που μένει αδρανής, είτε κάθε 5 ms που μένει αδρανής όσο η προτεραιότητά της είναι μικρότερη από 6 και κάθε 2 ms αδράνειας όσο η προτεραιότητά της είναι από 6 και πάνω. Αντίστοιχα, ως μέτρο αύξησης προτεραιότητας μιας κλήσης θα μπορούσε να χρησιμοποιηθεί ο αριθμός των rpc που ολοκληρώνονται όσο η κλήση παραμένει αδρανής. Για παράδειγμα, θα μπορούσε να υπάρχει αύξηση της προτεραιότητας μιας κλήσης για κάθε 4 rpc που ολοκληρώνονται όσο είναι αδρανής.

Επίλυση με time-out

Ένας άλλος μηχανισμός αντιμετώπισης του starvation προβλήματος θα ήταν η χρήση κάποιου χρόνου timeout, ο οποίος θα είναι και ο μέγιστος χρόνος που μπορεί να μείνει μία κλήση αδρανής πριν εκτελεστεί. Εάν μια κλήση, δηλαδή, είναι χαμηλής προτεραιότητας, τότε δεν θα υπάρχει κίνδυνος συνεχούς αναμονής της, καθώς θα πρέπει αναγκαστικά να εκτελεστεί μόλις περάσει ένα συγκεκριμένο χρονικό διάστημα αδράνειας, είτε διακόπτοντας την rpc κλήση που εκτελείται εκείνη τη στιγμή, είτε πολυπλέκοντας τα πακέτα της στο gRPC κανάλι.

Θα έπρεπε και πάλι να γίνει κάποια μελέτη σχετικά με τον μέγιστο χρόνο αναμονής μίας κλήσης. Ο χρόνος αυτός θα μπορούσε να είναι σταθερός για όλες τις κλήσεις, θα ήταν ωστόσο πιο λογικό για την ιεράρχηση προτεραιοτήτων, να είναι αντιστρόφως ανάλογος της προτεραιότητας του κάθε rpc. Ένα rpc, δηλαδή, μεγάλης προτεραιότητας θα πρέπει να έχει αντοχή να παραμείνει αδρανές περισσότερη ώρα από ένα rpc υψηλότερης προτεραιότητας. Θα μπορούσε, επίσης, ο time-out χρόνος να αλλάζει δυναμικά ανάλογα με τον αριθμό των κλήσεων που είναι έτοιμες κάθε στιγμή να εκτελεστούν. Για παράδειγμα, όσο περισσότερα rpc θέλουν να εκτελεστούν κάθε στιγμή, τόσο μικρότερο να είναι και το time-out time για κάθε ένα από αυτά. Ο χρόνος time-out για κάθε rpc i θα μπορούσε να υπολογίζεται κάθε στιγμή από μία συνάρτηση της μορφής:

$$time\_out\_time(i) = \frac{c}{priority(i) * n}$$

όπου c = μία σταθερά, $priority(i)$ = η προτεραιότητα του rpc i και n = ο συνολικός αριθμός rpc που είναι έτοιμα να εκτελεστούν τη συγκεκριμένη χρονική στιγμή.

Υλοποίηση χρονοδρομολογητή με time-out

Προκειμένου να γίνει μία σύντομη μελέτη του μηχανισμού αποφυγής starvation, πραγματοποιήθηκε μία υλοποίηση χρονοδρομολογητή με επίλυση της λιμοκτονίας με χρήση time-out. Συγκεκριμένα, επεκτάθηκε η πρώτη υλοποίηση με threads σε C++, ώστε κάθε νήμα να ξεκινάει ή να συνεχίζει αυτόματα το rpc που πρέπει να εκτελέσει αφού περάσει κάποιος χρόνος αδράνειας. Το time-out time ορίστηκε να είναι ένας σταθερός αριθμός για κάθε προτεραιότητα, σύμφωνα με τον Πίνακα 11. Οι χρόνοι αυτοί ορίστηκαν με βάση τους μέσους χρόνους εκτέλεσης της κάθε προτεραιότητας, οι οποίοι μετρήθηκαν στα πειράματα που έγιναν για τη συγκεκριμένη υλοποίηση.

Priority	1	2	3	4	5	6	7	8	9	10
Time-out time (ms)	2000	1950	1900	1800	1700	1600	1500	1300	1000	500

Πίνακας 11: Time-out χρόνοι για κάθε προτεραιότητα

Ο ψευδοκώδικας των threads για τη συγκεκριμένη υλοποίηση φαίνεται στον Πίνακα 12.

Ψευδοκώδικας threads

`for rpc_packets_to_send:`
`  while(sleep == true):`
`  if waiting_time > time_out_time[priority]:`
`  break`
`  send_rpc_packet()`
 	
`unsubscribe_queue.push(id)`

Πίνακας 12: Ψευδοκώδικας threads με προσθήκη time-out μηχανισμού

Στην περίπτωση του παραδείγματος της Εικόνας 16, αρχικά δημιουργείται ένα νήμα για να εκτελέσει ένα rpc με προτεραιότητα 1, το οποίο, εφόσον έχει τη μέγιστη προτεραιότητα μέχρι στιγμής, ξεκινάει να αποστέλει τα πακέτα του. Με την άφιξη του δεύτερου rpc με προτεραιότητα 3 και τη δημιουργία του thread_2, το thread_1 διακόπτει την εκτέλεσή του, περιμένοντας την ολοκλήρωση του thread_2 και στη συνέχεια και του thread_3, αφού και τα δύο νήματα αυτά εξυπηρετούν κλήσεις μεγαλύτερων προτεραιοτήτων. Εάν κατά την αναμονή του thread_1 περάσει ο time-out χρόνος των 2000 ms, τότε το thread_1 θα συνεχίσει την εκτέλεση του rpc του, πολυπλέκοντας τα πακέτα του με τα πακέτα του νήματος που χρησιμοποιεί εκείνη τη στιγμή το κανάλι, προκαλώντας καθυστέρηση σε αυτό. Το ίδιο θα συμβεί και με το thread_3: αν ο χρόνος αναμονής για την ολοκλήρωση της κλήσης του thread_2 ξεπεράσει τα 1900 ms, θα ξεκινήσει να στέλνει τα πακέτα του, χωρίς να περιμένει άδεια από τον χρονοδρομολογητή. Επομένως, γίνεται κατανοητό ότι ο μηχανισμός αυτός, παρόλο που εμποδίζει τη δημιουργία starvation και κρατάει του χρόνους αναμονής των νημάτων κάτω από κάποιο καθορισμένο όριο, προκαλεί καθυστέρηση στα υπόλοιπα νήματα που χρησιμοποιούν εκείνη τη στιγμή το κανάλι.

ΑΠΟΤΕΛΕΣΜΑΤΑ

Κλήσεις απομακρυσμένης διαδικασίας

Για τη μελέτη του gRPC framework αλλά και για την δοκιμή της υλοποίησης του χρονοδρομολογητή που περιγράφηκε σε προηγούμενο σημείο, αναπτύχθηκαν κάποιες rpc μέθοδοι, οι οποίες θα περιγραφούν στο συγκεκριμένο κεφάλαιο.

1. SayHello

```
rpc SayHello (HelloRequest) returns (HelloReply) {}
```

Εικόνα 22: Ορισμός μεθόδου SayHello

```
message HelloRequest {
  string name = 1;
}


message HelloReply {
  string message = 1;
}
```

Εικόνα 2313: Ορισμός μηνυμάτων HelloRequest και HelloReply

Η πρώτη rpc μέθοδος που αναπτύχθηκε είναι η SayHello. Η συγκεκριμένη μέθοδος είναι ένα unary rpc, κατά το οποίο ο client στέλνει ένα μήνυμα HelloRequest, και ο server απαντάει με ένα μήνυμα HelloReply.

Το μήνυμα HelloRequest περιέχει ένα string πεδίο name, ενώ το μήνυμα HelloReply περιέχει ένα string πεδίο message. Ουσιαστικά, ο πελάτης στέλνει ένα μήνυμα με το όνομά του, και ο server απαντάει με ένα μήνυμα Hello, {client's name}.

2. ComputeMean

```
rpc ComputeMean (stream IntNumber) returns (FloatNumber) {}
```

Εικόνα 24: Ορισμός μεθόδου ComputeMean

```
message IntNumber {
  int32 value = 1;
}


message FloatNumber {
  float value = 1;
}
```

Εικόνα 2514: Ορισμός μηνυμάτων IntNumber και FloatNumber

Η δεύτερη rpc μέθοδος είναι η ComputeMean, η οποία είναι ένα client-side streaming rpc. Ο client, δηλαδή, στέλνει πολλά πακέτα, κάθε ένα από τα οποία περιέχει ένα IntNumber μήνυμα, και ο server, αφού ο client ολοκληρώσει την αποστολή των πακέτων του, απαντάει με ένα μήνυμα FloatNumber.

Το μήνυμα Int Number περιέχει ένα πεδίο value, τύπου int32, ενώ το μήνυμα FloatNumber περιέχει ένα πεδίο value, τύπου float. Ο πελάτης, κατά την κλήση της συγκεκριμένης μεθόδου, στέλνει πλήθος από ακέραιους αριθμούς στον διακομιστή, ο οποίος, μόλις ολοκληρωθεί η αποστολή των πακέτων, απαντάει με έναν δεκαδικό αριθμό που είναι ο μέσος όρος των αριθμών που έλαβε.

3. ComputeMeanRepeated

```
rpc ComputeMeanRepeated (FloatNumberList) returns (FloatNumber) {}
```

Εικόνα 26: Ορισμός μεθόδου ComputeMeanRepeated

```
message FloatNumberList {
  repeated float value = 1;
}
```

Εικόνα 27: Ορισμός μηνύματος FloatNumberList

Η τρίτη μέθοδος που αναπτύχθηκε είναι η ComputeMeanRepeated, η οποία είναι ένα unary rpc, κατά το οποίο ο client στέλνει ένα μήνυμα FloatNumberList, και ο server απαντάει με ένα μήνυμα FloatNumber, όπως και προηγουμένως.

Το μήνυμα FloatNumberList περιέχει ένα repeated πεδίο value, τύπου float. Το keyword repeated υποδηλώνει ότι το συγκεκριμένο πεδίο μπορεί να επαναλαμβάνεται πολλές φορές, αντιστοιχεί δηλαδή σε ένα πίνακα από τιμές value. Ο πελάτης αυτή τη φορά στέλνει ένα πακέτο που περιέχει πολλούς δεκαδικούς αριθμούς, και ο διακομιστής απαντάει με έναν δεκαδιμό αριθμό που είναι ο μέσος όρος των αριθμών που έλαβε στο FloatNumberList μήνυμα. Σε περίπτωση που το μήνυμα FloatNumberList είναι πολύ μεγάλο, το HTTP/2 πλαίσιο μπορεί να «θρυμματιστεί» σε περισσότερα του ενός TCP πακέτα.

4. SendLongString

```
rpc SendLongString (LongString) returns (LongString) {}
```

Εικόνα 28: Ορισμός μεθόδου SendLongString

```
message LongString {
  string str = 1;
}
```

Εικόνα 29: Ορισμός μηνύματος LongString

Η τέταρτη μέθοδος που αναπτύχθηκε είναι η SendLongString, που είναι ένα unary rpc, κατά το οποίο ο πελάτης στέλνει ένα μήνυμα LongString και ο διακομιστής απαντάει με ένα μήνυμα επίσης LongString που περιέχει το ίδιο string με αυτό που έλαβε. Το μήνυμα LongString περιέχει ένα string πεδίο str. Η μέθοδος αυτή αναπτύχθηκε

προκειμένου να υπάρχει μία χρονική καθυστέρηση κατά την εκτέλεσή της, λόγω του μεγάλου μεγέθους του μηνύματος LongString.

5. ComputeMeanRepeatedOrSendLongString

```
rpc ComputeMeanRepeatedOrSendLongString (FloatNumberListOrLongString) returns (FloatOrLongString) {}
```

Εικόνα 30: Ορισμός μεθόδου ComputeMeanRepeatedOrSendLongString

```
message FloatOrLongString {
  oneof type {
    FloatNumber float_number_type = 1;
    LongString long_string_type = 2;
  }
}

message FloatNumberListOrLongString {
  oneof type {
    FloatNumberList float_number_list_type = 1;
    LongString long_string_type = 2;
  }
}
```

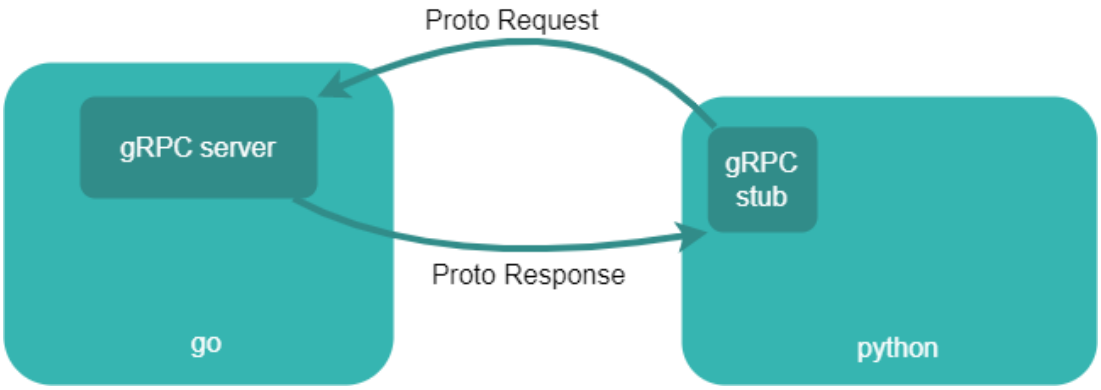
Εικόνα 31: Ορισμός μηνυμάτων FloatOrLongString και FloatNumberListOrLongString

Η τελευταία μέθοδος που αναπτύχθηκε είναι η ComputeMeanRepeatedOrSendLongString, η οποία είναι ένα client-side streaming rpc, το οποίο ουσιαστικά είναι ο συνδυασμός των μεθόδων ComputeMeanRepeated και SendLongString. Ο πελάτης στέλνει ένα μήνυμα FloatNumberListOrLongString και ο διακομιστής απαντάει με ένα μήνυμα FloatOrLongString.

Το μήνυμα FloatOrLongString περιέχει αυστηρά ένα μόνο εκ των πεδιών float_number_type, τύπου FloatNumber, και long_string_type, τύπου LongString, γεγονός που υποδηλώνεται από το keyword oneof. Το μήνυμα FloatNumberListOrLongString περιέχει ένα μόνο εκ των πεδιών float_number_list_type, τύπου FloatNumberList, και long_string_type, τύπου LongString. Κατά την κλήση της συγκεκριμένης μεθόδου, ο client στέλνει στον server ένα πακέτο που περιέχει είτε ένα μήνυμα FloatNumberList με έναν πίνακα από δεκαδικούς αριθμούς, είτε ένα μήνυμα LongString. Ο server στην πρώτη περίπτωση απαντάει με ένα μήνυμα FloatNumber με τον δεκαδικό μέσο όρο των αριθμών που έλαβε, ενώ στη δεύτερη με ένα μήνυμα LongString.

Wireshark

Με τη χρήση του εργαλείου Wireshark έγινε μία αρχική μελέτη της ανταλλαγής πακέτων μεταξύ client και server κατά την κλήση ενός ή και περισσότερων rpc. Στα παρακάτω πειράματα χρησιμοποιήθηκε η αρχιτεκτονική συστήματος της Εικόνας 32, δηλαδή ένας server υλοποιημένος σε Go και ένας client υλοποιημένος σε Python, και με τους δύο να βρίσκονται σε localhost.



Εικόνα 32: Αρχική αρχιτεκτονική client – server

Αρχικά, δοκιμάστηκε η κλήση της μεθόδου SayHello, για την παρακολούθηση των πακέτων που ανταλλάσσονται σε ένα unary rpc. Το αποτέλεσμα της παρακολούθησης της εκτέλεσης της κλήσης με το Wireshark φαίνεται στην Εικόνα 33.

Source	Destination	Protocol	Length	Info
127.0.0.1	127.0.0.1	TCP	56	22215 → 50051 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
127.0.0.1	127.0.0.1	TCP	56	50051 → 22215 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
127.0.0.1	127.0.0.1	TCP	44	22215 → 50051 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	126	Magic, SETTINGS[0], WINDOW_UPDATE[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 22215 [ACK] Seq=1 Ack=83 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	59	SETTINGS[0]
127.0.0.1	127.0.0.1	TCP	44	22215 → 50051 [ACK] Seq=83 Ack=16 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	53	SETTINGS[0]
127.0.0.1	127.0.0.1	TCP	44	22215 → 50051 [ACK] Seq=83 Ack=25 Win=2619648 Len=0
127.0.0.1	127.0.0.1	GRPC	326	SETTINGS[0], HEADERS[1]: POST /helloworld.Greeter/SayHello, WINDOW_UPDATE[1], DATA[1] (GRPC) (PROTOBUF) helloworld.HelloRequest, WINDOW_UPDATE[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 22215 [ACK] Seq=25 Ack=365 Win=2619392 Len=0
127.0.0.1	127.0.0.1	HTTP2	74	WINDOW_UPDATE[0], PING[0]
127.0.0.1	127.0.0.1	TCP	44	22215 → 50051 [ACK] Seq=365 Ack=55 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	61	PING[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 22215 [ACK] Seq=55 Ack=382 Win=2619392 Len=0
127.0.0.1	127.0.0.1	GRPC	125	HEADERS[1]: 200 OK, DATA[1] (GRPC) (PROTOBUF) helloworld.HelloReply, HEADERS[1]
127.0.0.1	127.0.0.1	TCP	44	22215 → 50051 [ACK] Seq=382 Ack=136 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	61	PING[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 22215 [ACK] Seq=136 Ack=399 Win=2619392 Len=0
127.0.0.1	127.0.0.1	HTTP2	61	PING[0]
127.0.0.1	127.0.0.1	TCP	44	22215 → 50051 [ACK] Seq=399 Ack=153 Win=2619648 Len=0
127.0.0.1	127.0.0.1	TCP	44	22215 → 50051 [FIN, ACK] Seq=399 Ack=153 Win=2619648 Len=0
127.0.0.1	127.0.0.1	TCP	44	50051 → 22215 [ACK] Seq=153 Ack=400 Win=2619392 Len=0
127.0.0.1	127.0.0.1	TCP	44	50051 → 22215 [FIN, ACK] Seq=153 Ack=400 Win=2619392 Len=0
127.0.0.1	127.0.0.1	TCP	44	22215 → 50051 [ACK] Seq=400 Ack=154 Win=2619648 Len=0

Εικόνα 33: Παρακολούθηση της κλήσης SayHello στο Wireshark

Παρατηρείται ότι τα πρώτα τρία πακέτα αντιστοιχούν στην τριμερή χειραψία μεταξύ των client και server για την εδραίωση της TCP σύνδεσης που χρησιμοποιεί το gRPC. Ο πελάτης, που είναι αυτός που ξεκινάει την κλήση, στέλνει ένα πακέτο [SYN] στον διακομιστή ο οποίος απαντάει με ένα πακέτο [SYN, ACK]. Ο πελάτης στη συνέχεια στέλνει ένα πακέτο [ACK] και από εκείνη τη στιγμή έχει εδραιωθεί η TCP σύνδεση στην οποία θα πραγματοποιηθεί η ανταλλαγή πακέτων. Από την πηγή και τον προορισμό των πακέτων φαίνεται ότι και ο πελάτης και ο διακομιστής βρίσκονται localhost (127.0.0.1), με τον πελάτη να ζει στη θύρα 22215 και τον διακομιστή στη θύρα 50051.

Στη συνέχεια, στέλνονται κάποια HTTP/2 πλαίσια SETTINGS που αφορούν το root stream, τα οποία μεταφέρουν πληροφορίες και ρυθμίσεις σχετικά με την επικοινωνία μεταξύ

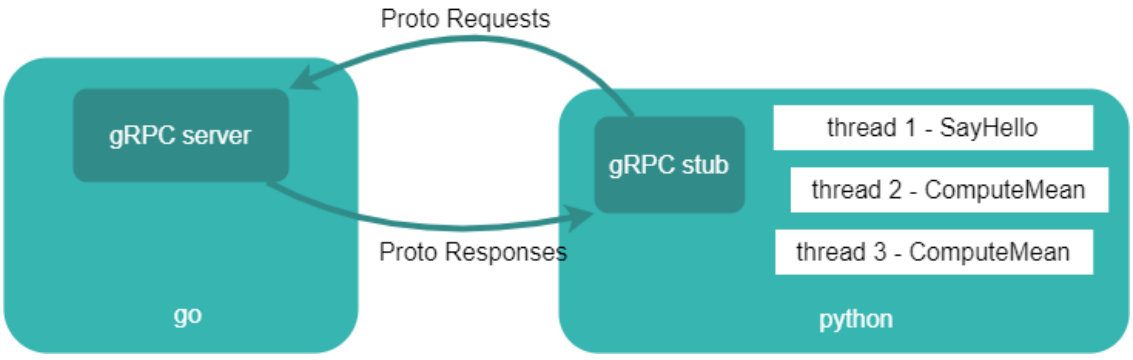
πελάτη και διακομιστή, ενώ με την αποστολή ενός WINDOW_UPDATE πλαισίου ο πελάτης ενημερώνει σχετικά με το αρχικό παράθυρό του, δηλαδή τον αριθμό των bytes από data που επιτρέπεται να του στείλει ο διακομιστής. Ο πελάτης έπειτα στέλνει τα πλαίσια που αντιστοιχούν στο stream 1, δηλαδή στο stream της rpc κλήσης SayHello. Αρχικά στέλνει ένα HEADERS πλαίσιο που ενημερώνει τον διακομιστή σχετικά με το ποιά μέθοδος θα κληθεί, και ύστερα ένα DATA πλαίσιο που περιέχει το μήνυμα HelloRequest. Στέλνει, ακόμα ένα WINDOW_UPDATE πλαίσιο, τόσο για το stream 1 όσο και για το stream 0, που ενημερώνει για την μείωση του παραθύρου του λόγω της αποστολής ενός DATA πλαισίου.

Όταν ο server λάβει το data μήνυμα, απαντάει με ένα WINDOW_UPDATE πλαίσιο, ώστε να ενημερώσει τον πελάτη ότι μπορεί να αυξήσει ξανά το παράθυρό του. Έπειτα στέλνει ένα HEADERS πλαίσιο, που στέλνει επιβεβαίωση για την κατάσταση της κλήσης (200 OK), ένα DATA πλαίσιο με το HelloReply μήνυμα και τέλος ακόμα ένα HEADERS πλαίσιο που σηματοδοτεί τη λήξη του stream.

Παρατηρείται ότι ενδοιάμεσα από τα πλαίσια που αφορούν την κλήση rpc, στέλνονται PING πλαίσια, και από τις δύο πλευρές, τα οποία είναι ένας μηχανισμός μέτρησης του ελάχιστου round-trip χρόνου του αποστολέα. Επίσης, μετά από κάθε TCP πακέτο που περιέχει HTTP/2 πλαίσια, στέλνεται ένα ACK TCP πακέτο από τον παραλήπτη, ως επιβεβαίωση λήψης του προηγούμενου πακέτου. Τέλος, να σημειωθεί ότι σε ένα TCP πακέτο μπορούν να υπάρχουν πολλά HTTP/2 πλαίσια.

Μετά την ολοκλήρωση της gRPC κλήσης, ανταλλάσσονται τέσσερα πακέτα FIN και ACK, τα οποία ξεκινούν από τον πελάτη, ώστε να τερματιστεί η TCP σύνδεση.

Στο επόμενο πείραμα δοκιμάστηκε η χρήση παράλληλων νημάτων στην πλευρά του πελάτη, τα οποία εκτελούν ανεξάρτητα μεταξύ τους rpc. Συγκεκριμένα ένα νήμα εκτελεί μία SayHello rpc κλήση, ενώ τα άλλα δύο νήματα εκτελούν μία client-side streaming ComputeMean κλήση το κάθε ένα, με διαφορετικούς όμως αριθμούς, όπως φαίνεται στην Εικόνα 34. Τα αποτελέσματα της συγκεκριμένης παρακολούθησης φαίνονται στην Εικόνα 35.



Εικόνα 3415: Αρχιτεκτονική client multithreading

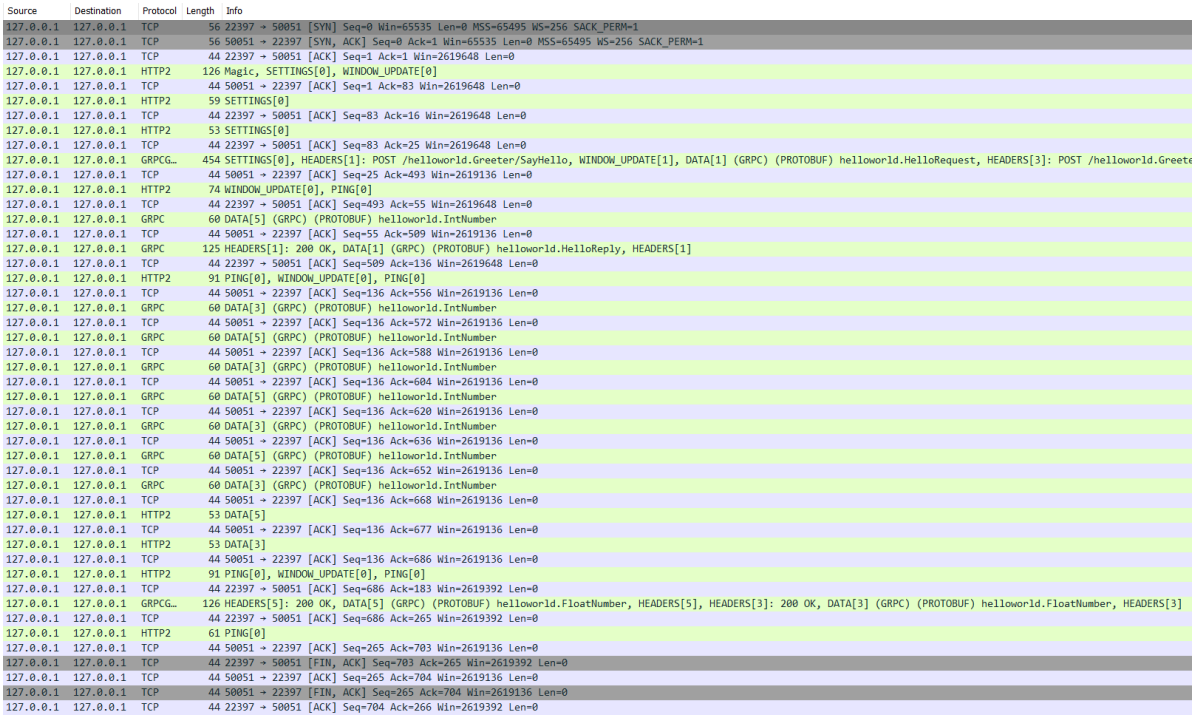
Παρατηρείται ότι και πάλι στέλνονται τα αρχικά TCP πακέτα για την εδραίωση της TCP σύνδεσης καθώς και κάποια αρχικά SETTINGS πλαίσια. Στη συνέχεια ο πελάτης στέλνει στο ίδιο πακέτο:

- HEADERS, WINDOW_UPDATE πλαίσια καθώς και το DATA πλαίσιο που περιέχει το HelloRequest μήνυμα για το stream 1

- HEADERS, WINDOW_UPDATE πλαίσια καθώς και το DATA πλαίσιο που περιέχει το IntNumber μήνυμα με τον πρώτο αριθμό για το stream 3
- HEADERS, WINDOW_UPDATE πλαίσια καθώς και το DATA πλαίσιο που περιέχει το IntNumber μήνυμα με τον πρώτο αριθμό για το stream 5
- Ένα WINDOW_UPDATE πλαίσιο για το root stream 0.

Ο διακομιστής στη συνέχεια απαντάει με το κατάλληλο HelloReply μήνυμα και τα HEADERS πλαίσια που το συνοδεύουν για το stream 1, ενώ ο πελάτης συνεχίζει να στέλνει τους υπόλοιπους αριθμούς για τα ComputeMean rpc των stream 3 και 5. Τα DATA πλαίσια που αφορούν τα streams 3 και 5 και αντιστοιχούν στους αριθμούς των ComputeMean rpc που αποστέλλονται από τα δύο διαφορετικά νήματα του πελάτη, στέλνονται με round-robin τρόπο από τον client. Η αποστολή των IntNumber μηνυμάτων των δύο streams ολοκληρώνεται σχεδόν την ίδια στιγμή, εφόσον τα δύο νήματα στέλνουν τον ίδιο αριθμό πακέτων. Ο διακομιστής απαντάει με ένα κοινό TCP πακέτο που περιέχει δύο DATA πλαίσια με το FloatNumber μήνυμα του μέσου όρου, μαζί με τα αντίστοιχα HEADERS πλαίσια του κάθε stream. Τέλος ανταλλάσσονται και πάλι κάποια πακέτα για τον τερματισμό της TCP σύνδεσης.

Παρατηρείται ότι και σε αυτή την περίπτωση ανταλλάσσονται PING πλαίσια καθώς και ACK πακέτα. Παρατηρείται επίσης ότι σε ένα TCP πακέτο μπορούν να συνυπάρχουν πλαίσια από πολλά διαφορετικά streams. Να σημειωθεί επίσης ότι τα streams 1, 3, 5 έχουν λάβει τα συγκεκριμένα αναγνωριστικά, καθώς τα streams που δημιουργούνται από τον πελάτη πρέπει να έχουν περιττό αριθμό ως αναγνωριστικό.



Source	Destination	Protocol	Length	Info
127.0.0.1	127.0.0.1	TCP	56	22397 → 50051 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
127.0.0.1	127.0.0.1	TCP	56	50051 → 22397 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
127.0.0.1	127.0.0.1	TCP	44	22397 → 50051 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	126	Magic, SETTINGS[0], WINDOW_UPDATE[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=1 Ack=83 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	59	SETTINGS[0]
127.0.0.1	127.0.0.1	TCP	44	22397 → 50051 [ACK] Seq=83 Ack=16 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	53	SETTINGS[0]
127.0.0.1	127.0.0.1	TCP	44	22397 → 50051 [ACK] Seq=83 Ack=25 Win=2619648 Len=0
127.0.0.1	127.0.0.1	GRPCG…	454	SETTINGS[0], HEADERS[1]: POST /helloworld.Greeter/SayHello, WINDOW_UPDATE[1], DATA[1] (GRPC) (PROTOBUF) helloworld.HelloRequest, HEADERS[3]: POST /helloworld.Greete
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=25 Ack=493 Win=2619136 Len=0
127.0.0.1	127.0.0.1	HTTP2	74	WINDOW_UPDATE[0], PING[0]
127.0.0.1	127.0.0.1	TCP	44	22397 → 50051 [ACK] Seq=493 Ack=55 Win=2619648 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[5] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=55 Ack=509 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	125	HEADERS[1]: 200 OK, DATA[1] (GRPC) (PROTOBUF) helloworld.HelloReply, HEADERS[1]
127.0.0.1	127.0.0.1	TCP	44	22397 → 50051 [ACK] Seq=509 Ack=136 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	91	PING[0], WINDOW_UPDATE[0], PING[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=136 Ack=556 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[3] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=136 Ack=572 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[5] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=136 Ack=588 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[3] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=136 Ack=604 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[5] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=136 Ack=620 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[3] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=136 Ack=636 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[5] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=136 Ack=652 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[3] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=136 Ack=668 Win=2619136 Len=0
127.0.0.1	127.0.0.1	HTTP2	53	DATA[5]
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=136 Ack=677 Win=2619136 Len=0
127.0.0.1	127.0.0.1	HTTP2	53	DATA[3]
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=136 Ack=686 Win=2619136 Len=0
127.0.0.1	127.0.0.1	HTTP2	91	PING[0], WINDOW_UPDATE[0], PING[0]
127.0.0.1	127.0.0.1	TCP	44	22397 → 50051 [ACK] Seq=686 Ack=183 Win=2619392 Len=0
127.0.0.1	127.0.0.1	GRPCG…	126	HEADERS[5]: 200 OK, DATA[5] (GRPC) (PROTOBUF) helloworld.FloatNumber, HEADERS[5], HEADERS[3]: 200 OK, DATA[3] (GRPC) (PROTOBUF) helloworld.FloatNumber, HEADERS[3]
127.0.0.1	127.0.0.1	TCP	44	22397 → 50051 [ACK] Seq=686 Ack=265 Win=2619392 Len=0
127.0.0.1	127.0.0.1	HTTP2	61	PING[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=265 Ack=703 Win=2619136 Len=0
127.0.0.1	127.0.0.1	TCP	44	22397 → 50051 [FIN, ACK] Seq=703 Ack=265 Win=2619392 Len=0
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [ACK] Seq=265 Ack=704 Win=2619136 Len=0
127.0.0.1	127.0.0.1	TCP	44	50051 → 22397 [FIN, ACK] Seq=265 Ack=704 Win=2619136 Len=0
127.0.0.1	127.0.0.1	TCP	44	22397 → 50051 [ACK] Seq=704 Ack=266 Win=2619392 Len=0

Εικόνα 35: Παρακολούθηση των κλήσεων από τον multithreading client στο Wireshark

Στο επόμενο πείραμα δοκιμάστηκε η παράμετρος maxConcurrentStreams στον Go server. Η παράμετρος αυτή καθορίζει το μέγιστο αριθμό παράλληλων streams στην TCP σύνδεση, ουσιαστικά δηλαδή το μέγιστο αριθμό διαφορετικών stream που μπορούν να συνυπάρχουν στα πλαίσια ενός TCP πακέτου (εξαιρείται το root stream 0). Ο τρόπος που ρυθμίζεται αυτή

η παράμετρος στον διακομιστή φαίνεται στην Εικόνα 36, ενώ τα αποτελέσματα του πειράματος στο Wireshark φαίνονται στην Εικόνα 37.

```
// set maximum number of concurrent streams in tcp connection
opts := []grpc.ServerOption{grpc.MaxConcurrentStreams(1)}

s := grpc.NewServer(opts...)
```

Εικόνα 36: Ορισμός maxConcurrentStreams = 1 στον Go client

Χρησιμοποιήθηκαν και πάλι τρία threads στον πελάτη που εκτελούν μία SayHello κλήση και δύο ComputeMean κλήσεις. Τα αποτελέσματα του πειράματος αυτού μοιάζουν αρκετά με τα αποτελέσματα του προηγούμενου. Συγκεκριμένα:

- Ανταλλάσσονται πακέτα στην αρχή και στο τέλος της σύνδεσης για την εδραίωση και τον τερματισμό του connection αντίστοιχα.
- Αρχικά στέλνονται κάποια SETTINGS πλαίσια από τον πελάτη.
- Τα αναγνωριστικά των τριών streams είναι οι συνεχόμενοι περιττοί αριθμοί 1, 3 και 5.
- Τα τρία νήματα στέλνουν αρχικά HEADERS και DATA πλαίσια ενώ ο διακομιστής απαντάει με ένα DATA και δύο HEADERS πλαίσια για κάθε stream.
- Ανταλλάσσονται PING πλαίσια και ACK πακέτα καθ' όλη τη διάρκεια της σύνδεσης.

Source	Destination	Protocol	Length	Info
127.0.0.1	127.0.0.1	TCP	56	12497 → 50051 [SYN] Seq=0 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
127.0.0.1	127.0.0.1	TCP	56	50051 → 12497 [SYN, ACK] Seq=0 Ack=1 Win=65535 Len=0 MSS=65495 WS=256 SACK_PERM=1
127.0.0.1	127.0.0.1	TCP	44	12497 → 50051 [ACK] Seq=1 Ack=1 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	126	Magic, SETTINGS[0], WINDOW_UPDATE[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=1 Ack=83 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	65	SETTINGS[0]
127.0.0.1	127.0.0.1	TCP	44	12497 → 50051 [ACK] Seq=83 Ack=22 Win=2619648 Len=0
127.0.0.1	127.0.0.1	HTTP2	53	SETTINGS[0]
127.0.0.1	127.0.0.1	TCP	44	12497 → 50051 [ACK] Seq=83 Ack=31 Win=2619648 Len=0
127.0.0.1	127.0.0.1	GRPC	326	SETTINGS[0], HEADERS[1]: POST /helloworld.Greeter/SayHello, WINDOW_UPDATE[1], DATA[1] (GRPC) (PROTOBUF) helloworld.HelloRequest, WINDOW_UPDATE[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=31 Ack=365 Win=2619392 Len=0
127.0.0.1	127.0.0.1	HTTP2	74	WINDOW_UPDATE[0], PING[0]
127.0.0.1	127.0.0.1	TCP	44	12497 → 50051 [ACK] Seq=365 Ack=61 Win=2619648 Len=0
127.0.0.1	127.0.0.1	GRPC	125	HEADERS[1]: 200 OK, DATA[1] (GRPC) (PROTOBUF) helloworld.HelloReply, HEADERS[1]
127.0.0.1	127.0.0.1	HTTP2	61	PING[0]
127.0.0.1	127.0.0.1	TCP	44	12497 → 50051 [ACK] Seq=382 Ack=142 Win=2619648 Len=0
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=142 Ack=382 Win=2619392 Len=0
127.0.0.1	127.0.0.1	GRPC	158	HEADERS[3]: POST /helloworld.Greeter/ComputeMean, WINDOW_UPDATE[3], DATA[3] (GRPC) (PROTOBUF) helloworld.IntNumber, WINDOW_UPDATE[0], PING[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=142 Ack=496 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[3] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=142 Ack=512 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[3] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=142 Ack=528 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[3] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=142 Ack=544 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[3] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=142 Ack=560 Win=2619136 Len=0
127.0.0.1	127.0.0.1	HTTP2	53	DATA[3]
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=142 Ack=569 Win=2619136 Len=0
127.0.0.1	127.0.0.1	HTTP2	91	WINDOW_UPDATE[0], PING[0], PING[0]
127.0.0.1	127.0.0.1	TCP	44	12497 → 50051 [ACK] Seq=569 Ack=189 Win=2619392 Len=0
127.0.0.1	127.0.0.1	GRPC	85	HEADERS[3]: 200 OK, DATA[3] (GRPC) (PROTOBUF) helloworld.FloatNumber, HEADERS[3]
127.0.0.1	127.0.0.1	TCP	44	12497 → 50051 [ACK] Seq=569 Ack=230 Win=2619392 Len=0
127.0.0.1	127.0.0.1	HTTP2	61	PING[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=230 Ack=586 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	101	HEADERS[5]: POST /helloworld.Greeter/ComputeMean, WINDOW_UPDATE[5], DATA[5] (GRPC), WINDOW_UPDATE[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=230 Ack=643 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[5] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=230 Ack=659 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[5] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=230 Ack=675 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[5] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=230 Ack=691 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	60	DATA[5] (GRPC) (PROTOBUF) helloworld.IntNumber
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=230 Ack=707 Win=2619136 Len=0
127.0.0.1	127.0.0.1	HTTP2	53	DATA[5]
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=230 Ack=716 Win=2619136 Len=0
127.0.0.1	127.0.0.1	GRPC	115	WINDOW_UPDATE[0], PING[0], HEADERS[5]: 200 OK, DATA[5] (GRPC) (PROTOBUF) helloworld.FloatNumber, HEADERS[5]
127.0.0.1	127.0.0.1	TCP	44	12497 → 50051 [ACK] Seq=716 Ack=301 Win=2619392 Len=0
127.0.0.1	127.0.0.1	HTTP2	74	PING[0], WINDOW_UPDATE[0]
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=301 Ack=746 Win=2619136 Len=0
127.0.0.1	127.0.0.1	TCP	44	12497 → 50051 [FIN, ACK] Seq=746 Ack=301 Win=2619392 Len=0
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [ACK] Seq=301 Ack=747 Win=2619136 Len=0
127.0.0.1	127.0.0.1	TCP	44	50051 → 12497 [FIN, ACK] Seq=301 Ack=747 Win=2619136 Len=0
127.0.0.1	127.0.0.1	TCP	44	12497 → 50051 [ACK] Seq=747 Ack=302 Win=2619392 Len=0

Εικόνα 3716: Παρακολούθηση των κλήσεων από τον multithreading client με maxConcurrentStreams = 1 στο Wireshark

Η διαφορά που παρατηρείται σε σχέση με προηγουμένως, είναι ότι σε κάθε TCP πακέτο υπάρχουν πλαίσια μόνο ενός stream από τα 1, 3, 5 ενώ μπορούν να υπάρχουν και πλαίσια του root stream 0. Ως αποτέλεσμα, ο πελάτης στέλνει σε τρία διαφορετικά πακέτα τα αρχικά HEADERS και DATA πλαίσια των τριών streams, ενώ ο διακομιστής απαντάει επίσης με τρία διαφορετικά πακέτα. Η διαφορά αυτή οφείλεται στην παράμετρο maxConcurrentStreams και στη ρύθμισή της να ισούται με 1.

Ghz

Τα πειράματα που έγιναν με τη χρήση του εργαλείου ghz, είχαν ως στόχο τη μέτρηση της επίδοσης του gRPC framework. Χρησιμοποιήθηκε ένας server υλοποιημένος σε Go, και η rpc μέθοδος ComputeMeanRepeatedOrSendLongString, η οποία δρα ως ComputeMeanRepeated είτε ως SendLongString rpc κλήση, αναλόγως των δεδομένων της, εάν δηλαδή στέλνεται μήνυμα FloatNumberList ή LongString. Συγκεκριμένα, έγιναν πολλές εκτελέσεις της συγκεκριμένης rpc μεθόδου, προκειμένου να προκύψουν συμπεράσματα σχετικά με το πως μεταβάλλονται ορισμένες μετρικές, συναρτήσει κάποιων παραμέτρων που αφορούν την εκτέλεση των κλήσεων.

Τα μεγέθη που μετρήθηκαν είναι τα εξής:

- Το average latency των κλήσεων, δηλαδή ο μέσος χρόνος μετάδοσης μιας κλήσης στο gRPC κανάλι.
- Ο συνολικός χρόνος εκτέλεσης όλων των κλήσεων.
- Ο συνολικός χρόνος εκτέλεσης των κλήσεων προς τον αριθμό των κλήσεων.
- Ο αριθμός των κλήσεων ανά δευτερόλεπτο.

Οι παράμετροι συναρτήσει των οποίων μετρήθηκαν τα παραπάνω μεγέθη είναι οι εξής:

- Ο συνολικός αριθμός των rpc κλήσεων.
- Η rpc μέθοδος και το είδος των δεδομένων στα μηνύματα που στέλνονται από τον πελάτη.
- Ο αριθμός των νημάτων που εκτελούν τις rpc κλήσεις.
- Ο αριθμός των cpu cores που χρησιμοποιούνται από το τοπικό μηχάνημα.

Παρακάτω φαίνονται τα διαγράμματα των μετρικών συναρτήσει των παραμέτρων που περιγράφηκαν.

Στην Εικόνα 38 φαίνεται το average latency, ο αριθμός των requests ανά δευτερόλεπτο, ο συνολικός χρόνος εκτέλεσης των κλήσεων και ο συνολικός χρόνος εκτέλεσης προς τον αριθμό των κλήσεων, για την εκτέλεση 1000 rpc κλήσεων που στέλνουν FloatNumberList μηνύματα που περιέχουν πίνακες 100 float numbers, με χρήση 2 cpu cores, συναρτήσει του concurrency, δηλαδή του αριθμού των νημάτων που χρησιμοποιεί ο πελάτης.

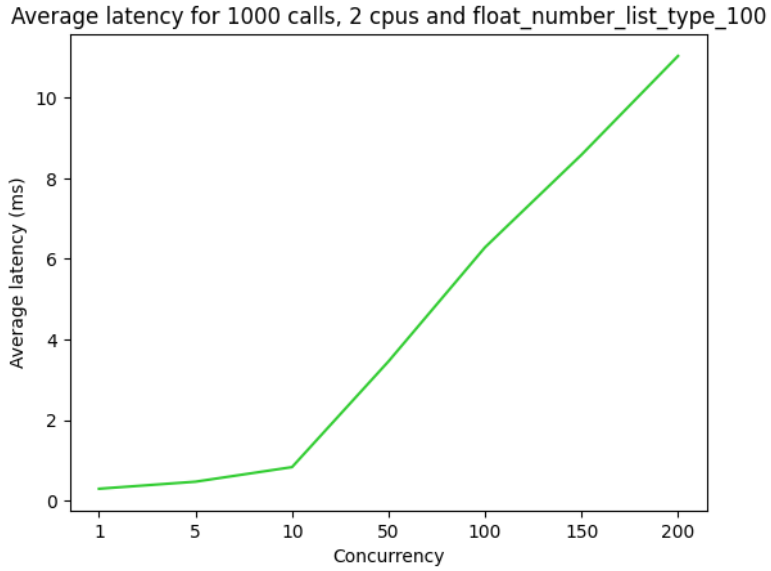
Στο πρώτο διάγραμμα παρατηρείται ότι όσο αυξάνεται το concurrency τόσο περισσότερο αυξάνεται το average latency, κάτι το οποίο συμβαίνει λόγω της πολυπλεξίας των πακέτων στο gRPC κανάλι. Όταν το concurrency ισούται με 1, τότε η εκτέλεση των rpc γίνεται σειριακά, αφού υπάρχει μόνο ένα νήμα. Με την εισαγωγή περισσοτέρων νημάτων δημιουργούνται περισσότερα ταυτόχρονα streams, δηλαδή περισσότερα HTTP/2 πλαίσια σε κάθε TCP πακέτο, με αποτέλεσμα ο μέσος χρόνος μετάδοσης να αυξάνεται.

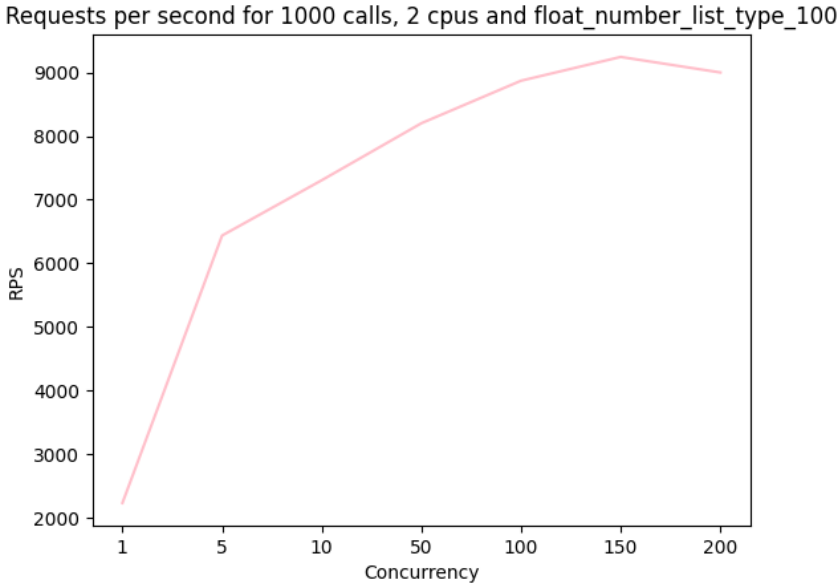
Παρά την αύξηση στο μέσο χρόνο μετάδοσης της κάθε κλήσης, στο διάγραμμα των κλήσεων ανά δευτερόλεπτο φαίνεται ότι η μετρική αυτή αυξάνεται. Παρόλο δηλαδή που το

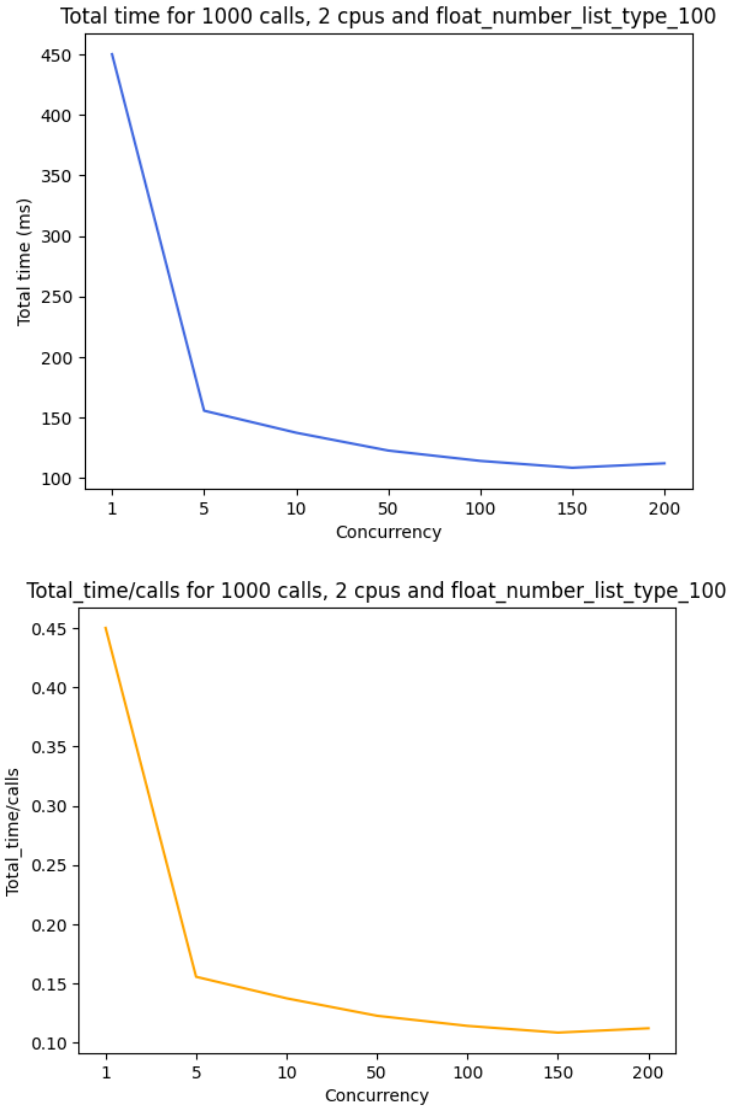
average latency της κάθε μίας κλήσης καθυστερεί λόγω της πολυπλεξίας, η επίδοση του συστήματος αυξάνεται λόγω της αποδοτικότερης χρήσης του καναλιού, εφόσον σε κάθε πακέτο μεταφέρονται πλαίσια περισσότερων streams. Παρατηρείται ωστόσο, ότι μετά τη χρήση των 150 νημάτων επέρχεται κορεσμός, καθώς η μετρική των requests/second μειώνεται για concurrency ίσο με 200. Αυτό μπορεί να οφείλεται στο γεγονός ότι από ένα σημείο και έπειτα η διαχείριση των νημάτων στον πελάτη αρχίζει να γίνεται περίπλοκη.

Η αύξηση της επίδοσης του συστήματος με την αύξηση του concurrency φαίνεται και στο διάγραμμα του συνολικού χρόνου, ο οποίος μειώνεται όσο μεγαλώνει ο αριθμός των νημάτων του πελάτη. Και σε αυτή την περίπτωση, μετά το concurrency των 150 νημάτων παρατηρείται κορεσμός, αφού ο συνολικός χρόνος εκτέλεσης των rpc κλήσεων αυξάνεται κατά τη χρήση 200 νημάτων.

Το διάγραμμα του συνολικού χρόνου εκτέλεσης προς τον συνολικό αριθμό κλήσεων είναι ανάλογο του διαγράμματος του συνολικού χρόνου, εφόσον χρησιμοποιείται σταθερός αριθμός κλήσεων.





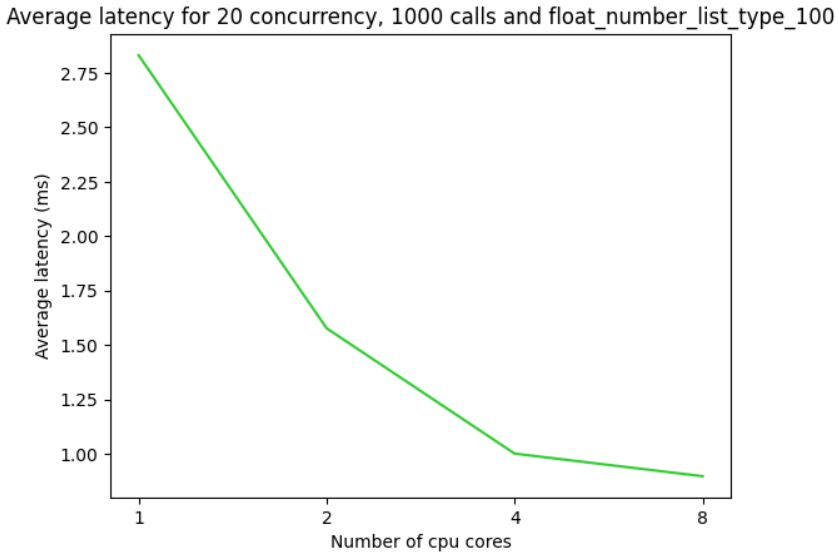


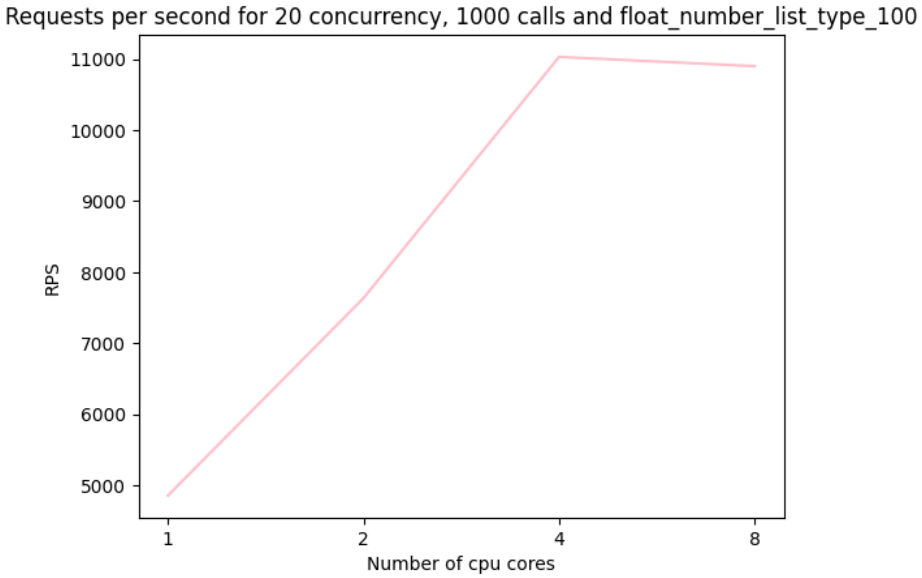
Εικόνα 38: Average latency, rps, total time και total time/calls συναρτήσει του concurrency

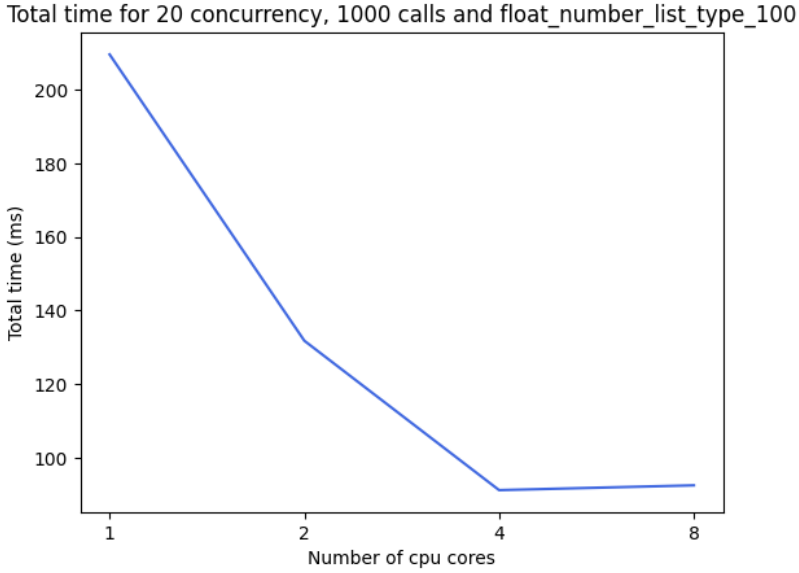
Στην Εικόνα 39 φαίνονται τα διαγράμματα του average latency, των requests per second, του συνολικού χρόνου και του χρόνου προς τον αριθμό των κλήσεων για 20 νήματα πελάτη, και εκτέλεση 1000 rpc calls με FloatNumberList μηνύματα με 100 αριθμούς σε συνάρτηση με τα cpu cores.

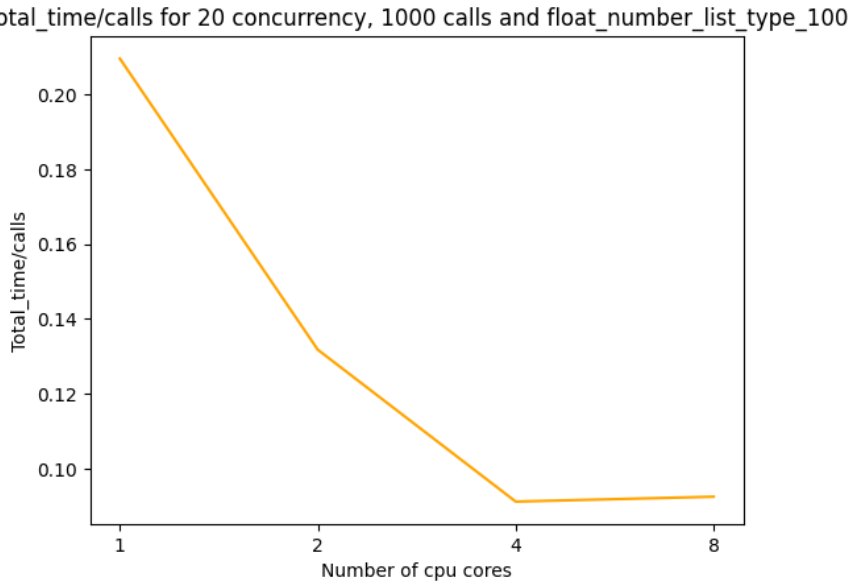
Στο πρώτο διάγραμμα, παρατηρείται ότι όσο περισσότερο αυξάνεται ο αριθμός των cpu cores, τόσο μειώνεται το average latency, κάτι το οποίο είναι λογικό εφόσον αυξάνεται η υπολογιστική ισχύς του μηχανήματος.

Αντίστοιχα αποτελέσματα φαίνονται και στα υπόλοιπα διαγράμματα. Ωστόσο παρατηρείται κορεσμός κατά την αύξηση των cpu cores από 4 σε 8, εφόσον ο αριθμός των κλήσεων ανά δευτερόλεπτο μειώνεται και ο συνολικός χρόνος αυξάνεται. Αυτό μπορεί να οφείλεται στο γεγονός ότι το μηχάνημα στο οποίο εκτελέστηκαν τα πειράματα διαθέτει 4 cpu cores, και επομένως η χρήση 8 cores από το ghz είναι εικόνική.


Average latency for 20 concurrency, 1000 calls and float_number_list_type_100
Average latency (ms)
2.75
2.50
2.25
2.00
1.75
1.50
1.25
1.00
1
2
4
8
Number of cpu cores


Requests per second for 20 concurrency, 1000 calls and float_number_list_type_100
RPS
11000
10000
9000
8000
7000
6000
5000
1
2
4
8
Number of cpu cores


Total time for 20 concurrency, 1000 calls and float_number_list_type_100
Total time (ms)
200
180
160
140
120
100
1
2
4
8
Number of cpu cores

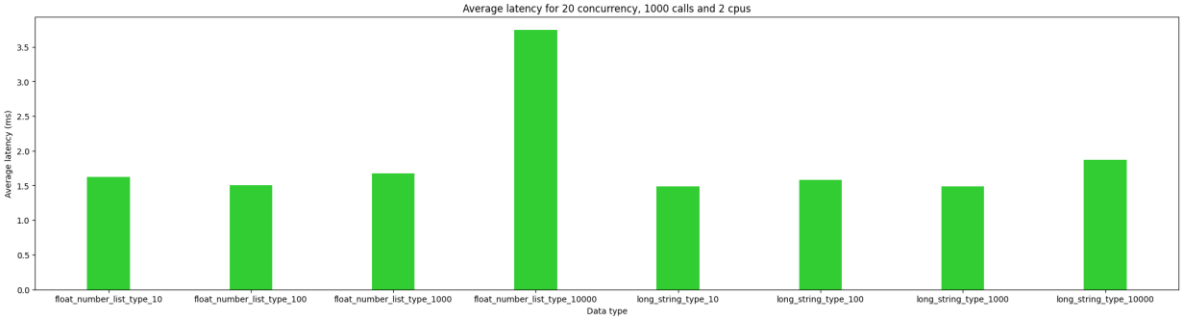


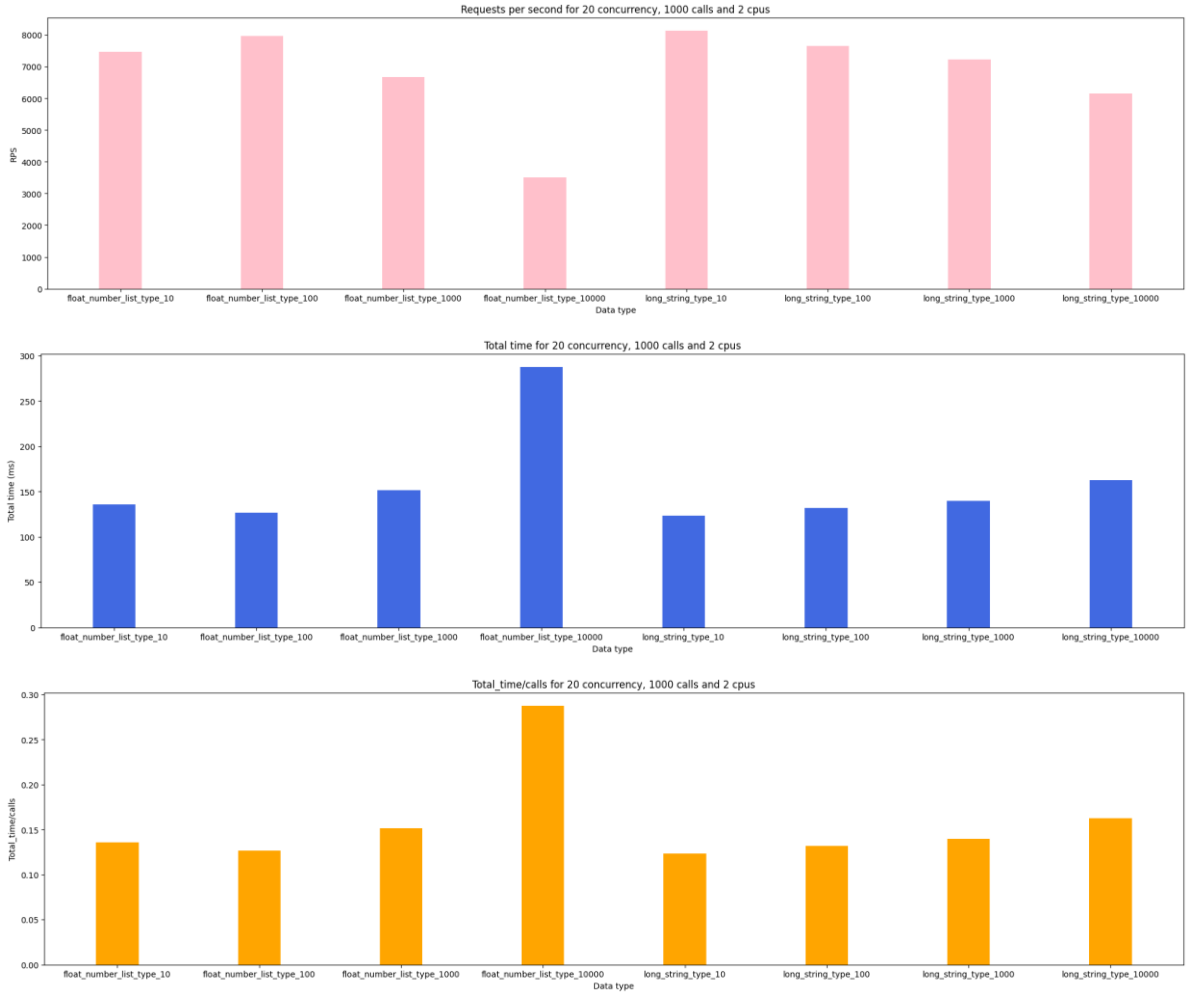
Εικόνα 39: Average latency, rps, total time και total time/calls συναρτήσει του αριθμού των cpu cores

Στην Εικόνα 40 φαίνεται η μεταβολή του average latency, του rps (requests per second), του συνολικού χρόνου εκτέλεσης, και του χρόνου εκτέλεσης προς τον αριθμό των κλήσεων για την εκτέλεση 1000 rpc calls με χρήση 2 cpu cores και 20 νημάτων πελάτη συναρτήσει του είδους και του μεγέθους των δεδομένων των κλήσεων.

Παρατηρείται ότι γενικά το average latency είναι παρόμοιο για όλα τα είδη κλήσεων και αυξάνεται με την αύξηση του μεγέθους των δεδομένων, αφού μεγαλώνουν τα πακέτα που μεταφέρουν τα μηνύματα Ανάλογες μεταβολές παρατηρούνται και στον συνολικό χρόνο εκτέλεσης των κλήσεων. Αντίστοιχα, ο αριθμός των κλήσεων ανά δευτερόλεπτο μειώνεται με την αύξηση του μεγέθους των δεδομένων, κάτι το οποίο είναι λογικό εφόσον αυξάνεται ο χρόνος μετάδοσης των πακέτων. Εξαίρεση σε όλα τα διαγράμματα αποτελεί η χρήση FloatNumberList μεγέθους 10000 η οποία αυξάνει κατά πολύ περισσότερο το average latency και το total time, ενώ μειώνει κατά πολύ το requests per second, λόγω του θρυμματισμού που προκαλείται στα HTTP/2 πλαίσια που μεταφέρουν τα μηνύματα αυτά.

Η χρήση ενός LongString που αποτελείται από 10000 χαρακτήρες δεν προκαλεί αντίστοιχο θρυμματισμό και αύξηση του average latency, κάτι το οποίο οφείλεται στο μεγαλύτερο μέγεθος σε bytes των int αριθμών από τους χαρακτήρες.





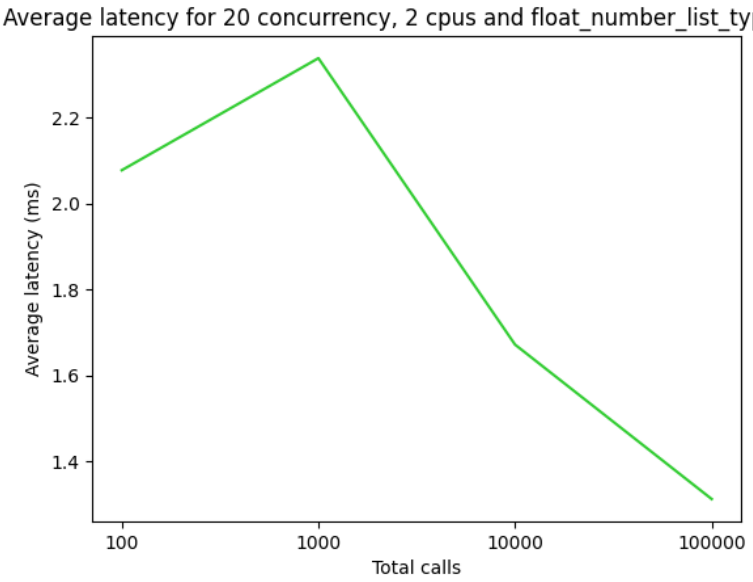
Εικόνα 40: Average latency, rps, total time και total time/calls συναρτήσει των δεδομένων

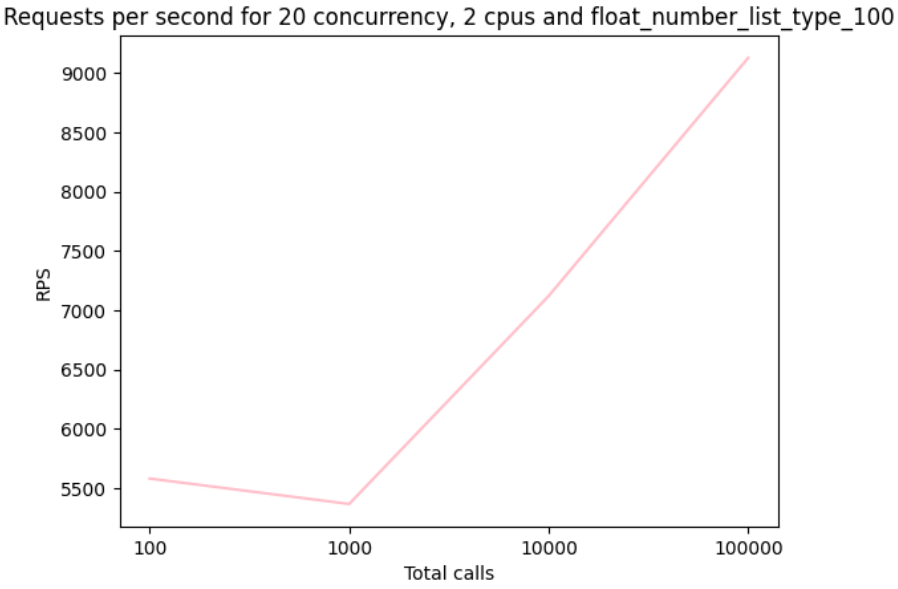
Στην Εικόνα 41 παρουσιάζεται το διάγραμμα του average latency, του rps, του total time και του total time/calls συναρτήσει του αριθμού των rpc κλήσεων που εκτελούνται, για FloatNumberList μηνύματα μεγέθους 100 και τη χρήση 2 cpu cores και 20 νημάτων πελάτη.

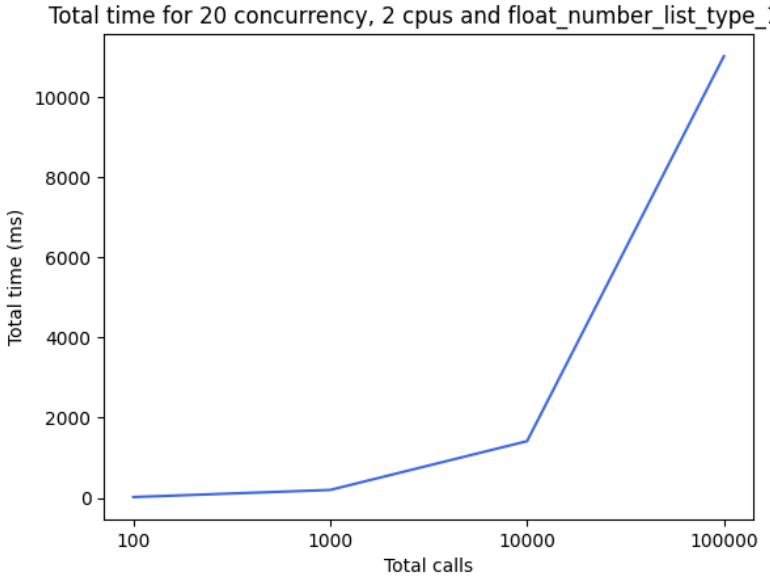
Παρατηρείται ότι το average latency παρουσιάζει ένα μέγιστο στις 1000 κλήσεις και έπειτα μειώνεται. Κάτι τέτοιο μπορεί να συμβαίνει λόγω της μέγιστης εκμετάλλευσης των πόρων του καναλιού λόγω της αύξησης των requests.

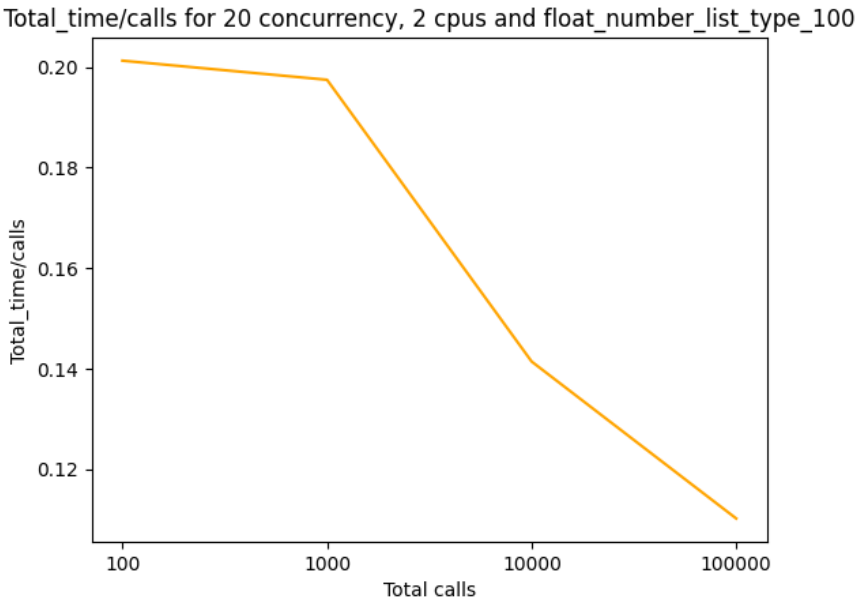
Ο αριθμός των κλήσεων ανά δευτερόλεπτο παρουσιάζει αύξηση όσο αυξάνεται ο αριθμός των κλήσεων, με μία μικρή μείωση κατά τη μετάβαση από τις 100 στις 1000 κλήσεις. Να σημειωθεί ότι στις περιπτώσεις του μικρού πλήθους κλήσεων, όπου ο χρόνος εκτέλεσης είναι μικρότερος του ενός δευτερολέπτου, ο αριθμός rps είναι ένα projection του ghz, καθώς δεν υπάρχουν επαρκή δεδομένα και η μέτρηση της επίδοσης του συστήματος γίνεται σε χρόνο πολύ μικρότερο του ενός second.

Ο συνολικός χρόνος εκτέλεσης των κλήσεων προφανώς αυξάνεται με την αύξηση του πλήθους τους. Ωστόσο, από το διάγραμμα total time/calls φαίνεται ότι ο μέσος χρόνος εκτέλεσης ενός rpc μειώνεται όσο αυξάνεται το συνολικό πλήθος των rpc. Φαίνεται λοιπόν ότι η συνολική επίδοση του συστήματος βελτιώνεται όσο περισσότερες rpc κλήσεις εκτελούνται.


Average latency for 20 concurrency, 2 cpus and float_number_list_type_100
Average latency (ms)
2.2
2.0
1.8
1.6
1.4
100
1000
10000
100000
Total calls


Requests per second for 20 concurrency, 2 cpus and float_number_list_type_100
RPS
9000
8500
8000
7500
7000
6500
6000
5500
100
1000
10000
100000
Total calls

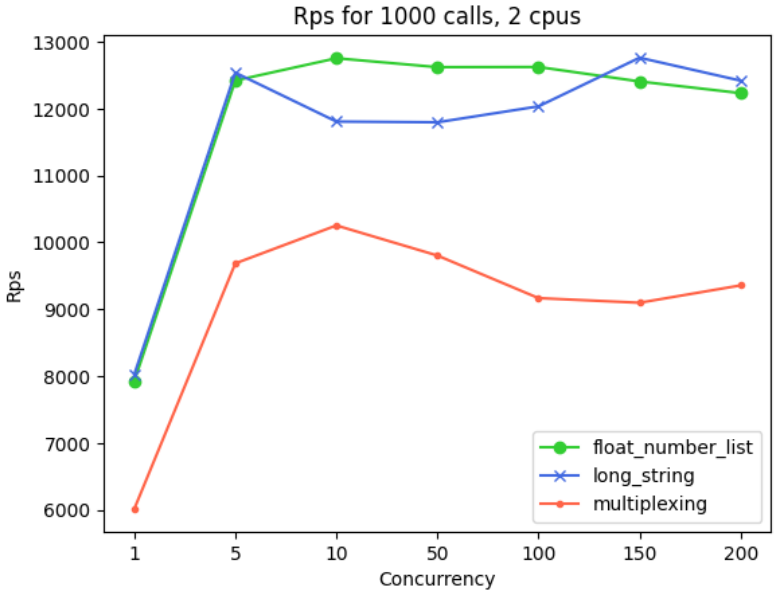

Total time for 20 concurrency, 2 cpus and float_number_list_type_100
Total time (ms)
10000
8000
6000
4000
2000
0
100
1000
10000
100000
Total calls

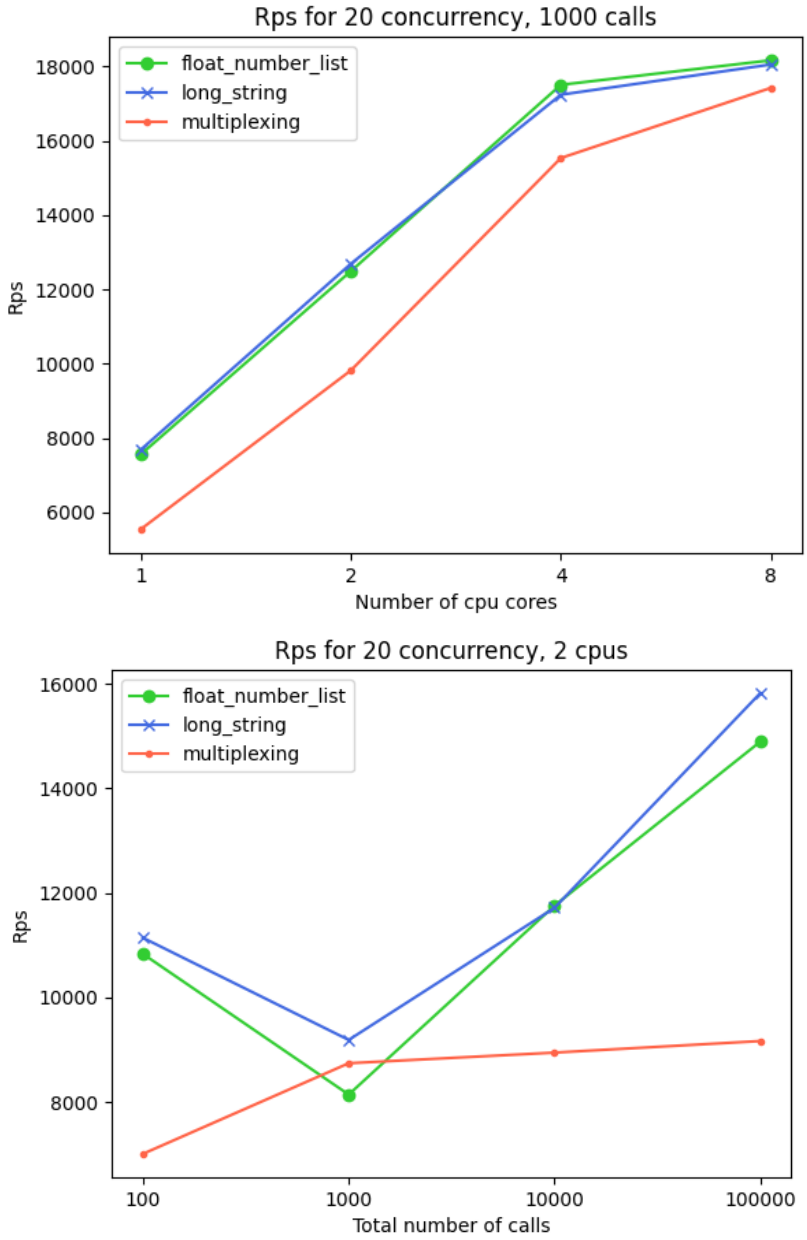


Εικόνα 41: Average latency, rps, total time και total time/calls συναρτήσει του συνολικού αριθμού των κλήσεων

Σκοπός των επόμενων πειραμάτων ήταν να καθοριστεί το πώς επηρεάζει ένα είδος rpc την εκτέλεση ενός διαφορετικού είδους rpc όταν αυτά πολυπλέκονται. Χρησιμοποιήθηκε, λοιπόν, η μέθοδος ComputeMeanRepeatedOrSendLongString σε τρεις περιπτώσεις:

- Ως ComputeMeanRepeated rpc, με χρήση FloatNumberList μηνυμάτων που περιέχουν πίνακες μεγέθους 100 αριθμών.
- Ως SendLongString rpc, με χρήση LongString μηνυμάτων που περιέχουν string μεγέθους 1000 χαρακτήρων.
- Εναλλάξ ως ComputeMeanRepeated και ως SendLongString rpc, με μηνύματα των παραπάνω μεγεθών, στο ίδιο πείραμα ώστε να παρατηρηθεί η πολυπλεξία μεταξύ των δύο ειδών rpc.

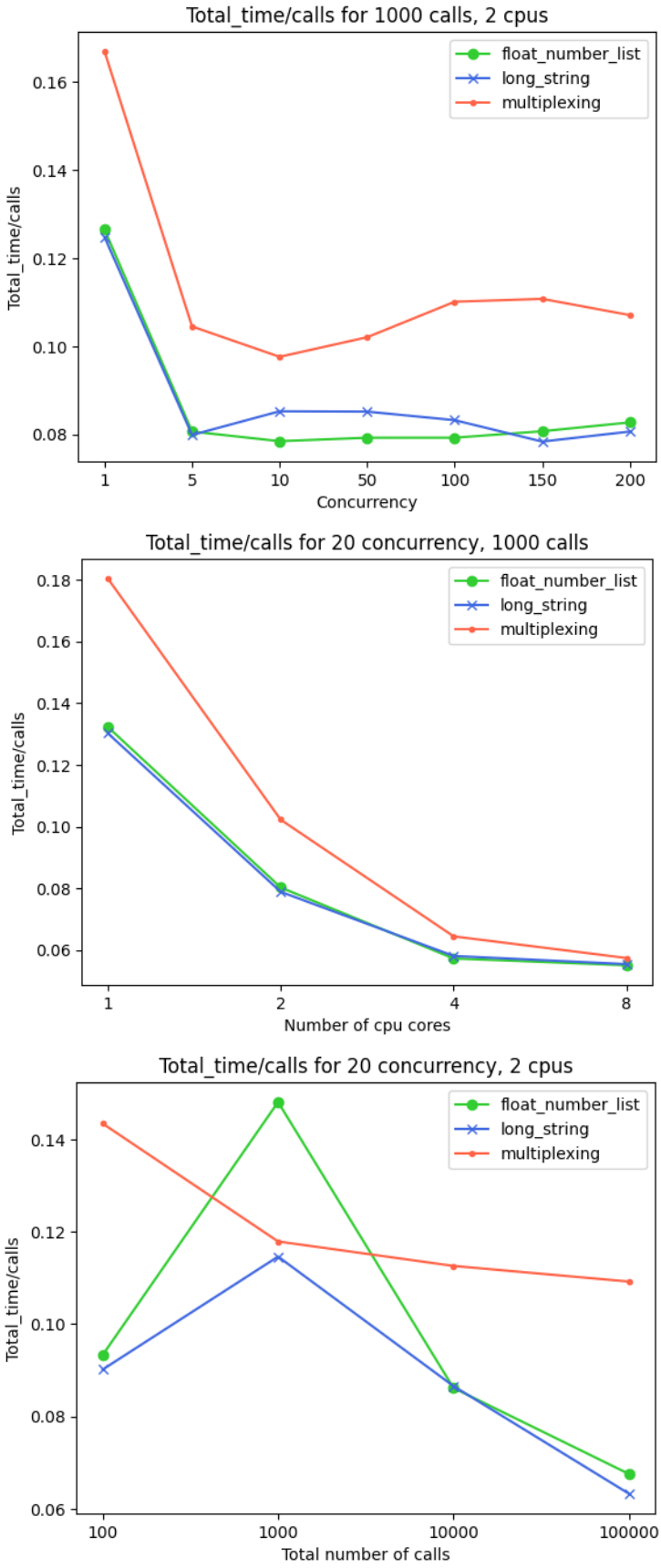




Εικόνα 42: Requests per second συναρτήσει παραμέτρων για δύο είδη rpc και multiplexing

Στην Εικόνα 42 φαίνονται τα διαγράμματα requests per second συναρτήσει του concurrency, του αριθμού των cpu cores και του πλήθους των rpc κλήσεων, ενώ στην Εικόνα 43 φαίνονται τα αντίστοιχα διαγράμματα για το συνολικό χρόνο εκτέλεσης προς των αριθμό των κλήσεων.

Παρατηρείται ανάλογη με προηγουμένως μεταβολή των μετρικών συναρτήσει των παραμέτρων. Ωστόσο φαίνεται ότι σε κάθε περίπτωση οι μετρικές επίδοσης του συστήματος χειροτερεύουν όσο εμπλέκονται τα δύο είδη rpc. Συμπεραίνεται επομένως, ότι η πολυπλεξία ενός rpc με ένα άλλο προκαλεί καθυστερήσεις στο σύστημα, γεγονός μη επιθυμητό σε περίπτωση που κάποιο rpc πρέπει να σταλεί γρήγορα και άρα θα έπρεπε να έχει προτεραιότητα σε σχέση με κάποιο άλλο που το καθυστερεί.



Εικόνα 43: Total time/calls συναρτήσει παραμέτρων για δύο είδη rpc και multiplexing

Εισαγωγή

Οι υλοποιήσεις του συστήματος με τον χρονοδρομολογητή, που αναπτύχθηκαν και σχολιάστηκαν παραπάνω, δοκιμάστηκαν προκειμένου να προκύψει μία εκτίμηση της επίδοσής τους και να επιλεγεί η καλύτερη. Συγκεκριμένα, οι υλοποιήσεις δοκιμάστηκαν για πολλές συνεχόμενες εκτελέσεις του client-side streaming rpc ComputeMean, το οποίο δέχεται πολλούς αριθμούς και επιστρέφει τον μέσο όρο τους.

Τα rpc που έπρεπε να σταλούν ήταν όλα του ίδιου είδους (ComputeMean) και με ίδιο πλήθος αριθμών που δόθηκαν ως input, αλλά με διαφορετική προτεραιότητα. Ο λόγος που επιλέχθηκε αυτό, είναι επειδή έγιναν μετρήσεις του χρόνου εκτέλεσης του κάθε rpc, οι οποίες προκειμένου να συγκριθούν δεν θα έπρεπε να περιλαμβάνουν διαφορές στο χρόνο αποστολής των πακέτων λόγω διαφορετικού πλήθους πακέτων ή διαφορές στο computation time από πλευράς server λόγω διαφορετικού είδους rpc.

Τα rpc που έπρεπε να σταλούν δόθηκαν στο πρόγραμμα ως αρχείο .txt με την κάθε γραμμή να αντιστοιχεί σε ένα rpc, το οποίο ο client διάβαζε ως input. Χρησιμοποιήθηκαν προτεραιότητες εύρους 1-10, πλήθος rpc, της τάξεως των 1000, και πλήθος input αριθμών, της τάξεως των 100. Σε κάθε πείραμα μετρήθηκε ο μέσος χρόνος ολοκήρωσης των rpc για κάθε προτεραιότητα, καθώς και ο συνολικός χρόνος ολοκλήρωσης του πειράματος, με και χωρίς χρήση scheduler για το ίδιο input file.

Στην Εικονα 44 φαίνεται ένα παράδειγμα του input file. Αρχικά αναγράφεται το rpc που θα εκτελεστεί, στη συνέχεια η αριθμοί που θα σταλούν ως IntNumber μηνύματα από τον client στο server, και τέλος η προτεραιότητα της rpc κλήσης.

python_scripts > scheduler_scripts > input_file.txt

```
1   compute_mean 78,35,4,7,5,84,85,49,85,45,9,65,0,6,82,64,59,76,40,28 4
2   compute_mean 33,74,23,45,49,82,65,23,36,88,62,73,36,20,84,90,84,7,24,18 6
3   compute_mean 41,99,50,76,14,38,2,51,53,61,5,48,17,83,41,60,48,67,77,60 2
4   compute_mean 10,86,24,29,76,86,93,22,22,11,10,24,94,96,54,7,97,74,23,21 10
5   compute_mean 37,14,68,29,65,80,20,89,35,13,16,21,84,48,37,60,56,45,12,35 3
6   compute_mean 14,66,55,29,91,29,17,20,0,88,69,38,56,92,6,17,33,13,15,74 9
7   compute_mean 46,56,84,14,43,4,79,7,8,100,75,70,23,1,87,41,9,87,73,45 4
8   compute_mean 41,67,62,59,97,56,21,77,31,92,20,37,80,60,39,58,2,92,23,0 8
9   compute_mean 15,6,84,1,50,93,86,26,27,64,60,54,80,40,9,87,60,71,13,59 8
10  compute_mean 19,58,45,41,59,91,77,94,65,61,9,45,88,73,69,69,27,9,56,45 5
11  compute_mean 89,8,98,91,100,20,25,83,44,0,73,95,53,40,47,48,5,54,45,85 4
12  compute_mean 39,39,46,63,64,69,36,97,91,43,28,52,32,37,47,41,83,20,23,46 5
13  compute_mean 15,12,85,34,75,86,81,65,82,49,99,11,75,32,79,28,29,60,86,10 1
14  compute_mean 4,59,46,62,20,19,68,83,18,26,24,33,32,10,54,59,45,86,43,58 7
15  compute_mean 5,18,81,87,71,38,71,45,31,36,5,67,77,47,36,4,83,84,31,7 4
16
```

Εικόνα 44: Παράδειγμα input file

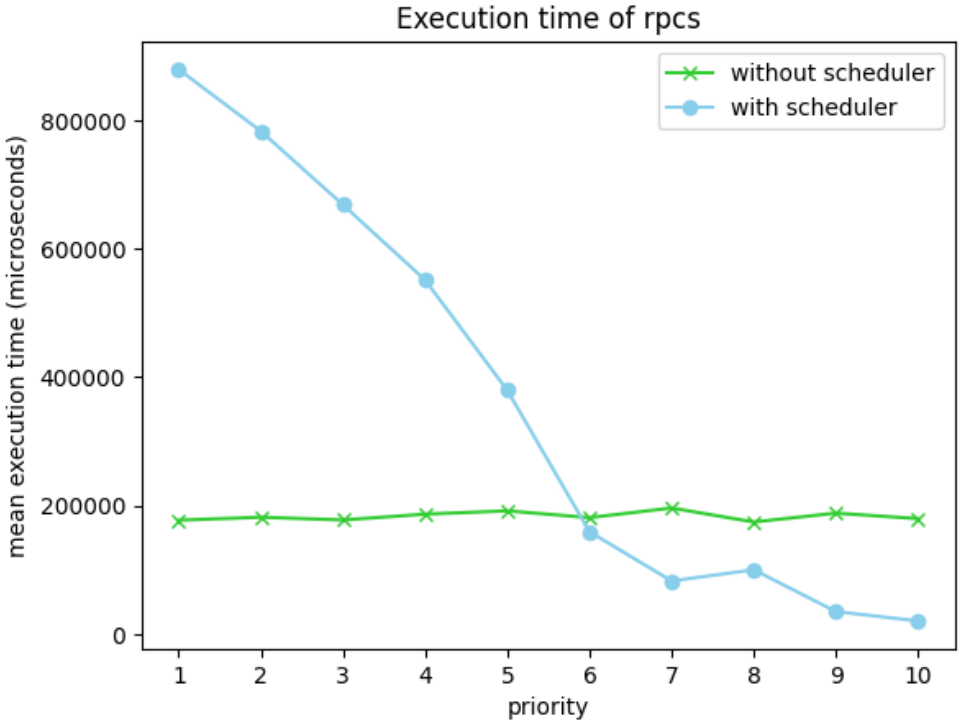
Επίδοση υλοποίησης σε C++ με processes

Στην Εικόνα 45 φαίνεται ο μέσος χρόνος εκτέλεσης ενός rpc συναρτήσει της προτεραιότητάς του, με χρήση scheduler (γαλάζιο χρώμα) και χωρίς (πράσινο χρώμα). Συγκεκριμένα, για κάθε rpc που εκτελέστηκε κατά τη διάρκεια του πειράματος, χρονομετρήθηκε το χρονικό διάστημα από τη στιγμή που διαβάστηκαν οι παράμετροι της κλήσης από το input file, έως τη στιγμή που λήφθηκε η απάντηση από τον διακομιστή. Στη συνέχεια υπολογίστηκε ο μέσος όρος των χρόνων εκτέλεσης, για τα rpc που έχουν την ίδια προτεραιότητα. Οι παραπάνω μετρήσεις έγιναν δύο φορές: μία με τη χρήση του scheduler

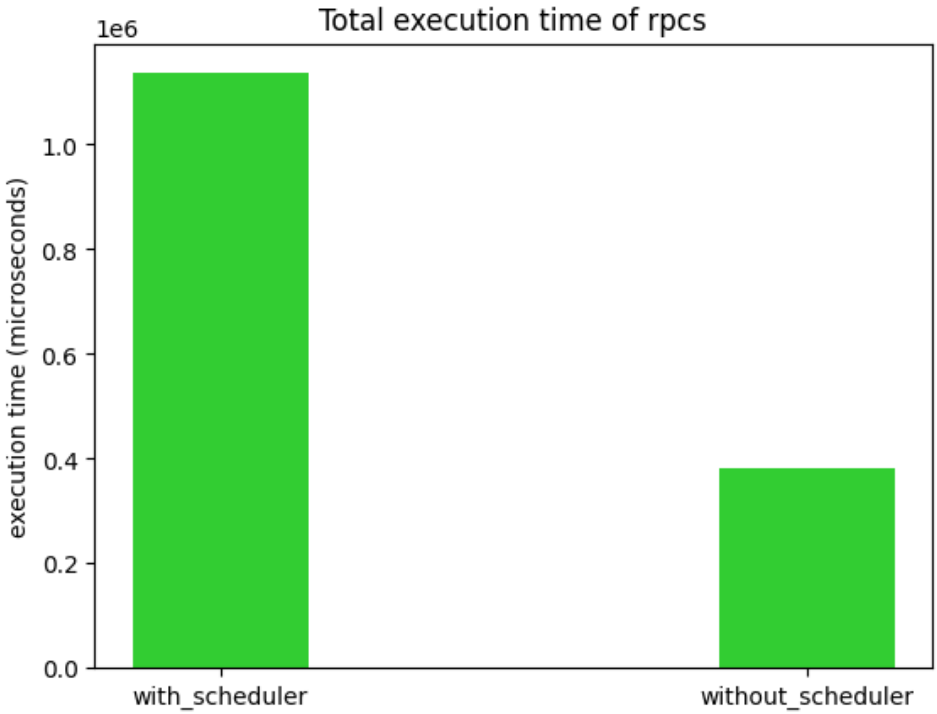
component στον client, όπου προηγούνται rpc κλήσεις μεγαλύτερων προτεραιοτήτων σύμφωνα με τον τρόπο που περιγράφηκε κατά την ανάλυση της κάθε υλοποίησης σε προηγούμενο κεφάλαιο, και μία χωρίς την ύπαρξη χρονοδρομολογητή, όπου ο πελάτης αρχίζει να εκτελεί τα rpc με τη σειρά που τα διαβάζει από το input file, πολυπλέκοντας τα πακέτα στο gRPC κανάλι ανεξαρτήτως των προτεραιοτήτων.

Στο διάγραμμα της Εικόνας 45, αυτό που φαίνεται είναι ότι χωρίς τη χρήση scheduler ο μέσος χρόνος ολοκλήρωσης είναι περίπου ίδιος για κάθε priority, κάτι το οποίο είναι λογικό εφόσον το gRPC δεν διακρίνει προτεραιότητες μεταξύ rpc – streams. Με τη χρήση της πρώτης υλοποίησης scheduler φαίνεται ότι οι χαμηλές προτεραιότητες εκτελούνται αρκετά πιο αργά σε σχέση με την απουσία scheduler, γεγονός που οφείλεται στο ότι τα rpc με τα χαμηλά priority περιμένουν να εκτελεστούν οι πιο σημαντικές κλήσεις πριν ολοκληρωθούν. Αντιθέτως, τα rpc που έχουν υψηλή προτεραιότητα ολοκληρώνονται χωρίς μεγάλη αναμονή. Στο διάγραμμα φαίνεται ότι οι προτεραιότητες 6-10 εκτελούνται πιο γρήγορα σε σχέση με την απουσία scheduler, ενώ οι προτεραιότητες 1-5 πιο αργά.

Στην Εικόνα 46 φαίνεται ο συνολικός χρόνος ολοκλήρωσης των δύο πειραμάτων, ο οποίος είναι σαφώς μεγαλύτερος για τη χρήση scheduler. Κατά τη χρήση χρονοδρομολογητή, δηλαδή, προστίθεται κάποιο χρονικό overhead στο πείραμα το οποίο οφείλεται σε καθυστερήσεις λόγω της διαδικασίας εναλλαγής του rpc που έχει την άδεια να στείλει κάθε φορά, καθώς και σε χρόνους που απαιτούνται για την εγγραφή μηνυμάτων στο socket object και την αποστολή των διαφόρων signals που ανταλλάσσονται. Παρατηρείται ότι ενώ προστίθεται κάποια καθυστέρηση λόγω της ύπαρξης scheduler, οι κρίσιμες προτεραιότητες (6-10) εκτελούνται αρκετά πιο γρήγορα, κάτι το οποίο ήταν και ζητούμενο. Υπάρχει, λοιπόν ένα trade-off αύξησης του συνολικού χρόνου εκτέλεσης με αντάλλαγμα την μείωση του χρόνου ολοκλήρωσης σημαντικών rpc κλήσεων.



Εικόνα 4517: Μέσος χρόνος εκτέλεσης rpc για την C++ processes υλοποίηση



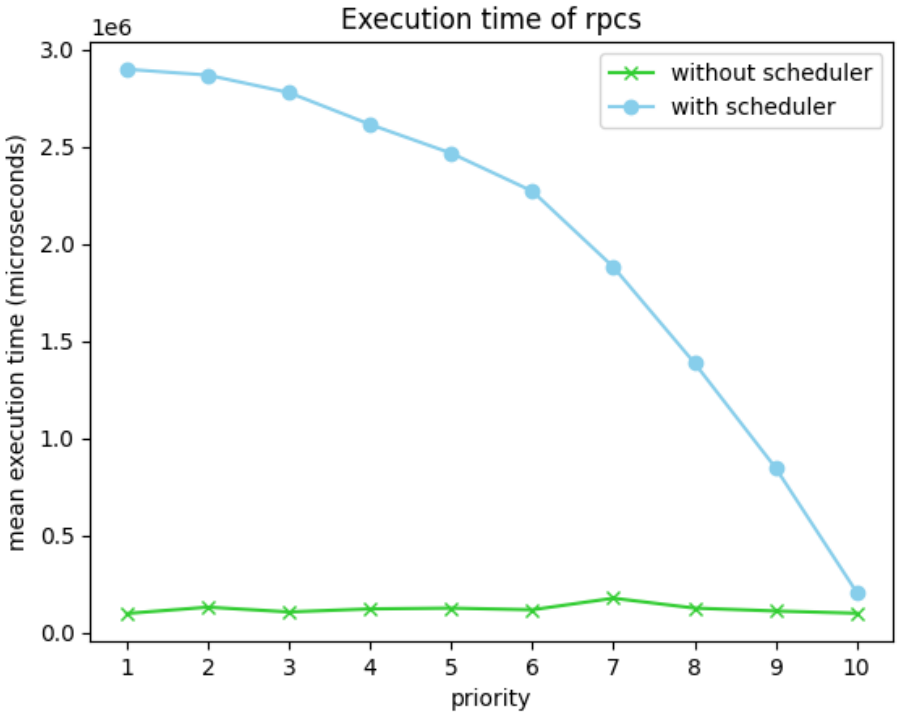
Εικόνα 46: Συνολικός χρόνος εκτέλεσης rpc για την C++ processes υλοποίηση

Να σημειωθεί ότι η συγκεκριμένη υλοποίηση παρουσιάζει κάποια μειονεκτήματα λόγω της χρήσης processes και της ανάγκης αποστολής σημάτων για τη μεταξύ τους επικοινωνία. Τα signals εμπλέκουν το kernel ενός λειτουργικού συστήματος, επομένως ο χρόνος που χρειάζεται ένα σήμα για να δημιουργηθεί και να σταλεί εξαρτάται κατά πολύ από τη διαθεσιμότητα του kernel τη συγκεκριμένη χρονική στιγμή. Υπάρχει, δηλαδή, εξάρτηση του συστήματος από το εκάστοτε λειτουργικό σύστημα που χρησιμοποιείται, τις διαθέσιμες βιβλιοθήκες που παρέχει αυτό κλπ.

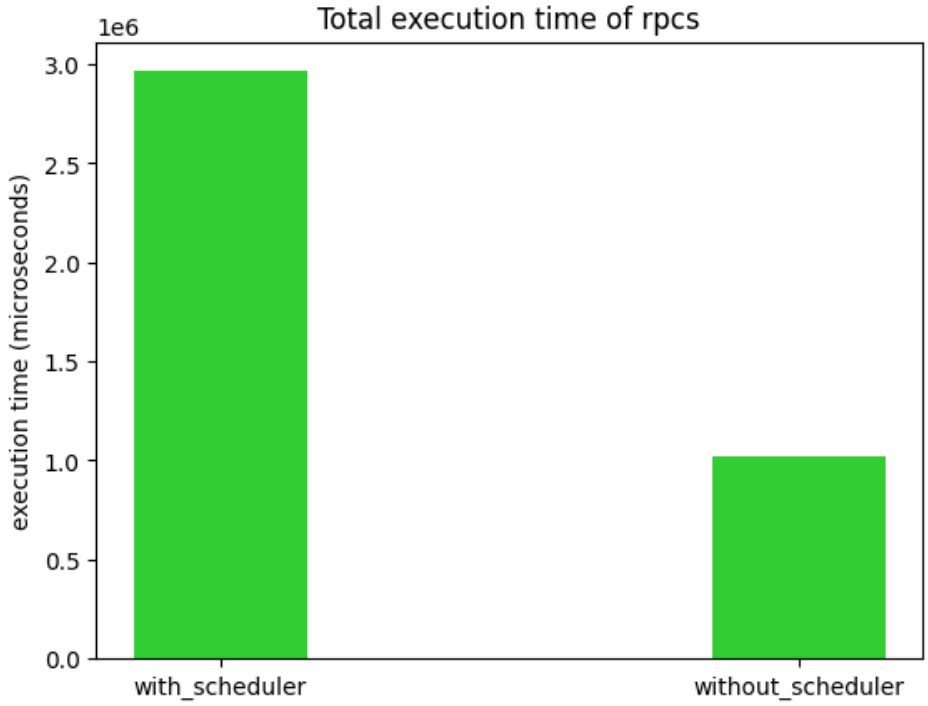
Επίδοση πρώτης υλοποίησης σε C++ με threads

Στις Εικόνες 47 και 48 φαίνονται τα αντίστοιχα διαγράμματα για την πρώτη υλοποίηση του scheduler με threads σε C++. Παρατηρείται ότι και πάλι ο μέσος χρόνος εκτέλεσης των rpc συναρτήσει της προτεραιότητας για απουσία scheduler είναι σταθερός και ανεξάρτητος της προτεραιότητας. Με τη χρήση χρονοδρομολογητή, υπάρχει μείωση του χρόνου ολοκλήρωσης όσο αυξάνεται η προτεραιότητα ενός rpc, ωστόσο καμία προτεραιότητα δεν εμφανίζει καλύτερους χρόνους σε σχέση με τη μη χρήση scheduler. Ο συνολικός χρόνος εκτέλεσης με τη χρήση scheduler είναι και πάλι μεγαλύτερος, σε σχέση με την απουσία scheduler.

Οι τόσο μεγάλες καθυστερήσεις οφείλονται στο χρόνο που απαιτείται για την απόκτηση locks, η οποία είναι απαραίτητη εφόσον τα threads χρησιμοποιούν κοινή μνήμη. Τα threads που εκτελούν τα rpc ελέγχουν συνεχώς κάποιες κοινές μεταβλητές προκειμένου να μάθουν εάν μπορούν να χρησιμοποιήσουν το κανάλι ή όχι. Επομένως, εάν είναι η σειρά ενός νήματος να στείλει τα πακέτα του, αυτό μπορεί να περιμένει αρκετή ώρα ώστε να αποκτήσει πρόσβαση στον κοινό πόρο και να ενημερωθεί σχετικά με αυτό. Το κανάλι κατά τη διάρκεια της διαδικασίας αυτής μένει ανεκμετάλλευτο. Συμπεραίνεται, λοιπόν, ότι η υλοποίηση αυτή δεν είναι καλή, καθώς απλά εισάγεται καθυστέρηση χωρίς να υπάρχει κάποιο κέρδος στο prioritization.



Εικόνα 187: Μέσος χρόνος εκτέλεσης rpc για την πρώτη C++ threads υλοποίηση

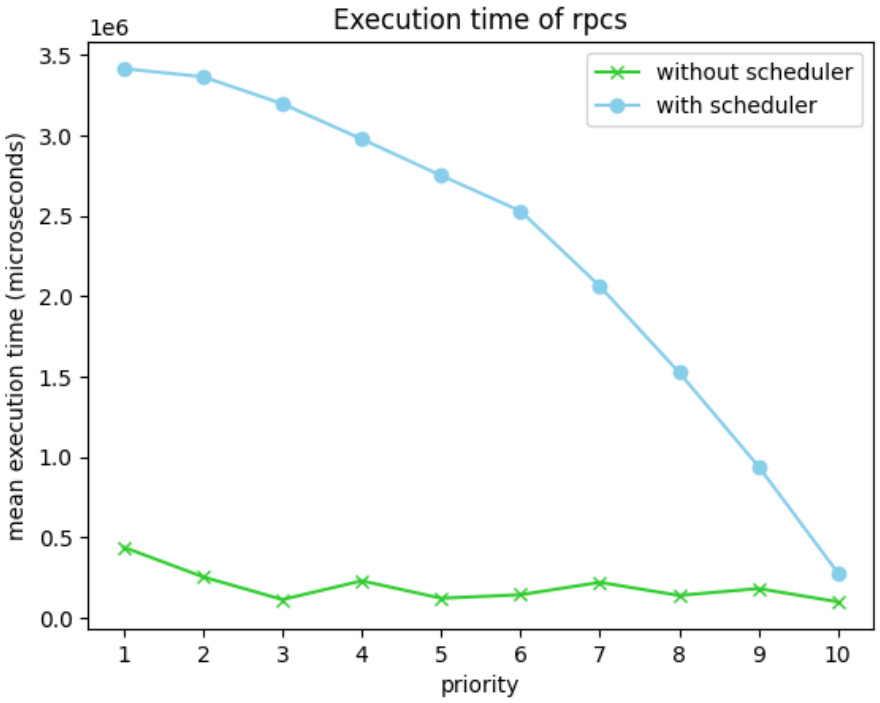


Εικόνα 48: Συνολικός χρόνος εκτέλεσης rpc για την πρώτη C++ threads υλοποίηση

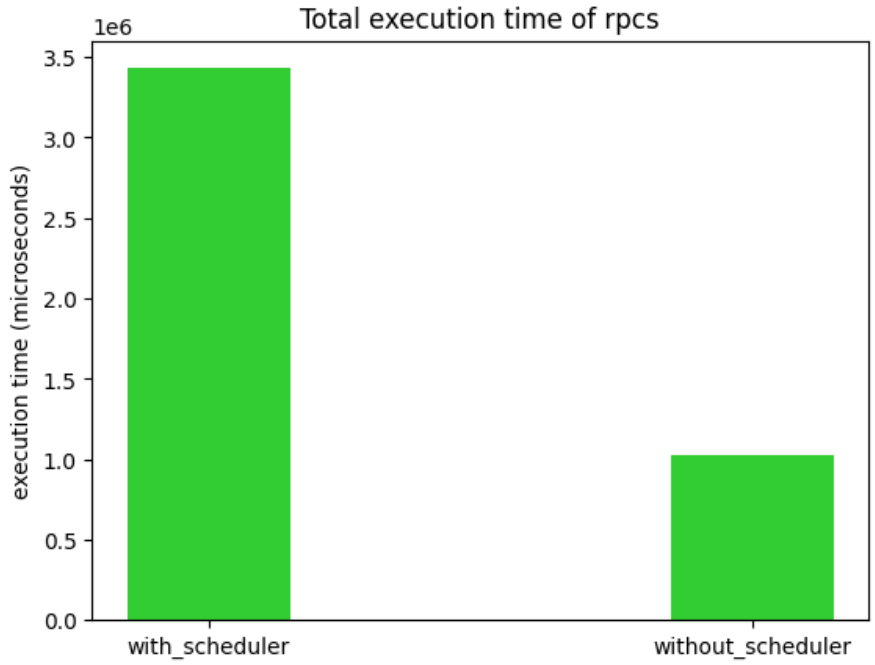
Επίδοση δεύτερης υλοποίησης σε C++ με threads

Στις Εικόνες 49 και 50 φαίνονται τα διαγράμματα του μέσου χρόνου εκτέλεσης και του συνολικού χρόνου εκτέλεσης των rpc για την δεύτερη υλοποίηση του πελάτη με threads σε C++, με και χωρίς τη χρήση του scheduler component. Τα αποτελέσματα των μετρήσεων

για αυτή την υλοποίηση μοιάζουν αρκετά με εκείνα της προηγούμενης, κάτι το οποίο είναι λογικό εφόσον και οι δύο υλοποιήσεις είναι στην ίδια γλώσσα, χρησιμοποιούν νήματα και πρόσβαση σε κοινούς πόρους για τη μεταξύ τους επικοινωνία.



Εικόνα 49: Μέσος χρόνος εκτέλεσης rpc για την δεύτερη C++ threads υλοποίηση

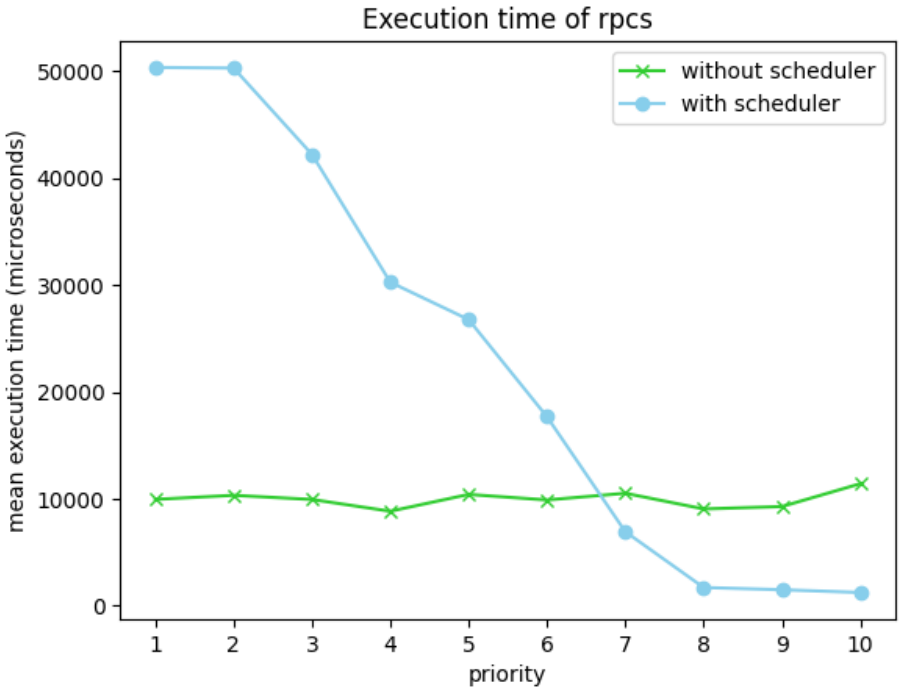


Εικόνα 50: Συνολικός χρόνος εκτέλεσης rpc για την δεύτερη C++ threads υλοποίηση

Παρατηρείται και πάλι μεγάλη χρονική καθυστέρηση από τη χρήση του χρονοδρομολογητή στο συνολικό χρόνο εκτέλεσης, γεγονός που οφείλεται στην αναμονή που απαιτείται για την πρόσβαση στα κλειδώματα που προστατεύουν τις δομές που βρίσκονται στη κοινή μνήμη και στα race conditions που δημιουργούνται. Υπάρχει μείωση του μέσου χρόνου εκτέλεσης των rpc κλήσεων με την αύξηση της προτεραιότητάς τους, ωστόσο ο χρόνος ακόμα και των κλήσεων με τη μέγιστη προτεραιότητα (10) είναι μεγαλύτερος από το μέσο χρόνο εκτέλεσης των rpc χωρίς τη χρήση χρονοδρομολογητή. Επομένως, και σε αυτή την περίπτωση υπάρχει μεγάλο χρονικό overhead με την προσθήκη του scheduler component, χωρίς να υπάρχει κέρδος από την ιεράρχηση των προτεραιοτήτων, αφού ακόμα και τα σημαντικότερα rpc αργούν αρκετά. Να σημειωθεί ότι όλοι οι χρόνοι αυτής της υλοποίησης είναι ελάχιστα υψηλότεροι από τους αντίστοιχους χρόνους στην προηγούμενη υλοποίηση. Προκύπτει, λοιπόν, ότι καμία από τις υλοποιήσεις που αναπτύχθηκαν με χρήση νημάτων σε γλώσσα C++ δεν είναι αποδοτική.

Επίδοση πρώτης υλοποίησης σε Go με goroutines

Στην Εικόνα 51 φαίνεται το διάγραμμα του μέσου χρόνου εκτέλεσης των rpc κλήσεων ως συνάρτηση της προτεραιότητάς τους. Όπως είναι αναμενόμενο, στην περίπτωση χρήσης χρονοδρομολογητή, ο χρόνος εκτέλεσης είναι αντιστρόφως ανάλογος της προτεραιότητας, εφόσον οι κλήσεις μιας προτεραιότητας πρέπει να περιμένουν τις κλήσεις των υψηλότερων προτεραιοτήτων να ολοκληρωθούν πριν εκτελεστούν. Οι προτεραιότητες 1-6 εκτελούνται πιο αργά σε σχέση με την απουσία του scheduler component, ενώ οι σημαντικές προτεραιότητες 7-10 ολοκληρώνονται πιο γρήγορα. Οι τρεις πιο κρίσιμες προτεραιότητες 8-10 εκτελούνται αρκετά γρήγορα και σε σχεδόν ίδιο χρόνο μεταξύ τους.

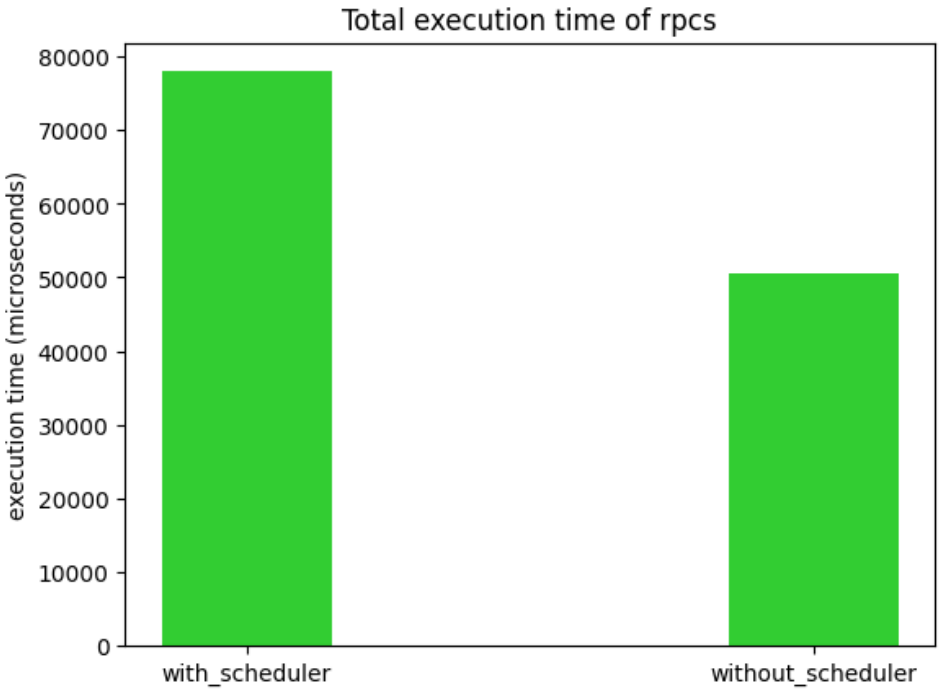


Εικόνα 51: Μέσος χρόνος εκτέλεσης rpc για την πρώτη Go υλοποίηση

Στην Εικόνα 52 παρουσιάζεται ο συνολικός χρόνος διεκπεραίωσης των δύο πειραμάτων. Κατά τη προσθήκη του χρονοδρομολογητή προστίθεται ένα χρονικό overhead, το οποίο όμως είναι αρκετά μικρότερο σε σχέση με τις προηγούμενες υλοποιήσεις. Αυτό μπορεί να οφείλεται στο γεγονός ότι δεν προστίθονται καθυστερήσεις λόγω αποστολής σημάτων και

πρόσβασης σε κλειδώματα. Τα channels της γλώσσας Go που χρησιμοποιούνται για την επικοινωνία ανάμεσα στα νήματα, είναι εξ ορισμού thread-safe, το οποίο σημαίνει ότι δεν δημιουργείται πρόβλημα κατά την ταυτόχρονη πρόσβασή τους από διαφορετικά νήματα. Τα channels χρησιμοποιούν build-in κλειδώματα, κάτι το οποίο καθιστά τη διαδικασία πρόσβασης σε αυτά πιο γρήγορη σε σχέση με την προσθήκη κλειδωμάτων από τον προγραμματιστή στον κώδικα, όπως συμβαίνει στις προηγούμενες υλοποιήσεις.

Τέλος να σημειωθεί ότι όλοι οι χρόνοι είναι μικρότεροι από τους αντίστοιχους χρόνους προηγούμενων υλοποιήσεων, επομένως η γλώσσα Go κρίνεται αρκετά αποδοτική και κατάλληλη για την υλοποίηση ενός gRPC πελάτη.



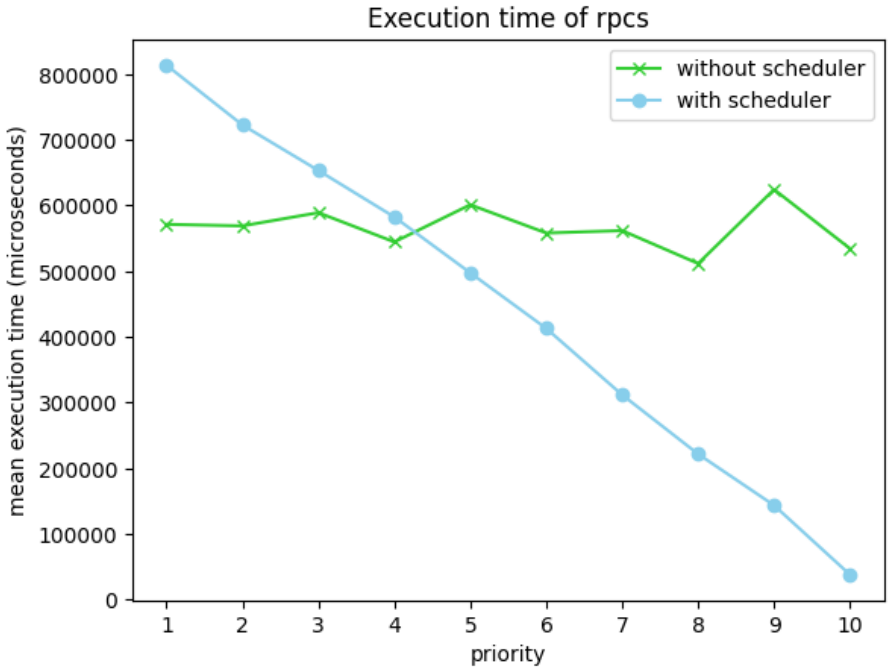
Εικόνα 52: Συνολικός χρόνος εκτέλεσης rpc για την πρώτη Go υλοποίηση

Επίδοση δεύτερης υλοποίησης σε Go με goroutines

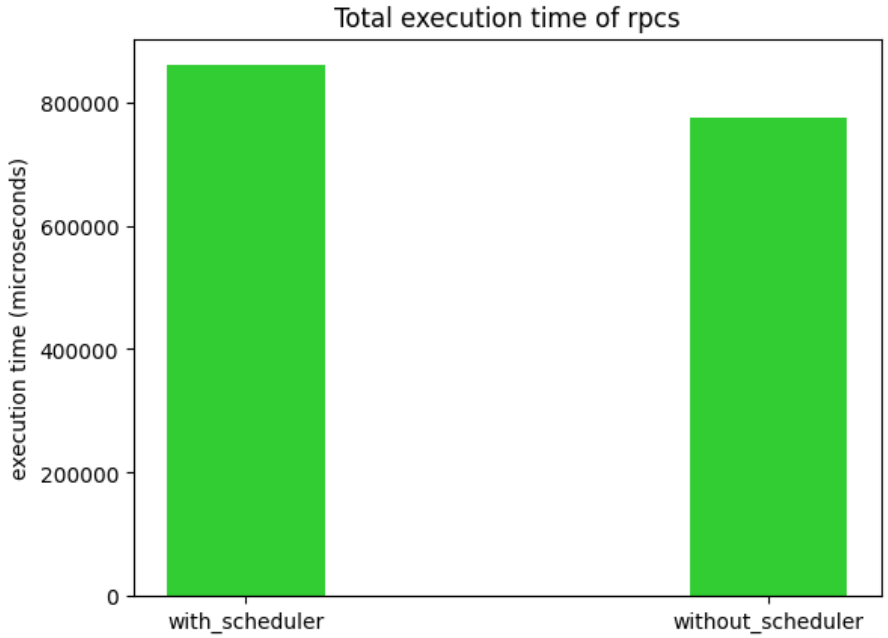
Στα σχήματα των Εικόνων 53 και 54 φαίνονται τα αντίστοιχα διαγράμματα για την τελευταία υλοποίηση του πελάτη, σε Go με goroutines. Σε αυτή την περίπτωση φαίνεται ότι τα rpc με προτεραιότητες 5-10 ολοκληρώνονται κατά μέσο όρο αρκετά πιο γρήγορα σε σχέση με την απουσία scheduler, ενώ μόνο τέσσερεις από τις μικρότερες προτεραιότητες εκτελούνται πιο αργά. Ακόμα όμως και οι προτεραιότητες 1-4, που βρίσκονται πιο ψηλά στο διάγραμμα από το χρόνο εκτέλεσης χωρίς scheduler, δεν παρουσιάζουν μεγάλη απόκλιση σε σχέση με αυτόν. Μάλιστα η ύπαρξη scheduler εισάγει πολύ μικρή καθυστέρηση στο συνολικό χρόνο διεκπεραίωσης του πειράματος, όπως φαίνεται στην Εικόνα 54.

Οι καλύτερες επιδόσεις του συστήματος στη συγκεκριμένη υλοποίηση οφείλονται στο γεγονός ότι rpc με ίδιο priority τρέχουν ταυτόχρονα και όχι σειριακά, σύμφωνα με τον τρόπο που το gRPC πολυπλέκει ήδη τα streams, και επομένως γίνεται καλύτερη χρήση του καναλιού. Επίσης δεν υπάρχουν μεγάλες καθυστερήσεις στη δρομολόγηση, εφόσον δεν χρειάζεται ο χρονοδρομολογητής να επέμβει για να δώσει την άδεια σε goroutines να χρησιμοποιήσουν το κανάλι κάθε φορά που ολοκληρώνει ένα rpc κάποιο goroutine, αλλά κάθε φορά που ολοκληρώνουν τα rpc τους όλα τα goroutines μιας συγκεκριμένης

προτεραιότητας. Η καθυστέρηση, δηλαδή, που προκαλείται από την επικοινωνία αυτή εισάγεται σε κάθε εναλλαγή προτεραιότητας και όχι σε κάθε εναλλαγή rpc, όπως γινόταν στις προηγούμενες υλοποιήσεις, και άρα εξαρτάται από των αριθμό των διαφορετικών προτεραιοτήτων που υπάρχουν και όχι από το πλήθος των rpc κλήσεων που πρέπει να εκτελεστούν. Το χρονικό overhead, λοιπόν, της ύπαρξης scheduler γίνεται τόσο πιο αμελητέο όσο περισσότερο αυξάνεται ο αριθμός των rpc που πρέπει να εκτελεστούν.



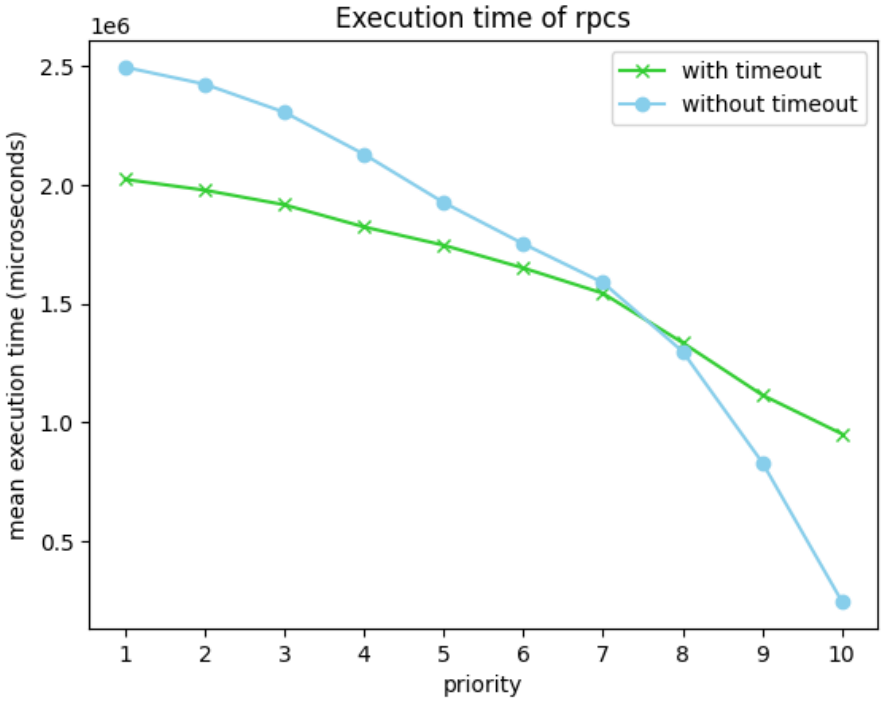
Εικόνα 53: Μέσος χρόνος εκτέλεσης rpc για την δεύτερη Go υλοποίηση



Εικόνα 54: Συνολικός χρόνος εκτέλεσης rpc για την δεύτερη Go υλοποίηση

Επίδοση υλοποίησης χρονοδρομολογητή με time-out

Στην Εικόνα 55 φαίνεται ο μέσος χρόνος εκτέλεσης των rpc κλήσεων συναρτήσει της προτεραιότητάς τους. Το γαλάζιο διάγραμμα αντιστοιχεί στην πρώτη υλοποίηση του χρονοδρομολογητή με νήματα σε C++, ενώ το πράσινο αφορά την ίδια υλοποίηση με την προσθήκη μηχανισμού time-out, όπως αναλύθηκε σε προηγούμενο κεφάλαιο.



Εικόνα 55: Μέσος χρόνος εκτέλεσης rpc για την υλοποίηση με time-out

Στην εικόνα φαίνεται ότι ο μέσος χρόνος εκτέλεσης των χαμηλών προτεραιοτήτων με χρήση time-out είναι μικρότερος από ότι χωρίς τον μηχανισμό. Το γεγονός αυτό οφείλεται στους μέγιστους χρόνους αναμονής που έχουν οριστεί. Για παράδειγμα, τα rpc με την χαμηλότερη προτεραιότητα (1), τίθενται σε αδράνεια προκειμένου πρώτα να ολοκληρωθούν οι κλήσεις μεγαλύτερων προτεραιοτήτων. Ωστόσο, μόλις παραμείνουν αδρανή για 2000 ms ξεκινούν ή συνεχίζουν αυτόματα την εκτέλεσή τους, χωρίς να περιμένουν τη σειρά τους. Χωρίς τη χρήση του μηχανισμού time-out παρατηρείται ότι ο μέσος χρόνος εκτέλεσης των κλήσεων με προτεραιότητα 1 είναι περίπου 2500 ms, κάτι το οποίο σημαίνει ότι ο χρόνος αναμονής τους ξεπερνάει τα 2000 ms που είναι το time-out time. Επομένως κατά την προσθήκη του time-out μηχανισμού παρατηρείται ότι ο μέσος χρόνος εκτέλεσης μειώνεται και βρίσκεται λίγο πιο ψηλά από το time-out time, εφόσον τα νήματα παραμένουν αδρανή μέχρι τότε και στη συνέχεια ολοκληρώνονται.

Αντίστοιχοι χρόνοι παρατηρούνται και για τις υπόλοιπες προτεραιότητες. Ωστόσο, όσο αυξάνεται η προτεραιότητα, τόσο μεγαλύτερος είναι ο μέσος χρόνος εκτέλεσης σε σχέση με το time-out time, κατά τη χρήση του μηχανισμού time-out. Για παράδειγμα, η προτεραιότητα 8 έχει απόκλιση 1000 ms ανάμεσα στο time-out time (13000 ms) και το μέσο χρόνος εκτέλεσης (~1400 ms), ενώ για την προτεραιότητα 10, η απόκλιση είναι σχεδόν 5000 ms. Η αύξηση της απόκλισης οφείλεται στο γεγονός ότι όταν οι κλήσεις μικρότερων προτεραιοτήτων ξεκινούν την αποστολή πακέτων πριν έρθει η σειρά τους, εισάγουν καθυστέρηση στις κλήσεις μεγαλύτερων προτεραιοτήτων που εκτελούνται εκείνη τη στιγμή.

Παρατηρείται, λοιπόν, ότι παρόλο που κατά τη χρήση του time-out μηχανισμού υπάρχει και πάλι η αντιστρόφως ανάλογη σχέση ανάμεσα στον μέσο χρόνο εκτέλεσης και την προτεραιότητα των rpc κλήσεων, ενώ έως κάποιο σημείο (προτεραιότητα 7) οι χρόνοι εκτέλεσης είναι μικρότεροι με τη χρήση του μηχανισμού από ότι χωρίς, για μεγαλύτερες προτεραιότητες ο μέσος χρόνος εκτέλεσης αυξάνεται αρκετά σε σχέση με την απλή υλοποίηση του χρονοδρομολογητή χωρίς time-out times. Το γεγονός αυτό βέβαια, οφείλεται εν μέρει και στους χαμηλούς χρόνους time-out, καθώς εάν αυτοί είχαν τεθεί μεγαλύτεροι, τέτοιοι δηλαδή ώστε κλήσεις μεγάλων προτεραιοτήτων να προλαβαίνουν να ολοκληρωθούν πριν λήξει ο χρόνος αναμονής κλήσεων χαμηλότερων προτεραιοτήτων, δεν θα παρατηρούταν καθυστέρηση σε ορισμένες μεγάλες προτεραιότητες.

ΣΥΜΠΕΡΑΣΜΑΤΑ

Συμπεράσματα

Κατά την ανάπτυξη της παρούσας διπλωματικής έγινε κατανοητή η χρησιμότητα του gRPC framework για την ανάπτυξη γρήγορων και κλιμακούμενων APIs σε συστήματα εφαρμογών που επικοινωνούν μεταξύ τους.

Το gRPC framework προσφέρει λειτουργίες οι οποίες το καθιστούν ιδιαίτερα εύκολο στη χρήση του και φιλικό προς τον προγραμματιστή. Μία πολύ σημαντική λειτουργία είναι η παροχή client-side, server-side και bidirectional streaming κλήσεων, η οποία είναι χρήσιμη σε περιπτώσεις όπου ο πελάτης χρειάζεται κάποια υπηρεσία από τον διακομιστή, η οποία όμως δεν μπορεί να ολοκληρωθεί με την ανταλλαγή μόνο δύο πακέτων, επειδή για παράδειγμα οι πληροφορίες που πρέπει να ανταλλαχθούν εξαρτώνται από κάποια εξωτερική πηγή (πχ. input από χρήστη). Επίσης, μέσω των protocol buffers μπορεί να γίνει εύκολα ο ορισμός των διαφόρων υπηρεσιών, μεθόδων και μηνυμάτων στα .proto αρχεία. Το gRPC έχει τη δυνατότητα να παράγει αυτόματα τον κώδικα υλοποίησης των RPC κλήσεων, ανεξάρτητα από τη γλώσσα και την πλατφόρμα που χρησιμοποιείται από το υπόλοιπο σύστημα. Με αυτόν τον τρόπο, μπορούν να επικοινωνήσουν εύκολα εφαρμογές υλοποιημένες σε διαφορετικές γλώσσες, όπως στην περίπτωση κάποιων από τις υλοποιήσεις της παρούσας διπλωματικής, όπου, για παράδειγμα, ο server αναπτύχθηκε σε Go ενώ ο client σε C++.

Το gRPC προσφέρει επίσης πλεονεκτήματα που αφορούν την επίδοση ενός συστήματος, καθώς έχει τη δυνατότητα να εκτελεί μεγάλο πλήθος κλήσεων σε πολύ μικρό χρονικό διάστημα, όπως φάνηκε από τις μετρήσεις των πειραμάτων που εκτελέστηκαν (1000 rpc σε ~600000 microseconds), κάτι το οποίο οφείλεται στη χρήση των protocol buffers και του HTTP/2 πρωτοκόλλου. Τα protocol buffers σειριοποιούν τα μηνύματα στην πλευρά του πελάτη και του διακομιστή με γρήγορο ρυθμό, με αποτέλεσμα να προκύπτουν μικρά και συμπαγή οφέλιμα φορτία στα HTTP/2 πλαίσια. Το HTTP/2 πρωτόκολλο συμβάλλει στην απόδοση του gRPC, προσφέροντας πολυπλεξία των rpc στην ίδια TCP σύνδεση, όπως φάνηκε στην απεικόνιση των πακέτων με τη χρήση του Wireshark, μειώνοντας έτσι το overhead που θα δημιουργούταν από την εδραίωσης μιας νέας σύνδεσης για κάθε κλήση, καθώς και την ύπαρξη φαινομένων head-of-line blocking, στα οποία μικρά αιτήματα χρειάζεται να περιμένουν την ολοκλήρωση μεγάλων αιτημάτων πρωτού εκτελεστούν.

Η επίδοση του gRPC framework μπορεί να επηρεαστεί από διάφορες παραμέτρους, όπως είναι το πλήθος των νημάτων που εκτελούν τις rpc κλήσεις, ο αριθμός των cpu cores που χρησιμοποιεί ο πελάτης, ο συνολικός αριθμός των rpc κλήσεων που εκτελούνται, το είδος και το μέγεθος των μηνυμάτων που ανταλλάσσονται. Συγκεκριμένα, όσο αυξάνεται ο αριθμός των ταυτόχρονων νημάτων που καλούν τις rpc μεθόδους στην πλευρά του client, ο μέσος χρόνος μετάδοσης των κλήσεων αυτών αυξάνεται, γεγονός που οφείλεται στην αύξηση της πολυπλεξίας στο gRPC κανάλι. Ωστόσο η συνολική απόδοση γίνεται καλύτερη εφόσον μειώνεται ο συνολικός χρόνος εκτέλεσης των κλήσεων και αυξάνεται η μετρική του αριθμού των κλήσεων που εκτελούνται ανά δευτερόλεπτο. Η βελτίωση των μετρικών αυτών παρουσιάζει έναν κορεσμό, καθώς κατά την αύξηση των νημάτων σε περισσότερα από 150, παρατηρείται μία μικρή αύξηση του συνολικού χρόνου εκτέλεσης και αντίστοιχα μείωση των αιτημάτων που εξυπηρετούνται ανά δευτερόλεπτο. Με την αύξηση του αριθμού των cpu cores του πελάτη υπάρχει βελτίωση της επίδοσης του συστήματος, και πάλι όμως υπάρχει κορεσμός κατά την αύξηση σε περισσότερα από 4 cpu cores. Όσον αφορά στο συνολικό αριθμό των αιτημάτων που πραγματοποιούνται, όσο περισσότερα είναι αυτά τόσο μειώνεται το average latency και ο μέσος χρόνος εκτέλεσης ενός rpc, ενώ αύξηση παρουσιάζουν ο συνολικός χρόνος εκτέλεσης και τα requests per second, το σύστημα δηλαδή εμφανίζει καλύτερη επίδοση. Τέλος, οι μετρικές απόδοσης του συστήματος

βελτιώνονται όσο μικρότερο είναι το μέγεθος των μηνυμάτων που ανταλλάσσονται ανάμεσα στον πελάτη και τον διακομιστή.

Παρά τα πλεονεκτήματα και την υψηλή επίδοση που παρουσιάζει το gRPC framework, δεν καλύπτει όλες τις ανάγκες που θα μπορούσε να έχει ένας πελάτης, όπως είναι για παράδειγμα η απόδοση προτεραιοτήτων στα rpc calls και η ιεράρχηση αυτών με βάση την προτεραιότητά τους. Αυτό συμβαίνει επειδή τα streams που εξυπηρετούν τα rpc calls είναι όλα ισότιμα μεταξύ τους, οπότε πολυπλέκονται ισόποσα στο gRPC κανάλι και στα TCP πακέτα.

Αποτέλεσμα του παραπάνω προβλήματος, είναι η εκτέλεση σημαντικών rpc να καθυστερεί λόγω της αποστολής λιγότερο σημαντικών rpc στην ίδια TCP σύνδεση. Μία λύση θα μπορούσε να είναι η αποστολή πακέτων rpc διαφορετικών προτεραιοτήτων μέσω διαφορετικής σύνδεσης, δηλαδή η δημιουργία μίας νέας TCP σύνδεσης για κάθε νέα προτεραιότητα. Με αυτόν τον τρόπο, τα rpc διαφορετικών προτεραιοτήτων θα αποστέλλονται χωρίς να επηρεάζει και να καθυστερεί η μία προτεραιότητα την άλλη. Ένα πρόβλημα σε αυτή την προσέγγιση είναι η δέσμευση παραπάνω πόρων και επομένως η μείωση του quality of service.

Μια διαφορετική προσέγγιση που εξετάστηκε στο πλαίσιο της συγκεκριμένης διπλωματικής είναι η προσθήκη ενός scheduler component στην πλευρά του πελάτη, το οποίο θα ελέγχει την δρομολόγηση των rpc κλήσεων στο gRPC κανάλι, ώστε η TCP σύνδεση να χρησιμοποιείται από μία μόνο προτεραιότητα κάθε φορά, την εκάστοτε μεγαλύτερη, και να μην υπάρχει καθυστέρηση λόγω της πολυπλεξίας. Η λύση αυτή προσθέτει μία χρονική επιβάρυνση λόγω των καθυστερήσεων στη διαδικασία της δρομολόγησης, ωστόσο εάν αυτή η επιβάρυνση είναι σχετικά μικρή, μικρότερη δηλαδή από κάποιο επιθυμητό όριο, η συγκεκριμένη προσέγγιση είναι προτιμότερη εφόσον δεν δεσμεύει παραπάνω resources από όσα χρειάζονται.

Στην παρούσα διπλωματική εξετάστηκαν διάφορες υλοποιήσεις του ζητούμενου scheduler component. Η πρώτη υλοποίηση έγινε σε C++ με τη χρήση processes, τα οποία αναλαμβάνουν να διεκπεραιώσουν τα rpc που πρέπει να εκτελεστούν. Ο μέσος χρόνος εκτέλεσης των rpc μειώνεται με την αύξηση της προτεραιότητάς τους, με τα rpc μεγάλων προτεραιοτήτων να εκτελούνται πιο γρήγορα σε σχέση με την απουσία χρονοδρομολογητή, κάτι το οποίο είναι και το ζητούμενο. Με την προσθήκη του scheduler component όμως προστίθεται αρκετά μεγάλη χρονική καθυστέρηση, με το συνολικό χρόνο εκτέλεσης σχεδόν να τριπλασιάζεται. Η υλοποίηση αυτή, χρησιμοποιεί sockets και σήματα για την επικοινωνία μεταξύ των διεργασιών, κάτι το οποίο εισάγει καθυστερήσεις στην όλη διαδικασία, ενώ εξαρτάται αρκετά από τη διαθεσιμότητα του kernel και το λειτουργικό σύστημα του μηχανήματος του πελάτη.

Η δεύτερη και η τρίτη υλοποίηση του χρονοδρομολογητή, έγινε σε C++ και χρήση νημάτων, αντί για διεργασιών. Πειράματα που έγιναν πάνω στις συγκεκριμένες υλοποιήσεις, έδειξαν ότι ενώ υπάρχει μείωση του μέσου χρόνου εκτέλεσης των rpc με την αύξηση της προτεραιότητάς τους, εισάγονται πολύ μεγάλες καθυστερήσεις από τη δρομολόγηση. Οι καθυστερήσεις αυτές έχουν ως αποτέλεσμα ακόμα και τα rpc με τη μέγιστη δυνατή προτεραιότητα να εκτελούνται πιο αργά σε σχέση με το πως θα εκτελούνταν χωρίς την ιεράρχηση προτεραιοτήτων από τον χρονοδρομολογητή. Η χρονική επιβάρυνση με την προσθήκη χρονοδρομολογητή οφείλεται εν μέρει στους χρόνους αναμονής των νημάτων μέχρι να αποκτήσουν κλειδώματα για να έχουν πρόσβαση σε πόρους που βρίσκονται στην κοινή τους μνήμη. Συμπεραίνεται, επομένως, ότι οι υλοποιήσεις σε C++ με threads δεν είναι αποδοτικές.

Οι δύο τελευταίες υλοποιήσεις που αναπτύχθηκαν ήταν σε Go με τη χρήση goroutines για την εκτέλεση των διαφόρων rpc. Η πρώτη υλοποίηση από τις δύο εμφάνισε παρόμοια επίδοση με την υλοποίηση της C++ με processes. Κάθε προτεραιότητα εμφάνιζε μικρότερους χρόνους εκτέλεσης σε σχέση με τις μικρότερές της και μεγαλύτερους σε σχέση με τις μεγαλύτερες, με τα rpc των κρίσιμων, μεγαλύτερων, προτεραιοτήτων να εκτελούνται πιο γρήγορα σε σχέση με το πώς θα εκτελούνταν χωρίς την ύπαρξη χρονοδρομολογητή. Και σε αυτή την υλοποίηση όμως εισάγονται καθυστερήσεις από την εναλλαγή των rpc στο gRPC κανάλι. Στην τελευταία υλοποίηση όμως, οι συνολικοί χρόνοι εκτέλεσης είναι σχεδόν ίδιοι είτε χρησιμοποιείται χρονοδρομολογητής είτε όχι. Με την προσθήκη του scheduler, ωστόσο, τα rpc μεγαλύτερων προτεραιοτήτων εκτελούνται σημαντικά πιο γρήγορα σε σχέση με την απουσιά χρονοδρομολογητή.

Με τη χρήση της τελευταίας, λοιπόν, υλοποίησης χρονοδρομολογητή, η οποία είναι και η αποδοτικότερη, ο πελάτης έχει τη δυνατότητα να διαχωρίσει τα rpc που θέλει να στείλει με βάση το πόσο γρήγορα θέλει να τα εκτελέσει αποδίδοντάς τους προτεραιότητες. Στη συνέχεια ο χρονοδρομολογητής αναλαμβάνει να καθορίσει τη σειρά με την οποία θα σταλούν τα πακέτα των rpc στο gRPC κανάλι, ώστε κάθε κλήση να ολοκληρώνεται με ρυθμό ανάλογο της προτεραιότητάς της. Η ιεράρχηση αυτή των προτεραιοτήτων πραγματοποιείται με πολύ μικρή επιπλέον χρονική επιβάρυνση και χωρίς να χρειάζεται η δημιουργία παραπάνω TCP συνδέσεων και η δέσμευση παραπάνω πόρων.

Ένα τελευταίο θέμα που εξετάστηκε είναι ο κίνδυνος δημιουργίας starvation κατά τη χρήση του priority based αλγόριθμου του χρονοδρομολογητή που αναπτύχθηκε. Για την αποφυγή του προβλήματος της συνεχούς αναμονής κλήσεων χαμηλών προτεραιοτήτων, προστέθηκε στην πρώτη υλοποίηση του χρονοδρομολογητή με νήματα σε C++ ένας μηχανισμός time-out, ώστε κάθε νήμα να παραμένει αδρανές για ένα μέγιστο καθορισμένο χρόνο. Τα αποτελέσματα της υλοποίησης αυτής έδειξαν ότι κατά τον καθορισμό σχετικά μικρών χρόνων time-out, μικρότερων από τους μέσους χρόνους εκτέλεσης των κλήσεων στην απλή υλοποίηση χρονοδρομολογητή, ο μέσος χρόνος εκτέλεσης των rpc με χαμηλή προτεραιότητα μειώνεται, ενώ των rpc με υψηλή προτεραιότητα αυξάνεται, σε σχέση με τη μη χρήση time-out μηχανισμού. Το γεγονός αυτό οφείλεται στην καθυστέρηση που εισάγουν οι κλήσεις των οποίων λήγει ο χρόνος αναμονής και ξεκινούν να εκτελούνται πολυπλέκοντας τα πακέτα τους με πακέτα κλήσεων μεγαλύτερων προτεραιοτήτων που δεν έχουν προλάβει να ολοκληρωθούν. Επομένως ο μηχανισμός time-out προκαλεί μία καθυστέρηση στις υψηλές προτεραιότητες, ωστόσο μειώνει τις μεγάλες αποκλίσεις στους χρόνους εκτέλεσης των διαφόρων προτεραιοτήτων και διατηρεί τους χρόνους ολοκλήρωσης των rpc χαμηλών προτεραιοτήτων κάτω από κάποιο καθορισμένο όριο. Εάν μοναδικός στόχος του μηχανισμού είναι η αποφυγή του starvation, μπορούν να οριστούν αρκετά υψηλοί χρόνοι time-out ώστε να μην προκαλείται μεγάλη καθυστέρηση στις υψηλές προτεραιότητες.

Μελλοντικές επεκτάσεις

Η προσέγγιση που αναπτύχθηκε για το πρόβλημα της ιεράρχησης προτεραιοτήτων των gRPC μηνυμάτων στο πλαίσιο της παρούσας διπλωματικής εργασίας, θα μπορούσε να βελτιωθεί και να επεκταθεί περαιτέρω, τουλάχιστον ως προς τρεις κατευθύνσεις.

Η παρούσα χρονοδρομολόγηση των rpc στο κανάλι είναι καθαρά priority based, που σημαίνει ότι μοναδικό κριτήριο για το εάν θα εκτελεστεί ένα rpc είναι το αν έχει τη μέγιστη προτεραιότητα τη δεδομένη χρονική στιγμή. Σε περίπτωση ισοτιμίας προτεραιοτήτων, σε κάποιες υλοποιήσεις ως δεύτερο κριτήριο θεωρείται η χρονική στιγμή στην οποία δημιουργήθηκε η ανάγκη εκτέλεσης του κάθε rpc, ενώ στην τελευταία υλοποίηση του χρονοδρομολογητή υπάρχει πολυπλεξία των rpc με ίδια προτεραιότητα στο κανάλι. Στην τελευταία υλοποίηση έγινε μια αρχική προσθήκη ενός time-out μηχανισμού προκειμένου να αποφευχθεί το φαινόμενο του starvation. Ένα επόμενο βήμα θα ήταν η επέκταση του μηχανισμού αυτού και η υλοποίηση εναλλακτικών αλγορίθμων δρομολόγησης στο scheduler component, όπως αναλύθηκαν θεωρητικά σε προηγούμενο κεφάλαιο.

Στις παρούσες υλοποιήσεις, η δρομολόγηση πραγματοποιήθηκε μόνο στην πλευρά του client, με τον έλεγχο των πακέτων που αποστέλλονται μέσω του gRPC καναλιού. Ο διακομιστής στο gRPC αφέθηκε ελεύθερος να εξυπηρετεί όποιο αίτημα του γίνεται. Μια μελλοντική επέκταση, λοιπόν, θα ήταν η υλοποίηση ενός scheduler component και στην πλευρά του server. Το scheduler component σε αυτή τη περίπτωση θα ήταν ένας χρονοδρομολογητής που θα έλεγχε με ποιά σειρά ικανοποιούνται τα αιτήματα από τον server και στέλνονται πίσω στον client μέσω του καναλιού.

Μία τελευταία μελλοντική επέκταση, και ίσως η πιο δύσκολη στην υλοποίησή της, θα ήταν η προσθήκη του scheduler component στο gRPC framework. Στις υλοποιήσεις της παρούσας διπλωματικής, ο χρονοδρομολογητής αποτελεί μέρος του πελάτη και όχι του gRPC, το οποίο δεν γνωρίζει για την ιεράρχηση προτεραιοτήτων που συμβαίνει στα rpc πριν αυτά εισέλθουν στο gRPC κανάλι. Μία πιο σωστή και ολοκληρωμένη προσέγγιση, θα ήταν ο πελάτης να καλεί κανονικά τις rpc μεθόδους που επιθυμεί, εισάγοντας μία προτεραιότητα στην κάθε μία ως metadata. Στη συνέχεια το gRPC μελετώντας τις προτεραιότητες των rpc κλήσεων που καλείται να εκτελέσει, θα πραγματοποιούσε εκείνο δρομολόγηση αυτών ώστε οι πιο σημαντικές να προηγούνται ως προς την ολοκλήρωσή τους. Μια τέτοια υλοποίηση θα μπορούσε να αναπτυχθεί με τη χρήση των gRPC interceptors, οι οποίοι είναι components του gRPC framework που μπορούν να αναπτυχθούν προκειμένου να ελέγχουν τα μηνύματα στην πλευρά του client και στην πλευρά του server, πριν αυτά φύγουν στο κανάλι ή προωθηθούν στον server αντίστοιχα, προσθέτοντάς τους πληροφορίες και metadata.

Πηγαίος κώδικας

Η ανάπτυξη των υλοποιήσεων του χρονοδρομολογητή έγινε με χρήση του gRPC framework. Ο διακομιστής του συστήματος αναπτύχθηκε σε γλώσσα Go, ενώ υλοποιήθηκαν πελάτες σε Python, C++ και Go. Τα διάφορα scripts για την εκτέλεση των πειραμάτων γράφτηκαν σε Python.

Ο πηγαίος κώδικας μπορεί να βρεθεί στο παρακάτω δημόσιο αποθετήριο κώδικα της πλατφόρμας GitHub:

https://github.com/mariartc/gRPC

ΒΙΒΛΙΟΓΡΑΦΙΑ

gRPC Authors (2022) *Official gRPC documentation*, https://grpc.io/

Rosencrance L., Matturro B. (2021), *Remote Procedure Call (RPC)*, https://www.techtarget.com/searchapparchitecture/definition/Remote-Procedure-Call-RPC

RFC 7540 (2015) Hypertext *Transfer Protocol Version 2 (HTTP/2)*, https://datatracker.ietf.org/doc/html/rfc7540

Jean de Klerk (2018, July 3) *HTTP/2: Smarter at scale*, https://www.cncf.io/blog/2018/07/03/http-2-smarter-at-scale/

Rukshani Athapathu (2019) *HTTP/2 Flow Control*, https://medium.com/coderscorner/http-2-flow-control-77e54f7fd518

gRPC Authors (2022) *gRPC on HTTP/2 Engineering a Robust, High-performance Protocol*, https://grpc.io/blog/grpc-on-http2/

Wang X., Zhao H., Zhu J., *GRPC: a communication cooperation mechanism in distributed systems*, https://dl.acm.org/doi/abs/10.1145/155870.155881#

Wallarm, *The Concept Of GRPC*, https://www.wallarm.com/what/the-concept-of-grpc

Grigorik I., Surma (2016, September 29) *Introduction to HTTP/2*, https://web.dev/performance-http2/

Google developers, *Protocol Buffers*, https://developers.google.com/protocol-buffers

gRPC Authors (2022) *gRPC Load Balancing*, https://grpc.io/blog/grpc-load-balancing/

Wireshark Wiki, *gRPC*, https://wiki.wireshark.org/gRPC

gRPC Authors (2022) *Analyzing gRPC messages using Wireshark*, https://grpc.io/blog/wireshark/

Ben Ibinson (2018) *Our experience designing and building gRPC services*, https://www.bugsnag.com/blog/using-grpc-in-production

The Edgehogs (2021, June 16) *A Guide to gRPC and Interceptors*, https://edgehog.blog/a-guide-to-grpc-and-interceptors-265c306d3773

AfterAcademy (2019, November 4) *What is Starvation and Aging?*, https://afteracademy.com/blog/what-is-starvation-and-aging