ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Runtime Resource Management on Serverless Computing Architectures

Ιωάννης Φακίνος
Α.Μ. : 03116158

Επιβλέπων **:** Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Αθήνα,
Φεβρουάριος 2022

ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ
ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ
ΤΟΜΕΑΣ ΤΕΧΝΟΛΟΓΙΑΣ ΠΛΗΡΟΦΟΡΙΚΗΣ ΚΑΙ ΥΠΟΛΟΓΙΣΤΩΝ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

Runtime Resource Management on Serverless Computing Architectures

Ιωάννης Φακίνος
Α.Μ. : 03116158

Επιβλέπων **:** Δημήτριος Ι. Σούντρης
Καθηγητής ΕΜΠ

Τριμελής Επιτροπή Εξέτασης

(Υπογραφή)	(Υπογραφή)	(Υπογραφή)
....	....	....
Δημήτριος Σούντρης	Παναγιώτης Τσανάκας	Σωτήριος Ξύδης
Καθηγητής	Καθηγητής	Αναπληρωτής Καθηγητής
ΕΜΠ	ΕΜΠ	Χαροκόπειο Πανεπιστήμιο

Ημερομηνία Εξέτασης:
04 Μαρτίου 2022

Copyright ©- All rights reserved Ιωάννης Φακίνος, 2022.
Με επιφύλαξη κάθε δικαιώματος.

Απαγορεύεται η αντιγραφή, αποθήκευση και διανομή της παρούσας εργασίας, εξ΄ ολοκλήρου ή τμήματος αυτής, για εμπορικό σκοπό. Επιτρέπεται η ανατύπωση, αποθήκευση και διανομή για σκοπό μη κερδοσκοπικό, εκπαιδευτικής ή ερευνητικής φύσης, υπό την προϋπόθεση να αναφέρεται η πηγή προέλευσης και να διατηρείται το παρόν μήνυμα. Ερωτήματα που αφορούν τη χρήση της εργασίας για κερδοσκοπικό σκοπό πρέπει να απευθύνονται προς τον συγγραφέα.

(Υπογραφή)

....
Ιωάννης Φακίνος
Διπλωματούχος Ηλεκτρολόγος Μηχανικός και Μηχανικός Υπολογιστών Ε.Μ.Π.
©2022 - All rights reserved.

In memory of my grandmother,
Rodanthe Maria.

Περίληψη

Η έννοια *Συναρτήσεις-σαν-Υπηρεσία* μπορεί να χαρακτηριστεί ως το μέλλον της νεφοϋπολογιστικής, αποτελώντας μία αναδυόμενη φιλοσοφία που απαλλάσσει τον προγραμματιστή από διαχειριστικά θέματα. Σύμφωνα με αυτό το μοντέλο, η γνωστή μονολιθική αρχιτεκτονική κάθε εφαρμογής απαιτείται να αντικατασταθεί από έναν γράφο ανεξάρτητων και εφήμερων συναρτήσεων. Μάλιστα, αυτές είθισται να ενεργοποιούνται και να εκκινούν έπειτα από εξωτερικά γεγονότα, όπως την αποστολή διαπιστευτηρίων από έναν χρήστη. Ταυτόχρονα, από την πλευρά του παρόχου, προβλήματα όπως η διαθεσιμότητα, η κλιμακωσιμότητα, η κατανομή φορτίου κ.α. πρέπει να αντιμετωπιστούν χωρίς προϋπάρχουσα γνώση της συμπεριφοράς και των υπολογιτικών απαιτήσεων του κώδικα των χρηστών τους.

Σε αυτά τα πλαίσια, οι πάροχοι προσφέρουν διάφορα πακέτα χρεώσεων ανάλογα με τους διαθέσιμους πόρους (επεξεργαστής, μνήμη, αποθηκευτικός χώρος κ.τ.λ.) για τα κοντέινερ, που αξιοποιούνται για την εκτέλεση των συναρτήσεων αυτών. Δυστυχώς, αυτά αναγκαστικά συνυπάρχουν με άλλα ομοειδή μέσα σε ένα μηχάνημα οικοδεσπότη πεπερασμένων πόρων. Η προαναφρθείσα, λοιπόν, ανεπίβλεπτη τεχνική κατανομής πόρων δεν εγγυάται καμία καλώς ορισμένη ποιότητα υπηρεσιών αναφορικά με τον χρόνο εκτέλεσης των συναρτήσεων και συνόλων αυτών.

Κατά καιρούς, διάφορες εναλλακτικές έχουν προταθεί για την επίλυση αυτής της αδυναμίας και την παροχή της *Ποιότητας Υπηρεσίας*, οι οποίες αν και επιτυγχάνουν ενδιαφέροντα αποτελέσματα, υστερούν σε συμβατότητα με υπάρχοντες βιβλιοθήκες/εργαλεία ή επικεντρώνονται σε περιορισμένες εφαρμογές. Σε αυτήν τη διπλωματική, εξερευνούμε το Ρολόι Ακολουθίας, ένα εργαλείο ελέγχου χρονοκαθυστέρησης το οποίο δυναμικά παρακολουθεί τις κλήσεις συναρτήσεων σε μία συστοιχία υπολογιστών και επιτρέπει την εκτέλεση συναρτησιακών ακολουθιών εντός του προκαθορισμένου χρονικού ορίου. Δύο μεθοδολογίες ελέγχου ακολουθήθηκαν, με την μία να πετυχαίνει έως και 82% μείωση στη σοβαρότητα των χρονικών παραβιάσεων και σε ορισμένες περιπτώσεις να τις εξαλείφει πλήρως.

Λέξεις Κλειδιά— υπολογιστική χωρίς διακομιστή, Συναρτήσεις-σαν-Υπηρεσία, ποιότητα υπηρεσίας, ακολουθίες συναρτήσεων, Κυβερνήτης, OpenWhisk

Abstract

Function as a service or FaaS represents the next frontier in the evolution of cloud computing being an emerging paradigm that removes the burden of configuration and management issues from the developer's perspective. However, this relatively new technology, like any other, surely comes with its caveats. For starters, the whole well known monolithic approach has to be replaced by a DAG of standalone, small, stateless, event driven components called functions. At the same time, at the cloud provider's side, problems like availability, load balancing, scalability and others has to be resolved without ever knowing the functionality, behavior or resource requirements of their tenants code.

In this context, vendors offer certain billing plans concerning the available resources (CPU, memory & cold storage size etc) of the containers/sandboxes that functions run on. Unfortunately, these containers have to coexist with others in a runtime of a host with finite shared resources. Thus, with the latter passive resource allocation technique there's no guarantee of a well defined quality of service or QoS in regards to functions' and function sets' latency.

Various efforts have been made towards the holy grail of QoS, but they either lack in compatibility with existing serverless frameworks, or they are limited in specific applications. In this thesis, we explore Sequence Clock, a latency targeting tool that actively monitors serverless invocations in a cluster and offers execution of sequential chain of functions, also known as pipelines or sequences, while achieving the targeted time latency. It was developed in Go, wrapped as a helm chart (a packaging format for Kubernetes) and focuses on OpenWhisk deployments on top of Kubernetes' clusters. Two regulation methods were utilized, with one of them achieving up to a 82% decrease in the severity of time violations and in some cases even eliminating them completely.

Keywords— serverless computing, faas, QoS, target latency, sequences, OpenWhisk, Kubernetes

Ευχαριστίες

Αρχικά, θα ήθελα να εκφράσω την ευγνωμοσύνη μου στον επιβλέποντα Καθηγητή Δημήτριο Ι. Σούντρη του Εθνικού Μετσόβιου Πολυτεχνείου. Αφενώς, μου προσέφερε τη δυνατότητα και την ευκαιρία να αποτελέσω μέλος του Εργαστηρίου Μικροϋπολογιστών και Ψηφιακών Συστημάτων (MicroLab), εκπονώντας την διπλωματική μου εργασία σε αυτό. Αφετέρου, αφιέρωσε εκτενή χρόνο στην διαδικασία αναζήτησης ενός θέματος, που ταιριάζει με τα ερευνητικά μου ενδιαφέροντα και αρμόζει στις κλίσεις και ικανότητές μου.

Ταυτόχρονα, η δεδομένη εργασία δεν θα μπορούσε να πραγματοποιηθεί χωρίς την συνεισφορά και καθοδήγηση των μελών της ομάδας Edge & Cloud Computing, του εν λόγω εργαστηρίου, η οποία και απαρτίζεται από τα μέλη Σωτήριο Ξύδη, Δημοσθένη Μασούρο & Αχιλλέα Τζενετόπουλο. Οι μακροσκελείς συναντήσεις και ο καταιγισμός ιδεών και προτάσεων, που έλαβαν χώρα σε αυτές, συντέλεσαν στην αρτιότητα του ερευνητικού αποτελέσματος, αλλά και στην προσωπική μου αυτοβελτίωση γύρω από τον χώρο του serverless computing.

Όσον αφορά το στενό οικογενειακό μου περιβάλλον, θα ήθελα να ευχαριστήσω τους γονείς μου, οι οποίοι με στήριξαν καθ΄ όλη τη διάρκεια των σπουδών μου στο αρχαιότερο τεχνολογικό ίδρυμα της χώρας. Τέλος, επιθυμώ να αναφερθώ στην κοπέλα μου, Θεοδώρα, της οποίας η ψυχολογική και συναισθηματική υποστήριξη ήταν η κινητήριος δύναμη για να πετύχω τους στόχους μου.

Acknowledgments

Firstly, I would like to express my gratitude to my supervisor, Professor Dimitrios I. Soudris of the National Technical University of Athens, who offered me the privilege and opportunity to be part of the Microprocessors and Digital Systems Laboratory (Microlab). Not only that, he also devoted a significant amount of his time, in order for me to find a subject, that resonated with my interest of research and corresponded to my skills and personal capabilities.

In addition, this thesis couldn't be developed, if it wasn't for the contribution and guidance of the members of Edge & Cloud Computing team, which consists of Sotirios Xydis, Dimosthenis Masouros & Achilleas Tzenetopoulos. Time consuming meetings and non trivial discussions with these fine colleagues were the main reasons that lead to this flawless scientific result.

As far as my closest persons are concerned, I would like to thank my parents, that supported me throughout my studies in the oldest higher education institution of the country. Last but not least, it is proper to mention my girlfriend, Theodora, whose psychological and emotional support comprised the driving force that helped me achieve my goals.

Contents

List of Figures

List of Tables

Εκτεταμένη Περίληψη

1 Εισαγωγή

Την σήμερον ημέρα, όλο και περισσότερες υπηρεσίες προωθούνται και εκτελούνται στο Υπολογιστικό Νέφος. Με τον συνεχώς αυξανόμενο ανταγωνισμό και τον χρόνο διάθεσης στην αγορά να διαδραματίζει καίριο ρόλο σε εταιρείες συνιφασμένες με το λογισμικό, είναι απολύτως φυσιολογικό προγραμματιστές και μηχανικοί να αναζητούν λύσεις που μειώνουν τον χρόνο υλοποίησης και τους επιτρέπουν να αφοσοιωθούν στην επιχειρηματική λογική της εκάστοτε εφαρμογής.

Με την πάροδο των χρόνων έχουν προκύψει διάφορα μοντέλα της νεφοϋπολογιστικής, καθένα από τα οποία προσφέρει διαφορετικά επίπεδα αφαίρεσης και διαφορετικό βαθμό διαχωρισμού της διαχείρισης υποδομών από την ανάπτυξη εφαρμογών, ήτοι *Υποδομή-σαν-Υπηρεσία* (*Infrastructure-as-a-Service, IaaS*) [1, 2, 3], *Πλατφόρμα-σαν-Υπηρεσία* (*Platform-as-a-Service, PaaS*) [2, 3, 4, 5], *Λογισμικό-σαν- Υπηρεσία* (*Software-as-a-Service, SaaS*) [2] και πλέον *Συναρτήσεις- σαν-Υπηρεσία* (*Function-as-a-Service, FaaS*). Για το τελευταίο, ισοδύναμος όρος αποτελεί και η *Υπολογιστική χωρίς Διακομιστή* (*Serverless Computing*), ενώ έχει χαρακτηριστικεί από πολλούς και ως ένα καινούριο προγραμματιστικό μοντέλο από μόνο του [6].

Συγκεκριμένα, το μοντέλο *Συναρτήσεις-σαν-Υπηρεσία* έχει χαρακτηριστεί ως μία αναδυώμενη αρχιτεκτονική ανάπτυξης[1]εφαρμογών, η οποία αποκρύπτει τη διαχείριση των διακομιστών από τους ενοικιαστές του υπολογιστικού νέφους [7]. Προφανώς, ένα παρόμοιο επίπεδο αφαίρεσης προσφέρεται σε διαφορετικό βαθμό και στα υπόλοιπα μοντέλα των εταιριών προμηθευτών του υπλογιστικού νέφους. Το χαρακτηριστικό που κάνει το μοντέλο *Συναρτήσεις-σαν-Υπηρεσία* να ξεχωρίζει είναι το γεγονός πως η συμβατική και ευρέως γνωστή λογική μίας μονολιθικής εφαρμογής πρέπει να αντικατασταθεί από ένα σύνολο εφήμερων και χωρίς κατάσταση συναρτήσεων. Κάθε συνάρτηση εκτελείται σε ένα ξεχωριστό περιβάλλον, γνωστό και ως κοντέινερ, το οποίο είναι επίσης βραχύβιο. Αυτό σημαίνει ότι ένα τέτοιο κοντέινερ δημιουργείται, όταν η κλήση της αντίστοιχης συνάρτησης πραγματοποιηθεί και αντίστοιχα αφαιρείται αφότου το αντίστοιχο αίτημα έχει εξυπηρετηθεί.

[1]Έχει χρησιμοποιηθεί ο όρος *ανάπτυξη* ως μετάφραση του αγγλικού όρου *deployment*, ο οποίος δυστυχώς δεν είναι απόλυτα ισοδύναμος.

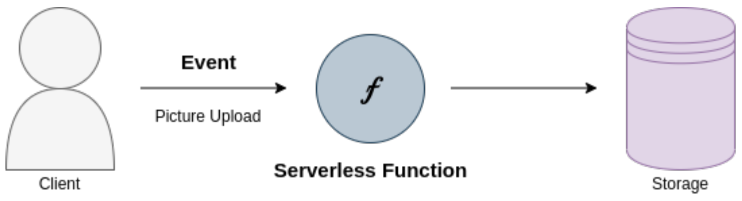


Figure 1: Απλοποιημένο Παράδειγμα *Συνάρτησης-σαν-Υπηρεσία*.

Με αυτή τη λογική, οι προγραμματιστές χρεώνονται ανά κλήση συνάρτησης και δεν πληρώνουν αδρανείς πόρους. Επιπλεόν, αυτή η προσσέγγιση προσφέρει μηχανισμούς αυτόματης κλιμακωσιμότητας, διαχείριση σφαλμάτων και διαθεσιμότητα σε πολλές γεωγραφικές περιοχές. Παρ΄ όλα αυτά, κανείς μπορεί εύκολα να διατυπώσει ανησυχίες σχετικές με την ασφάλεια και την ποιότητα απομόνωσης, καθώς οι παρεμβολές μεταξύ συναρτήσεων των ίδιων και διαφορετικών ενοικιαστών είναι αναπόφευκτες [7].

Σε κάθε περίπτωση, οι *Συναρτήσεις-σαν-Υπηρεσία* παραμένουν στο προσκήνιο και αποτελούν μία εντυπωσιακή τεχνολογία, εάν αξιοποιούνται καταλλήλως, ήτοι σε εφαρμογές με διακοπτόμενη λειτουργία όπου η συντήρηση μακρόβιων περιβάλλοντων είναι κοστοβόρα [8].

Από τον Μονόλιθο στις Ακολουθίες

Όπως προαναφέραμε, στην *Αρχιτεκτονική χωρίς Διακομιστή*, μία εφαρμογή πρέπει να μεταφραστεί σε σύνολο από ανεξάρτητες συναρτήσεις. Αυτές καλούν η μία την άλλη, επικοινωνούν μεταξύ τους και ανταλάσουν δεδομένα, με την διαδικασία αυτή να μπορεί να αναπαρασταθεί ως ένα μαθηματικό γράφο. Η πιο απλή μορφή μίας τέτοιας διαδικασίας αποτελεί η ακολουθιακή εκτέλεση τέτοιων συναρτήσεων και αντισοιχεί σε μία λίστα. Από δω και στο εξής θα αποκαλούμε τέτοιους γράφους *ακολουθίες*.

Όσον αφορά τον χρόνο εκτέλεσης τέτοιων γράφων, δεν υπάρχει καμία εγγύηση (με τα μέχρι στιγμής γνωστά εργαλεία) ότι θα είναι κάτω από ένα προκαθορισμένο όριο, κάτι το οποίο θα μπορούσε να μεταφραστεί ως *Ποιότητα Υπηρεσίας* (*Quality of Service, QoS*). Για να επιτευχθεί κάτι τέτοιο, κρίνεται απαραίτητη η χρήση ενός πιο εξελιγμένου συστήματος διαχείρισης πόρων.

Περιεχόμενο Διπλωματικής

Στην παρούσα διπλωματική, παρουσιάζεται το Sequence Clock (Ρολόι Ακολουθίας), ως μία πιθανή λύση στις παραβιάσεις του QoS στον χρόνο εκτέλεσης ακολουθιών συναρτήσεων τύπου serverless. Το Ρολόι Ακολουθίας αποτελεί ένα κατανεμημένο εργαλείο ελέγχου, το οποίο παρακολουθεί όλες τις κλήσεις συναρτήσεων τύπου serverless σε μία συστοιχία υπλογιστών και προσπαθεί να καθορίσει τον χρόνο εκτέλεσης στο επίπεδο ακολουθίας. Στο πυρήνα του αξιοποιεί μία *άπληστη* προσέγγιση υπολογισμού του σφάλματος (γνωστό και ως slack) από τον στόχο, μαζί με έναν ελεγκτή τύπου PID για τον υπολογισμό της ποσόστωσης της Κεντρικής Μονάδας Επεξεργασίας (ΚΜΕ) ως μεταβλητή ελέγχου.

2 Θεωρητικό Υπόβαθρο

Ακολουθεί μία παρουσίαση των τεχνολογιών και των εργαλείων που μελετήθηκαν και αξιοποιήθηκαν κατά τη διάρκεια αυτής της διπλωματικής εργασίας.

2.1 Κυβερνήτης

Ο Κυβερνήτης (Kubernetes) είναι μία πλατφόρμα ανοικτού κώδικα για τη διαχείριση εφαρμογών εγκιβωτισμένων σε κοντέινερ, η οποία προσφέρει δυνατότητα διαμόρφωσης αυτών και αυτοματισμούς [9]. Εν περιλήψει, προσφέρει διαλακτικότητα στην λειτουργία κατανεμημένων συστημάτων. Μεταξύ άλλων, δυνατοτήτες που διαθέτει αποτελούν η ενορχίστρωση αποθηκευτικών μέσων, η επαναφορά εφαρμογών σε παλαιότερες εκδόσεις, η κατανομή υπολογιστικού φορτίου κ.α. Όλα αυτά είναι εφικτά με την διαμόρφωση αρχείων ρυθμίσεων και με τη χρήση εργαλείων της γραμμής εντολών. Επιπλεόν, αναλαμβάνει την αποθηκεύση και διαχείριση ευαίσθητων πληροφοριών, όπως διαπιστευτήρια χρηστών, με τη χρήση συνιστωσών που αποκαλεί *μυστικά* (secrets) χωρίς να υπάρχει ο φόβος διαρροής στον κώδικα [10, 9].

Μία συστοιχία διακομιστών, στην οποία έχει τοποθετηθεί ο Κυβερνήτης, αποτελείται από ένα μη κενό σύνολο υπολογιστών, οι οποίοι χαρακτηρίζονται ως κόμβοι εργάτες (nodes) και αξιοποιούνται για την φιλοξενεία των απαραίτητων συνιστωσών των εφαρμογών. Από την άλλη, ένα σύνολο από συνιστώσες ελέγχου διαχειρίζονται αυτούς τους κόμβους, ενώ ταυτόχρονα εντοπίζουν και ανταποκρίνονται σε εξωτερικά γεγονότα, παίρνοντας καθολικές αποφάσεις [11, 4].

Αντικείμενα του Κυβερνήτη

Τα αντικείμενα ή αλλιώς πόροι (να μην γίνεται σύγχηση με την έννοια των πόρων ενός κόμβου) του Κυβερνήτη αποτελούν οντότητες που χαρακτηρίζουν και περιγράφουν την κατάσταση του συστήματος. Αφότου δημιουργηθούν, ο Κυβερνήτης προσπαθεί να εξασφαλίσει την υπαρξή τους. Η δημιουργία, μεταβολή και διαγραφή αυτών των αντικειμένων είναι εφικτή μέσω της Διεπαφής Προγραμματισμού Εφαρμογών (API) του Κυβερνήτη. Όλες οι σχετικές πληροφορίες για αυτές τις ενέργειες τοποθετούνται σε αρχεία μορφότυπου yaml , τα οποία μπορούν να αξιοποιηθούν από διάφορα εργαλεία όπως το *kubectl* (η διεπαφή γραμμής εντολών του Κυβερνήτη). Ένα παράδειγμα τέτοιου αρχείου έχει τοποθετηθεί παρακάτω.

```
    apiVersion: apps/v1
    kind: Deployment
```

```yaml
  metadata:
    name: nginx-deployment
  spec:
    selector:
      matchLabels:
        app: nginx
    replicas: 2 # tells deployment to run 2 pods matching the template
    template:
      metadata:
        labels:
          app: nginx
      spec:
        containers:
        - name: nginx
          image: nginx:1.14.2
          ports:
          - containerPort: 80
```

Κάθε αντικείμενο του Κυβερνήτη περιλαμβάνει δύο πεδία, που το διαμορφώνουν, ήτοι το *spec* και το *status*. Το πρώτο πρέπει να οριστεί κατά την δημιουργεία του, περιγράφοντας πληροφορίες για την επιθυμητή κατάσταση του αντικειμένου. Αντιθέτως, το *status* περιγράφει την τρέχουσα κατάσταση του πόρου, η οποία ενημερώνεται από τον ίδιο τον Κυβερνήτη και τις συνιστώσες του. Στην ουσία, οι συνιστώσες ελέγχου εργάζονται συνεχώς ώστε η πραγματική κατάσταση να ταυτίζεται με την επιθυμητή [12].

Επιγραμματικά, ορισμένα από τα πιο γνωστά αντικείμενα του Κυβερνήτη είναι τα ακόλουθα:

- **Ονοματοχώροι (Namespaces)**: Στοχεύουν στην απομόνωση διαφορετικών συνόλων από πόρους μέσα στην ίδια συστοιχία [13].
- **Pod**: Η μικρότερη υπολογιστική μονάδα που μπορεί να διαχειριστεί ο Κυβερνήτης [14]. Στην ουσία αποτελείται από ένα σύνολο ενός ή περισσοτέρων κονταίνερ.
- **Deployment**: Μπορεί να χαρακτηριστεί ως ένα σύνολο από υπολογιστικούς πόρους, παραδείγματος χάρην ένα σύνολο από ξεχωριστά pods [15].
- **DeamonSet**: Ένα τέτοιο αντικείμενο εξασφαλίζει ότι κάθε κόμβος ή ένα υποσύνολο κόμβων φιλοξενούν ένα αντίγραφο από ένα ορισμένο pod [16].
- **Υπηρεσίες (Services)**: Αποτελούν μία αφαίρεση που ορίζει ένα νοητό σύνολο από pods και τον τρόπο επικοινωνίας οποιασδήποτε συνιστώσας εκτός αυτού του συνόλου με αυτά [17].
- **Μακρόβιοι/Επίμονοι Τόμοι (Persistent Volumes)**: Στον πυρήνα τους οι τόμοι αποτελούν κατάλογοι αρχείων, στους οποίους διάφορα pod έχουν πρόσβαση [18]. Οι μακρόβιοι/επίμονοι τόμοι αποτελούν ειδική κατηγορία αυτών, οι οποίοι δεν ανήκουν σε κανέναν ονοματοχώρο και υπάρχουν στο σύστημα ανεξαρτήτως από την ύπαρξη ενδεχόμενων pod, που έχουν πρόσβαση στα περιεχόμενά τους [19].

2.2 Apache OpenWhisk

Το Apache OpenWhisk αποτελεί μία κατανεμημένη πλατφόρμα υπολογιστικής χωρίς διακομιστή, που αναλαμβάνει την εκτέλεση συναρτήσεων ως απόκριση σε εξωτερικά γεγονότα [20]. Με την χρήση κοντέινερ τύπου Docker ως περιβάλλον για κάθε συνάρτηση, προσφέρει συμβατότητα με πληθώρα γνωστών εργαλειών, όπως Κυβερνήτης, OpenShift και Docker Compose [20, 21]. Οι συναρτήσεις, που αποκαλούνται actions, δύνανται να αναπτυχθούν χρησιμοποιώντας μία πληθώρα από γλώσσες προγραμματισμού, ενώ ταυτόχρονα προσφέρεται η δυνατότητα να ομαδοποιηθούν σε ακολουθίες, των οποίων η επίδοση επιχειρείται να βελτιωθεί από το Sequence Clock.

Αρχιτεκτονική

Το όλο σύστημα αποτελείται από τις ακόλουθες συνιστώσες, κάθε μία από τις οποίες τελεί έναν διαφορετικό ρόλο:

- **NGiNX**: Αποτελεί το σημείο εισόδου της κλήσης μίας συνάρτησης. Γενικά, το NGiNX είναι λογισμικό ανοικτού κώδικα, που αξιοποιείται ως διακομιστής ιστοσελίδων, διαμεσολαβιτής, εξισορροπητής φορτίου κ.τ.λ. [22].

- **Ελεγκτής (Controller)**: Λειτουργεί ως ο ρυθμιστής του συστήματος, καθώς προσφέρει την διεπαφή προγραμματισμού εφαρμογών τύπου REST για διαχείριση οντοτήτων που περιλαμβάνονται στο OpenWhisk. Μάλιστα αναλαμβάνει την επικοινωνία με την βάση δεδομένων CouchDB για την αυθεντικοποίηση και την εξιουσιοδότηση των χρηστών.

- **CouchDB**: Η CouchDB είναι ένα παράδειγμα μη σχεσιακής βάσης δεδομένων, που αξιοποιείται από το OpenWhisk για την καταγραφή διαπιστευτηρίων, μεταδεδομένων, ονοματοχώρων, καθώς επίσης και των ορισμών των συναρτήσεων. Οι εγγραφές των τελευταίων περιλαμβάνουν κυρίως τον κώδικα, τις προκαθορισμένες παραμέτρους εισόδου και τυχόν περιορισμούς στους απαιτούμενους υπολογιστικούς πόρους της συνάρτησης, οι οποίοι της επιβάλλονται κατά την εκτέλεση.

- **Εξισορροπητής Φορτίου (Load Balancer)**: Το συγκεκριμένο στοιχείο δεν αποτελεί ξεχωριστή συνιστώσα, αλλά μέρος του Ελεγκτή, το οποίο ελέγχει συνεχώς την κατάσταση των συνιστωσών που αποκαλούνται ως Επικαλεστές. Διατηρώντας μία σφαιρική εικόνα για το σύστημα, γνωρίζει ποιο Επικαλεστές είναι διαθέσιμοι, ώστε να τους προωθήσει τα αίτηματα περί εκτέλεσης των συναρτήσεων [23].

- **Kafka**: Ένα ακόμη σύστημα ανοικτού κώδικα από το ίδρυμα Apache , που χρησιμοποιείται για ενδιάμεση αποθήκευση και επικοινωνία του Ελεγκτή με τους Επικαλεστές.

- **Επικαλεστές (Invokers)**: Μπορούν εύλογα να χαρακτηριστούν ως η καρδιά του συστήματος, καθώς αναλαμβάνουν να εκκινήσουν τις διάφορες συναρτήσεις σε ξεχωριστά κοντέινερς. Για κάθε συνάρτηση δημιουργείται και εκκινείται ένα ξεχωριστό κοντέινερ, στο οποίο οι Επικαλεστές εισάγουν τις απαραίτητες παραμέτρους εισόδου και από το οποίο ανακτούν το αποτέλεσμα. Μόλις η εκτέλεση της εκάστοτε συνάρτησης ολοκληρωθεί και μετά το πέρας του απαιτούμενου χρονικού διαστήματος, το σχετικό κοντέινερ διαγράφεται από το σύστημα.

Βήματα Εκτέλεσης μίας Συνάρτησης

Στις επόμενες γραμμές, θα επιχειρήσουμε να περιγράψουμε συνοπτικά τα βήματα που πραγματοποιούνται έπειτα της κλήσης μίας συνάρτησης του OpenWhisk και θα περιγράψουμε πως οι προαναφερθείσες συνιστώσεις συνεργάζονται και επικοινωνούν μεταξύ τους [23]. Η διαδικασία εκκινεί με την λήψη ενός HTTP αιτήματος τύπου POST και έπειτα ακολουθούν οι εξής ενέργειες:

1. Το αίτημα λαμβάνεται από τον διακομιστή NGiNX και παραδίδεται στον Ελεγκτή.

2. Ο Ελεγκτής επικοινωνεί με τη βάση δεδομένων, ώστε να πιστοποιήσει τον χρήστη και να βεβαιωθεί ότι είναι εξουσιοδοτημένος για την ενέργεια που αιτείται.

3. Εφόσον το προηγούμενο βήμα ολοκληρωθεί επιτυχώς, ο Ελεγκτής υποβάλλει νέο ερώτημα στη βάση, αυτή τη φορά ζητώντας την εγγραφή της συνάρτησης και τις πληροφορίες που αυτή περιέχει.

4. Ο Εξισορροπητής επιλέγει τον βέλτιστο Επικαλεστή, στον οποίο στέλνει το σχετικό μήνυμα μέσω του συστήματος Kafka.

5. Αυτός με τη σειρά του ζητά από την βάση δεδομένων τον κώδικα της συνάρτησης και είτε δημιουργεί στο περιβάλλον αυτόματα ένα κοντέινερ για τη συνάρτηση αυτή, είτε αιτείται στον Κυβερνήτη τη δημιουργεία νέου pod[2] [24]. Σε κάθε περίπτωση, οι παράμετροι εισόδου τροφοδοτούνται στην συνάρτηση, της οποίας η εκτέλεση εκκινεί.

6. Μόλις η εκτέλεση ολοκληρωθεί επιτυχώς, το αποτέλεσμα ανακτάται από τον αντίστοιχο Επικαλεστή, ο οποίος το αποθηκεύει μαζί με διάφορες μετρικές και δεδομένα (σχετικά με την εκτέλεση της συνάρτησης) στη βάση δεδομένων.

[2]Η εμπλοκή του Κυβερνήτη πραγματοποιείται εάν η επιλογή *KubernetesContainerFactory* είναι ενεργοποιημένη για τους Επικαλεστές.

Η δημιουργεία νέου pod/container θα πραγατοποιηθεί στη περίπτωση που δεν υπάρχει διαθέσιμο στο σύστημα. Αποκαλούμε αυτήν την κατάσταση ως *ψυχρή εκκίνηση* (*cold start*). Σε αντίθετη περίπτωση, το αίτημα θα ικανοποιηθεί από τον υπάρχων μηχανισμό και η κατάσταση αυτή χαρακτηρίζεται ως *θερμή εκκίνηση* (*warm start*).

3 Δυναμική Διαχείριση Πόρων σε Αρχιτεκτονικές χωρίς Διακομιστή

3.1 Μαθηματικοί Φορμαλισμοί & Ορισμοί

Έστω s μία ακολουθία από n συναρτήσεις f_0, f_1,..., f_{n-1}. Αξιοποιούμε διανυσματικό συμβολισμό και γράφουμε $s = (f_0, f_1, ..., f_{n-1})$. Έστω l ο επιθυμητός συνολικός χρόνος εκτέλεσης για την ακολουθία s. Κατά την εκτέλεση I της ακολουθίας αυτής συμβολίζουμε με t_i τον μετρούμενο χρόνο εκτέλεσης για κάθε συνάρτηση f_i για $0 \leq i \leq n-1$. Για την εκτέλεση αυτή, θα λέμε ότι η ακολουθία s δεν ικανοποίησε τον στοχευμένο χρόνο εκτέλεσης ή ότι παραβίασε τον στόχο και την Ποιότητα Υπηρεσίας, εάν ισχύει η ανιστότητα $T_I(s) = \Sigma_{i=0}^{n-1} t_i > l$. Σε αντίθετη περίπτωση, λέμε ότι η ακολουθία ικανοποιεί τον στόχο ή την Ποιότητα Υπηρεσίας. Τέλος, βαφτίζουμε την ποσότητα $\frac{T_I(s)-l}{l}$ ως συντελεστή παραβίασης.

Με βάση τα παραπάνω, μπορούμε εύκολα να ορίσουμε το πρόβλημα της Ποιότητας Υπηρεσίας για της ακολουθίες συναρτήσεων χωρίς διακομιστή, ως την διαδικασία ελαχιστοποιήσης του πλήθους των χρονικών παραβιάσεων για ένα ορισμένο πλήθος εκτελέσεων. Εναλλακτικά, κανείς δύναται να επιχειρίσει την ελαχιστοποίηση του συντελεστή παραβίασης για τις παραβιάσεις που συμβαίνουν. Τονίζεται πως τα δύο προβλήματα δεν είναι ισοδύναμα, αλλά ένα καλώς σχεδιασμένο σύστημα οφείλει να λαμβάνει υπόψην και τα δύο.

Επιπλέον, εάν ο προβλεπόμενος χρόνος εκτέλεσης τ_i κάθε συνάρτησης f_i για $0 \leq i < n$ είναι γνωστός, μπορούμε να ορίσουμε ως χαλαρώτητα (slack) της συνάρτησης f_i την ποσότητα $slack_{f_i} = \Sigma_{j=0}^{i-1} \tau_j - \Sigma_{k=0}^{i-1} t_k = \Sigma_{j=0}^{i-1} (\tau_j - t_j)$. Για την αρχική συνάρτηση f_0 ορίζουμε μηδενική χαλαρότητα. Εάν η ποσότητα αυτή είναι θετική, η εκτέλεση της ακολουθίας προπορεύεται του στόχου, ενώ σε αντίθετη περίπτωση έπεται.

3.2 Προτεινόμενη Λύση: *Ρολόι Ακολουθίας*

Επισκόπηση της Αρχιτεκτονικής

Το *Ρολόι Ακολουθίας* (ΡΑ) είναι ένα κατανεμημένο σύστημα που αποτελείται από τρεις διαφορετικές συνιστώσες, ήτοι τον *Προγραμματιστή*, τον *Ανώτατο Παρατηρητή* και τους *Παρατηρητές*. Κάθε μία από αυτές αποτελεί ξεχωριστό αντικείμενο του Κυβερνήτη. Επιπλεόν, υπάχει μία επιπρόσθετη συνσιτώσα που ονομάζεται *Διαχειριστής Ακολουθίας*, ο οποίος αξιοποιεί τις δυνατότητες και ευελιξία του OpenWhisk, αποτελώντας μία ξεχωριστή συνάρτηση για κάθε ακολουθία. Μία αφηρημένη αναπαράσταση του συστήματος παρουσιάζεται στην εικόνα 2.

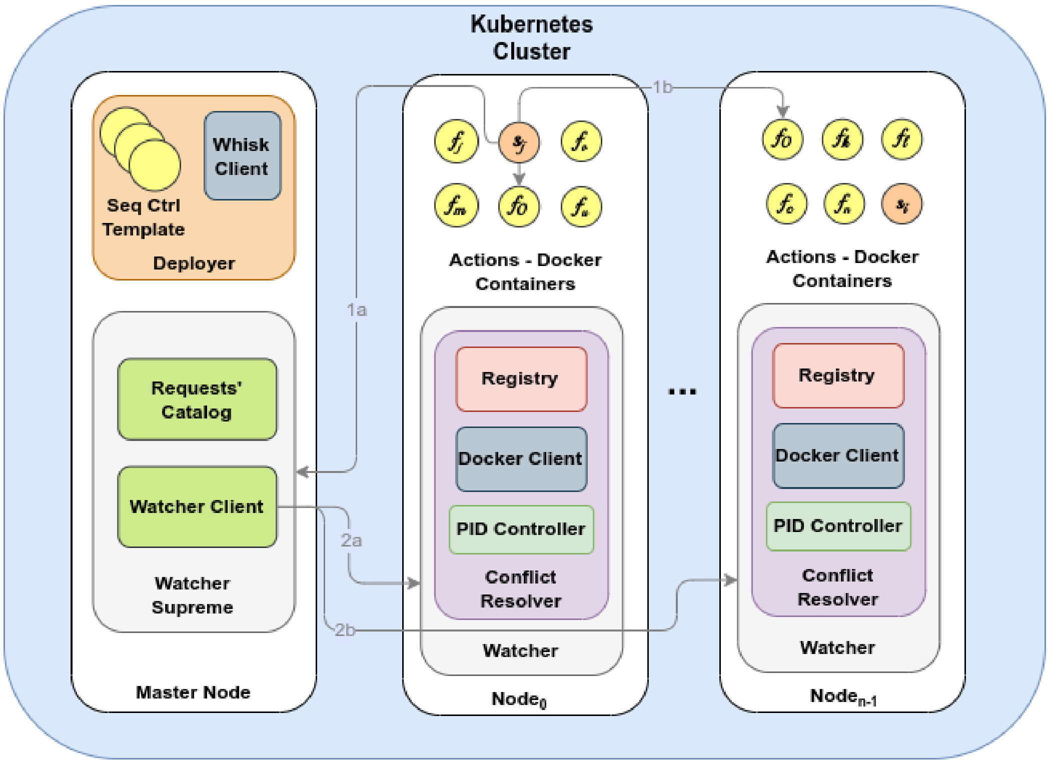


Figure 2: Αρχιτεκτονική του *Ρολογιού Ακολουθίας*

Επιγραμματικά, το σύστημα παράγει για κάθε ακολουθία προς ρύθμιση μία ξεχωριστή συνάρτηση μέσα στο OpenWhisk, τον *Διαχειριστή Ακολουθίας* (πορτοκαλί κύκλοι στην εικόνα 2). Ο μοναδικός σκοπός της ύπαρξής του είναι αφενώς να εκτελεί την κλήση (1b) της κάθε συνάρτησης της ακολουθίας (κάθε συνάρτηση αναπαριστάται με έναν κίτρινο κύκλο στην εικόνα 2) και αφετέρου να συλλέγει διάφορες μετρικές για την εκάστοτε εκτέλεση. Η ίδια συνιστώσα επικοινωνεί (1α) με τον *Ανώτατο Παρατηρητή*, αιτούμενος για παροχή υπολογιστικών πόρων. Αυτός, εν συνεχεία, μεταφέρει το αντίστοιχο αίτημα στους *Παρατηρητές*, οι οποίο βρίσκονται σε κάθε κόμβο διαχειρίζοντας και κατανέμοντας όσο το δυνατόν πιο δίκαια τους πόρους του. Κάθε *Παρατηρητής*, σύμφωνα με τα ληφθέντα αιτήματα, προσφέρει περισσότερη υπολογιστική ισχύ σε κοντέινερ ακολουθιών με αρνητική *χαλάρωση*, ενώ ταυτόχρονα αφαιρεί υπολογιστική ισχύ από κοντέινερ ακολουθιών με θετική *χαλάρωση*. Η μεταβλητή ελέγχου, που χρησιμοποιείται, είναι η ποσόστωση της ΚΜΕ (CPU quotas).

Προγραμματιστής

Ο *Προγραμματιστής* είναι μία πολύ απλή αλλά ταυτόχρονα αρκετά ισχυρή συνιστώσα. Συγκεκριμένα, στοχεύει στην αυτοματοποίηση της δημιουργείας των διαφόρων ακολουθιών για τον χρήστη. Κάτι τέτοιο είναι εφικτό μέσω της διεπαφής προγαμματισμού (API) που προσφέρει και της χρήσης του πελάτη του OpenWhisk.

Διαχειριστής Ακολουθίας

Ο *Διαχειριστής Ακολουθίας* είναι η μοναδική συνιστώσα του συστήματος η οποία δεν είναι μακρόβια. Το Ρολόι Ακολουθίας έχει σχεδιαστεί με σεβασμό προς το ίδιο το OpenWhisk και επομένως θα αποτελούσε μία σημαντική αστοχία, εάν ο μηχανισμός συλλογής μετρικών και χρονομέτρου είχε υλοποιηθεί με συγκεντρωτικό τρόπο. Αντί για αυτό, για την εκτέλεση κάθε ακολουθίας μία ξεχωριστή συνάρτηση, ο *Διαχειριστής Ακολουθίας*, εκτελείται με σκοπό να καλεί κάθε επιμέρους συνάρτηση και να προωθεί την έξοδό της στην επόμενη. Ταυτόχρο- να, είναι υπεύθυνος για την καταγραφή της *χαλάρωσης* της ακολουθίας πριν από κάθε εκτέλεση, της οποίας την τιμή μαζί με άλλες μετρικές προωθεί στην συγκεντρωτική συνιστώσα του *Ανώτατου Παρατηρητή*. Το τελευταίο, πραγματοποιείται όταν ο *άπληστος* αλγόριθμος είναι ενεργοποιημένος, ο οποίος ζητάει από τους *Παρατηρητές* τους απαραίτητους πόρους, με σκοπό να ρυθμίσει τον χρόνο εκτέλεσης. Κατά την υλοποίηση αυτής της διπλωματικής μελετήθηκε ένας επιπλέον ¨αλγόριθμος¨, αποκαλούμενος ως dummy, ο οποίος αγνοεί την ύπαρξη των *Παρατηρητών* και δεν πραγματοποιεί αιτήματα προς κανέναν κόμβο για παροχή επιπλέον ποσόστωσης της ΚΜΕ, αφήνωντας την μεταβλητή ελέγχου ελεύθερη.

Ανώτατος Παρατηρητής

Ο ρόλος του *Ανώτατου Παρατηρητή* είναι κεντρικός, αλλά όχι ιδιαίτερα περίπλοκος, καθώς απλά χρησιμεύει ως ο διαμεσολαβητής μεταξύ των *Διαχειριστών Ακολουθιών* και των *Παρατηρητών*. Αντί οι πρώτοι να επικοινωνούν με κάθε κόμβο ξεχωριστά (κάτι το οποίο θα εισήγαγε επιπλέον χρονοκαθυστέρηση), προτιμάται να επικοινωνούν ασύγχρονα με μία και μόνο συνιστώσα, αυτή του *Ανώτατου Παρατηρητή*. Ο τελευταίος, ρωτά κάθε κόμβο ξεχωριστά ώστε να μάθει σε ποιο σημείο της συστοιχίας βρίσκεται το κοντέινερ, σχετικό με την εκτέλεση της εκάστοτε συνάρτησης. Μόλις αυτή η πληροφορία γίνει γνωστή, ο *Ανώτατος Παρατηρητής* μεταφέρει το σχετικό αίτημα παροχής πόρων στον σχετικό κόμβο, ώστε να εκκινήσει η διαδικασία ρύθμισης.

Στην περίπτωση της *ψυχρής εκκίνησης* δεν υπάρχει διαθέσιμο κοντέινερ σε κανέναν από τους κόμβους της συστοιχίας. Αυτό συνεπάγεται πως κανένας κόμβος δεν θα απαντήσει θετικά. Σε αυτήν ακριβώς την περίπτωση, επιλέξαμε ο *Ανώ-*

τατος Παρατηρητής να μην εκτελεί καμία περαιτέρω ενέργεια και η εκτέλεση της ακολουθίας να συνεχίζει χωρίς ρύθμιση. Σε πρώιμες εκδόσεις, είχε αξιοποιηθεί ένας μηχανισμός σφυγομέτρησης (polling) για την σταδιακή εύρεση του κόμβου εγκατάστασης του εκάστοτε κοντέινερ. Δυστυχώς η συγκεκριμένη προσέγγιση παρήγαγε έντονη κίνηση στο δίκτυο και οδηγούσε σε υπολειτουργία. Για αυτόν ακριβώς το λόγο επιλέχθηκε η αφαίρεση αυτού του χαρακτηριστικού στην πορεία της ανάπτυξης. Σε κάθε περίπτωση, περαιτέρω μελέτη απαιτείται γύρω από την διαχείριση των *ψυχρών εκκινήσεων*.

Παρατηρητής

Αποτελεί την κύρια συνιστώσα του συστήματος. Κάθε κόμβος της συστηχίας περιέχει ένα κοντέινερ στο οποίο τρέχει ένας *Παρατηρητής* με την αρμοδιότητα διαμοιρασμού των υπολογιστικών πόρων και την ρύθμιση του χρόνου εκτέλεσης των ακολουθιών, με βάση τα αντίστοιχα αιτήματα.

Όσον αφορά τη δομή του, εμπεριέχει δύο εσωτερικούς μηχανισμούς, ήτοι έναν διακομιστή και τον *Επιλυτή Συγκρούσεων*. Ο εξυπηρετητής απλά προσφέρει την κατάλληλη διεπαφή, ώστε να είναι διαθέσιμες οι διάφορες λειτουργίες σε λοιπά στοιχεία της συστοιχίας. Από την άλλη, ο *Επιλυτής Συγκρούσεων* καταγράφει τα εισερχόμενα αιτήματα, τους προσφερόμενους πόρους σε κάθε κοντέινερ και επικοινωνεί με τον μηχανισμό Docker του κάθε κόμβου. Για την καταγραφή των τρέχοντων αιτημάτων αξιοποιείται μία δομή δεδομένων, το *Αρχείο* (*Registry*), στο οποίο καταχωρείται, επιπλεόν, η κατάσταση των κοντέινερ του κόμβου.

Ελεγκτής PID

Ο *Ελεγκτής PID* είναι μία απλή συνάρτηση, μέσα στους *Παρατηρητές*, η οποία δέχεται μετρικές όπως την *χαλαρώτητα* κάθε συνάρτησης (αναλογικός όρος), το άθροισμα των προηγούμενων μετρήσεων αυτού του μεγέθους (ολοκληρωτικός όρος) και την χαλαρώτητα της προηγούμενης συνάρτησης (όρος παραγώγου). Με βάση αυτές τις τιμές επιστρέφει την ποσότητα κατά την οποία πρέπει να αυξηθεί ή να μειωθεί η ποσόστωση της ΚΜΕ του εκάστοτε κοντέινερ.

Αλγόριθμος Κατανομής Πόρων

Όπως είναι φυσικό κάθε κόμβος διαθέτει έναν πεπερασμένο αριθμό πυρήνων. Επομένως, απαιτείται η εύρεση ενός τρόπου κατά τον οποίο θα γίνεται ένας δίκαιος διαμοιρασμός της ποσόστωσης της ΚΜΕ στα διάφορα κοντέινερ, όταν ο κάθε κόμβος φτάνει σε κατάσταση κορεσμού. Κάτι τέτοιο είναι δυνατόν να συμβεί, εάν το άθροισμα των ποσοστώσεων, που αιτείται η κάθε συνάρτηση, υπερβεί το πλήθος των διαθέσιμων πυρήνων. Για αυτόν ακριβώς τον λόγο, ο κάθε *Παρατηρητής* δεν προσφέρει σε κάθε κοντέινερ το πλήθος των quotas που του υπέδειξε ο *Ελεγκτής*

PID, αλλά ένα ποσοστό λ αυτής. Ο παράγοντας λ προκύπτει από την ακόλουθη μαθηματική φόρμουλα:

$$\lambda = \begin{cases} \frac{N_{cores} \cdot T}{\sum\limits_{i=0}^{n-1} r_i} & , saturation \\ 1 & , otherwise \end{cases} \tag{1}$$

Στην εξίσωση 1 ως N_{cores} συμβολίζουμε το πλήθος των πυρήνων του κόμβου, ως T την προεπιλεγμένη περίοδο της ΚΜΕ και ως r_i την ποσότητα quotas της ΚΜΕ, που αιτήθηκε η i-οστή συνάρτηση f_i.

Για την αποφυγή του εξονυχιστικού υπολογισμού του αθροίσματος στην παραπάνω φόρμουλα, κανείς μπορεί να εξάγει την ακόλουθη αναδρομική σχέση:

$$\lambda_{new} = \frac{N_{cores} \cdot T}{\frac{N_{cores} \cdot T}{\lambda_{old}} + r_{new} - r_{old}} = \frac{N_{cores} \cdot T \cdot \lambda_{old}}{N_{cores} \cdot T + (r_{new} - r_{old}) \cdot \lambda_{old}} \tag{2}$$

Με τους όρους r_{new} και r_{old} αναπαριστούμε την νέα και την παλαιά τιμή των quotas της ΚΜΕ που αιτήθηκε η εκάστοτε συνάρτηση.

Πολιτική Συγκρούσεων

Κατά την εκτέλεση πολλαπλών ακολουθιών, οι *Παρατηρητές* καλούνται να επιλύσουν ένα επιπλέον πρόβλημα, αυτό του διαμοιρασμού πόρων μεταξύ αιτημάτων, που αφορούν την ίδια συνάρτηση. Μία τέτοια κατάσταση ενδέχεται να προκύψει εάν πολλαπλές κλήσεις της ίδιας ακολουθίας πραγματοποιούνται ταυτόχρονα, με αποτέλεσμα διαφορετικά αιτήματα να εμφανιστούν για το ίδιο κοντέινερ. Σε αυτήν την περίπτωση, αποφασίσαμε οι *Παρατηρητές* να ακολουθήσουν μία πολιτική κατά την οποία θα δώσουν προτεραιότητα στο αίτημα που αφορά την ακολουθία με τη μεγαλύτερη χρονική επιβάρυνση.

4 Αποτελέσματα & Αξιολόγηση

4.1 Πειραματική Διάταξη

Για το πειραματικό μέρος της Διπλωματικής αξιοποιούμε μία συστοιχία τεσσάρων εικονικών μηχανημάτων. Σε καθένα από αυτά είναι εγκατεστημένη η διανομή Ubuntu 20.04 LTS με την 5.4 έκδοση του πυρήνα Linux. Περαιτέρω λεπτομέρειες, σχετικές με τις προδιαγραφές αυτών, βρίσκονται στον παρακάτω πίνακα.

#	Ρόλος	Ρόλος στο OpenWhisk	Αριθμός Πυρήνων	Μνήμη	Swap
0	master	none	2	7.77 GiB	off
1	worker	invoker	4	15.6 GiB	off
2	worker	invoker	4	15.6 GiB	off
3	worker	invoker	2	15.6 GiB	off

Table 1: Προδιαγραφές Πειραματικής Συστοιχίας.

Από τους τέσσερις κόμβους, ένας αξιοποιείται ως ο κύριος κόμβος, στον οποίο τοποθετούνται οι κεντρικές συνιστώσες του Κυβερνήτη και του OpenWhisk. Οι υπόλοιποι χρησιμεύουν ως δευτερεύοντες κόμβοι, στους οποίους τοποθετούνται συνιστώσες χρηστών, καθώς επίσης και τα κοντέινερ των συναρτήσεων.

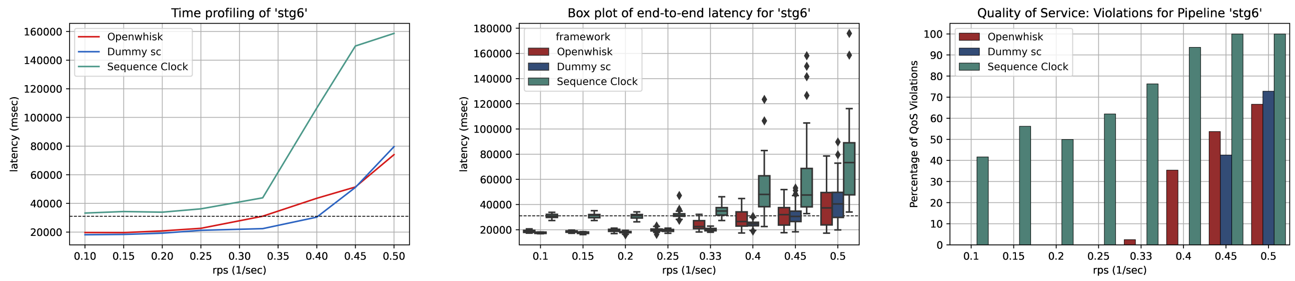
Ο υπέρτατος στόχος για την δεδομένη συστοιχία είναι η αξιολόγηση της επίδοσης των ακολουθιών, υπό την επιβολή διαφόρων επιπέδων ¨πίεσης¨. Ο τρόπος για να ποσοτηκοποιηθεί κάτι τέτοιο είναι μέσω της μεταβολής του ρυθμού κλήσεων των ακολουθιών, ήτοι η εκτέλεση αλλεπάλληλων κλήσεων μίας ακολουθίας με έναν σταθερό ρυθμό rps ή ισοδύναμα ανά σταθερό χρονικό διάστημα $\Delta t = \frac{1}{rps}$

4.2 Σύγκριση OpenWhisk & Ρολογιού Ακολουθίας

Για τα πειράματά μας αξιοποιούμε μία πληθώρα ακολουθιών που έχουν προκύψει τόσο από γενικές συναρτήσεις απαιτητικές ως προς την ΚΜΕ, όσο και από πιο ρεαλιστικές εφαρμογές της σουίτας SeBS [25, 26]. Από το σύνολο αυτών παράγουμε τις ακολουθίες μήκους τριών συναρτήσεων *stg3, mtg3, mpg3, p021, p643* και τις ακολουθίες μήκους έξι συναρτήσεων *stg6, mtg6, mpg6, p024879, p051463*. Παρακάτω παραθέτουμε τα αποτελέσματα για ορισμένες από αυτές.

Ανάλυση Γενικών Ακολουθιών

Ως ακολουθίες αναφοράς αξιοποιούνται η *stg6*, με χρόνο 31064 mseconds και η *stg3* με χρόνο 8917 mseconds. Η διαφορά τους στο μήκος αποδεικνύεται, εν τέλει, πως έχει σημαντική επίπτωση στον τελικό χρόνο.



(a) 99-ο Εκατοστημόριο. (b) Κατανομή του Χρόνου. (c) Ποσοστό Παραβιάσεων.

Figure 3: Συγκρίνοντας το OpenWhisk με το Ρ.Α. για την ακολουθία *stg6*.

Από το διάγραμμα (3α) κανείς μπορεί να συμπεράνει τα ακόλουθα:

- Η επιλογή διαχείρισης της κάθε ακολουθίας από μία ξεχωριστή συνάρτηση και οι απαραίτητες μεταφορές δεδομένων, που αυτή συνεπάγεται, δεν επιφέρει κάποια καθυστέρηση δικτύου, εξού και η συμπεριφορά του Dummy PA.
- Ο *άπληστος* αλγόριθμος και ο ελεγκτής καταφέρνουν να περιορίσουν τον χρόνο εκτέλεσης γύρω από τον επιθυμητό στόχο για τιμές του *rps* μικρότερες

του 0.33, ενώ για μεγαλύτερες δυστυχώς επιφέρουν χρόνους χειρότερους από το ίδιο το OpenWhisk.

Έχοντας αναφέρει τα παραπάνω, διαπιστώνουμε από το διάγραμμα (3c) ότι το PA εμφανίζει παραβιάσεις για όλα τα επίπεδα *rps*. Για τιμές μικρότερες του 0.33, τα διαγράμματα (3b) και (4) υποδηλώνουν ότι δεν είναι μεγάλης κρισιμότητας, αλλά κανείς μπορεί να υποστηρίξει ότι τέτοιου είδους παραβιάσεις δεν παρουσιάζονταν στο ίδιο το OpenWhisk εξ' αρχής για αυτά τα επίπεδα πίεσης.

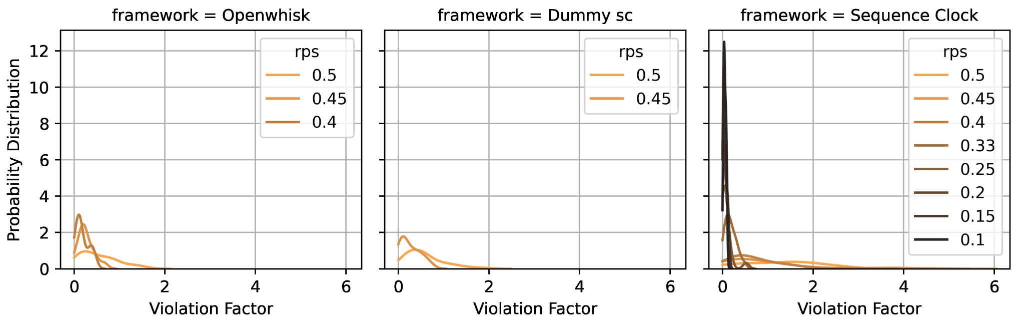
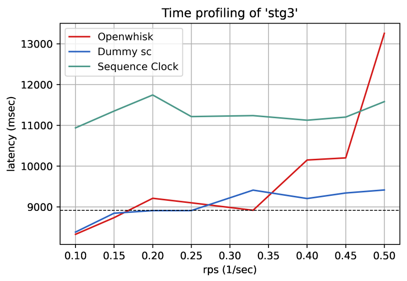
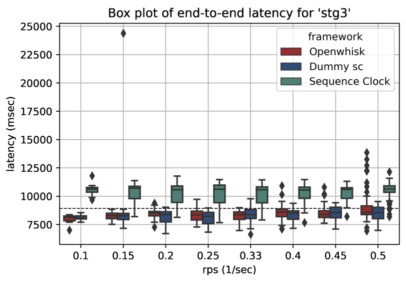


Figure 4: Κατανομή του Συντελεστή Παραβιάσεων για την ακολουθία *stg6*.

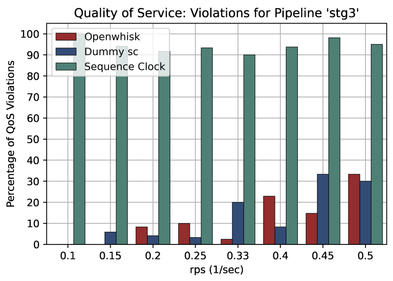
Προχωρώντας με το *stg3*, έπειτα από παρατήρηση των αντίστοιχων διαγραμμάτων, γίνεται σαφές πως ακολουθίες μικρότερου μήκους υποφέρουν λιγότερο από αύξηση του χρόνου εκτέλεσης. Ταυτόχονα, όμως, το διάγραμμα (5a) δείχνει μία ακόμη πτυχή του συστήματος, η οποία είναι η αδυναμία βελτιστοποιήσης του PID ελεγκτή για όλες τις ακολουθίες και συναρτήσεις, καθώς το *stg3* παρουσιάζει σφάλμα σταθερής κατάστασης.



(a) 99-ο Εκατοστημόριο.



(b) Κατανομή του Χρόνου.



(c) Ποσοστό Παραβιάσεων.

Figure 5: Συγκρίνοντας το OpenWhisk με το P.A. για την ακολουθία *stg3*.

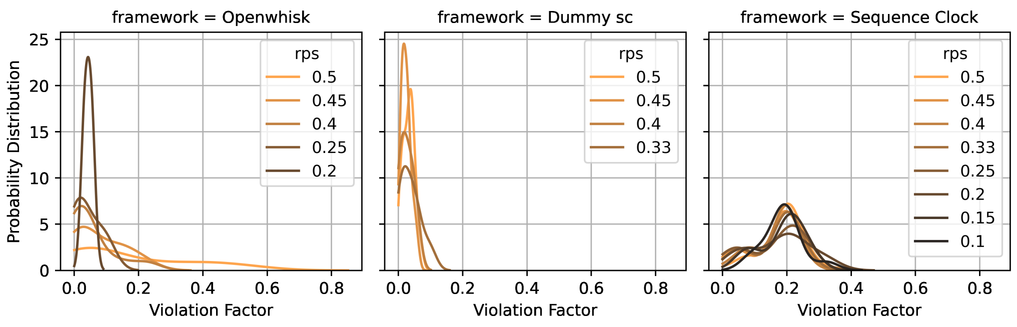
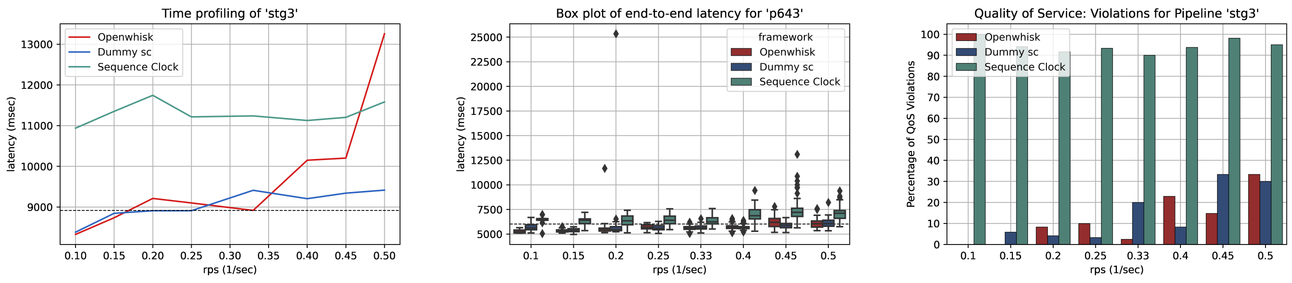


Figure 6: Κατανομή του Συντελεστή Παραβιάσεων για την ακολουθία *stg3*.

Ανάλυση Ακολουθιών της Σουίτας SeBS

Η συμπεριφορά των πιο ρεαλιστικών ακολουθιών δεν αποκλίνει από αυτή των πιο γενικών, με το PA να συμπεριφέρεται καλύτερα σε μεγάλου μήκους ακολουθίες. Παρακάτω παρατίθονται τα αποτελέσματα για την ακολουθία *p643*.



(a) 99-ο Εκατοστημόριο. (b) Κατανομή του Χρόνου. (c) Ποσοστό Παραβιάσεων.

Figure 7: Συγκρίνοντας το OpenWhisk με το P.A. για την ακολουθία *p643*.

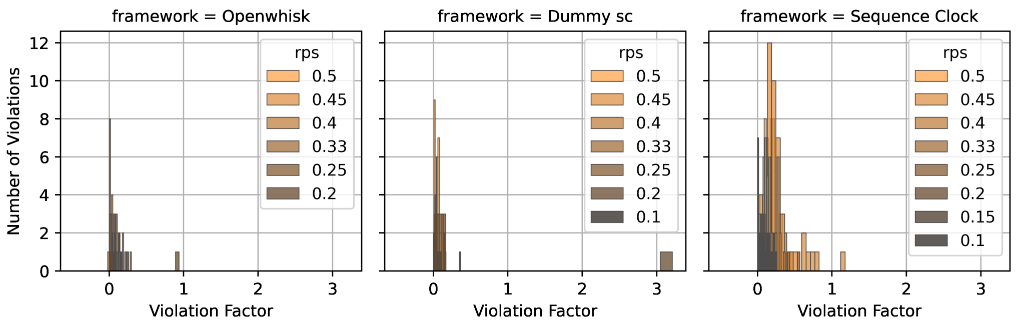


Figure 8: Κατανομή του Συντελεστή Παραβιάσεων για την ακολουθία *p643*.

4.3 Ερμηνεία Αποτελεσμάτων & Παράδοξα

Στις γραμμές που ακολουθούν παραθέτουμε αιτίες που οδήγησαν στην αδυναμία του Ρολογιού Ακολουθίας να υπερτερήσει έναντι του OpenWhisk. Όπως έγινε σαφές στα παραπάνω διαγράμματα, ο χρόνος εκτέλεσης των ακολουθιών συμπεριφέρεται ως μία τυχαία μεταβλητή με την διασπορά της αντίστοιχης κατανομής να αυξάνεται για μεγαλύτερα επίπεδα πίεσης. Η ρύθμιση μέσω της ποσόστωσης της ΚΜΕ, που αντιστοιχεί σε κάθε κοντέινερ, φαίνεται ότι καταφέρνει να περιορίσει την κατανομή αυτήν για τιμές δυστυχώς μικρότερες του επιπέδου *rps*, το οποίο συντέλεσε στην διαμόρφωση του στοχευμένου χρόνου εκτέλεσης. Αυτό ερμηνεύεται ως δυνατότητα επιβράδυνσης των συναρτήσεων για μικρότερες τιμές των quotas και ταυτόχρονη αδυναμία επιτάχυνσης για τιμές μεγαλύτερες από μία περίοδο της ΚΜΕ. Επιπλέον, κρίνεται απαραίτητο να θίξουμε το θέμα συντονισμού των κερδών του ελεγκτή PID. Συγκεκριμένα, σε ορισμένες ακολουθίες συμπεριφέρεται ως βελτιστοποιημένος, με την χρονική κατανομή να περιορίζεται σημαντικά γύρω από τον προκαθορισμένο στόχο, ενώ σε άλλες οδηγεί σε σφάλμα σταθερής κατάστασης.

Εν συνεχεία, μία εξήγηση επείγει για την αποδοτικότερη συμπεριφορά της *dummy*, χωρίς επίβλεψη έκδοσης του Ρολογιού Ακολουθίας. Τα διαγράμματα έδειξαν μειωμένο ποσοστό παραβιάσεων και ταυτόχρονη μετατόπιση της κατανομής του συντελεστή παραβιάσεων εγγύτερα στο μηδέν. Το τελευταίο συνεπεγάται πως οι παραβιάσεις που προκύπτουν είναι λιγότερο έντονες. Η ερμηνεία που προσφέρουμε για αυτήν την συμπεριφορά βασίζεται στο γεγονός πως το OpenWhisk αναλαμβάνει την εκτέλεση των διαφόρων ακολουθιών με συγκεντρωτικό τρόπο, το οποίο αποδεικνύεται ότι δεν κλιμακώνεται εύκολα. Αντιθέτως, το προτεινόμενο σύστημα αξιοποιεί ξεχωριστή συνάρτηση (επομένως και κοντέινερ) για την διαχείριση κάθε ξεχωριστής ακολουθίας, κατανέμοντας το επιβαλλόμενο φορτίο και οδηγώντας σε καλύτερους χρόνους.

5 Σύνοψη & Μελλοντική δουλειά

5.1 Σύνοψη

Σε αυτήν την Διπλωματική Εργασία, ερευνήσαμε τις προκλήσεις παροχής Ποιότητας Υπηρεσίας σε αρχιτεκτονικές χωρίς διακομιστή. Το υπολογιστικό μοντέλο *Συνάρτηση-σαν-Υπηρεσία* παρουσιάστηκε και αξιολογήθηκε παράλληλα με διάφορες βιβλιοθήκες, εργαλεία και τη δομή αυτών, όπως το OpenWhisk και το OpenFaas, παράλληλα με την δομή τους.

Μέσω εκτενών πειραμάτων, αναγνωρίσαμε ποικίλους παράγοντες που έχουν μία επίδραση στον χρόνο εκτέλεσης των ακολουθιών και επικεντρώσαμε το ενδιαφέρον μας στον κορεσμό των υπολογιστικών πόρων. Τα πειράματα έδειξαν, κατά μέσο όρο, μία εκθετική αύξηση στον χρόνο εκτέλεσης των ακολουθιών κατά την αύξηση του ρυθμού κλήσεων, με την προβλεψιμότητα του συτήματος να ακολουθεί μία παρόμοια μοίρα.

Το επόμενο βήμα στην τρέχουσα Διπλωματική ήταν η ανάπτυξη του Ρολογιού Ακολουθίας, το οποίο αποτελεί ένα εργαλείο ελέγχου του χρόνου εκτέλεσης ακολουθιών αρχιτεκτονικής χωρίς διακομιστή. Μέσω μίας σειρά πειραμάτων πραγματοποιήθηκε σύγκριση με το ίδιο το OpenWhisk, που έδειξε πως το Ρολόι

Ακολουθίας είναι ικανό να επιβραδύνει ακολουθίες συναρτήσεων και να πετύχει σε χαμηλούς ρυθμούς κλήσεων κατανομές μικρότερης διασποράς του χρόνου εκτέλεσης με μέση τιμή κοντά στον προεπιλεγμένο στόχο. Από την άλλη, η χωρίς επίβλεψη έκδοση του *Διαχειριστή Ακολουθιών* κατάφερε να μειώσει το ποσοστό των παραβιάσεων και να περιορίσει το μέγιστο της κατανομής του *συντελεστή παραβίασης*.

5.2 Μελλοντική δουλειά

Σαν μελλοντική δουλειά, κανείς θα μπορούσε να εξετάσει έναν συνδυασμό του *άπληστου* αλγόριθμου και της χωρίς επίβλεψη έκδοσης του *Διαχειριστή Ακολουθίας*. Αυτή η πρόταση προκύπτει από την διαπίστωση πως ο ελεγκτής PID, με την ρύθμιση των quotas της ΚΜΕ, είναι ικανός να περιορίσει τον χρόνο εκτέλεσης γύρω από τον προκαθορισμένο στόχο με μικρή διακύμανση για χαμηλές τιμές του ρυθμού κλήσεων. Αντιθέτως σε συνθήκες αυξημένης πίεσης, αποδεικνύεται πως η χωρίς επίβλεψη προσέγγιση οδηγεί σε μικρότερο αριθμό παραβιάσεων από το ίδιο το OpenWhisk.

Επιπλεόν, επείγει η εύρεση παραμέτρων που θα αποτελέσουν αποδοτικότερες μεταβλητές ελέγχου για την ρύθμιση της επίδοσης των συναρτήσεων και των ακολουθιών ως άμεση συνέπεια. Υποψήφια μεγέθη αποτελούν η συχνότητα ρολογιού της ΚΜΕ του μηχανήματος οικοδεσπότη, το εύρος ζώνης δικτύου κτλ. Παράλληλα, ενδιαφέρουσα θα ήταν μία μελέτη που θα αξιολογούσε έναν δυναμικό μηχανισμό *ψυχρής εκκίνησης* και οριζόντιας κλιμάκωσης των συναρτήσεων. Κάτι τέτοιο δύναται να αποτελέσει ένα επιπλέον βήμα προς τον στόχο της πολυπόθητης Ποιότητας Υπηρεσίας, καθώς ενδέχεται σε ορισμένες περιπτώσεις να συμφέρει, από χρονικής άποψης, η δημιουργεία ενός νέου κοντέινερ έναντι της αναμονής ή της παράλληλης εκτέλεσης στο ίδιο κοντέινερ.

Chapter 1

Introduction

Nowadays, there is in ever-increasing number of workloads pushed and executed on the Cloud. As the competition increases and the Time-to-Market plays a significant role in software oriented companies and startups, it is natural for developers and engineers to be interested in solutions that would potentially decrease development time and allow them to focus on their application's business logic. An abstract comparison of various deployment models concerning the relation of business logic focus and the programming stack implementation overhead is placed in Figure 1.1.

Over the years various cloud computing models have emerged offering different layers of abstraction and decoupling infrastructure maintenance from application development, i.e. the well known infrastructure as a service, platform as a service, and now function as a service. The latter, to which this thesis is related, is often used interchangeably with the term serverless computing and it is something more than just a cloud computing service, as it has been described by many as a whole new programming model [6].

Of course, for completeness purposes, before moving to FaaS presentation, it is wise to familiarize the reader with the concepts of cloud computing in general and the other types of cloud computing services.

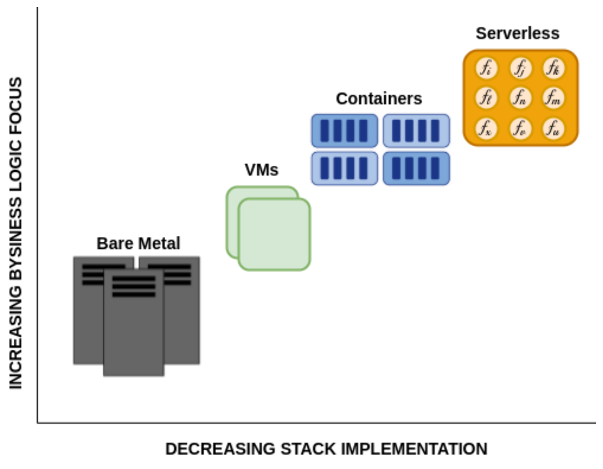

Serverless
Containers
VMs
Bare Metal
INCREASING BYSINESS LOGIC FOCUS
DECREASING STACK IMPLEMENTATION

Figure 1.1: Popularized comparison of various deployment models.[6]

1.1 Cloud Computing

According to PCMag Encyclopedia, cloud computing etymology is *"hardware and software services from a provider on the internet (the cloud)"* [1]. In more depth, cloud computing is a delivery of computing services, i.e. storage, databases, networking, and analytics, over the internet to offer faster innovation, resources, and economies of scale [2]. Some key benefits of cloud computing are the following:

- *Cost*: It eliminates capital expense of purchase and maintenance of hardware equipment
- *Speed*: It is so called *"self service"*. Users manage the entire process from start to finish with flexibility and without the pressure of capacity planning even for vast amounts of computing resources.
- *Scalability & Performance*: The ability to scale elastically and provide a service in various geographic locations would be impossible otherwise.
- *Productivity*: There are no time consuming IT management chores.
- *Reliability*: It makes data backup and recovery easier and less expensive as data can be mirrored at multiple redundant sites on the cloud providers' network.
- *Security*: A broad set of policies, technologies, and controls strengthen your security posture overall, helping protect your data, apps, and infrastructure from potential threats.

As we mentioned above, most cloud computing models fall into four broad categories which include IaaS, PaaS, SaaS, and FaaS with tech giants such as Amazon, Google, and Microsoft offering all of them. A brief description of each one is placed below.

1.1.1 Infrastructure as a Service (IaaS)

It is also called *cloud hosting* or *utility computing* and it provides only a base infrastructure. That means that through high level APIs, it is possible for users to rent IT infrastructure like servers, virtual machines, storage, network bandwidth and operating systems on a pay-as-you-go basis [1, 2, 3]. In case of VMs, a hypervisor hosts a large number of virtual machines, while offering the ability to scale them up or down depending on tenants' requirements [4]. Examples of IaaS are among others AWS EC2, Google Computing Engine, Microsoft Azure, and Digital Ocean.

1.1.2 Platform as a Service (PaaS)

This type of cloud computing service offers the user an on-demand environment for developing, testing, delivering, and managing software applications onto the cloud. Building and maintenance of the underlined infrastructure is completely shifted from the user to the provider, while control over the deployed application and configuration settings are still feasible from the consumer's perspective [2, 3, 4, 5]. Famous examples of PaaS are Google App Engine, Cloud Foundry, Heroku, AWS Elastic Beanstalk, Openshift and others.

1.1.3 Software as a Service (SaaS)

Software as a service is a method for delivering software applications over the Internet on demand and typically on a subscription basis [2]. The applications are accessible from various thin client interfaces, such as a web browser or a program interface [4], which means that nowadays one has definitely used a SaaS application at least once in their lifetime, as typical examples include YouTube, Slack, DropBox, Google Suite and the list goes on. Similarly to PaaS, the underline infrastructure, i.e. applications, runtime, data, middleware, operating systems, virtualization, servers, storage and networking, are managed completely by vendors, while consumers are limited in specific configurations of application-dependent settings.

1.2 Serverless Computing: Is it the future?

Having said all the above, an interesting question rises. What exactly is serverless or FaaS and where does it lie between all this? Serverless can be defined as an emerging application deployment architecture that completely hides server management from tenants [7]. Of course, this is also the case in previously mentioned and older vendor plans. Its main idea and original characteristic, though, is the fact that a monolithic application must be replaced and implemented by a set of short-lived, stateless and event-driven functions, thus the name FaaS. An oversimplified example of this, shown in Figure 1.2, is a picture upload from a user as the event and the transfer of this data into a database as the function [6]. Each function is being executed in a dedicated instance (usually a container), which is also ephemeral. That means that it will be deployed whenever an invocation occurs and will be deleted after handling the current request.

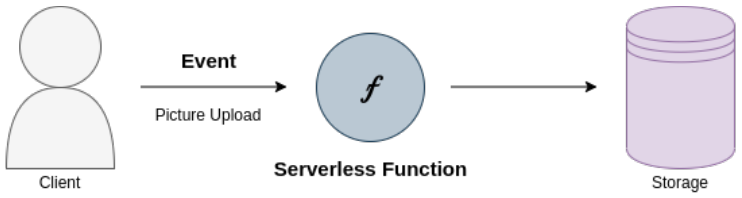


Figure 1.2: Simple example of a serverless function.

In these manner, tenants are charged on per invocation basis without paying for idle resources. In addition, due to its core idea and the lack of boot sequence in containers, this approach potentially offers auto-scaling mechanisms, fault tolerance and multiregional availability. For example, an increase in incoming requests can be handled with the addition of new containers and a load balancer. Faulty containers can again be replaced by new ones in a matter of seconds. On the contrary, one might express some significant concerns about security and quality of isolation as interference is inevitable between different functions or even different tenants [7]. Additionally, functions' impermanence leads to the need of persistent storage of data, in order for subsequent requests to operate on it, e.g. S3 in AWS Lambda. This may introduce significant overheads compared to in-memory computation of relative traditional applications [8].

Nevertheless, serverless is here to stay and remains an impressive technology, when used properly, i.e. in applications with intermittent activity, where maintaining long-running instances is cost inefficient [8].

1.2.1 Microservices vs Serverless

Before moving to further depth, it is considered necessary to distinguish serverless computing from another, also much discussed, concept like Microservices. The latter is not a cloud service type but an architectural pattern in which an application is divided into a series of services as opposed to the so-called monoliths (referring to applications where all functionality runs as a single entity). The key difference with FaaS programming model is that microservices are not stateless nor short-lived. It is common for a serverless application to contain dozens of functions, that implement discrete units of application functionality. On the other hand, it is common for a service-oriented-application or SOA (implemented with the microservices' architecture), to contain only a few of them, which is also destined to be run continuously and not be triggered by events [27].

1.3 From Monolith to Workflows

1.3.1 FaaS as a programming model

As we previously mentioned, in serverless computing an application must be translated into a set of stateless and event-driven functions. These functions need to call each other and share data in the process, resulting to the creation of graphs, which from now on we are going to call workflows or pipelines. Concerning the structure of these workflows, developers have the privilege to select from various options that are best suited for each occasion. The simplest of all is sequential execution of functions, which can be presented by a list-like graph and be characterized as a sequence. Input dependence is being introduced with pipeline branching and usage of if - else conditions. Even parallel execution of functions can be utilized whenever the massive amount of resources for a brief period of time is beneficial [8, 28].

1.3.2 Latency as a QoS metric

As far as the execution runtime of a pipeline is concerned, whichever its structure may be, certain users' configurations include various thresholds for computing resources (memory limit, number of processes, timeout etc.) in the function level [29]. Unfortunately, there is no guarantee for a certain time latency concerning the overall workflow, which could be interpreted as the well-known concept of quality-of-service or QoS. Even if a developer has find a tuned configuration where their function behaves in a certain way, e.g. terminates after t ms, this analysis is irrelevant if not useless. Each function instance coexists with a plethora of others and competes for the resources of the host machine. Having this said, for achieving a QoS concerning a workflow's tail latency, a more sophisticated runtime resource management system is required.

1.4 Thesis Overview

In this thesis, Sequence Clock (SC) is been presented as a potential solution to QoS violations of the execution time latency of function sequences, i.e., the simplest type of serverless functions' workflows. SC is a distributed, latency targeting tool that actively monitors serverless invocations inside a cluster and tries to regulate execution duration at the sequence level. In its core, it utilizes a greedy method for measuring the error (slack) from the target latency and a PID controller for estimating the required amount of cpu quotas as the tuning parameter. Moreover, a seemingly fair algorithm was developed for balancing resources between the containers placed on the same host.

In the chapters that follow, we present among others, technologies like Openwhisk, an opensource & distributed Serverless platform and Kubernetes, an open-source system for automating deployment & scaling management of containerized applications. Chapter 5 includes an in-depth description of Sequence Clock, it's components and functionalities, while chapter 6 evaluates our proposed framework, by comparing it with the default unsupervised Openwhisk sequences, across different stressing scenarios and workloads.

Chapter 2

Related Work

With Sequence Clock being a multi component system, built on top of existing frameworks and developed with various technologies, a *related work* chapter oughts to reference similar approaches concerning the ultimate goal, i.e. automatic resource management, and mention methods for serverless functions' benchmarking.

2.1 Metrics Collection

2.1.1 Deathstar Bench

Although DeathStarBench [8] is a benchmark suite for microservices which is a completely different concept from serverless, a simple reference to this is considered necessary as the methods that proposes provided us with some useful guidelines for our system's smoke test.

In short, it was developed upon the design principles mentioned below:

- *Representativeness*: inclusion of popular open source applications, such as nginx, memcached, MongoDB, RabbitMQ, MySQL, Appache HTTP server, ardrone-autonomy and others.

- *End-to-end operation*: accountability for the impact of inter service dependencies.

- *Heterogeneity:* support for low and high level languages including C/C++, Java, JavaScript, node.js, Python, html, Ruby, Go, Scala.

- *Modularity*: Conway's law at it's finest. In in a nutshell, following its creators communication structure, microservices of DeathStarBench suite minimize two-way communication and remain single concerned and loosely coupled.

- *Reconfigurability*: easy swapping out of microservices for alternate versions.

The aforementioned suite consists from five independent applications, a social network, a media service, an e-commerce service and the drone swarm coordination system.

2.1.2 Other Approaches

As every major cloud provider supports serverless, e.g., AWS lambda, Azure Functions, and Google cloud functions, one should wonder about the characterization of these provided services' performance and resource management efficiency. With this in mind, researchers [7] performed an extensive measurement study. Areas of interest include tenant isolation, scalability, function placement, cold start events, and instance lifetime and reusage.

2.2 Resource Management in serverless computing

2.2.1 Llama

Llama is a heterogeneous serverless framework for auto-tuning video pipelines [28]. By the time of writing this thesis, it was the closest approach to ours as it also uses time latency as target. In more depth, it calculates the target latency for each invocation of the pipeline and utilizes a cost optimizer in order to determine efficient configurations for each one of them. These configurations correspond to parameters (the so called knobs) like sampling rate, batch size, and type of hardware and can be tuned to meet the target latency while minimizing execution cost. As far as its architecture is concerned, it is a serverless framework by itself and doesn't rely on commercial platforms. Its main advantages are compatibility with heterogeneous hardware (CPU, GPU, FPGA, Vision Chips e.t.c.) and support for input dependent execution flow which is crucial for performance and efficiency.

2.2.2 FaastLane

FaastLane [30] doesn't offer a quality of service constraint, but it tries to minimize workflow latency by eliminating function interaction and exchanges of data between them. Instead of executing each function in a separate, isolated container, it encapsulates the entire pipeline inside a single container instance. Functions are executed as threads within a single process, while data sharing is been replaced by simple load/store instructions. Isolation of sensitive data is being implemented with Intel Memory Protection Keys (MPK) [31], while parallelism is possible with fork of processes or spawning of new containers.

2.3 Our Approach

Our approach focuses on serverless pipelines that utilize OpenWhisk framework deployed inside Kubernetes. In order to achieve the target time latency, it monitors all invocations inside the cluster and tries to distribute computing resources of each node fairly to function instances inside the same host. This distribution of finite resources follows a *"help the most needy"* policy, while an estimation for the requirements of each invocation is being given by a PID controller. For each sequence, metrics are collected by a separate serverless function who also acts as a timer by computing the current slack before each invocation. As far as efficiency characterization is concerned, various test cases were performed with different pipelines and sequence configurations. A comparison is been conducted with the default scheme of sequences that OpenWhisk provides, alongside the actively controlled Sequence Clock's pipelines.

Chapter 3

Background

The purpose of this chapter is the familiarization of the reader with technologies, tools and frameworks that are been studied and used extensively in serverless computing and the cloud world in general. Firstly, we introduce Kubernetes along with its features, as one of the most common orchestration engines. Then, concerning Serverless Frameworks, we focus on Openwhisk, while we also mention OpenFaas. Lastly, a brief presentation of the programming language named Go is included.

3.1 Cloud, Virtualization and Containerization

Understanding the importance of the following tools and their capabilities, can be achieved by looking various deployment options over the years. Traditionally, organizations ran applications on physical servers, where there is no way for defining resource boundaries. Thus, resource allocation issues had emerged and the proposed solution of running everything in an isolated and unique machine was not viable for obvious reasons. The introduction of virtualisation allowed the deployment of multiple virtual machines (VMs) on single physical servers' CPUs, while offering a level of security as the information of one application cannot be freely accessed by another application. Of course, drawbacks of this approach do exist with first and foremost performance. Virtualizing the entire OS is slow and that is why containerization, a new concept emerged. Just like VMs, containers utilize their own filesystem, share of CPU, memory, process space etc., but they are considered ligthweight and portable across platforms. [4, 9]

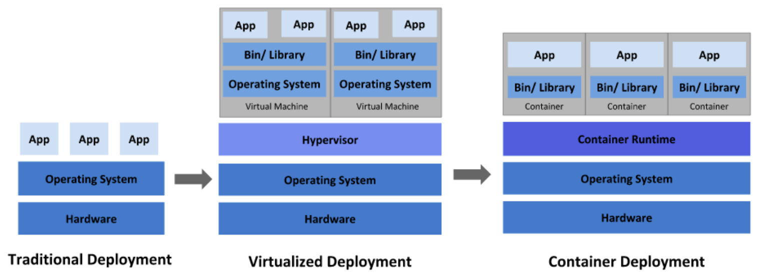


Figure 3.1: The evolution of Virtualisation and Containerisation [1]

Most serverless frameworks utilize containerization technology for isolating functions of the same or different tenants. This raises the problem of how automation of containerized workloads, load balancing, provisioning, and scaling are being handled. So, container orchestration engines come in place, while in fact some serverless frameworks utilize one. In general, container orchestration tools are mainly responsible for performing the following tasks [32]:

- Allocating resources among different containers.
- Scaling containers up or down based on workloads.
- Routing network traffic and balancing loads.
- Assigning services and applications to specific containers.
- Deployment and Provisioning.

As far as Apache Openwhisk's approach is concerned, it offers compatability with state-of-the-art orchestrators like Kubernetes, OpenShift and even Docker Compose. However, we are going to focus mainly on Kubernetes, as SC is built on top of it.

3.2 Kubernetes: A Container Orchestration Platform

Kubernetes, also known as K8s, is an open-source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation [9]. It was developed by Google in 2008 and handed over to the Cloud Native Computing Foundation (CNCF) in 2014 [32]. Its name originates from Greek (κυβερνήτης), meaning helmsman or pilot.

Figure 3.2: K8s Logo [2]

[1]https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
[2]https://github.com/kubernetes/kubernetes/blob/master/logo/logo.svg

K8s, in general, provides a framework for running distributed systems resiliently. Among others, this means that service discovery and load balancing is controlled with ease. In addition, storage orchestration is possible by mounting storage systems of various types, while self healing and automatic rollouts/rollbacks can be handled via simple configuration files or command line interfaces. Resource allocation or automated bin packing sets thresholds on the resources, e.g., amount of CPU and RAM, that containers have access to, making the best use of a system's resources. Last but not least, storage and management of sensitive information like passwords, OAuth tokens, and SSH keys is possible without the fear of exposure in the code stack, through the usage of secrets [9]

3.2.1 Kubernetes Architecture

A Kubernetes cluster consists of a non empty set of worker machines, called nodes, that are used for running containerized applications. The worker node(s) host the necessary components of the application workload. On the other hand, a set of control plane's components manages the worker nodes inside the cluster, making global decisions as well as detecting and responding to cluster events. For simplicity reasons, the latter components are placed on a single machine, called main or master[3], where placement of user oriented elements is discouraged. [4, 11]

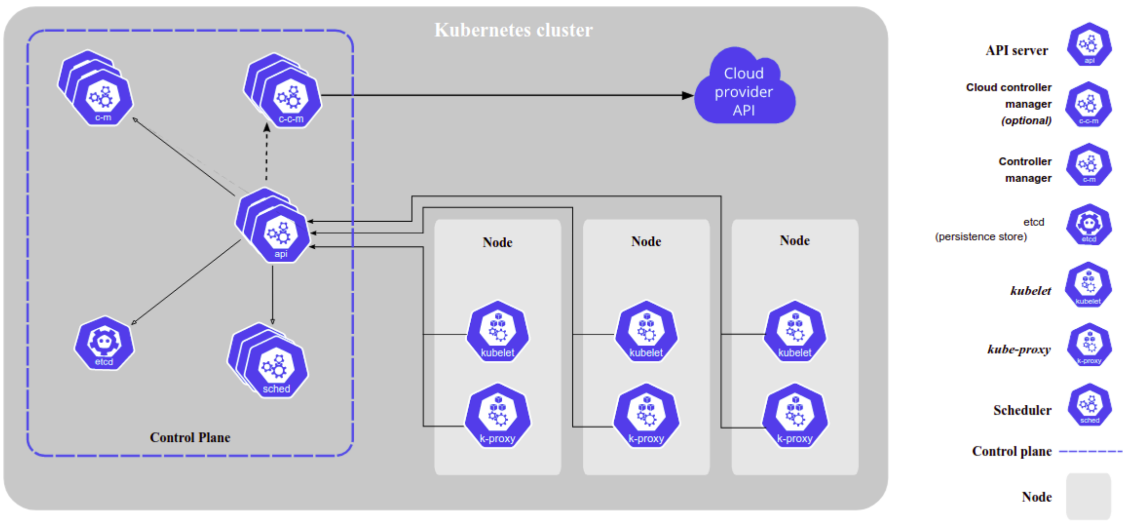


Figure 3.3: Components inside a Kubernetes cluster [4]

[3]The term was used in the past bibliography extensively, making its complete elimination difficult. As we acknowledge its political incorrectness, we will try to minimize its appearances in this text. We apologize beforehand for this situation.

[4]https://kubernetes.io/docs/concepts/overview/components/

Control Plane Components

This category of distinct K8s' units includes the following:

- *kube-apiserver*: The unit that exposes components inside the cluster to the outside world, through Kubernetes API. It is designed to be scaled horizontally, with traffic being balanced between its instances.
- *etcd*: Consistent and highly-available key value store used as Kubernetes' backing store [33].
- *kube-scheduler*: Responsible for watching newly created Pods with no assigned node, and selecting the proper node for placement. Key factors that result to this decision include: individual and collective resource requirements, hardware/software/policy constraints, affinity and anti-affinity specifications, data locality, inter-workload interference, and deadlines.
- *kube-control-manager*: The component/s that run control processes. Various types of controllers fall into this category, like node controllers, endpoint controllers, and service account & token controllers.
- *cloud-control-manager*: It embeds cloud-specific control logic, linking the local cluster into a cloud provider's API. It also separates out the components that interact with that cloud platform from components that only interact with the local cluster. If developers run Kubernetes on their own premises (e.g. inside a learning environment of a single PC), this unit is not present.

Worker Nodes Components

This category includes the following:

- *kubelet.* An agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. The kubelet takes a set of specs that are provided through various mechanisms and ensures that these containers are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.
- *kube-proxy.* A network proxy for each node, implementing part of the Kubernetes Service concept. It maintains network rules on nodes, allowing network communication between internal components of internal components with external elements.
- *container runtime.* The mechanism that runs containers inside a single node. Kubernetes supports several container runtimes like Docker, con-

tainerd, CRI-O, and any implementation of the Kubernetes CRI (Container Runtime Interface).

3.2.2 Kubernetes Objects

Kubernetes objects or resources (not to be confused with node resources) are persistent entities in the Kubernetes system. Kubernetes uses these entities in order to represent the state of the cluster and once they are created the entire system will constantly work to ensure their existence.

Creation, modification or deletion of objects is being held via the Kubernetes API. All the necessary information for such operation can be described into a yaml file, which then can be passed to various tools, like *kubectl* (kubernetes cli), or used by Helm Charts (more on that later). An example of this can be seen bellow.

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-deployment
spec:
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.14.2
        ports:
        - containerPort: 80
```

Almost every Kubernetes object includes two nested object fields that govern the object's configuration and that is *spec* and *status*. Spec must be set during the creation of the object, providing a description of the resource's characteristics, i.e. its desired state. On the contrary, status describes the current state of the object, supplied and updated by the Kubernetes system and its components. The Kubernetes control plane continually and actively manages every object's actual state to match the desired state [12]. Unfortunately, during this thesis we will not get in more depth concerning yaml format. Nevertheless, we encourage potentially interested readers to read the documentation for further understanding, if this is desirable.

It's almost impossible to mention and explain every single object type of Kubernetes, let alone user defined resources, but in the section that follows, we will try to present a series of important object types, which also played a significant role in SC's implementation.

Namespaces

Namespaces provide a mechanism for isolating groups of resources within a single cluster [13]. An object, depending on its type, might be either coupled with a specific namespace (services, deployments, etc.) or considered a cluster-wide object, which exists independent from namespaces' presence (e.g. persistent volumes). Same namespaced-objects ought to have different names, where this obligation isn't be cared accross different namespaces. We should point out that namespaces cannot be nested inside one another and each resource must only be in one namespace.

Pods

Pods are the smallest deployable units of computing that can be created and managed in Kubernetes [14]. To put it simply, a pod is a group of one or more containers, with shared storage and network resources, and a specification for how to run these containers. Details about how to be configured is kept inside the etcd component [4].

Usually, explicit pod definition and creation is not common for developers, as a workload resource might be used for this purpose, such as Deamon Sets or Deployments. These offer a layer of abstraction for dealing with the logistics of how pods are spun up, rolled out, and spun down [4]. Further more, pods can be used by two main ways. These include either the *one-container-per-pod* model, where the pod acts as a simple wrapper for the internal container, or the encapsulation of an application composed of multiple co-located tightly coupled containers. The aforementioned model should not be confiscated with a method called replication. This term refers to the procedure of scaling an application horizontally, by deploying multiple pods. [14]

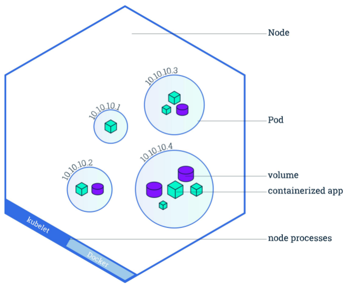


Figure 3.4: Node-Pod-Container abstraction levels [5]

[5]https://kubernetes.io/docs/tutorials/kubernetes-basics/explore/explore-intro/

Deployments

As it was described above, a Deployment can be characterized as workload resources. This means that a desired state is been described in a deployment and the deployment controller will try to configure the actual state to the desired one [15]. Among others, it is used to scale an app up or down, update an app to a new version, or roll back an app to a known-good version. In case of failures, deployments will try to reschedule faulty pods and always ensure the user that the required number of pods is present.

DeamonSets

A DaemonSet ensures that all nodes or a specified subset of them run a copy of a certain pod [16]. Addition or deletion of pods of interest is being handled dynamically with the addition or removal of cluster nodes. Possible cases, where this configuration is desirable, include:

- A cluster storage daemon on every node.
- A logs collection daemon on every node.
- A node monitoring daemon on every node.

StatefulSets

StatefulSets are yet another workload resource and used for provisioning statefull applications [34]. Unlike a Deployment, a StatefulSet maintains a persistent identifier for each of its pods, while providing a guarantee about the ordering and uniqueness of those. Thus, these pods are not equivalent nor interchangeable. In case of failure, these identifiers are used for matching existing Volumes (a term that we will analyze later) to new pods, that would be rescheduled and replace the old ones.

Services

Until now, we have discussed kubernetes resources relative to application deployment. It is obvious, that most of them will require communication with the external world or with others applications inside the cluster. The reason why this raises some basic concerns is the fact that pods are not permanent resources (thus the need for workload resources), while remain time independent. This means if a pod exists with a specific IP address at a certain point in time, there is no guaranty that this will continue to be valid in the future. This is where services fit into place. A Service is an abstraction which defines a logical set of Pods and a policy by which to access them [17]. In other words, it exposes an application running on a set of pods as a network service. Of course, there are various types of services, with actually most of them to be characterized by a superset-subset relation[35, 17]:

1. *ClusterIP*: The simplest option which exposes the service on a cluster-internal IP.
2. *NodePort*: Just like ClusterIP, it supplies the service with an internal IP, but in addition exposes the hole service on a static port of each node. In this way, the service is automatically reachable from within and outside the cluster, provided apriori knowledge of a node's external IP.
3. *LoadBalancer*: If the usage of an external cloud provider's load balancer is possible, then this service type is the way to go. Just like NodePort and ClusterIP, a LoadBalancer typed service is also a NodePort type service, meaning that it remains reachable with all the previously mentioned ways.
4. *ExternalName*: It's the only category that may be considered separate from the others. In short, it creates an internal service with an endpoint pointing to a DNS name.

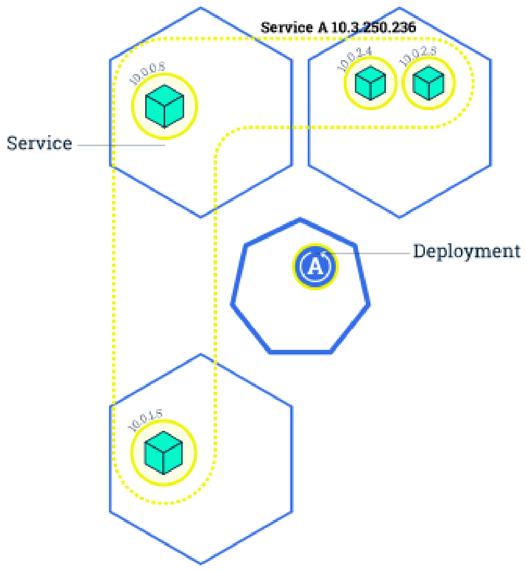


Figure 3.5: Kubernetes Resources Abstraction[6]

Persistent Volumes

On-disks files are ephemeral in containers just like containers themselves, which might introduce some serious problems. For instance, failure of a con-

[6]https://kubernetes.io/docs/concepts/overview/components/

tainer implies loss of any files that were in it. In addition, with the existing concepts, that we have already presented, file sharing between containers is just not feasible (ignoring of course transfer over the network). The suggested solution to these are, of course, Volumes. At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a pod [18]. A specific category of those are Persistent Volumes.

In contrast with all the above types of Kubernetes objects, Persistent Volumes are not namespace specific resources and can be referenced and used by any namespace object. A PersistentVolume (PV) is a piece of storage in the cluster that has been provisioned by an administrator or dynamically provisioned using Storage Classes[7]. PVs are volume plugins like Volumes, but have a lifecycle independent of any individual pod that uses the PV [19].

Persistent Volume Claims

As pods consumes node computing resources, a way for consuming persistent volumes and volumes in general is needed. Persistent Volumes Claims or PVCs is exactly that, a request for storage by a user. Claims can request specific size and access modes, (e.g., ReadOnlyMany) [19]. At this point we should clarify that persistent volume claims in contrast with persistent volumes are namespace specific.

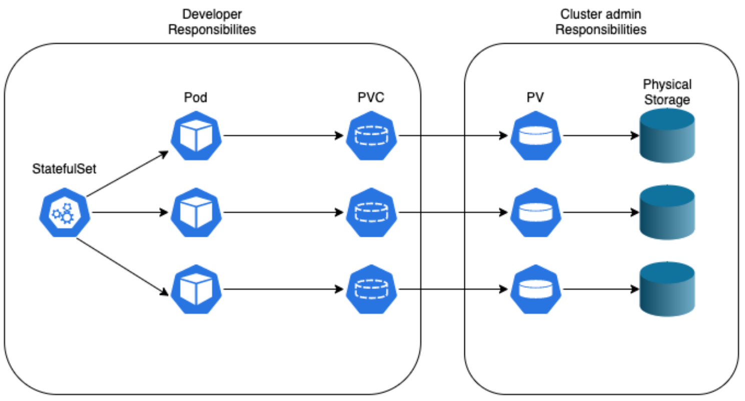


Figure 3.6: Abstract Representation of a StatefulSet, PVs & PVCs [8]

ConfigMaps

A ConfigMap is an API object used to store non-confidential data in key-value pairs [37]. This kind of information can be consumed by pods as environmental variables, command-line arguments, or as configuration files in a volume. It

[7]A Storage Class is another K8s resource, that provides a way for administrators to describe the "classes" of storage they offer. Different classes might map to quality-of-service levels, or to backup policies, or to arbitrary policies determined by the cluster administrators [36].

[8]https://kylezs.blog/persistent-storage-kubernetes/

must be noted, that they are not designed to store large chunks of data (actually there's a 1 MiB limit) and for such cases a type of Volume must be preferred.

Secrets

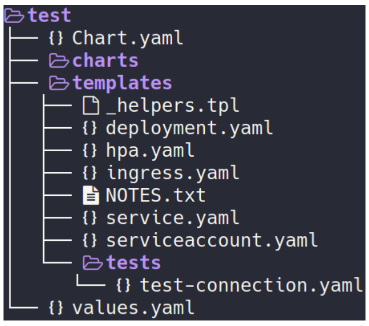
A Secret is an object that contains a small amount of sensitive data such as a password, a token, or a key [10]. As confidential data must not be exposed in application workflows, secrets are created independent of pods and deployments. Thus, during operations of creating, viewing, and editing pods and other resources of the cluster, secrets remain safe from potential intruders. Usage of secrets is similar with ConfigMaps but their purpose is storing sensitive information.

3.2.3 Helm & Helm Charts

As every resource is described by a yaml file, it should be clear by now that in cases of large applications and systems, it's not wise to explicitly load every single component of this application into the K8s cluster separately. In some cases, also, the developer might want to provide a specific parameter of the system during deployment time (e.g. an API key to be stored using a K8s secret). The need for addressing these issues gave birth to another concept, the concept of a package manager just for K8s resources.

Figure 3.7: Helm Logo[9]

In our case, Helm [38] is the package manager of Kubernetes and its packaging format is called Helm Cart. A chart is a collection of (mainly yaml) files that describe a related set of Kubernetes resources. This collection is stored using a particular directory tree and can be versioned, deployed, and published in specific archives. An example of this structure can be seen in the next figure.



```
test
├── Chart.yaml
├── charts
├── templates
│   ├── _helpers.tpl
│   ├── deployment.yaml
│   ├── hpa.yaml
│   ├── ingress.yaml
│   ├── NOTES.txt
│   ├── service.yaml
│   ├── serviceaccount.yaml
│   └── tests
│       └── test-connection.yaml
└── values.yaml
```

Figure 3.8: Helm Chart Directory Structure

[9]https://cncf-branding.netlify.app/projects/helm/

3.3 Apache OpenWhisk

In previous sections we explored Kubernetes, as an container orchestration engine. However, as the main goal of this thesis is resource management in serverless architectures it would be plausible to present a serverless framework. The one that we will focus on is called OpenWhisk [20]. It was initially developed by IBM, but now it is considered an Apache Incubator Project [39] and it is been administrated by the Apache Software Foundation (ASF).

Figure 3.9: OpenWhisk Logo[10]

3.3.1 General Information

Apache OpenWhisk is an open source, distributed serverless platform that executes functions in response to events at any scale [20]. Under the hood, it is using Docker containers for its functions, thus auto-scaling is easy and fast. As far as deployments options are concerned, it offers compatibility with many container frameworks such as Kubernetes, OpenShift, and Docker Compose. However, the community endorses deployment on Kubernetes using Helm charts (can be found in [21]), since it provides many easy and convenient implementations for both developers and operators alike [20].

Functions, which in OpenWhisk's case are called actions, can be developed in any supported programming language and can be dynamically scheduled and run in response to associated events (called triggers) from external sources (called feeds) or from HTTP requests. The growing list of those includes Go, Java, NodeJS, .NET, PHP, Python, Ruby, Rust, Scala and Swift. By the time of writing this thesis, an experimental runtime for Deno is in-development. Most importantly, even if the targeted runtime is not supported out of the box, developers have the option to create, customize, and bundle their own executables inside zip archives, which run on the Docker runtime. Last but not least, actions can be declaratively chained together to form higher-level programming constructs like sequences, whose performance will be targeted for regulation by SC.

[10]https://github.com/apache/openwhisk/tree/master/docs/images

3.3.2 OpenWhisk Architecture

Being an open-source project, OpenWhisk utilizes known projects like NGiNX, Kafka, Docker, and CouchDB. All of these components come together to form a “serverless event-based programming service” [23].

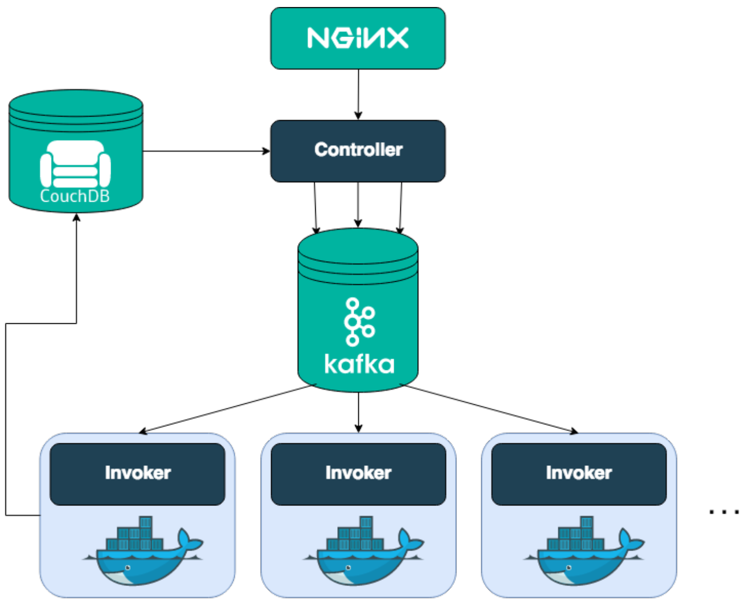


Figure 3.10: OpenWhisk Architecture

NGiNX

The entry point of an invocation is the NGiNX component. In general, NGiNX is an open source software for web serving, reverse proxying, caching, load balancing, media streaming, and other [22]. It was started out as a web server designed for maximum performance and stability, but also functions as a proxy server for email, a reverse proxy and load balancer for HTTP, TCP, and UDP servers. In the case of Openwhisk, it is used for SSL termination and forwarding appropriate HTTP calls to the next component, the Contoller.

Controller

It is a Scala-based implementation of the actual REST API. Thus, it acts as the governor and the interface for every user operation, including CRUD requests[11] for OpenWhisk entities and invocation of actions. Based on the HTTP method of received requests, it clarifies the type of operation. The controller communicates with the component of CouchDB, for authentication/authorization purposes and for fetching the necessary records of the requested resource.

[11]Create, Read, Update, Delete operations

CouchDB

CouchDB is a document-oriented database (NoSQL paradigm), in which each document field stores a key-value map. Fields can be either a simple key/value pairs, lists, or maps. A document-level unique identifier, as well as a revision number (for each change that is made and saved to the database) are been given to each document [40, 41].

Concerning OpenWhisk, CouchDB is used for registering information regarding credentials, metadata, namespaces as well as the definitions of actions, triggers[12] and rules[13]. Actions' records mainly comprise function execution code, default input parameters and resource restrictions imposed on the action itself during execution, such as the memory.

Load Balancer

Although Load Balancer is not a separate component, rather than a subpart of the aforementioned Controller, it is considered a crucial element of OpenWhisk. Its liability is maintaining a global view of the executors, called Invokers, available in the system by checking their health status continuously. In case of an action invocation, the Load Balancer, knowing which Invokers are available, selects the most appropriate of them to invoke the action requested [23].

Kafka

Apache Kafka is an open-source distributed event streaming platform used for high-performance data pipelines, streaming analytics, data integration, and mission-critical applications [42]. In other words, it is a framework for storing, reading, and analysing streaming data. Kafka was originally created at LinkedIn, for analysing the connections between millions of professional users. It was given open source status and passed to the Apache Foundation in 2011 [43].

As a messaging system, it is used for buffering and persisting communications messages between the controller and the invoker/s. This means that it lifts the burden of buffering in memory, risking an *OutOfMemoryException*, of both the Controller and the Invoker, while also making sure that messages are not lost in case the system crashes.

[12]Triggers are named channels for classes or kinds of events sent from Event Sources.

[13]Rules are used to associate one trigger with one action. After this kind of association is created, each time a trigger event is fired, the action is invoked.

Invokers

They are are been characterized as the heart of OpenWhisk. Their main goal is, of course, invoking actions in an isolated and safe environment using containers (mostly Docker containers). In order to do that, a container is spawned per function invocation, in which the action code gets injected. There, it gets executed using the parameters passed to it and after the result is obtained and a short period of time has passed, the container gets destroyed (approximately 10 minutes).

At this point, a specific configuration choice for the invoker/s must be mentioned and that is *Invoker Container Factory*. In more depth, there are two ways of deploying invoker components on Kubernetes [24]:

- *Docker Container Factory*: This implementation matches the architecture used by the non-Kubernetes deployments of OpenWhisk. When developers opt for it, invokers run on every cluster node and communicate directly with the local docker deamon, to whom they send requests for creating and managing user function containers. The primary advantages of this configuration are lower latency on container management operations and robustness of the code paths being used. Unfortunately, it does not leverages the simplicity in resource management and security offered by Kubernetes. Also, it can be used only if the underlying container engine is Docker.

- *Kubernetes Container Factory*: This implementation is more of a Kubernetes-native design. In this case, each invoker relies on Kubernetes to create, schedule, and manage the pods that contain the user function containers (a single container per pod). Obviously, that implies a higher latency is introduced in the system, which with no additional configurations may result to poor performance. On the other hand, it surely takes advantage of Kubernetes capabilities.

During SC's development, the latter option was preferred and studied extensively, whereas it provided us with higher control over the system.

3.3.3 Action Invocation's Steps

In the next lines, we will attempt to trace an action invocation and see how the various components behave in this simple scenario. Starting off, an invocation is the core concept of a serverless-engine functionality. That includes execution of code fed by the user into the system and return of execution results. It may occur either from associated events, external sources, or HTTP requests. As OpenWhisk API is completely HTTP based and follows a RESTful design (cacheable, stateless, etc.), an invocation is essentially an HTTP request against the OpenWhisk system, with roughly the below structure.

```
    POST /api/v1/namespaces/$userNamespace/actions/myAction
    Host: $openwhiskEndpoint
```

The variable *$userNamespace* is referred to that namespace (at least one), that the invoking user has access to. The concept of OpenWhisk's namespaces is similar to the one of Kubernetes, since its purpose is the same: user isolation between actions. However, these two terms are not equivalent and must not be confused. From this point on, references to both will occur and the distinction will be clear to the reader by the context.

So, the much discussed operation starts with a POST request to the API and the procedure contains the next steps:

1. After the HTTP request is handled from the NGiNX component, it is handed over to the controller.

2. The controller communicates with CouchDB (the *subjects* database to be precise) for authenticating the user and checking if he is authorized for such operation as the specified action invocation.

3. If the previous steps succeed, the controller queries from the *whisks* database of CouchDB, the action record and the information that comes with it.

4. The Load Balancer chooses the appropriate Invoker and a message is send to the selected component via Kafka.

5. The selected Invoker fetches action's code for CouchDB. Then, it either creates a Docker container immediately (when *Docker Container Factory* is enabled) or will request from Kubernetes the creation of a new pod (when *Kubernetes Container Factory* is enabled).

6. Input parameters are provided inside the function and the execution initiates.

7. After the function is executed successfully, the result is obtained by the Invoker.

8. The Invoker, then, stores the result into the *activations* database of CouchDB, alongside the log data that were fetched from the container. Also, it stores the starting and ending time of the invocation of the action in the same record.

Step 5 will occur in case that there is no available container/pod for this function. We will call this situation a *cold start*. In case there is already an available one, no new creation is needed as the function can be handled by that containerized runtime. This situation is characterized as a *warm start*.

3.3.4 Actions' Runtime

In this section, we will try to further explain the procedure that we call previously as *"function execution"*. Actions are stateless functions (code snippets) that run on the OpenWhisk platform. Each function receives a dictionary-like input parameter and returns another dictionary. In case of python, for instance, an actual dictionary is used for that purpose and in case of a Go function a map is used instead. An example of a python action is the following:

```python
    def main(args):
        name = args.get("name", "stranger")
        greeting = "Hello " + name + "!"
        print(greeting)
        return {"greeting": greeting}
```

In order for the specified container to run a function like the above, OpenWhisk uses a small HTTP server inside each container to provide two endpoints, */init* and */run*, while exposing them at port 8080. The first one is used for retrieving action code and storing it as a runable entity into the */action* directory. Compiled type runtimes have also the need to compile the given code on the fly. The previous stage is called *initialization*. After initialization, */run* endpoint is used for passing the necessary parameters into the runtime and finally run the handler [44].

Openwhisk uses Alpine Linux as the base image and it has created new Docker images for each supported language [45]. In each runtime, it has installed some basic dependencies and libraries. If external dependencies' usage is desirable, they must be placed inside the runtime. The process depends on language option, for example in Python and Go zip archives are an option, with virtual environments and vendor folders respectively [46, 47]. In case one provides their own image, it has to expose the endpoints */init* and */run* at port 8080, as it was described [45].

3.3.5 Special Topics

Pre Warm Containers

Cold and *Warm* starts have already been defined. However, there is also another type called *pre warm* and it refers to the situation where an available container is already deployed inside the cluster, but the initialization stage hasn't yet started [44]. This scenario is feasible with NodeJS containers, which remain deployed even after function termination. These containers/pods are not coupled with a specified function, but after an invocation the hole process of initialization must be restarted.

Concurrency

By default, each deployed pod/container is able to handle one request at a time. If an invocation of the same action is been triggered before the termination of the running one, OpenWhisk will spawn a new pod/container resulting a *cold start*. There is an option for increasing concurrency capabilities of a function, which behaviour will be discussed extensively in the next chapters as it has much to offer from a research perspective. In a nutshell, from our personal expirementation for functions based on Python and Go, increased concurrency will not result to parallel execution of invocations, but to a queuing effect.

3.4 OpenFaas

OpenFaas [48] is another popular serverless framework, which was studied during this thesis. Although, it was not used as the targeted environment for Sequence Clock, we believed that a brief overview is required. OpenFaaS is an open source serverless function engine where users can publish, run, and manage functions on Kubernetes clusters. It is worth mentioning OpenFaas Pro, a commercial distribution of the project for companies and enterprises with some additional features [49].

Figure 3.11: OpenFaas Logo[14]

The specified framework, when it's deployed inside a Kubernetes cluster, is managing the lifecycle of the containers through a custom Kubernetes controller, called faas-netes [50]. Other key differences with OpenWhisk are related to deployment and autoscaling methods. First of all, it does not dynamically pack and execute code at runtime, which means that developers need to predefine and pre-built docker containers with their funtion inside. In addition, two types of autoscaling mechanisms can be leveraged [51]. One is based on the simple metric of requests per second, while the other is scaling by CPU and/or memory utilization (only when OpenFaas is deployed using Kubernetes). Furthermore, scaling-up from zero replicas is enabled by default, which means that for every deployed function, there's always a running container inside the cluster. This leads to cold start elimination, but raises the problem of binding idle resources.

For creating a function, one must use existing templates with various supporting languages like Go, Java, Python, JavaScript, PHP, Ruby and C#. However, a template store provides many more. An example of a Go serverless function is presented below.

```go
package function

import (
  "fmt"
)

// Handle a serverless request
func Handle(req []byte) string {
  return fmt.Sprintf("Hello, Go. You said: %s", string(req))
}
```

[14]https://github.com/openfaas/faas

3.5 The Go Programming Language

Go [52] is a statically typed, compiled language aiming high performance server side applications. It was started back in 2007 by Robert Griesemar, Rob Pike, and Ken Thompson and was published in 2012 as an OpenSource project. Its design principles are simplicity and efficiency, but is mostly recognized for the extremely fast compile times (compilation result is a single binary).

Import and export of code between projects is handled with ease, as Go offers its own packaging and moduling system. Linking of published packages from GitHub is also available. Moreover, as it was developed in a world of multicore processors, parallelism and concurrency are supported out of the box with goroutines, which are go functions that can run at the same time by utilising multiple threads on a CPU.

```go
package main

import "fmt"

func main() {
  fmt.Println("Hello fellow Gophers!")
}
```

Notable projects that were implemented or still are maintained using Go are Docker, Kubernetes itself, Helm, and of course Sequence Clock.

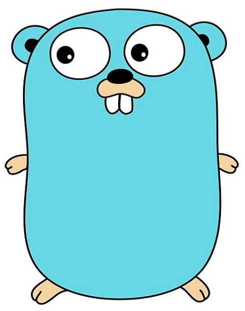


Figure 3.12: Go's mascot, the gopher

[14]Kenneth Lane Thompson (born February 4, 1943) is an American pioneer of computer science. Thompson worked at Bell Labs for most of his career where he designed and implemented the original Unix operating system. He also invented the B programming language, the direct predecessor to the C programming language.

Chapter 4

Motivational Analysis & Observations

In this chapter, we describe the computational infrastructure that was utilized during SC development, alongside a detailed presentation of the benchmarks suites, which were used. The latter include a third-party framework and a suite of benchmarks that were developed alongside Sequence Clock. Later in the same chapter, we explore various reasons that may lead a sequence's invocation into time latency violations.

4.1 Experimental Infrastructure

4.1.1 System setup

For the rest of the thesis, we consider a cluster of four virtual machines, deployed on a set of physical servers. In each one of them Ubuntu 20.04 LTS with 5.4 version of the Linux Kernel is installed as the operating system. Further specifications are illustrated in table 4.1.

#	Role	OpenWhisk Role	Cores	RAM	Swap
0	master	none	2	7.77 GiB	off
1	worker	invoker	4	15.6 GiB	off
2	worker	invoker	4	15.6 GiB	off
3	worker	invoker	2	15.6 GiB	off

Table 4.1: Cluster Specifications.

The first of these four nodes is used as the master node component (needed by Kubernetes), while the others are marked as workers. At the same time, each worker node is labeled with $openwhisk - role = invoker$ label, which means that actions' containers are placed only on those machines. All the workers have the same amount of memory equal to 15.6 GiB, while we opted for some diversity on CPU core count as the controlling parameter was based on it. The high ratio between system memory and CPU cores allowed us to deploy &

invoke several function containers inside the cluster, without worrying about RAM shortage and the cold starts that this would have resulted into.

Framework	**Version**
Docker	19.03.8
Kubernetes	v1.20.5
OpenWhisk	v1.0.0
Helm	v3.5.3

Table 4.2: Frameworks' Versions.

As far as OpenWhisk deployment is concerned, the yaml file with the configuration options is placed below. In more detail, *KubernetesContainerFactory* was selected (lines 23-26) as this provides more control over functions' containers over the *DockerContainerFactory* option. The available user memory was set at ~34.8 GiB (line 9), which allowed 4 GiB of free memory in each node[1]. Various limits (individual action concurrency, number of invocations per minute, and number of concurrent invocations) were increased (lines 10-15), while at the same time *persistence* was enabled (lines 16-19). The latter has been made possible by deploying a separate NFS server inside the Kubernetes' cluster (specifically inside master node), which manages automatic Persistent Volumes' creation. This configuration choice was vital as it permitted data preservation during node failures/restarts without a complete loss of application state or the need for the OpenWhisk Helm Chart re-installation.

```
1  controller:
2    replicaCount: 2
3  whisk:
4    ingress:
5      type: NodePort
6      apiHostName: 192.168.1.243
7      apiHostPort: 31001
8    containerPool:
9      userMemory: "35635m"
10   limits:
11     actionsInvokesConcurrent: 999
12     actionsInvokesPerminute: 999
13     actions:
14       concurrency:
15         max: 999999
16 k8s:
17   persistence:
18     hasDefaultStorageClass: false
19     explicitStorageClass: "nfs-client"
20 nginx:
21   httpsNodePort: 31001
22 invoker:
23   containerFactory:
24     impl: "kubernetes"
25     kubernetes:
26             replicaCount: 3
```

[1]Ubuntu Desktop 20.04 LTS recommends 4 GiB of system memory for seemingless experience, thus this threshold was used as a reference point.

4.1.2 Benchmarks Suites

Serverless Benchmarks Suite - SeBS

SeBS [25, 26] is a standardized platform for continuous evaluation, analysis, and comparison of FaaS performance, reliability, and cost-effectiveness. It provides support for automatic build, deployment, and invocation of benchmarks on AWS Lambda, Azure Functions, Google Cloud Functions, and a custom, Docker-based local evaluation platform.

It consists of six categories of benchmarks, with each one having separate serverless functions written in Python 3.6. These are:

- MicroBenchmarks: (*sleep* function)
- WebApps: (*uploader*, *dynamic-html* functions)
- Multimedia Apps:(*thumbnailer*, *video-processing* functions)
- Utilities: (*compression* function)
- Inference: (*image-recognition* function - not used in our experiments)
- Scientific: (*graph-bfs*, *graph-mst*, *graph-pagerank*, and *dna-visualization* functions)

Characterization is based on resource consumption footprint and applications' requirements (cpu utilization, memory, I/O, code size etc.).

As it was developed targeting known platforms and not for OpenWhisk usage out of the box, a lot of effort was put into converting them into deployable components. The first and more crucial problem that we encountered was about the usage of external libraries outside the default OpenWhisk's runtime for Python. More specifically, OpenWhisk uses pre-built images (with Alpine Linux as the final base image) that use version 3.6, 3.7 or 3.9 of Python with some common libraries, already embedded into them. In case one needs additional libraries, there are three specific ways of deploying such actions:

- Create a zip archive with action's code and additional packages' code in it, following a certain directory structure. Finally, provide this zip at the time of action's deployment.
- Create a virtual environment, where the additional packages will be installed. To ensure compatibility with the OpenWhisk container, package installations inside the virtualenv must be done in the target environment, thus the Docker image of OpenWhisk Python runtime must be used. Then this zip archive can be offered during action's deployment [46].

- Build an image from scratch by using the existing runtime of Python *openwhisk/python3action* as the base image. During build process, the developer has full control of what additional packages to install (Python modules or Alpine Linux Distribution's packages), what data files to store inside it etc. The final image can be finally deployed inside OpenWhisk as a blackbox [45].

The latter opens the door for third party Docker Images used for OpenWhisk actions. The caveat to this, is the fact that if none of the given runtimes is used as the base image, the endpoints (described in 3.3.4) that each action exposes must be implemented manually. The option that allowed us to successfully deploy the benchmarks was the third one with some of the reasons be the following:

- Some libraries required external package installation with the default package manager for Alpine Linux, *apk*.
- Some functions required additional configuration steps, like manual symbolic link creation for libraries [53].
- Some functions required some additional data as input in their runtime.
- Others needed compilation of a library from source code, as it was available in this specific distribution.

Further more, most of the non trivial benchmarks use external cold storage. For this reason, a MinIO service was deployed inside the cluster with a MinIO tenant placed stratigically in the master node, offering four *Persistent Volumes* of 5 GiB each for bucket creation. Last but not least, the functionality of each action, that was used during this thesis is presented below.

Uploader Function This function is a network intensive one as it downloads a file from a specific url and uploads it into the MinIO service.

Dynamic-HTML Function This function uses an HTML template stored inside the function's Docker runtime and according to input type creates a new one with dynamic size (the template contains an *ul* tag with a dynamic number of *li* elements). We note that we removed the actual HTML template from the result of this function, as it was violating the maximum output size that OpenWhisk allows.

Compression Function As its name implies, it downloads an entire directory of files from a bucket, compresses it into zip archive and re-uploads it in a different bucket.

Graph-BFS Function This function uses the *igraph* python module and generates a random graph based on the Barabási–Albert model[2] [54] with a variable size of vertices according to user's input. The function, then, performs a breadth-first search (BFS) on this graph and returns the computation time, alongside the size of the graph. We note that initially the function return value also included the result from the *bfs()* method, with the vertex IDs visited (in order), the start indices of the layers in the vertex list, and the parent of every vertex in the BFS. This information was later removed, due to the fact that large values of input size were producing even larger dictionaries as output, which violated OpenWhisk's maximum output size.

Graph-MST Function Similarly, to the previous function, *graph-mst* generates a random graph (based on the Barabási–Albert model) with a variable size of vertices according to user's input. For this graph, the function finds the minimum spanning tree (mst) by using Prim's algorithm.

Graph-PageRank Function This function performs the same operations as the two latter for creating a random graph. On this data structure, it calculates the Google PageRank[3] [55, 56] values of each vertex, i.e., a numerical weighting with the purpose of "measuring" its relative importance within the graph.

DNA-Visualization Function This functions utilizes the *squiggle* Python module [57], for visualizing a DNA sequence. Specifically, it downloads a file in the FASTA format[4], which stores the DNA sequence of the *bacilus subtilis* bacterium and then using this library it transforms it into a series of coordinates for 2D visualization. Unfortunately, for the processing of the entire file, that SeBS provided, the default 256 MiB of system memory that OpenWhisk provides was not enough. As the increase of the system's memory limit for only one function's container would have resulted into inconsistencies and asymmetry in our experiments, we used only the first 10000 lines of the related file as input.

Video-Processing Function This function calls the *ffmpeg* command for editing a video, downloaded again from the MinIO server. Available operations are

[2]A graph generated following th Barabási–Albert model is characterized as a scale-free network, which means that the probability $P(k)$ that a vertex in the network interacts with k other vertices decays as a power law of $P(k) \sim k^{-\gamma}$, with γ being typically in the range $2 < \gamma < 3$.

[3]PageRank algorithm was developed by Larry Page and Sergey Brin as the original Google Search algorithm. In April of 1998, Page and Brin published a research paper on the topic titled *"The Anatomy of a Large-Scale Hypertextual Web Search Engine"*.

[4]In bioinformatics and biochemistry, the FASTA format is a text-based format for representing either nucleotide sequences or amino acid (protein) sequences, in which nucleotides or amino acids are represented using single-letter codes.

watermark, gif or mp3 conversion, while we opted for the first one during our experiments. The final result is then re-uploaded in a different bucket.

Thumbnailer Function This action downloads a stream of data of an image, stored inside a bucket of the MinIO server, resizes it and re-uploads it as stream in a different bucket of the same server.

Sleep Function This function is a non trivial one, as its only purpose is to sleep for a certain amount of time, provided by user's input, and terminate afterwards. The largest available value that it was used during the experiments was one thousand seconds.

Each function was assigned a single number digit as an id, which was later used in the names of the created pipelines. For example, the pipeline *p021* consists of the functions *uploader,compression, dynamic-html* with that particular order. The related table with the id of each function is placed in 4.3.

ID	Function
0	uploader
1	dynamic-html
2	compression
3	graph-pagerank
4	graph-mst
5	graph-bfs
6	dnavisualization
7	videoprocessing
8	thumbnailer
9	sleep

Table 4.3: SeBS Functions IDs

Generic CPU Intensive Benchmarks

In order for the impact of CPU saturation to appear on the conducted experiments, the creation of CPU intensive benchmarks was crucial. In addition, these benchmarks had to require the minimum amount of all the other computational resources, i.e., system memory, I/O, cold storage usage, and network. For this purpose, six generic Python functions implemented from scratch with a structure like the following:

```python
def main(args):
    iterations = args.get("iterations",1000000)
        for i in range(4):
            foo = random.randrange(10,100)
            for _ in range(iterations):
                # ----------- Computational part -------------
                # The actual operations don't matter.
                foo += pow(foo + 42,2)/(pow(foo + 42,2) + foo)
                #---------------------------------------------
        return {"result":"Computation ended"}
```

The function consists of two nested for loops with the outer one having a constant iteration number of four and the inner part computing mathematical expressions over the same variable. The actual expressiom and result don't matter, so by trying various combination we can produce five additional actions. The sixth one was designed differently with sequential nested for loops and a light weighted mathematical operation into the final loop (a simple addition). Its code can be seen in the following snippet.

```
def main(args):
    iterations = args.get("iterations",10)
    for i in range(4):
        foo = random.randrange(10,100)
        for _ in range(iterations):
            for _ in range(iterations):
                for _ in range(iterations):
                    for _ in range(iterations):
                        for _ in range(iterations):
                            for _ in range(2):
                                foo = foo + random.choice([-1,1])
    return {"result":"Computation ended"}
```

As far as the constant outer loop pattern in every function is concerned, it was placed strategically in order to explore the behaviour of multi-threaded & multi-processed applications[5]. The aforementioned functions are by default single threaded applications, but by just using the *multiprocessing* and the *threading* modules of Python we can convert them into the desired ones with each iteration to be distributed to a single thread or process.

It is worth mentioning that by leveraging threads, the operation won't be executed in parallel but concurrently, so the application will not try to acquire more than one cores. The way of how this is done is possible is placed in the following code block.

```
from threading import Thread
import random

def bar(i,iterations):
    foo = random.randrange(10,100)
    for _ in range(iterations):
        foo += pow(foo + 42,2)/(pow(foo + 42,2) + foo)

def main(args):
    iterations = args.get("iterations",1000000)
    threads = []
    for i in range(4):
        threads.append(Thread(target=bar,args=(i,iterations)))
    for process in threads:
        process.start()
    for process in threads:
        process.join()
    return {"result":"Computation ended"}
```

[5]As we said earlier, the maximum number of cores per node in our cluster is four

In contrast, the *multiprocessing* module offers the usage of separate processes, which will leave their footprint in more than one CPU cores as they will try to get executed in parallel.

```python
from multiprocessing import Process
import random

def bar(i,iterations):
    foo = random.randrange(10,100)
    for _ in range(iterations):
        foo += pow(foo + 42,2)/(pow(foo + 42,2) + foo)

def main(args):
    iterations = args.get("iterations",2000000)
    processes = []
    for i in range(4):
        processes.append(Process(target=bar,args=(i,iterations)))
    for process in processes:
        process.start()
    for process in processes:
        process.join()
    return {"result":"Computation ended"}
```

Lastly, the naming convention for each function is *mt<i>generic* for the multithreaded, *mp<i>generic* for the mutli-processed and *st<i>generic* for the single-threaded. Accordingly, from those we constructed pipelines with three and six functions giving them the names *stg3*,*stg6*,*mtg3*, *mtg6*, *mpg3*, and *mpg6* respectively.

4.1.3 Monitoring & Metrics Collection Mechanisms

During the conduction of our experiments, a plethora of metrics were gathered concerning pipelines' behaviour, OpenWhisk's functionality, nodes' state, and more. For collecting all the necessary measurements, various bash[6] scripts were developed, with some of theme being presented in the next sections.

Functions' Metrics

As far as functions' behaviour is concerned, OpenWhisk offers extensive metrics from the logs of each function to the end to end latency, all accessible via the *CouchDB* component. In order to gather the above information automatically after each invocation, we developed the following bash script function.

```bash
function measurement(){
    local counter=0
    local id=$1
    local res=("norecord" 0)
    while [ $counter -le 2 ]
    do
        wsk -i activation get $id > temp 2>/dev/null
        if [ $? -eq 0 ]
        then
            res=($(awk -f ./measurement.awk temp))
            break
```

[6]Bash is the GNU Project's shell—the Borne Again Shell [58].

```
12          fi
13          counter=$((counter + 1))
14          sleep 30s
15      done
16      rm temp
17      echo ${res[@]}
18  }
```

This routine queries OpenWhisk's API by using the activation ID of the specified invocation (line 7) and returns the result status of this invocation alongside the total time of execution (line 17). The while loop of two iterations is used for polling the the database for the result. In situations of high load, OpenWhisk might write the new records after a period of time, thus the sleep command of 30 seconds. Even more, in extreme cases it is possible some records never to be inserted in the database. For this reason, we check if the query executed successfully or not (line 8), while after two unsuccessful queries, the functions returns the pair *"norecord" & 0*. In line 10, an awk[7] script is used for parsing only the necessary data from the entire activation record. Its contents are the following.

```
BEGIN {
  FPAT = "([^ ]+)|(\"[^\"]+\")"
}
/(\x22end\x22\x3A)|(\x22start\x22\x3A)|(\x22status\x22\x3A)/ {
    gsub(/,/,"")
    t=$2
    if ($1 == "\"start\":") {
        sum-=t
    }
    else if ($1 == "\"end\":") {
        sum+=t
    }
    else if ($1 == "\"status\":") {
        status=$2
    }
}
END {
    gsub(" ","",status)
    printf("%s %d",status,sum)
}
```

Pipelines' Metrics

Gathering pipelines' metrics follows the same pattern as the per function method. OpenWhisk puts executions times inside sequences' activation records, while at the same time adds the activation ID of each function inside the record of each sequence. The latter can be combined with the above scripts for gathering comprehensive information for each pipeline invocation. Thus, the *measurement()* function is used for per pipeline logging and the *measurement_per_function()* is used for the internal function logging.

[7]The awk language is a domain-specific programming language (DSL), specialized for textual data manipulation [59].

```bash
function measurement_per_function(){
    local id=$1
    local length=$2
    id_array=($(wsk -i activation logs $id | awk '{print $NF}'))
    res=()
    for function_id in ${id_array[@]}
    do
        res+=($(measurement $function_id))
    done
    echo ${res[@]}
}
```

Finally, all the data are saved in a csv format, for further manipulation with the Python seaborn library.

Nodes' Metrics

For completion purposes, nodes' utilization metrics need to be collected via the top command[8] and stored into csv format. This procedure has been made possible with the script below. In this, the top command runs in batch mode (it does not accept input) and fetches metrics every 0.5 seconds until the iterations limit (-*n* flag). The latter is computed using the experiment time (provided as a command line argument) plus a period of 120 seconds and multiplied by a factor of two, as each update is done every 0.5 seconds. Finally, output is piped into an awk script for parsing.

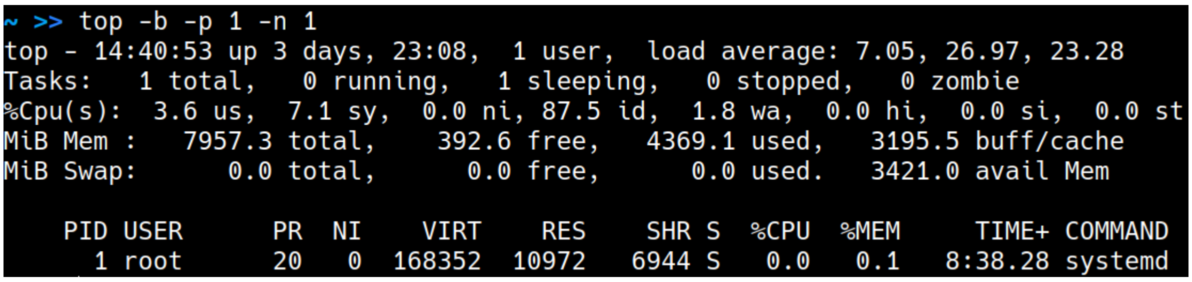
```bash
#!/bin/bash
DT="$1"
Texp="$2"
n=$(((Texp+120)*2))
LOG_FILE="top_usage.csv"

top -b -d .5 -p 1 -n $n | awk \
    -v cpuLog="$LOG_FILE"  -v dt="$DT" '
    /^top -/ {
        temp = 10
        for(i=1;i<=NF;i++) {
            if($i == "load") {
                temp = i + 2
                break
            }
        }

        gsub(/,/,".",$temp)
        load_avg = substr($temp, 1, length($temp)-1)
    }
    /Cpu\(s\)/ {
        cpu=$2
    }
    /^[K|M]iB Mem/ {
        used_mem = $8
        total_mem = $4
        printf "%s %.3f %.3f %.3f %s %s\n",load_avg,used_mem,total_mem,(used_mem/
    total_mem)*100,cpu,dt >> cpuLog
        fflush(cpuLog)
    }'
```

[8]The top program provides a dynamic real-time view of a running system. It can display system summary information as well as a list of processes or threads currently being managed by the Linux Kernel [60].

As one can understand from the awk part, the physical quantities that are used are the following:

- Load Average: This indicates the system load average over the last minute.
- CPU state percentage: The percentage of total time (since the last update) the CPU use for running un-niced user processes.
- Memory Usage: The awk script logs the amount of used system memory, the total amount of system memory and the related percentage.

For reference, a sample output (with the *-p* flag, output is being reduced into process' number 1 metrics, i.e., *systemd* in *systemd* systems.) of the top command for the master node is shown in Figure 4.1.



```
~ >> top -b -p 1 -n 1
top - 14:40:53 up 3 days, 23:08,  1 user,  load average: 7.05, 26.97, 23.28
Tasks:   1 total,   0 running,   1 sleeping,   0 stopped,   0 zombie
%Cpu(s):  3.6 us,  7.1 sy,  0.0 ni, 87.5 id,  1.8 wa,  0.0 hi,  0.0 si,  0.0 st
MiB Mem :   7957.3 total,    392.6 free,   4369.1 used,   3195.5 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.   3421.0 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+ COMMAND
      1 root      20   0  168352  10972   6944 S   0.0   0.1   8:38.28 systemd
```

Figure 4.1: Top sample output for master node.

4.2 Observing Target Latency

In the following lines we try to investigate various reasons that may lead a serverless sequence to time violations and by that to clarify the importance and complexity of this subject.

4.2.1 Impact of Resource Contention

It is only obvious that function containers, that coexist in the same host machine, compete for various resources. These might be the network, the time in the CPU, I/O, system memory etc. Moreover, the more resource types a function needs or to put it simply the more it depends on various resource types, the more severe and noticeable the effect will be in the function execution time elongation. Let alone function sequences where the delay stacks up.

To realize how this phenomenon is present and important even in well designed and much used frameworks like OpenWhisk, one can simply watch the behaviour of the end-to-end latency in regards to a quantity that represents the cluster's "pressure". For instance, in Figure 4.2 the end-to-end latency of the pipeline *p024879* as a function of rps values is plotted[9] (the values in each

[9]The steps of this experiment is explained thoroughly in Chapter 6.

rps level are placed in boxplots). Each invocation is using different containers in order to mimic resource contention of different tenants' sequences with similar footprint. This was done possible by using different aliases for both the sequences and the internal functions, for OpenWhisk to consider them as nonidentical.

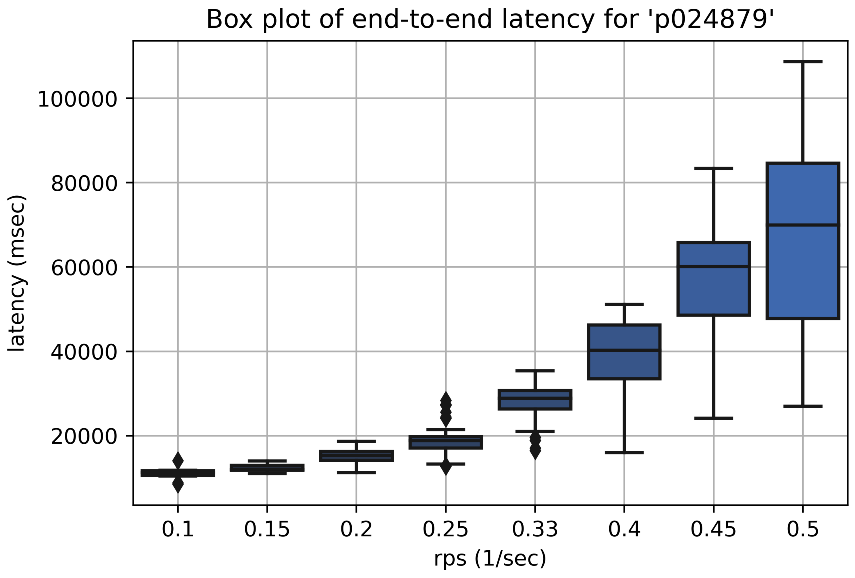


Figure 4.2: Pipeline *p024879* time latency for various rps levels.

From the above figure, two points should be made. First of all, it is clear that higher rps levels lead to higher values for end to end latency. In fact, for this pipeline, which depends on various system resources, the increase is exponential. Secondly, there's not only a mean level increase of the depending variable, but also an increase of the variance of each level distribution. This means, that not only a sequence gets delayed when more pressure is applied to the system, but also the physical quantity of time latency becomes unpredictable.

CPU Saturation

To make things worse, during our experiments we observe similar behaviour even for generic benchmarks that ping only the CPU. In Figure 4.3, the results of the sequence *stg6* are placed for the same experiment. In this case, CPU saturation is one of the main reasons (others are network delay, OpenWhisk queueing inside invokers etc.) behind this behaviour. In particular, for rps equal to 0.5, one invocation occurs every 2 seconds for 2 minutes. With the minimum value of time latency being over 20 seconds (actually the 25th-percentile minus the 1.5 of the interquartile range [61]), there is a descent amount of time, where at least one pipeline is executed at the same time with approximately

10 others[10]. More precisely, the cluster offers in total 10 cores (two nodes with 4 cores and one with 2 cores), thus the above statement implies that there is an amount of time where the total number of containers requiring a CPU is greater than the available one.

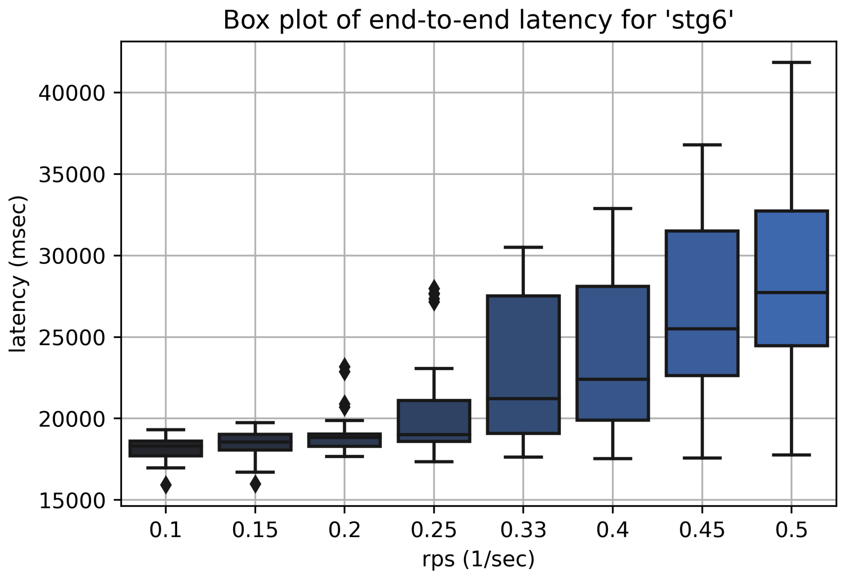


Figure 4.3: Pipeline *stg6* time latency for various rps levels.

[10]With $\frac{1}{rps}$ equal to 0.5 and execution time equal to 20 seconds, during the execution of a pipeline and after its invocation, $\frac{20}{2} - 1 = 9$ additional invocations will occur. By adding the 9 invocations that precede, during the entire life time of the sequence, 10 pipelines are running simultaneously.

4.2.2 Impact of Cold Starts

In Section 3.3.3, we introduced the concept of cold starts, according to which when there is no available Docker container for an action to be executed, a new one will be spawned adding a delay to the total execution time. How significant this delay is and how it may change in various states of the cluster was yet to be found. A way to test that, is to repeat the previous experiment and make sure that each function invocation suffers from a cold start. In the next figure, it is clear that such decision affects notably the time latency of the pipeline.

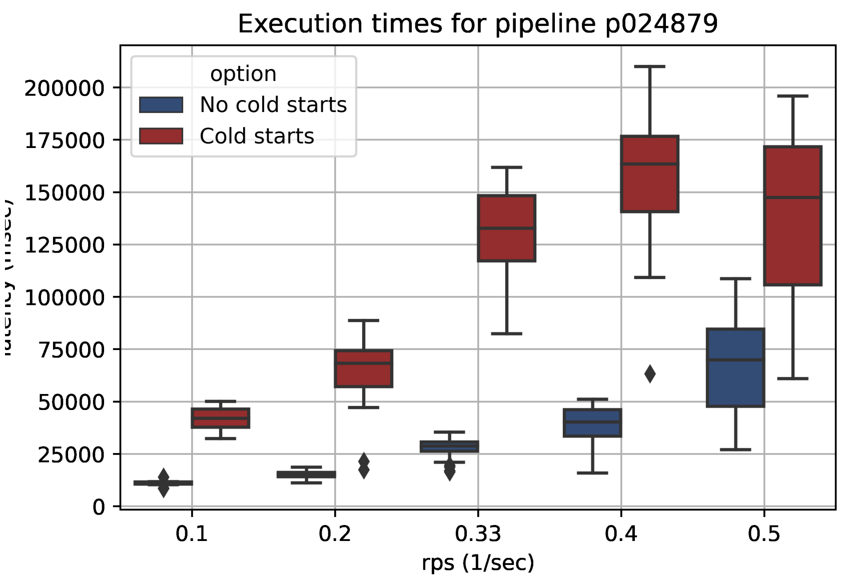


Figure 4.4: Cold Starts affection to time latency[11].

4.2.3 Impact of Concurrency & Queueing Phenomenona

Concurrency in actions is a subject that has already been mentioned in Section 3.3.5. One should easily understand that as resource contention is a concept to worry about in the node level, the same applies in the container level, when true concurrency is enabled (see prewarm containers in 3.3.5). However, this is not the purpose of this paragraph. Instead, it is important to note that in cases where OpenWhisk's runtime doesn't support truly concurrent invocations, even if the latter is enabled, a Queueing effect will rise.

For example, Python's runtime is implemented using the Flask framework version 1.0.2, which does not support concurrent requests. By conducting the already described experiment of sequential invocations of variable rps levels, while forcing each function to reuse the container of previous invocations, we

[11]We note that the decrease of the mean in the last rps level concerning the option for cold starts occurs as some containers may not got deleted by the last run resulting in some invocations to have found a ready-to-use container. However, the hole distribution remains worse than the other option.

observe the result of this design decision. In Figure 4.5, end-to-end latency is plotted as a function of the time of invocation for a period of 2 seconds and various rps values. Normally, a bell curve is expected. On the contrary, due to the aforementioned reason, a linear causality appears.

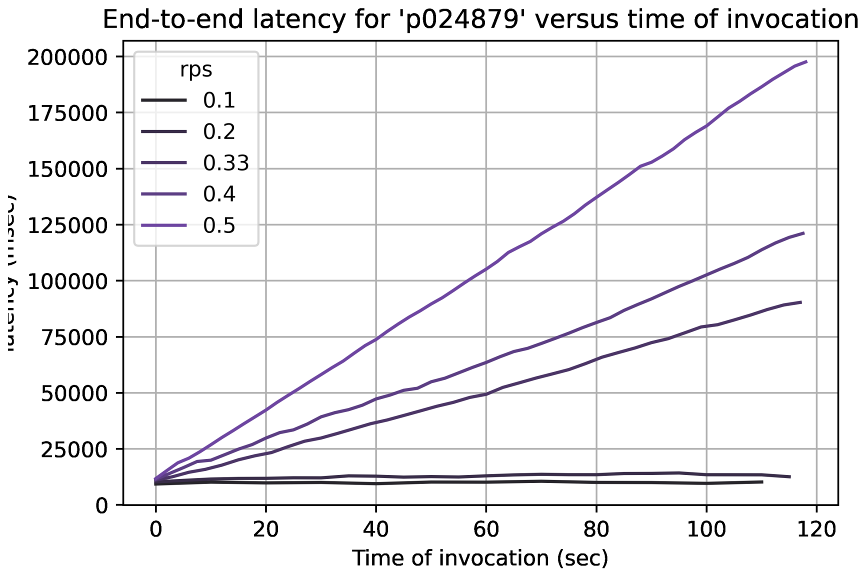


Figure 4.5: Queueing effect on time latency

4.2.4 CPU Quotas Affection

An additional factor that has an influence on target latency is of course how much time a function, i.e., a container, is allowed to use the host's CPU. A way to quantify this concept is through CPU quotas. In the lines that follow, we explore the affection of this metric to pipelines' latency. For this, we conduct a single invocation of a pipeline after previously we have tampered with the value of CPU quotas of the related Docker container. We repeat the process ten times, in order to get a proper sample, and for various percentages of the default CPU period of 100000 mseconds. The end-to-end latency the function *graph-bfs* in regards to these CPU quotas levels is placed inside Figure 4.6.

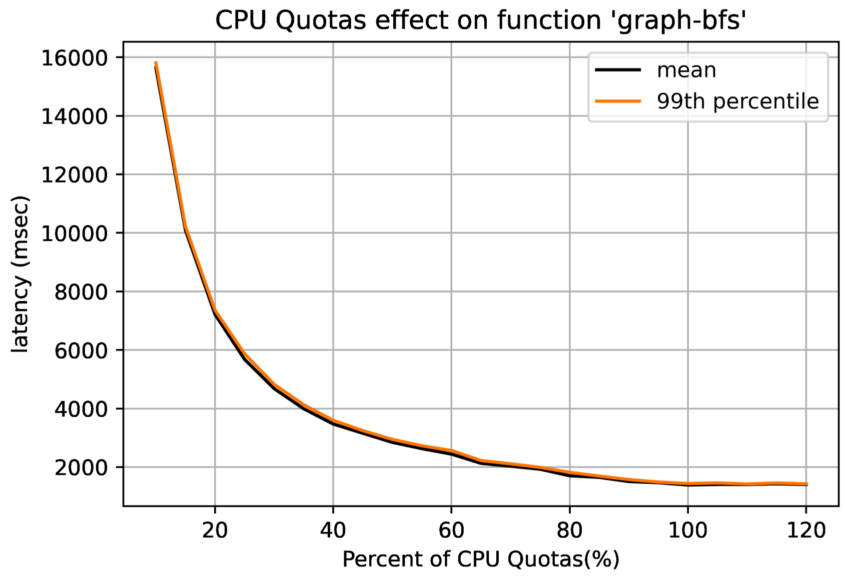


Figure 4.6: CPU Quotas affection on function time latency.

It is clear that with values less than 80 percent of a single CPU, the function's execution time starts to decelerate drastically. In addition, as the function does not leverage multiple cores, after 100 percent of the available CPU period the effects of quotas increase start to fade out. All the previous observations, that also apply for other functions as well, motivate us to use CPU quotas as a tuning parameter inside Sequence Clock.

Chapter 5

Sequence Clock: A latency targeting tool for serverless function sequences

5.1 Mathematical Modeling & Problem Definition

Before moving on with our proposal, it is worth mentioning some basic mathematical definitions and concepts that rule the entire system, alongside some conventions that we will follow from now on.

General Definitions

Let s be a serverless sequence or pipeline of n serverless functions f_0, f_1,..., f_{n-1}. We declare this by writing $s = (f_0, f_1, ..., f_{n-1})$, following a vector notation.

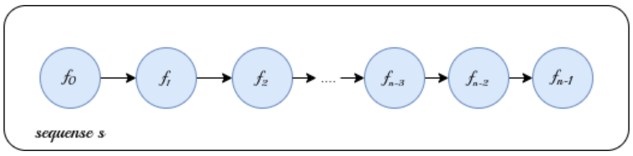


Figure 5.1: Abstract Representation of a Sequence

Let l be the targeted time latency of the sequence. During an invocation I, the measured execution time of function f_i is t_i for all $i, 0 \leq i < n$. We say that the sequence s did not met the target latency l, or violated the targeted latency l, or that a QoS violation occured when:

$$T_I(s) = \sum_{i=0}^{n-1} t_i > l \; for \; the \; invocation \; I$$

Respectively, we consider that sequence s met the targeted latency l, when:

$$T_I(s) = \sum_{i=0}^{n-1} t_i \leq l \; for \; the \; invocation \; I$$

In case of a QoS violation, we call the quantity $\frac{T_I(s)-l}{l}$ violation factor.

Problems' Definition

The QoS problem is defined as follows. For any set of m invocations I_0, I_1,..., I_{m-1} try to minimize the number $N_{violation} \leq m$ of the times that sequence s violated the target latency l. Equivalently, try to maximize the quantity $m - N_{violation}$. Another problem concering the QoS is the following. For any set of m invocations $I_0, I_1, ..., I_{m-1}$ and for any violation of s try to minimize the *violation factor*. The above two problems are not equivalent, but a well behaved resource management system should target both.

Slack

For a given experiment I of the sequence $s = (f_0, f_1, ..., f_{n-1})$ with target latency l, let t_i be the measured execution time of function f_i with $0 \leq i < n$. If we know that each function f_i should run under τ_i time, with $\sum_{i=0}^{n-1} \tau_i = l$, we define slack as follows:

$$slack_i = slack_{t_{i-1}} = slack_{f_i} = \sum_{j=0}^{i-1} \tau_j - \sum_{k=0}^{i-1} t_k = \sum_{j=0}^{i-1} (\tau_j - t_j)$$

for function f_i $1 \leq i < n$

- If $slack_i > 0$, then sequence execution is ahead of the target latency before the execution of function f_i.
- If $slack_i < 0$, then sequence execution is behind the target latency before the execution of function f_i.

We also consider *slack* for the first function f_0 to be equal to 0, $slack_0 = slack(f_0) = 0$

5.2 Proposed Solution: Sequence Clock

In this chapter, we present and analyze Sequence Clock in depth, a latency targeting tool for serverless function sequences for OpenWhisk deployments inside Kubernetes.

5.2.1 An Overview of the Sequence Clock's Architecture

SC is a distributed system, which consists of three different components, i.e. *Deployer*, *Watcher Supreme*, and *Watcher/s*. Each one of them is considered a Kubernetes resource and exists inside the cluster independently. In addition there is also another element called *Sequence Controller*, which is leveraging OpenWhisk by being a serverless function by itself and being mapped to a unique serverless sequence. An abstract representation of the entire stack is given in Figure 5.2.

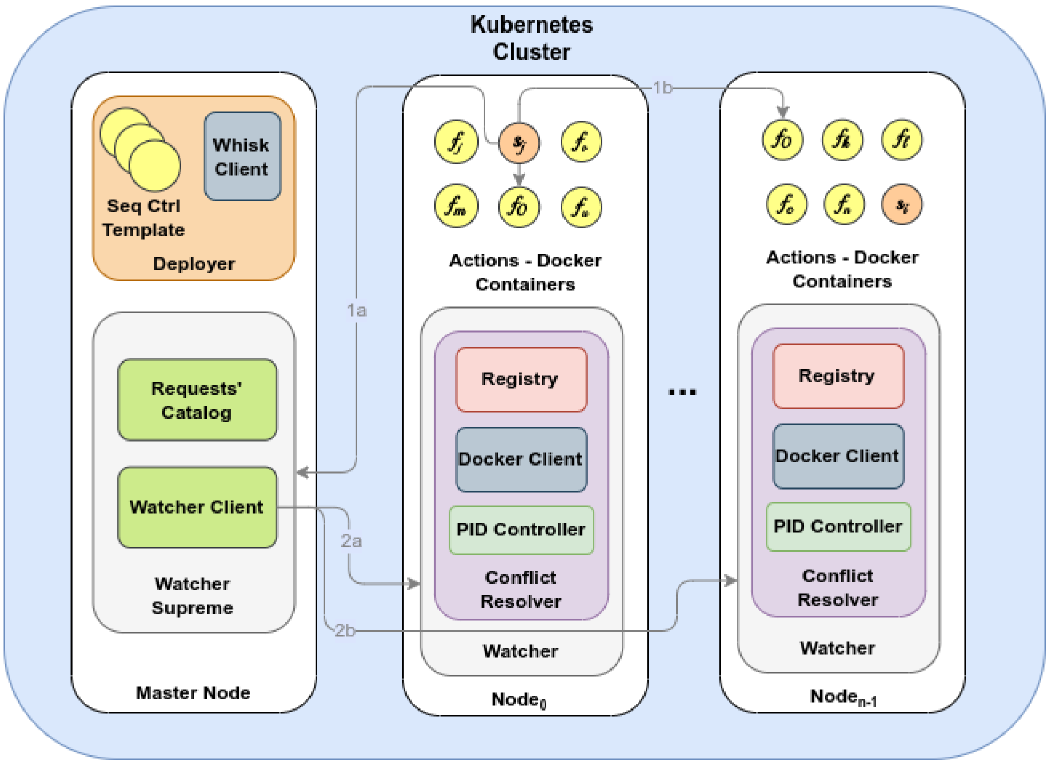


Figure 5.2: Sequence Clock Architecture

In a nutshell, Sequence Clock deploys each controlled sequence, named *Sequence Controller*) (marked as orange circles in Figure 5.2), as an OpenWhisk action, whose only purpose is to invoke (1b) each function (marked as yellow circles in Figure 5.2) of the pipeline with the proper order and gather metrics. The same component, during pipeline execution, communicates (1a) with the centralized service of *Watcher Supreme*, to whom it sends requests for resource allocation (computing resources). The latter transfers these requests (2a, 2b) to the *Watchers*, which are deployed in each node and they are responsible for managing and distributing computing resources in the node level as fair as possible. Each *Watcher*, depending on the received requests, offers further computing power to containers of sequences with negative slack, while it reduces the same quantity in containers of sequences with positive slack. The regulated parameter that was used is CPU Quotas, while its effects on containers' performance were studied extensively.

5.2.2 Deployer

The *Deployer* is a simple component that aims to automate sequence creation and deployment. It runs an HTTP server, implemented with the Gin

Web Framework [62] and exposes three endpoints */api/check*, */api/create*, and */api/delete* of a REST API.

- */api/check*. This endpoint is used as just the liveness and readiness probe for Kubernetes controllers. In short, K8s pings this periodically, in order to ensure healthyness of the resource.
- */api/create*. This endpoint is available for POST requests as it provides the functionality of creating a new sequence (*Sequence Controllers*) of SC. Information about this sequence, like the name, functions involved and others, are sended URL encoded using an HTTP form.
- */api/delete*. It is used for deleting existing sequences, while the name of the function is expected as a query parameter.

The structure of the Deployer is simple, yet important for leveraging Sequence Clock as a usable tool. In Figure 5.3, all the inner components are presented, as well as the operations that follow a request for sequence creation (seen as arrows). After a POST request is received, the server calls *template handler* (1), which is an abstraction for creating a new sequence. *File Zipper* is called (2) from the *template handler* and it fetches a template of code for the *sequence controller* (we are describing it more thoroughly in the following paragraphs). After injecting sequence's information in that template, it zips the template with all the necessary dependencies into an archive (3). Finally, the *template handler* provides (4) the archive to the inner *whisk client* (a wrapper for the OpenWhisk Go client), which sends it to the OpenWhisk's API deploying a new sequence.

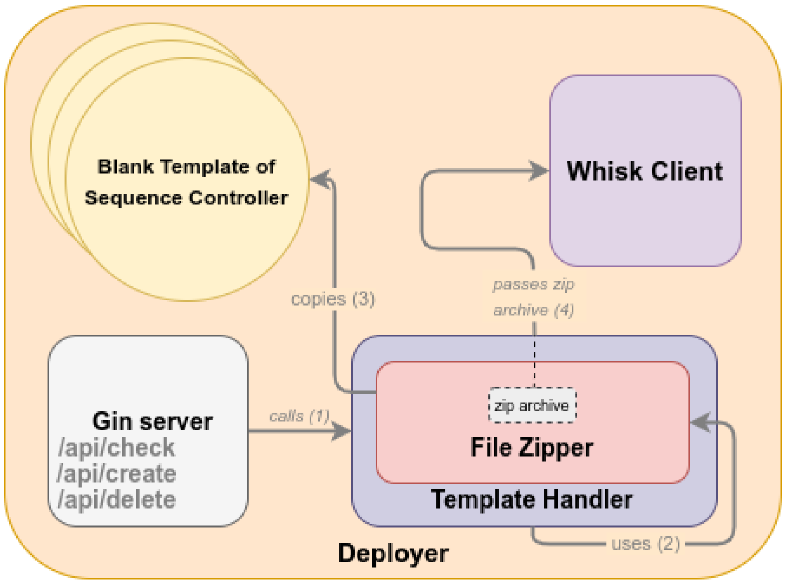
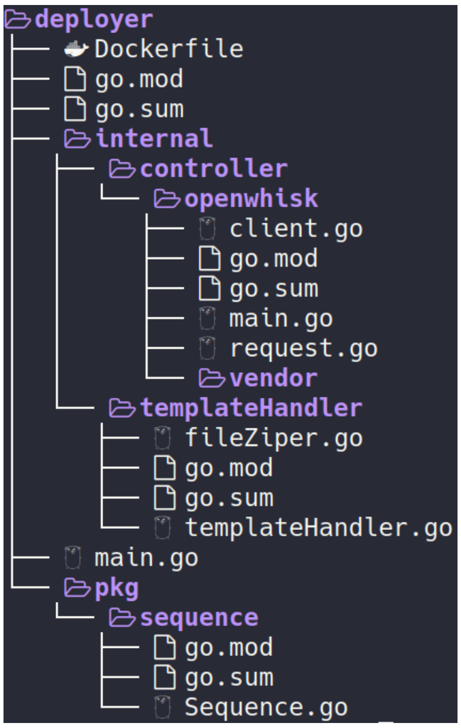


Figure 5.3: Deployer Abstract Representation

The file structure of this sub project, which functionality was discussed earlier, is given in the next figure.



```
deployer
├── Dockerfile
├── go.mod
├── go.sum
├── internal
│   ├── controller
│   │   └── openwhisk
│   │       ├── client.go
│   │       ├── go.mod
│   │       ├── go.sum
│   │       ├── main.go
│   │       ├── request.go
│   │       └── vendor
│   └── templateHandler
│       ├── fileZiper.go
│       ├── go.mod
│       ├── go.sum
│       └── templateHandler.go
├── main.go
└── pkg
    └── sequence
        ├── go.mod
        ├── go.sum
        └── Sequence.go
```

Figure 5.4: Deployer's Files' Structure

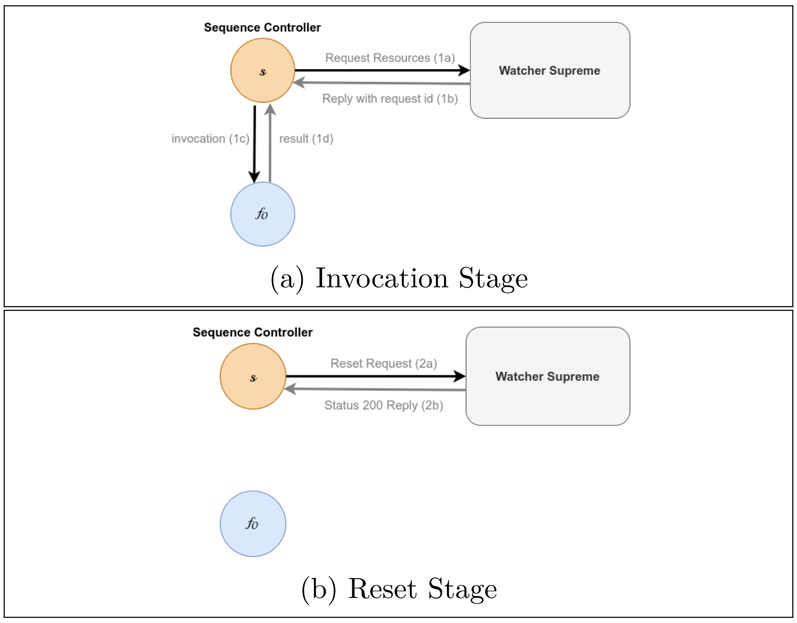
As far as Kubernetes leverage is concerned, the entire *Deployer* is a K8s' deployment, which handles one pod (with one container). Its API is exposed using a NodePort service, while the entire deployment is placed specifically into the master node. In addition, the large file structure of *sequence controller* is stored in a persistent volume inside the same node.

5.2.3 Sequence Controller

Sequence Controller is the only component that is not exactly deployed. SC is designed with respect to Apache OpenWhisk, thus it would be inappropriate to implement the metrics gathering mechanism and timer component of the system as a centralized and persistent element.

Instead, for each sequence invocation a serverless action, the *sequence controller* of this sequence, is executed, whose purpose is to invoke each function and transfer its output to the next one. In addition, it is responsible to compute the *slack* value for each function, as it was defined in section 5.1, as well as to gather other metrics like the sum of previous *slacks* and the *slack* of the previous function. All these measurements are then sent to the centralized element of SC, called *Watcher Supreme*, in order for the regulation mechanism to get initiated. This entire process is presented with pseudocode (python like) in the next lines and in the Figures 5.5a, 5.5b.

```
algorithms = {
    "greedy":greedy,
    "dummy":dummy,
}

def main(args):
    return algorithms[ALGORITHM](args)

def dummy(args):
    res = args
    for f in functions:
        res = invoke(f,res)

def greedy(args):
    result = args
    slack = 0
    sum_of_slacks = 0
    previous_slack = 0

    for f in functions:
        t_start = Now()
        request_resources_from_watcher()    # 1st request
        result = invoke(f,result)           # function invocation
        reset_request_to_watcher()          # 2nd request
        t_end = Now()
        elapsed_time = t_end - t_start

        # Gather metrics
        previous_slack = slack
        slack = slack + profiled_time(f) - elapsed_time
        sum_of_slack = sum_of_slack + slack

    return res
```



(a) Invocation Stage

(b) Reset Stage

Figure 5.5: Sequence Controller steps

Two elements must be pointed out. First, one can see two algorithms available for the *sequence controller*. They were implemented for experimental and comparison reasons, with *"greedy"* being the actual component of the system and *"dummy"* being, as the name implies, an approach that only calls the functions of the sequence. Second, two requests to *Watcher Supreme* exist. The first one is used for requesting resource allocation based on the slack value, while the second one informs the system for the termination of each function.

5.2.4 Watcher Supreme

The role of *Watcher Supreme* is central but not specifically complex. It acts as the entrypoint of *sequence controllers* to node *Watchers*. Instead of pinging each node separately for the aforementioned requests, a *sequence controller* communicates asynchronously with only one component, the *Watcher Supreme*. The latter has the responsibility of finding in which node a function's container is or will be placed. When it obtains this information, i.e., when a node replies positively, the specified component transfers the resource allocation request to the related *Watcher*, for the regulation to start.

In its core it uses two dictionary-like data structures, which are called *maps* in the Go programming language:

- *Request Catalog*: A map with a unique id for each request is used as the key value. Similarly, a reference[1] to the node[2], to which this request was handed over, is been stored as a value. The purpose for this structure is for the reset request that follows a function termination to be sent immediately to the specified node (that handled the initial request for resource allocation).
- *Function catalog*: It is used as a caching mechanism, while the *Watcher Supreme* tries to remember which node replied for which function[3]. It is also a map structure with function names as keys and references to the node *Watchers* as values.

The Gin server, inside it, exposes again three separate endpoints, offering a small RPC API:

- */api/check*: Likewise, a liveness and readiness probe for Kubernetes.
- */api/function/requestResources*: It is used for requests that demand resource allocation.
- */api/function/resetResources*: It is used for requests that inform the system for a function termination (in order to free the supplied resources).

Pseudocode of each handler of the above endpoints is placed below.

[1]Go supports pointers.

[2]Actually, the node client that is used for the communication between *Watcher Supreme* and *Watchers*

[3]Kubernetes prefers to place pods into nodes that already have downloaded the necessary image. Thus, it is wise to ask first the node that recently replied positive for the specified action.

Resource Request: /api/function/requestResources

```
def requestHandler():
    mutex_lock()
    request_id = counter
    counter = counter + 1
    mutex_unlock()

    requestResourceAllocationFromWatchers()

    return
```

As seen above, a mutex mechanism is utilized for synchronization purposes, when a new id is produced for a new request.The function *requestResourceAllocationFromWatchers()* is executed asynchronously with a go routine and its functionality was basically described above. That is node discovery for the container related to the specified function invocation.

```
def requestResourceAllocationFromWatchers(function):
    mutex_read_lock()
    read functionCatalog[function]
    mutex_read_unlock()

    if function in functionCatalog:
        sendRequest(node)
        if node repliess:
            mutex_lock()
            requestCatalog[id] = node
            mutex_unlock()
            return

    for node in all nodes:
        sendRequest(node)
        if node replies:
            mutex_lock()
            requestCatalog[id] = node
            functionCatalog[function] = node
            mutex_unlock()
            return
    return
```

If a cold start is going to occur, no related container is present inside the cluster by the time of this operation, thus no *Watcher* node will reply positively. In this case, nothing will happen and function execution will take place without SC provision and control mechanism. In early development stages, this situation was being handled with a polling mechanism by the *Watcher Supreme*. Unfortunately, this approach was producing a lot of network traffic (even if node polling had been preceded by a sleeping period) and the *Watcher* nodes were being used wrongfully. This resulted in poor performance and thus we chose to remove this feature in future implementations. In any case, further research is needed, related to cold start handling.

Reset Request: /api/function/resetRequest

As the previous request has demanded for further resource allocation (or resource reduction in case of a positive slack), a mechanism for returning each container into the initial state is needed. That is why this type of request is necessary.

```
def resetHandler(id,function):
    resetRequestToWatchers(id,function)
```

Similarly to *requestHandler*, *resetHandler* does nothing further than calling asynchronously *resetRequestToWatchers* (using a goroutine). Pseudo code for this code routine is placed below.

```
def resetRequestToWatchers(id,function):
    while True:
        mutex_read_lock()
        node = requestCatalog[id]
        mutex_read_unlock()
        if node found:
            break

        sleep for 5 msecs

    sendRequest(node)
    mutex_lock()
    delete(requestCatalog,id)
    mutex_unlock()
```

The loop, again, is used as a polling mechanism in the extreme case of really short lived functions (e.g. under 5 msec). In this case, there is a small chance of the reset request to start being handled before the termination of *requestResourceAllocationFromWatchers*, as the latter needs to communicate with the *Watchers*. If that occurs, without the loop mechanism, the specified request will not be found inside *requestCatalog* and the process will fail.

It is considered essential to observe that every operation upon *requestCatalog* and *functionCatalog* is executed with the usage of locking mechanisms to avoid unwanted, unpredictable behaviour, due to dirty read/write operations.

As far as Kubernetes resources are concerned, *Watcher Supreme* is a deployment, which admins one pod, strategically placed inside the master node, while a NodePort service is used for making the RPC API available to sequences.

5.2.5 Watcher

This is the main component of the system. A daemon set is used for deploying exactly one *Watcher* to each cluster node, (except master node), while port forwarding allows for each *Watcher* to listen on port 8080 of the host machine. Its main purpose is to balance computing resources of each node and to regulate sequences' time latency based on the metrics that *sequence controllers* send. As it can be seen from the figure below, it consists of two different components, i.e., the Gin server and the *Conflict Resolver*.

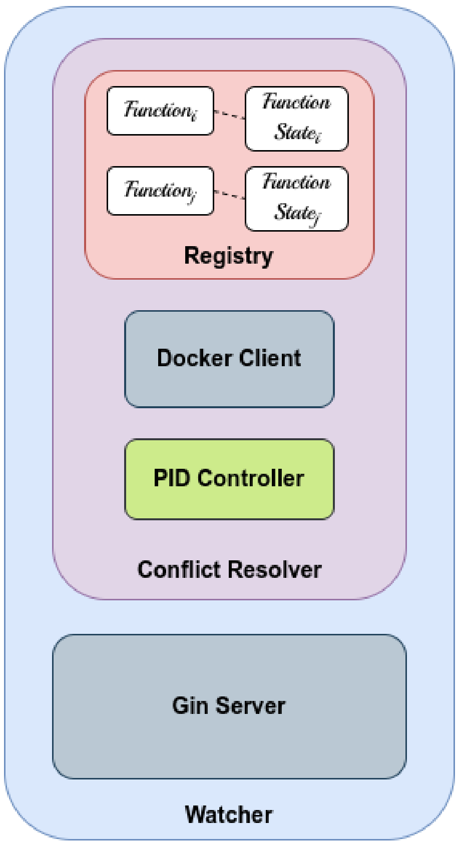


Figure 5.6: Watcher's Structure

RPC API

Gin server exposes again three endpoints: */api/check*, */api/function/resetRequest*, and */api/function/requestResources*. The first one shares the same existence purpose like the aforementioned same named endpoints, while the others are used from the same services of *Watcher Supreme*. In more depth:

- */api/function/requestResources* is used from *Watcher Supreme* to ask from the *Watcher* to allocate more or less resources to the specified function (to the related container inside the node, to be precise). If no such container exists, *Watcher* will reply with a status 404 error response.
- */api/function/resetResources* is used again from *Watcher Supreme* to inform the *Watcher* that the related action has ended and no more regulation is needed.

Conflict Resolver

Conflict Resolver maybe is the most important subcomponent, as it monitors containers' resources and communicates with the Docker Daemon inside the node via the socket /var/run/docker.sock, thus the *Docker Client* component in Figure 5.6. Moreover, it leverages the *Registry*, a data structure that stores all the ongoing action related requests and the state of each related container.

Registry

Registry is actually a dictionary that uses action names as keys. It was designed with the principle that each user action invocation results to one pod/-container inside the cluster. Such thing is not entirely true, as OpenWhisk will try to scale out and spawn more containers if the invocation rate is high, i.e., higher than the execution rate with concurrency of the function being equal to one. Fortunately, if one enables concurrency in such actions, the above statement corresponds to reality.

On the other hand, the values of the dictionary are structs of type *Function-State*, which store information like the container ID, CPU quotas of the same container and the desired CPU quotas (the two quantities may vary in some cases, as we will describe in 5.2.6). In addition to this, the id of the current execution request is stored along with the ids of similar requests and the desirable CPU quotas values.

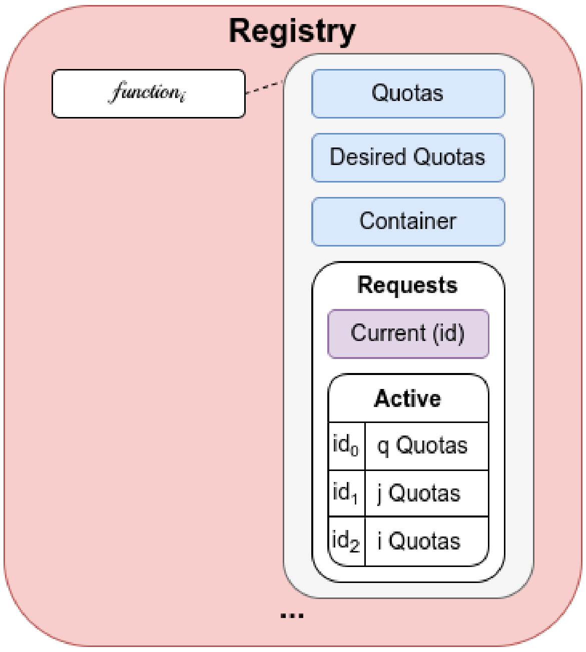


Figure 5.7: Registry Data Structure

PID Controller

The *PID Controller* is a Go function that accepts the metrics, that is the *slack* (used by the proportional term), the sum of previous *slacks* (used by the integral related term), and the previous *slack* (used by the derivative related term) sent from *sequence controllers*. Then, it returns the amount of CPU Quotas increase or decrease. As there is no fear of overshooting, only the proportional and the integral terms were used.

```
def controllerPID(metrics):
    return -1 * (
        Kp * metrics.slack +         # proportional term
        Ki * metrics.sumOfSlack +    # integral term
        Kd * metrics.previousSlack  # derivative term
    )
```

It is worth noticing the minus one term inside the function. It is used to invert the final output, as negative values of slack should produce positive output, thus allowing the *Conflict Resolver* to allocate more resources to the specified function.

A simple example

Until now, the complete architecture of *Watcher* has been presented, but no further information has been given on how these components are combined. In the next lines, we will try to cover this topic by studying an example of a *resourceRequest*, which is also being presented graphically in Figure 5.8. In this situation, the steps that follow are:

1. Server receives a POST request for the action f_i (which means that a *sequence controller* is about to invoke it).

2. The server asks *Conflict Resolver* if such container exists.

3. *Conflict Resolver* first checks the *Registry* and if an f_i key is not present, it will utilize the Docker Client to search the local Docker runtime.

4. *Conflict Resolver* uses the PID controller, to which sends the metrics from the specified request. An (algebraic) amount of CPU Quotas offset, relative to the base value of CPU quotas (equal to the default amount of CPU period in ms, in our case 100000 ms) is being produced.

5. The specified container gets updated with the offset CPU Quotas plus the default value. A trimming process occurs in order for the regulated value not to violate the total number of available quotas in the node, i.e., the number of cores multiplied by the default CPU period. Respectively, a minimum of five thousand (5000) quotas is used.

6. The *Conflict Resolver* updates the entire *Registry* and the Docker Runtime using the *Resource Distribution Algorithm* (5.2.6) and *Conflicts Policy* (5.2.7).

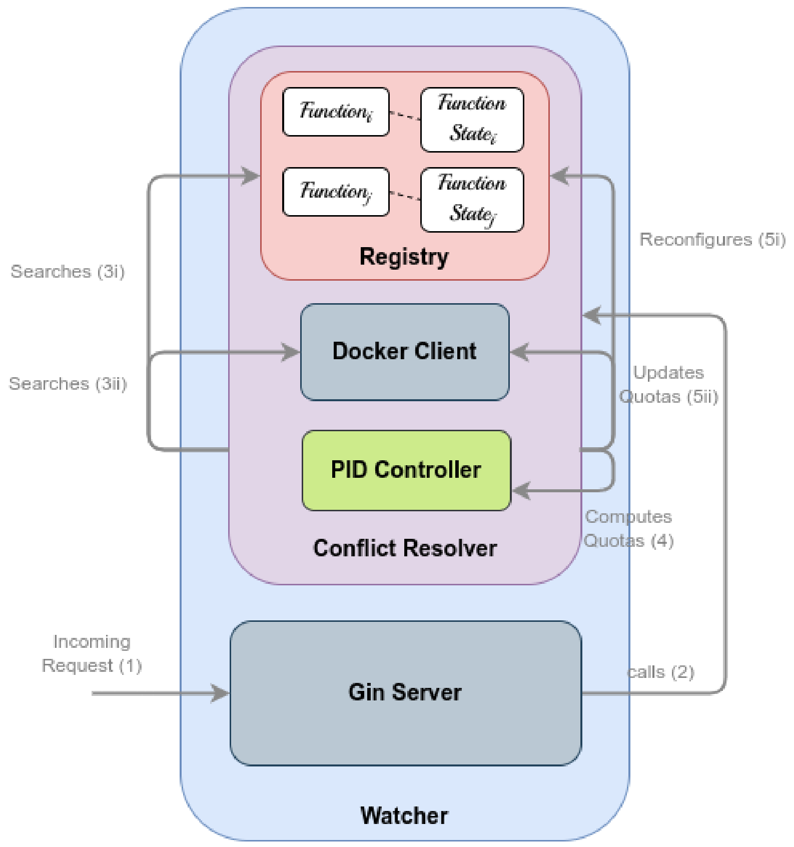


Figure 5.8: Watcher's Workflow

5.2.6 Resource Distribution Algorithm

In the previous section 5.2.5 , it was mentioned that the desired quotas of a function and its related container may be different from the actual quotas, that the *Conflict Resolver* provided. This may occur due to an algorithm that we developed in order to distribute fairly the available CPU quotas of the host machine (on which each *Watcher* lies). This is done by multiplying each value of the desired quotas with a lambda factor, whose formula is placed in equation 5.4. In the following lines, we place the proof for the computation of this factor and we also describe how this operation may be optimized.

Factor λ

For a random node of the system, let T be the default CPU period, N_{cores} be the number of available CPU cores and n be the number of resource requests handled by this node. Let, also, r_i be the desired CPU quotas, that the *Conflict Resolver* tried to provide for each $0 \leq i \leq n-1$ request, and q_i be the actual amount of the same entity. Given this information, there are two possible cases.

- Case I:

$$\sum_{i=0}^{n-1} r_i \leq N_{cores} \cdot T \tag{5.1}$$

In this case, no resource contention occurs and we can safely provide to each container the amount of CPU quotas that the *Conflict Resolver* computed using the *PID controller*, thus $q_i = r_i$ for $0 \leq i \leq n-1$.

- Case II:

$$\sum_{i=0}^{n-1} r_i > N_{cores} \cdot T \tag{5.2}$$

We characterize this situation as *saturation* and according to our approach the amount of the actual CPU quotas q_i, that each function will be provided with by the *Conflict Resolver*, can be computed as:

$$q_i = \lambda r_i \tag{5.3}$$

In order for the saturation to be resolved and the CPU resources to be distributed fairly to each container, we demand the following:

$$\begin{aligned} q_0 + q_1 + ... + q_{n-1} &= N_{cores} \cdot T \Leftrightarrow^{(5.3)} \\ \lambda \cdot (r_0 + r_1 + ... + r_{n-1}) &= N_{cores} \cdot T \Leftrightarrow \\ \lambda &= \frac{N_{cores} \cdot T}{\sum\limits_{i=0}^{n-1} r_i} \end{aligned}$$

Consequently, the general formula for the lambda factor is the following:

$$\lambda = \begin{cases} \frac{N_{cores} \cdot T}{\sum\limits_{i=0}^{n-1} r_i} & , \; saturation \\ 1 & , \; otherwise \end{cases} \tag{5.4}$$

Physical Meaning

The physical meaning of the equations 5.3 and 5.4 is that in case of resource contention/saturation, i.e., in case that the functions' containers require a higher amount of CPU quotas (according to *PID Controller's* output) than the available, each one of them will be provided with a percentage of the requested physical quantity. In addition, in each node the lambda factor lets the ratio between the quotas of two functions untouched, meaning that actions with negative slack will still be provided with higher resources than the ones with positive or less (by absolute value) slack.

A computational example might be the following. If a node with one core count and $T = 100000$ *msec* is responsible for the execution and provision of two positive slack actions and these actions' metrics indicate a CPU quotas value of 50000 (half a CPU each), then no saturation occurs and lambda will be equal to one. Alternatively, in the case that these actions' metrics indicate 110000 and 15000 CPU quotas respectively (the first one has a negative slack, while the second one has a positive slack), then λ will be equal to 0.8, which means that each function container will run with 80% of the required CPU resources.

Furthermore, the equation 5.4 implies that each time a sequence sends a *resourceRequest* to a *Watcher* node, the *Conflict Resolver* needs to iterate through the elements of the *Registry* in order to compute the factor λ again. However, with some basic algebra we can show, that having the previous value of the aforementioned entity with some additional information is more than enough for computing its next value.

Optimization

As far as the computation of the λ factor by the *Conflict Resolver* is concerned, it is worth mentioning three different cases, where apriori knowledge of its previous value can result to an $O(1)$ computation.

Case I: Container Addition

Let n be the number of function containers, inside a random node at the time point t, with r_i, $0 \leq i \leq n-1$ symbolizing the desired quotas of each one. Hypothetically, at the time point $t+1$, the n-th request shows up asking for r'_n quotas. In this case, concerning $\lambda(t)$ and $\lambda(t+1)$, the following is true:

$$\lambda(t+1) = \frac{N_{cores} \cdot T}{\sum\limits_{i=0}^{n-1} r_i + r'_n} \tag{5.5}$$

$$\lambda(t) = \frac{N_{cores} \cdot T}{\sum\limits_{i=0}^{n-1} r_i} \tag{5.6}$$

$$(5.5) \div (5.6) \Rightarrow \frac{\lambda(t+1)}{\lambda(t)} = \frac{\frac{N_{cores} \cdot T}{\sum_{i=0}^{n-1} r_i + r'_n}}{\frac{N_{cores} \cdot T}{\sum_{i=0}^{n-1} r_i}} \Leftrightarrow$$

$$\lambda(t+1) = \frac{\sum\limits_{i=0}^{n-1} r_i}{\sum\limits_{i=0}^{n-1} r_i + r'_n} \cdot \lambda(t) \Leftrightarrow^{(5.6)}$$

$$= \frac{N_{cores} \cdot T}{\frac{N_{cores} \cdot T}{\lambda(t)} + r'_n} \tag{5.7}$$

Case II: Container Removal

Let n be the number of function containers, inside a random node at the time point t, with r_i, $0 \leq i \leq n-1$, symbolizing the desired quotas of each one. Hypothetically, at the time point $t+1$, the function concerning the n-th request terminates and the related resources must be unbound. In this case, the following is true for $\lambda(t+1)$:

$$\lambda(t+1) = \frac{N_{cores} \cdot T}{\sum\limits_{i=0}^{n-2} r_i} = \frac{N_{cores} \cdot T}{\sum\limits_{i=0}^{n-1} r_i - r_n} \tag{5.8}$$

$$\begin{aligned}
(5.8) \div (5.6) \Rightarrow \frac{\lambda(t+1)}{\lambda(t)} &= \frac{\frac{N_{cores} \cdot T}{\sum_{i=0}^{n-1} r_i - r_n}}{\frac{N_{cores} \cdot T}{\sum_{i=0}^{n-1} r_i}} \Leftrightarrow \\
\lambda(t+1) &= \frac{\sum\limits_{i=0}^{n-1} r_i}{\sum\limits_{i=0}^{n-1} r_i + r'_n} \cdot \lambda(t) \Leftrightarrow^{(5.6)} \\
&= \frac{N_{cores} \cdot T}{\frac{N_{cores} \cdot T}{\lambda(t)} - r_n} \qquad (5.9)
\end{aligned}$$

Case III: Container Update

This case is about the situation where the $(n-1)$-th function is still being executed, the *Conflict Resolver* has already alloted $q_{n-1} = \lambda(t) \cdot r_{n-1}$ quotas to the related container and a new request shows up concerning the same function, thus the same container. The process of how the aforementioned component of SC decides the new value of r'_n for this container, will be presented in the next section 5.2.7, but for the steps that follow we will consider this as granted.

Let n be the number of function containers, inside a random node at the time point t, with r_i, $0 \leq i \leq n-1$ symbolizing the desired quotas of each one. At the time point $t+1$, the function concerning the $(n-1)$-th request needs $r'_{n-1} \neq r_{n-1}$ quotas, while all the others still need $r'_i = r_i$, $0 \leq i \leq n-2$ quotas. In this case, the following is true:

$$\lambda(t+1) = \frac{N_{cores} \cdot T}{\sum\limits_{i=0}^{n-1} r'_i} = \frac{N_{cores} \cdot T}{\sum\limits_{i=0}^{n-2} r_i + r'_{n-1}} = \frac{N_{cores} \cdot T}{\sum\limits_{i=0}^{n-1} r_i + r'_{n-1} - r_{n-1}} \tag{5.10}$$

$$(5.10) \div (5.6) \Rightarrow \frac{\lambda(t+1)}{\lambda(t)} = \frac{\frac{N_{cores} \cdot T}{\sum_{i=0}^{n-1} r_i + r'_{n-1} - r_{n-1}}}{\frac{N_{cores} \cdot T}{\sum_{i=0}^{n-1} r_i}} \Leftrightarrow$$

$$\lambda(t+1) = \frac{\sum_{i=0}^{n-1} r_i}{\sum_{i=0}^{n-1} r_i + r'_{n-1} - r_{n-1}} \cdot \lambda(t) \Leftrightarrow^{(5.6)}$$

$$= \frac{N_{cores} \cdot T}{\frac{N_{cores} \cdot T}{\lambda(t)} + r'_{n-1} - r_{n-1}} \tag{5.11}$$

In conclusion, all the equations (5.7), (5.9), and (5.11) can be narrowed down into one single recursive formula:

$$\lambda_{new} = \frac{N_{cores} \cdot T}{\frac{N_{cores} \cdot T}{\lambda_{old}} + r_{new} - r_{old}} = \frac{N_{cores} \cdot T \cdot \lambda_{old}}{N_{cores} \cdot T + (r_{new} - r_{old}) \cdot \lambda_{old}} \tag{5.12}$$

As far as cases (I) and (II) are concerned, the above formula holds true if we set r_{old} and r_{new} equal to zero respectively. Additionally, we should point out that for this recursive equation to give valid results, λ_{old} should always be computed by using the *saturation* term from (5.4) even if this gives values higher than one. However, in the previous described situation, the actual values for q quotas of each container should still be produced with a λ factor of one.

Pseudocode of the part of SC that handles λ factor computation and correction of CPU quotas of each Docker container accordingly is placed below. The sections of the snippet that are important is the usage of the non-recursive equation 5.4 (line 10) and the behaviour in no *saturation* cases (line 25).

In more depth, when no previous value of λ exists, i.e. when the *Registry* is empty, the only way to compute the proper value of λ without having the *Watcher* to panic[4], is by using equation 5.4. Last but not least, a full traversal of the *Registry* for correcting the actual quotas value of each container can be overlooked in only one case and that is if the previous state alongside the new state of the system is not characterized by *saturation*. This happens only when λ_{new} and λ_{old} are greater or equal to one (values higher than one may occur by using the well discussed formula even for no *saturation* situations).

```
1 def lowerBound(quotas):
2     # Docker Daemon doesn't permit
3     # values less than 1000
4     if quotas <= 1000:
5         return 1000
6     else:
```

[4]**Panic** is a built-in function that stops the ordinary flow of control and begins *panicking*. After all functions in the current goroutine have returned, the program crashes. It can be caused by runtime errors or it can be called be directly by a function [63].

```
        return quotas
def ReconfigureRegistry(q_new, q_old):
    if lambda_old == 0:
        # Non recursive computation
        sum = 0
        for s in Registry:
            sum = sum + s.r
        lambda_new = N*T / sum
    else:
        # Recursive computation
        lambda_new = N*T / (N*T/lambda_old + q_new - q_old)

    if not (lambda_new >= 1 and lambda_old >= 1):
        for f, s in Registry:
            if lambda_new < 1:
                # Saturation
                s.q = lowerBound(lambda_new * s.r)
            else:
                # No saturation
                s.q = s.r
            updateContainerQuotas(s.Container, s.q)

    lambda_old = lambda_new
    return
```

5.2.7 Conflicts Policy

During execution of various pipelines and their functions, SC's *Watchers* need to solve an additional problem, which appears when different users try to execute the same pipeline or similar pipelines at the same time. In more depth, when requests reach a node and demand resources for the same function, thus the same container, the node's *Watcher* has to make a decision on what request should be used. The problem occurs as different requests may and probably will demand different amount of resources, i.e., one might have a higher (by absolute value) slack than the other or even worse one might have a positive slack and the other a negative one. In this case, we decided to follow a *"help the one in need"* method, according to which it is wise to regulate the related container using the request with the higher amount of *DesiredQuotas* r.

Mathematically, this can be described as follow. Let f_i be a random function, whose container is being regulated by a node. Let m be the number of requests that have reached this node and are related to the function f_i. Each one of them demands a value of quotas equal to $r_j(f_i)$ with $0 \leq j < m$. Then, the value of quotas, that the *Conflict Resolver* will try to allocate, can be defined as:

$$r_i = r(f_i) = \max_{0 \leq j < m} r_j(f_i) \tag{5.13}$$

If two requests have the same value for r, we opt for the one with the lower value of id, i.e., we prioritize the older one.

The logic behind the formula 5.13 is that when an invocation of a function has negative slack it needs desperately to be provided with more resources. For regulation to exist, invocations that are far behind must be prioritized in contrast with invocations that are on time or ahead of it. The latter will be helped more than it is supposed and needs to, but it is obvious that there is no such a big of deal if a pipeline terminates faster. The opposite is what we try to eliminate.

As far as the implementation of this policy is concerned, it is placed in the below code block and consists of a simple linear search algorithm.

```python
def findNext(state):
    maxid = -1
    maxq = -1
    for id, quotas in state.Requests.Active:
        if quotas > maxq or quotas == maxq and id < maxid:
            maxid = id
            maxq = quotas
    return maxid, maxq
```

The existence of the data structures, called *Active*, inside the *Registry* provide a way of keeping track all the running requests for a certain function. A dictionairy-like structure was used over a max-heap (that would have optimized the searching time of the maximum from the linear $O(n)$ to $O(1)$), because we opted for having $O(1)$ lookup times during *reset* requests. Also, a max-heap would have increased computational time complexity in other operations of the *Watcher*, like the process of adding a new request.

At this point, it important to clarify a hypothesis that is hidden inside the latter approach and actually has already been mentioned in 3.3.5. In order for this policy to have a reason to be implemented, containers of each OpenWhisk action need to allow concurrent invocations. In case that a queue mechanism occurs, this policy is not optimal. In that case we believe that a queue-aware policy might achieve better results. Unfortunately, in order to have a clear answer on this, we believe that further research is needed.

5.2.8 Regulation Mechanism

Having explained extensively the architecture of each component of the system and presented various algorithms or policies that each component uses, it is now feasible to provide a full stack look inside the regulation mechanism. In the following lines, pseudocode blocks of two Go methods of the *Conflict Resolver* are presented, with the first being the *UpdateRegistry()*, while the second being the *RemoveFromRegistry()*.

Update Process

This method is called by the Gin server of the *Watcher* as it handles the requests that each function sends. Basic operations that this method accomplishes are:

- Search of related Docker containers
- Quotas updates
- Calls of *ReconfigureRegistry()* method, whenever this is necessary.

As one can notice, it returns a Boolean variable depending on if the related container was found inside the Docker runtime of the node. Based on this value, the Gin server will then replies with an HTTP Status 200 or HTTP Status 404 to the *Watcher Supreme*.

```
 1 # BASE_PERIOD
 2 T = 100000
 3 
 4 def UpdateRegistry(id, function, metrics):
 5     mutex.Lock()
 6     if not function in Registry:
 7         container = searchDockerRuntime(function)
 8         if no container found:
 9             mutex.Unlock()
10             return false
11         Registry[function] = newState(container)  # Data structure used
12                                                   # as values for the Registry
13 
14     quotas_new = upperLowerThresholds(controllerPID(metrics) + T)
15     state = Registry[function]
16     if quotas_new > state.DesiredQuotas:
17         if state not empty:
18             # Move current request
19             # into Active section
20             state.Active[state.Requests.Current] = state.r
21         quotas_old = state.r
22 
23         updateCurrentState(id,quotas_new)
24 
25         updateDockerContainer(state.Container, quotas_new)
26 
27         ReconfigureRegistry(quotas_new, quotas_old)
28     else:
29         # Just add request into Active section.
30         # Do nothing with the container!!
31         state.Requests.Active[id] = quotas_new
32 
33     mutex.Unlock()
34     return true
35 
36 def upperLowerThreshold(quotas):
37     if quotas > N * T:
38         return N*T
39     elif quotas < 5000:
40         return 5000
41     else:
42         return quotas
```

As it should be clear by now, mutex mechanisms are a necessity also here, protecting the *Registry* by dirty reads, simultaneous writes, and inconsistencies in general. Moreover, a key point of this procedure is that quotas correction of each container is happening with the desired new value (line 23) before the call of the method *ReconfigureRegistry()* (line 27), which might correct it alongside all the other containers by the factor λ. This is crucial, as regulation for negative slack invocations must be done as fast as possible.

Reset Process

The reversed operation is the *RemoveFromRegistry()* method, which handles the *reset* request that follows an action's termination. Basic operations of this method include:

- Removal of the related request from the registry.
- Quotas change of the now unwanted Docker container of the function that terminated.
- Registry Update in case of pending requests for the related function exist.

After a container is no longer needed, *Conflict Resolver* should reset the amount of CPU quotas (line 6). At this point, there are two options. The first is to reset it back to the default value that OpenWhisk uses and that is -1 (which means that there is no limit in the amount of CPU resources that the container will try to acquire). The second one is to force the minimum possible value, which is 1000 as we mentioned earlier. Between those two, after several trials we found that the default value is better. A possible explanation to this is that the minimum value of quotas slows down function execution dramatically during the time window between function invocation and actual regulation from each *Watcher*.

```
1  def RemoveFromRegistry(id, function):
2      mutex.Lock()
3      state = Registry[function]
4      if state.Requests.Current == id:
5          if state.Requests.Active is empty:
6              updateDockerContainer(state.Container,-1)
7              Registry.delete(function)
8              if Registry not empty:
9                  ReconfigureRegistry(0, state.r)
10             else:
11                 lambda_previous = 0
12         else:
13             quotas_old = state.r
14             state.Requests.Current, state.r = findNext(state)
15             state.q = state.r
16             quotaas_new = state.r
17             state.Requests.Active.delete(state.Requests.Current)
18             updateDockerContainer(state.Container,quotas_new)
19             ReconfigureRegistry(quotas_new,quotas_old)
20     else:
21         state.Requests.Active.delete(id)
22     mutex.Unlock()
23     return true
```

The more important parts are the following:

1. In line 14, the Go function *findNext()* is called, making usage of the *Conflicts Policy.*

2. In case of an empty *Registry* there's no reason of calling *ReconfigureRegistry()* for computing λ factor. During next request, the non recursive formula 5.4 should and will be used.

3. In case of remaining requests for the selected function (else statement in line 12), correction of quotas with the desired value (line 18) happens before *ReconfigureRegistry()* for the same reason that described in method *UpdateRegistry()*.

The entire code for Sequence Clock will become available at *https://github.com/john98nf/SequenceClock* after the publication of this Thesis.

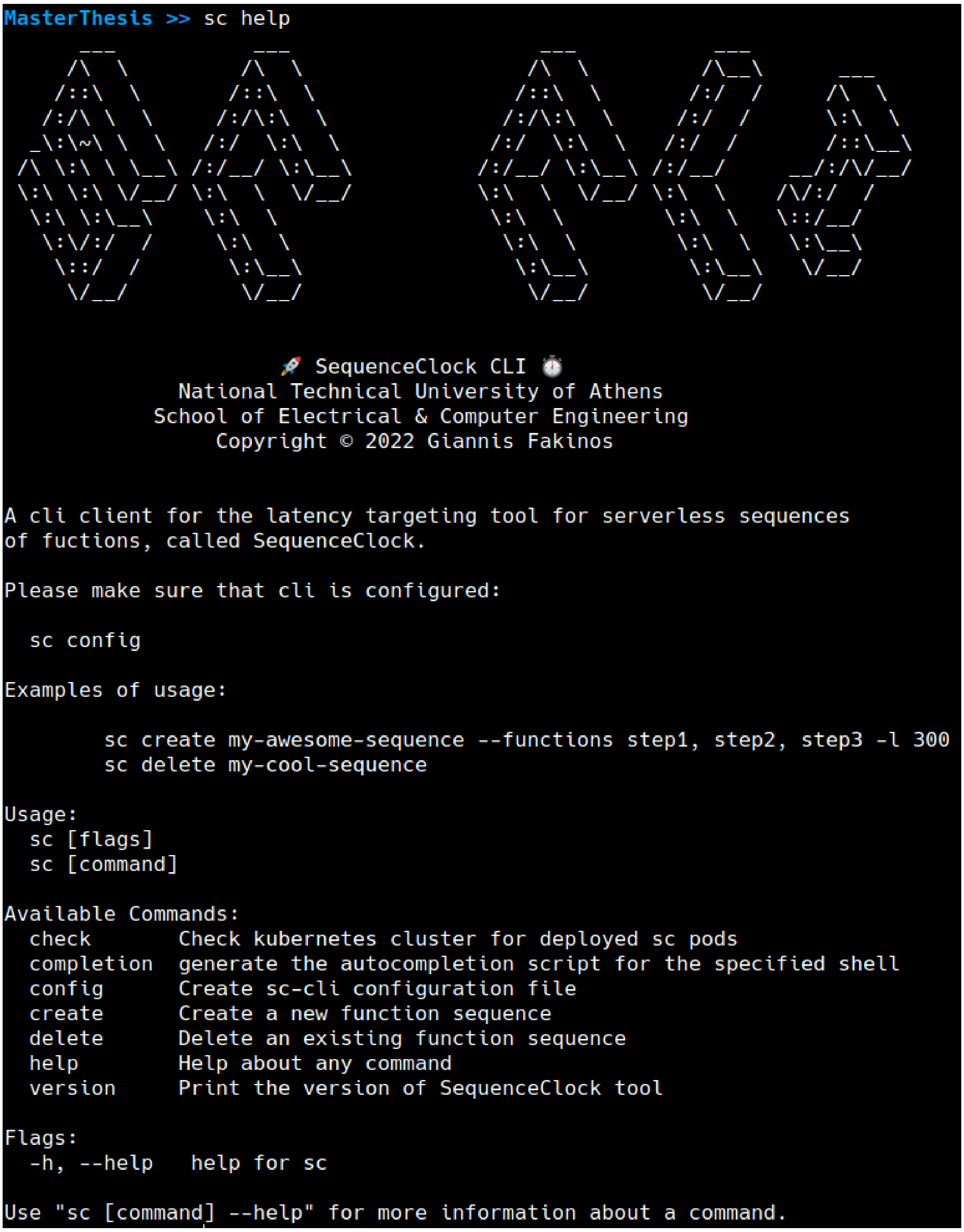
5.3 Sequence Clock CLI

5.3.1 Overview

Similar to OpenWhisk and wsk cli tool [64], Sequence Clock has also a command line tool for offering the developers an easy way of managing pipelines. The sequence-clock-cli is a command line tool developed in Go and it offers four simple tasks:

- Connectivity checking with Sequence Clock API
- Configuration for usage between multiple clusters
- Creation of SC pipelines.
- Deletion of SC pipelines.

The help message of this tool can be seen below.



```
MasterThesis >> sc help
      ___           ___                    ___           ___
     /\  \         /\  \                  /\  \         /\__\      ___
    /::\  \       /::\  \                /::\  \       /:/  /     /\  \
   /:/\ \  \     /:/\:\  \              /:/\:\  \     /:/  /      \:\  \
  _\:\~\ \  \   /:/  \:\  \            /:/  \:\  \   /:/  /       /::\__\
 /\ \:\ \ \__\ /:/__/ \:\__\          /:/__/ \:\__\ /:/__/     __/:/\/__/
 \:\ \:\ \/__/ \:\  \  \/__/          \:\  \  \/__/ \:\  \    /\/:/  /
  \:\ \:\__\    \:\  \                 \:\  \        \:\  \   \::/__/
   \:\/:/  /     \:\  \                 \:\  \        \:\  \   \:\__\
    \::/  /       \:\__\                 \:\__\        \:\__\   \/__/
     \/__/         \/__/                  \/__/         \/__/


                      🚀 SequenceClock CLI ⏱️
              National Technical University of Athens
            School of Electrical & Computer Engineering
                 Copyright © 2022 Giannis Fakinos


A cli client for the latency targeting tool for serverless sequences
of fuctions, called SequenceClock.

Please make sure that cli is configured:

  sc config

Examples of usage:

        sc create my-awesome-sequence --functions step1, step2, step3 -l 300
        sc delete my-cool-sequence

Usage:
  sc [flags]
  sc [command]

Available Commands:
  check       Check kubernetes cluster for deployed sc pods
  completion  generate the autocompletion script for the specified shell
  config      Create sc-cli configuration file
  create      Create a new function sequence
  delete      Delete an existing function sequence
  help        Help about any command
  version     Print the version of SequenceClock tool

Flags:
  -h, --help   help for sc

Use "sc [command] --help" for more information about a command.
```

Figure 5.9: Sequence Clock CLI Help message

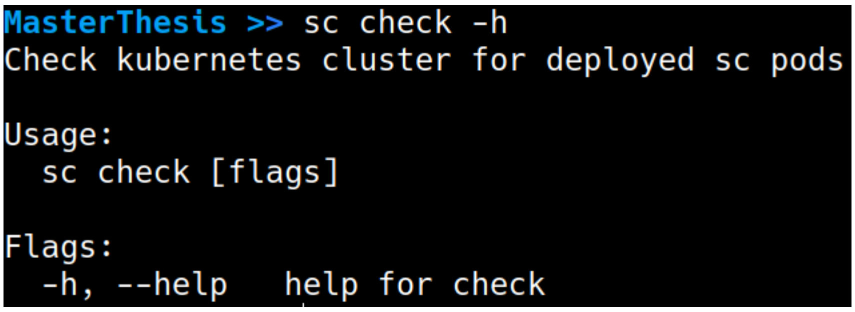
5.3.2 Architecture

This tool is developed using the well known libraries Viper [65] for managing configuration files & Cobra [66] for implementing the entire project. It consists of five commands and an HTTP client, designed specifically for the *Deployer*

API of Sequence Clock.

Check command

This command is used just for troubleshooting purposes as it hits the endpoint */api/check* of the *Deployer* and prints a related message according to the server's response.

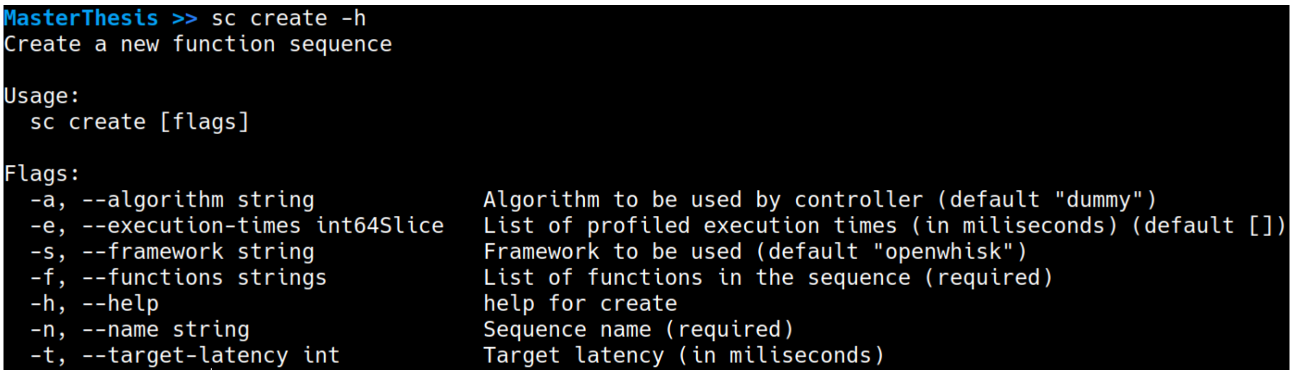


```
MasterThesis >> sc check -h
Check kubernetes cluster for deployed sc pods

Usage:
  sc check [flags]

Flags:
  -h, --help   help for check
```

Figure 5.10: Check command usage

Create command

In order for a developer to deploy a sequence, that leverages Sequence Clock, into OpenWhisk usage of the *Deployer's* API is needed. Instead of manually making the call to the related endpoint */api/create*, one can simple use the *sc* command, specify the internal structure of the pipeline, and lastly choose a target latency. It even provides the option of setting time targets per function of the pipeline, instead of the total time latency (which actually will be divided equally to each sub-action). Moreover, with an additional flag called *algorithm* a developer may set the algorithm that the *Sequence Controller* will use, i.e., *greedy* or *dummy*.

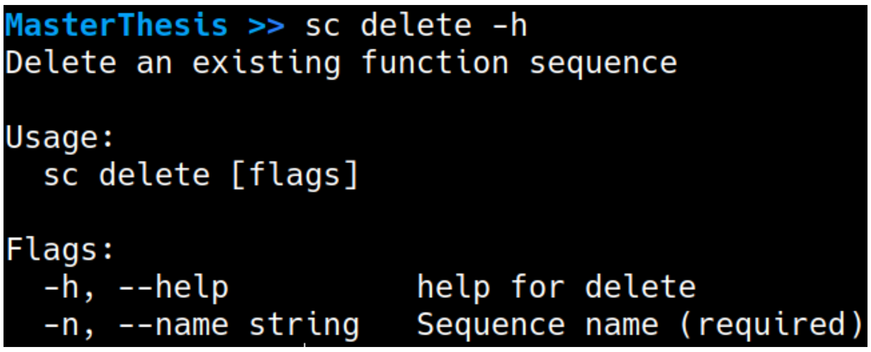


```
MasterThesis >> sc create -h
Create a new function sequence

Usage:
  sc create [flags]

Flags:
  -a, --algorithm string            Algorithm to be used by controller (default "dummy")
  -e, --execution-times int64Slice  List of profiled execution times (in miliseconds) (default [])
  -s, --framework string            Framework to be used (default "openwhisk")
  -f, --functions strings           List of functions in the sequence (required)
  -h, --help                        help for create
  -n, --name string                 Sequence name (required)
  -t, --target-latency int          Target latency (in miliseconds)
```

Figure 5.11: Create command usage

Delete command

As it is obvious, this command is used for deleting created sequences. The related help message with its usage is placed in Figure 5.12.

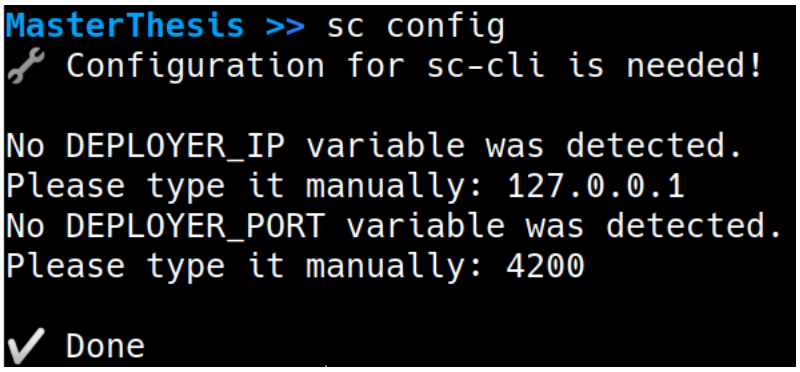


```
MasterThesis >> sc delete -h
Delete an existing function sequence

Usage:
  sc delete [flags]

Flags:
  -h, --help          help for delete
  -n, --name string   Sequence name (required)
```

Figure 5.12: Delete command usage

Config command

As it was mentioned earlier, in order for all the operations to be executed *sc* must have access to the *Deployer's* API. The latter is exposed by using a Kubernetes service of type *NodePort*, which means that one only needs to know the address of the master node of the cluster and the port that the service was exposed to. After the installation of the related Helm Chart the necessary commands for finding this information is shown on the screen, while the *config* command of *sc* provides a way of saving this information into the config file *~/.sc-cli/config.yaml* with ease.



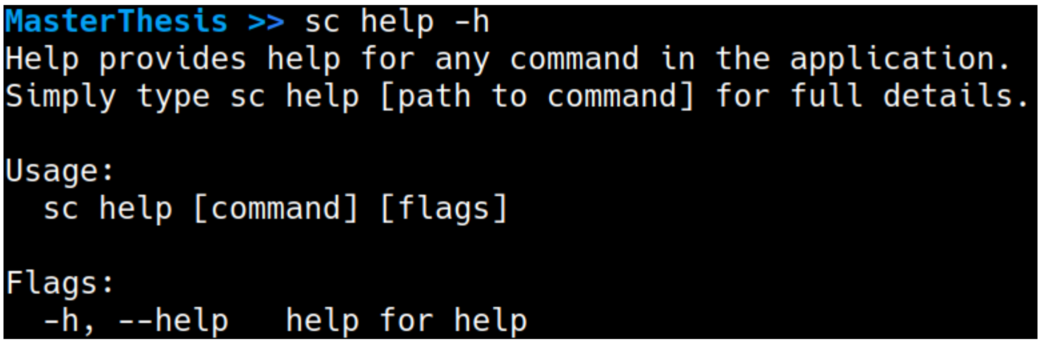
```
MasterThesis >> sc config
  Configuration for sc-cli is needed!

No DEPLOYER_IP variable was detected.
Please type it manually: 127.0.0.1
No DEPLOYER_PORT variable was detected.
Please type it manually: 4200

✓ Done
```

Figure 5.13: Config command execution example

By executing this command, *sc* will prompt the user for two things, i.e., the address and the port number or it will fetch this information from the *DEPLOYER_IP*, *DEPLOYER_PORT* environmental variables. After this step, *sc* is ready to be used.

Help command

This command offers a help message for the entire cli, as it was shown in Figure 5.9, or the usage message of a command, provided as command line argument. Its structure can been seen bellow.

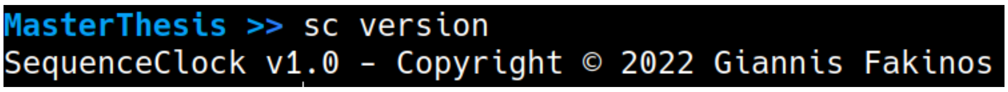


```
MasterThesis >> sc help -h
Help provides help for any command in the application.
Simply type sc help [path to command] for full details.

Usage:
  sc help [command] [flags]

Flags:
  -h, --help   help for help
```

Figure 5.14: Help command usage

Version command

This command offers a regular functionality of command line arguments tools, which is printing the current version of the software tool. By the time of writing this Thesis, the current version is v1.0. Sample output is the following.



```
MasterThesis >> sc version
SequenceClock v1.0 - Copyright © 2022 Giannis Fakinos
```

Figure 5.15: Version command output

The entire code for Sequence Clock CLI will become available at *github.com/john98nf/sequence-clock-cli* after the publication of this Thesis.

Chapter 6

Evaluation

In this chapter, we evaluate the results of our proposed approach in various scenarios. In more depth, a detailed architecture of the experiment that we performed for various pipelines is placed in Section 6.1, while the analysis of OpenWhisk's behaviour in this follows in Section 6.2. The Chapter ends with the presentation of Sequence Clock's performance in the same scenarios that OpenWhisk's pipelines were tested, both for the unsupervised (*Dummy*) and the *Greedy* version of SC.

6.1 Experimental setup

6.1.1 Abstract Analysis

The ultimate goal was to explore sequence's performance in various levels of system pressure, various "stressing" scenarios. A way to quantify this, is to tinker with requests per seconds concerning the invocations, i.e., to perform sequential invocations of a pipeline with a constant rate *rps* or equivalently with a constant time difference $\Delta t = \frac{1}{rps}$.

At this point, it is crucial for the integrity of the measurements to have a complete control over the way each function invocation is being executed, i.e., by which container each request is being handled. If one allows reusage of a container for the random function f_i, then there is a high possibility during the larger values of rps the queueing phenomenon to appear, as older invocations might not have finished before new ones arrive. On the other hand, if one forces *concurrency* of functions to be equal to one, then each invocation will spawn a new container, if an available one does not exist, and cold start penalty will be added in execution time.

For this exact reason, each function was deployed inside OpenWhisk with an additional ID or alias in order for each invocation to be distinguishable from other invocations (inside the experiment that we mentioned earlier) of the same function for the same pipeline. Mathematically, this can be interpreted as follows. Let a sequence $s = (f_0, f_1, ..., f_{n-1})$ of n functions be the one that will be tested. This pipeline will be deployed in OpenWhisk as $s_0 = (f_{00}, f_{10}, ..., f_{(n-1)0})$, ..., $s_0 = (f_{0(m-1)}, f_{1(m-1)}, ..., f_{(n-1)(m-1)})$ m times, where m is the number of invocations that will happen during the experiment. Thus the notation f_{ij} is about the i-th function that deployed inside openwhisk with the j-th alias (for example the action *uploader* will be deployed as $uploader_0$,... , $uploader_{(m-1)}$ m times). An abstract representation of the experiment and the notation is presented in Figure 6.1.

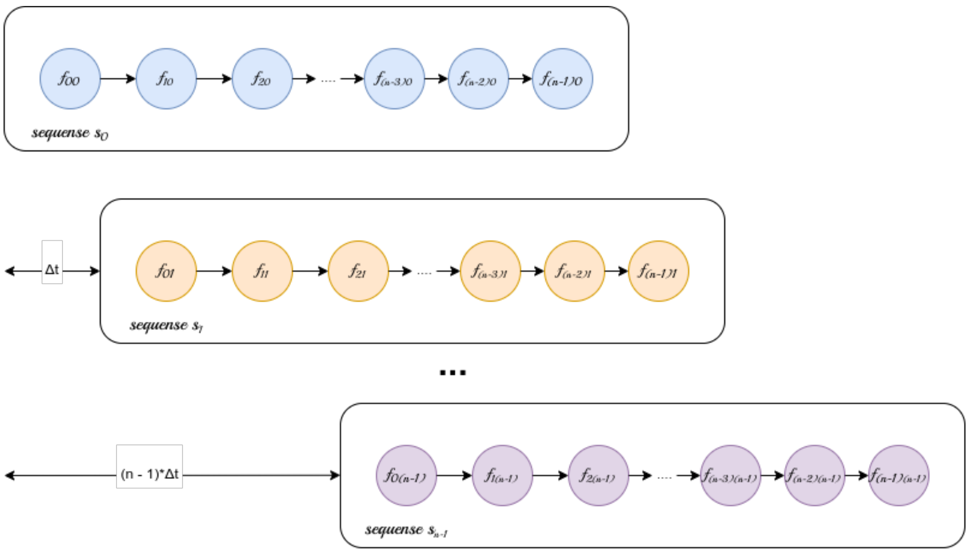


Figure 6.1: Abstract Representation of a pipeline experiment.

This approach guarantees that every invocation request will be placed on a separate container, eliminating wait times in Python actions and ensuring that Sequence Clock *Watchers* will keep track of the incoming requests. As far as cold starts times to be avoided, one can simple perform m dummy requests before each execution and *rps* level for the needed containers to be put in place inside the cluster.

6.1.2 Technical Configurations

As far as experiments' parameters are concerned, the default period for each *rps* level is set to 120 seconds, while Δt values are set to [10, 6.67, 5, 4, 3, 2.5, 2.22, 2] which makes *rps* value to fluctuate between 0.1 and 0.5 with an approximate step of 0.15.

According to this, the highest value of m (for $rps = 0.5$) is equal to $\frac{120}{2} = 60$. Unfortunately, 60 different implementations of a 3-function pipeline require $60 * 3 = 180$ containers and thus $180 * 256 = 46080$ MiB $= 45$ GiB of total system memory inside the cluster. Even worse, if this pipeline is deployed with SC $60 * 256 = 15360$ MiB $= 15$ GiB of additional memory are required due to *Sequence Controllers'* containers. On the contrary, we mentioned that 34.8 GiB

(35635, 2 MiB) were provided as user memory from OpenWhisk yaml file (see Section 4.1). Thus, $\lfloor \frac{35635,2}{256} \rfloor = \lfloor 139.2 \rfloor = 139$ containers can fit into memory of the cluster and $\lfloor \frac{139}{(3+1)} \rfloor = 34$ different aliases of SC 3-function pipelines can be deployed at the same time inside the cluster. For the additional invocations for *rps* levels higher than or equal to 0.33 ($\Delta t = 3 \Rightarrow m = \frac{120}{3} = 40$) a modulo operation is used, with no performance overhead[1]. For 6-function sequences, as it is desirable to retain the threshold of 34 aliases per pipeline, for each alias s_i the following is true $s_i = (f_{0(i \mod 17)}, f_{1(i \mod 17)}, \ldots, f_{6(i \mod 17)})$.

The entire smoke test is implemented using the Bash scripting language and an oversimplified version of it is presented with pseudocode above.

```
# Variables $pipeline & $framework (openwhisk, dummy, sc)
# are extracted by command line arguments
# Code not shown

# Actual Experiment
texp=120
max_implementations=34
duration=(10 6.67 5 4 3 2.5 2.22 2)
for dt in ${duration[@]}
do
    # ---- Actual Invocations ----
    echo "* Expirement for Dt = ${dt} sec"
    n_total=$(python -c "print(int(${texp}/${dt}))")
    cold_start
    echo " - Starting actual experiment"
    for ((n=0;n<$n_total;n++))
    do
        n_mod=$(mod ${n} ${max_implementations})
        invoke ${pipeline}i${n_mod}${framework} --async >> owdev_id_${pipeline}_${n}
        sleep $dt
    done
    echo " - Sleep for 2 mins"
    sleep 2m
    # ----------------------------
    # Measurements
    echo " - Fetching measurements"
    fetch_measurements
    echo "-------------------------------"
done

sleep 1m
echo "* Downloading logs from each node"
download_logs
echo "------ End of experiment ------"
```

The function *invoke* is a wrapper for the *wsk -i action invoke* command of the OpenWhisk CLI. As one can notice, the invocation happens with the non blocking option in order for the script to continue its run.

[1]The $(i + 34)$-th invocation uses the same containers with the i-th one but starts execution of the j-th function long after its termination from the previous invocation. Even if two requests overlap, the maximum number of requests that a function invocation will wait for will be one due to the total amount of invocations happening in 120 seconds. In addition, as we discussed in Section 4.2.3 the effect of latency elongation will be linear and not exponential.

```bash
function invoke(){
    local f=$1
    local option=$2
    case $option in
        --sync)
            wsk -i action invoke $f --result > /dev/null
            ;;
        --async)
            wsk -i action invoke $f
            ;;
        *)
            echo -n "Unkown option"
            exit -1
            ;;
    esac
}
```

The code for smoke tests' scripts of Sequence Clock will become available at *github.com/john98nf/sequence-clock-smoke-tests* after the publication of this Thesis.

6.2 OpenWhisk's Analysis

In this section, we present the figures that will be used for evaluating OpenWhisk's performance along side the behaviour of Sequence Clock and its various algorithms, i.e., dummy & greedy.

6.2.1 99-th Percentile, Mean & Time Latency Extraction

We draw the mean value and the 99-th percentile of time latency as functions of *rps* values. Graphical representation of the 99-th percentile of time latency was crucial for the extraction of a certain target time latency for each pipeline. It is worth mentioning that it would be a mistake to perform an invocation of each sequence and utilize the (almost random) execution time of each run as the reference point. The reason is that it would be impossible to achieve that kind of QoS, as it is unlikely for such sequences to run under those circumstances, i.e., isolated and without *"competition"*. Thus, a similar method with the one proposed in [67] is performed. In more depth, we set the target latency for each pipeline as the 99th percentile latency of the knee of the related plot, where the application experiences a rapid increase in tail latency after exceeding a load threshold (typically between $0.2 \sim 0.4$ *rps*).

For automating knee location for each plot we utilize the *knee* Python module, which is based on the findings of [68]. Whenever this algorithm is not capable to locate such point, e.g., when the latency experiences linear increase instead of exponential like, we perform a linear search to find at which point the growth between each value and the next one is higher than a certain percentage (normally 0.35%) relative to the original value. For instance, in Figure 6.2 on can observe the knee of the sequence *stg6* to appear at 0.33 of *rps* resulting to a target latency of 26956 mseconds.

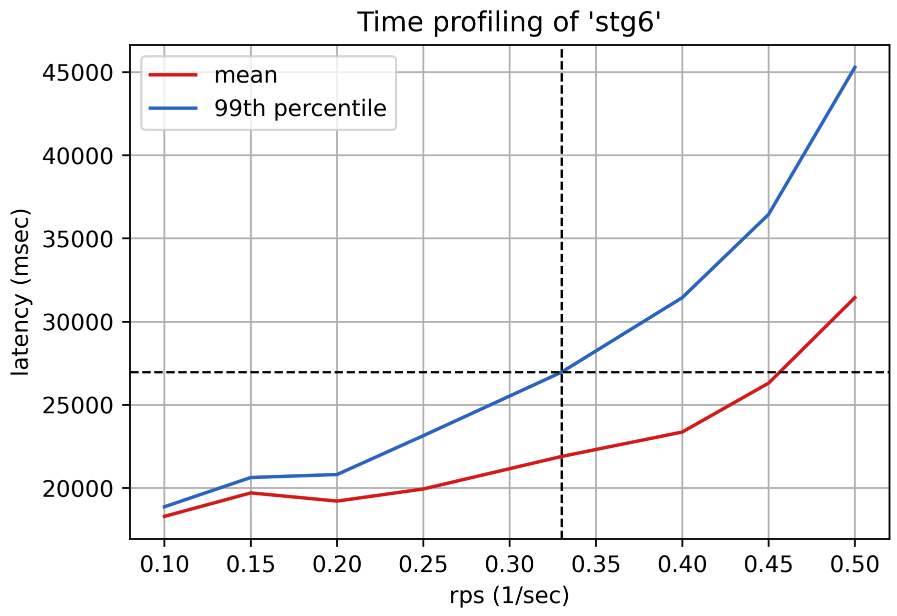


Figure 6.2: OpenWhisk: Target Latency Extraction from the 99th-percentile.

6.2.2 Time Latency's Distribution per Sequence & Function

In order to observe the potential increase of the variance in time latency we use box/violin plots for various *rps* values. A box plot assumes a single distribution resulting in outliers' exclusion, in contrast with a violin plot which assumes a mixture of Gaussian distributions with unknown parameters. This analysis is performed in both the sequence (seen in Figure 6.3) and the function level (seen in 6.4). The latter also serve as a way of discovering the function or functions that consume the most of the total time latency and suffer from higher values of variance.

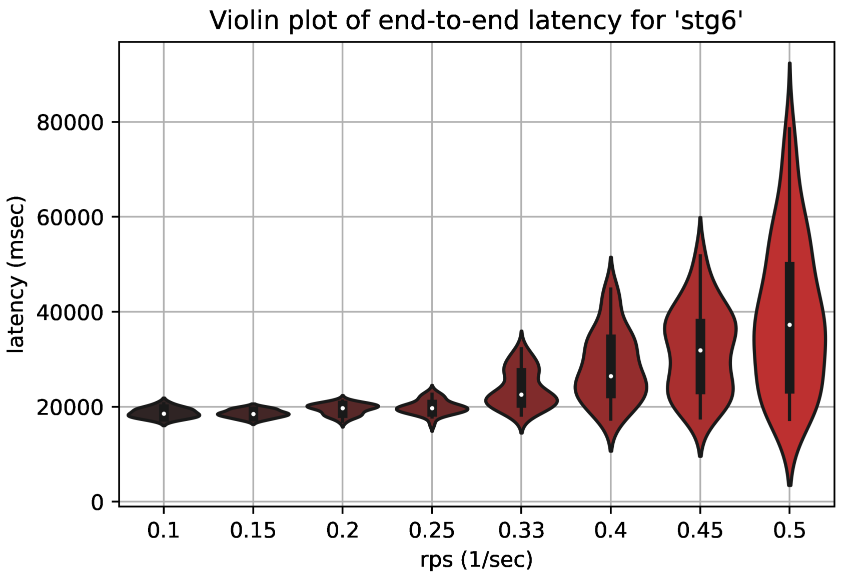


Figure 6.3: OpenWhisk: Distribution of Time Latency for *stg6* for various *rps* values

For example, from the next Figure we can safely conclude that in *stg6* the functions that consume the highest of the total end-to-end latency are *stg0generic* and *stg2generic*.

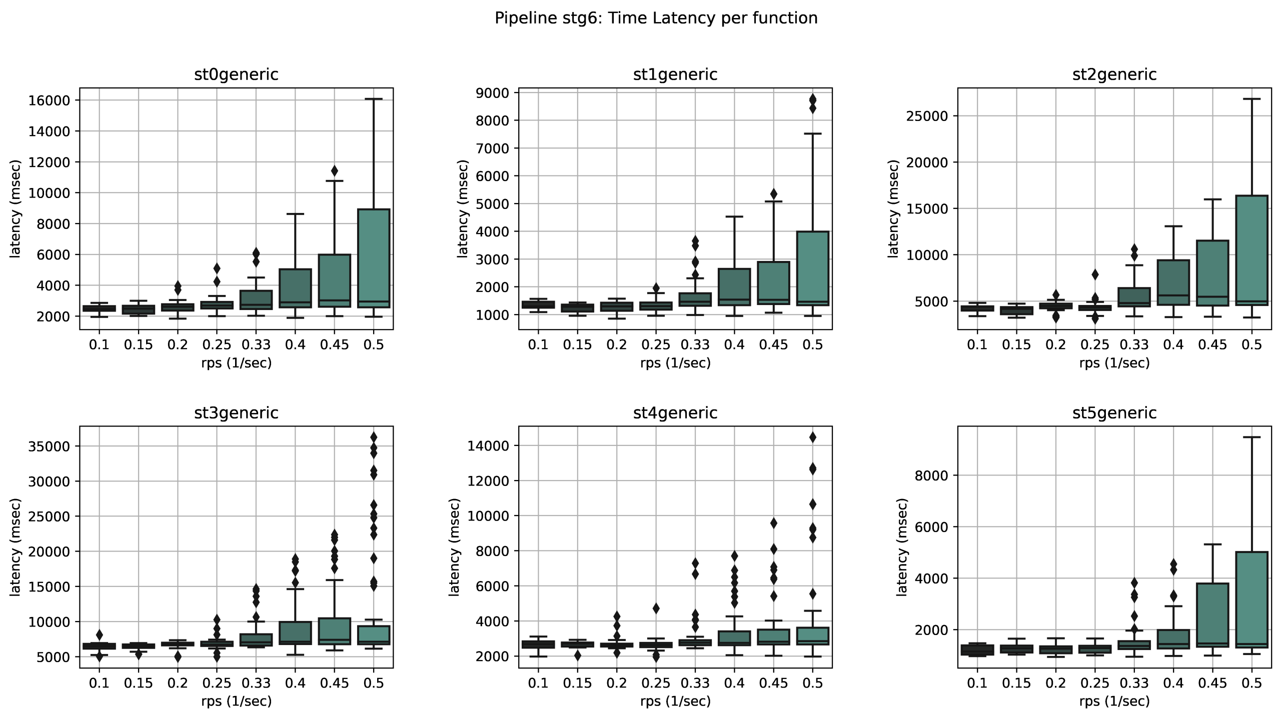


Figure 6.4: OpenWhisk: Distribution of Time Latency for each function of *stg6* for various *rps* values

One key point for both the sequence and the function level is that for higher pressure circumstances, i.e., higher values of *rps*, not only time latency increases but it also becomes more unstable and unpredictable.

6.2.3 Time Latency across Time

An additional way to explore and monitor sequence behaviour is to plot the latter as a function of time for the specified interval of 120 seconds. If everything runs as expected, a bell like curve must be observed, just like the one in Figure 6.5, where higher values of *rps* force higher values and curvature for this bell. The physical interpretation of this lies on the previously placed Diagram 6.1, where one can understand that invocations placed in the middle suffer from higher degree of resource contention, resulting in higher values of execution time.

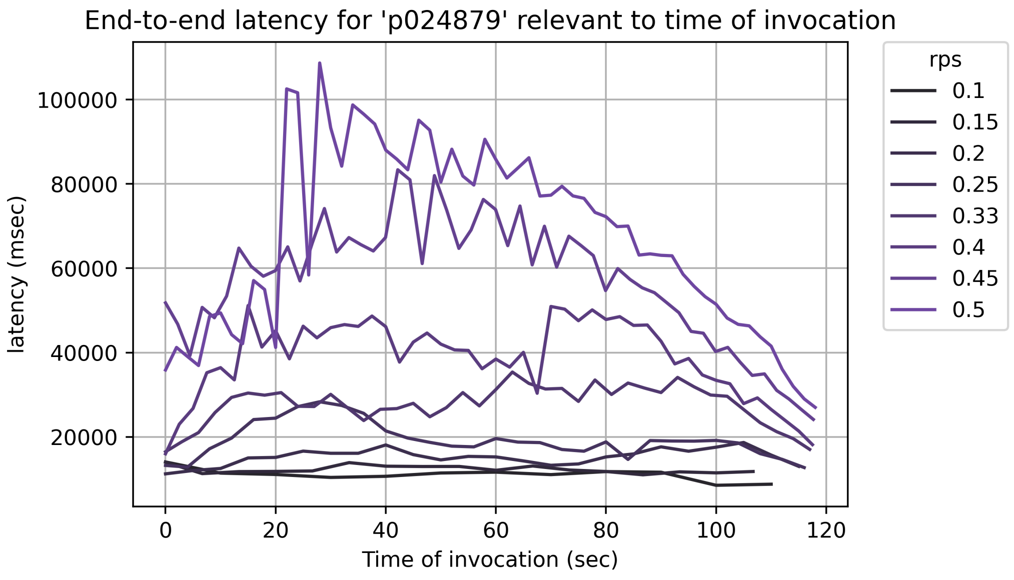


Figure 6.5: OpenWhisk: Time Latency of $p024879$ as a function of time.

6.2.4 Violations' Analysis

As we have previously noted, there are two factors that can unveil the violations' phenomenon, which are the percentage of the violations and the violation factor (see 5.1). If violations' percentage is present regardless its value, it doesn't reveal a lot by itself, as the violations might not be severe relative to the actual time latency. That is why, besides a bar plot of the previous physical entity, like the one in Figure 6.6, a way to visualize how much an invocation violated the target, is needed. This is done by computing the violation factor and by plotting its histogram[2] (just like the one in Figure 6.7a) or its probability distribution function (like Figure 6.7b).

[2]In the histograms that we opted for the height of each bin shows the number of observations inside the related interval.

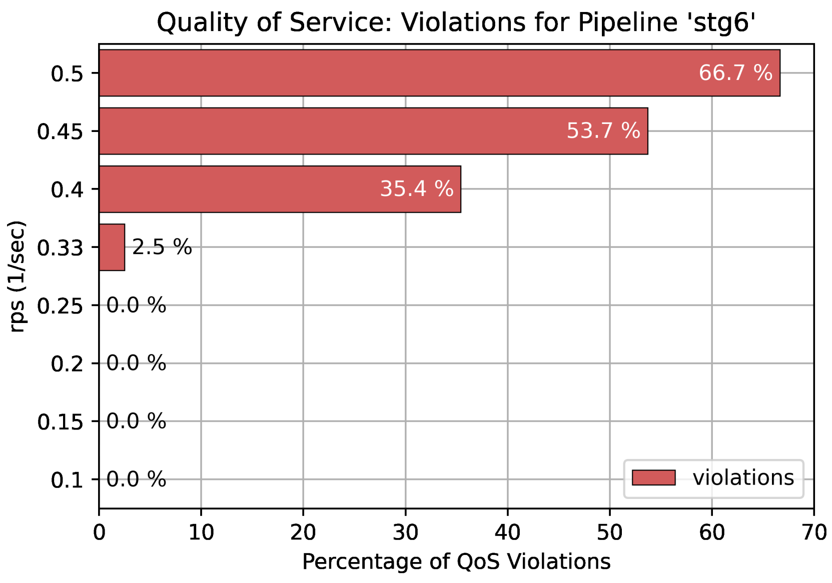
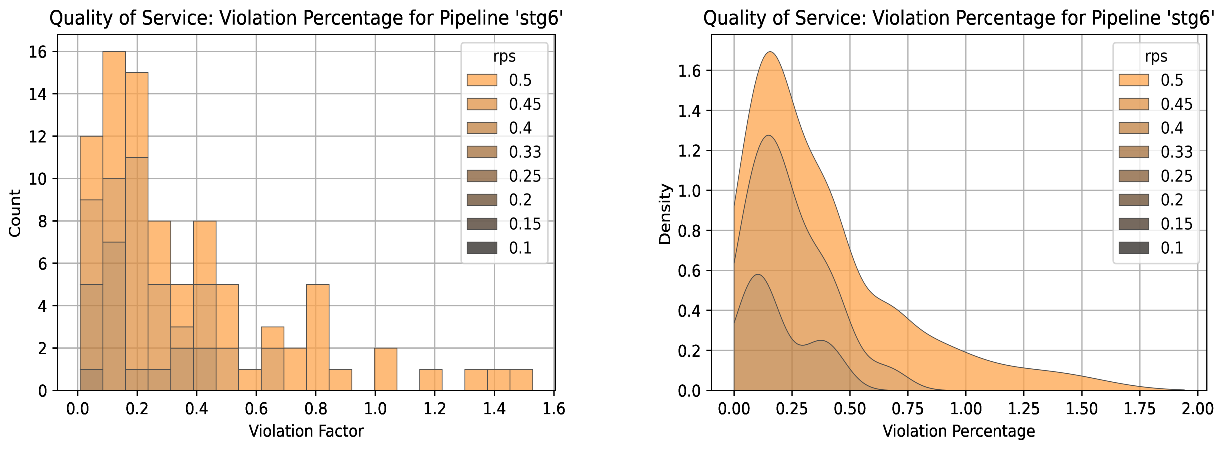


Figure 6.6: OpenWhisk: Violations' Percentage of the sequence *stg6* for various *rps* values.



(a) Histogram (b) Kernel Distribution Estimate (KDE) Plot

Figure 6.7: OpenWhisk: Distribution of Vioaltion Factor for *stg6*.

6.2.5 Resource Utilization Monitoring

In search of true understanding of the cluster state, metrics like the CPU Load Average, the CPU percentage for un-niced processes and the percentage of system memory usage provide a basic understanding of what is going on inside each worker node (invokers). All of them are functions of time and are plotted as such (see Figures 6.8, 6.9, and 6.10). Furthermore, it is worth mentioning that as the CPU percentage usage tends to reach its highest value and descent immediately back to lower values, a better approach is to plot the rolling average of the same quantity. An example is shown in Figure 6.10b, which looks more comprehensive than Figure 6.10a.

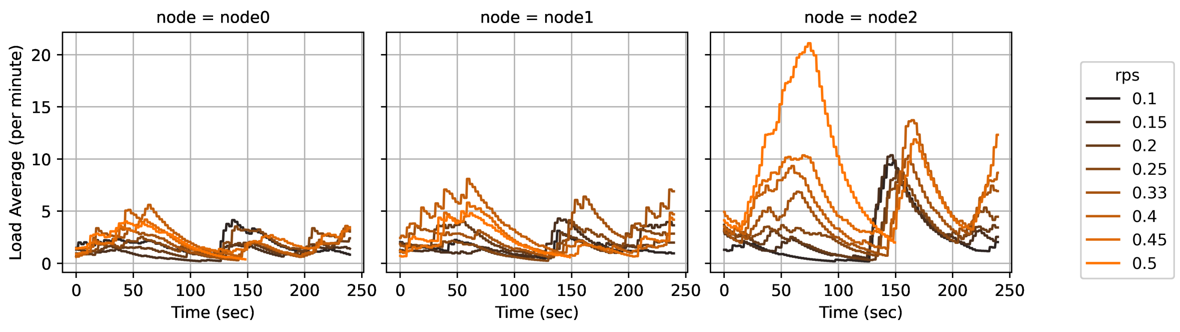


Figure 6.8: OpenWhisk: Per Node Load Average during experimentation for sequence *stg6*.

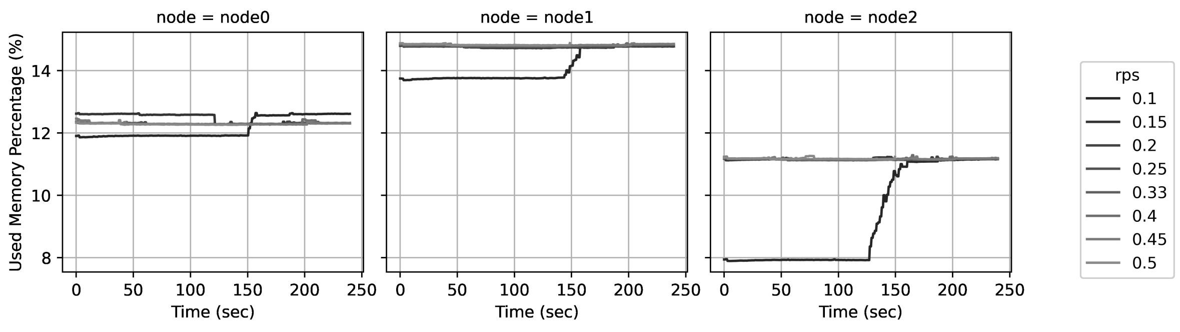
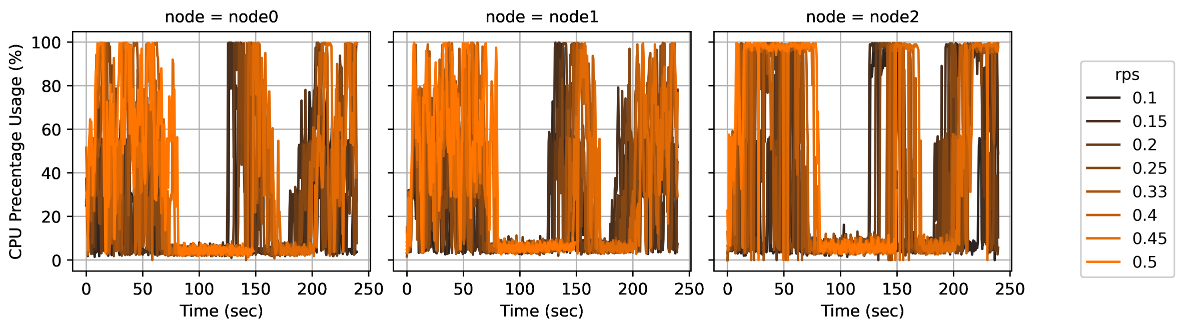
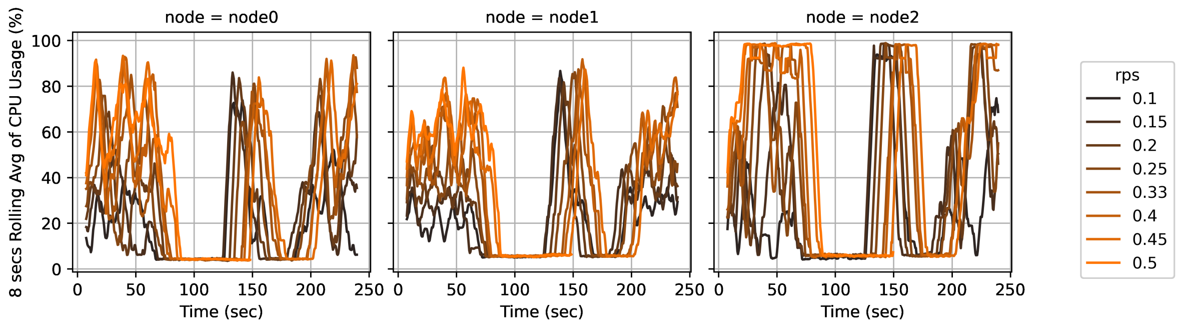


Figure 6.9: OpenWhisk: Per Node System Memory Utilization during experimentation for sequence *stg6*.



(a) OpenWhisk: Per Node Percentage of CPU.



(b) OpenWhisk: 8 seconds Roling Average of per Node CPU Percentage.

Figure 6.10: OpenWhisk: Per Node CPU utilization during *stg6* experimentation

Lastly, we note that metrics' gathering exceed the interval of 120 seconds and reach 240 seconds. As it would be impossible to know the exact moment, when the rear invocations of each experiment will finish, we decided to run the top script for a period of 240 seconds, which includes the querying process of *CouchDB*, after the actual invocations' termination. The above figures (especially 6.8 and 6.10b) show that weaker systems (like *node2*) suffer from higher values of CPU utilization, while the percentage of used system's memory remains constant at least for the time interval of 120 seconds.

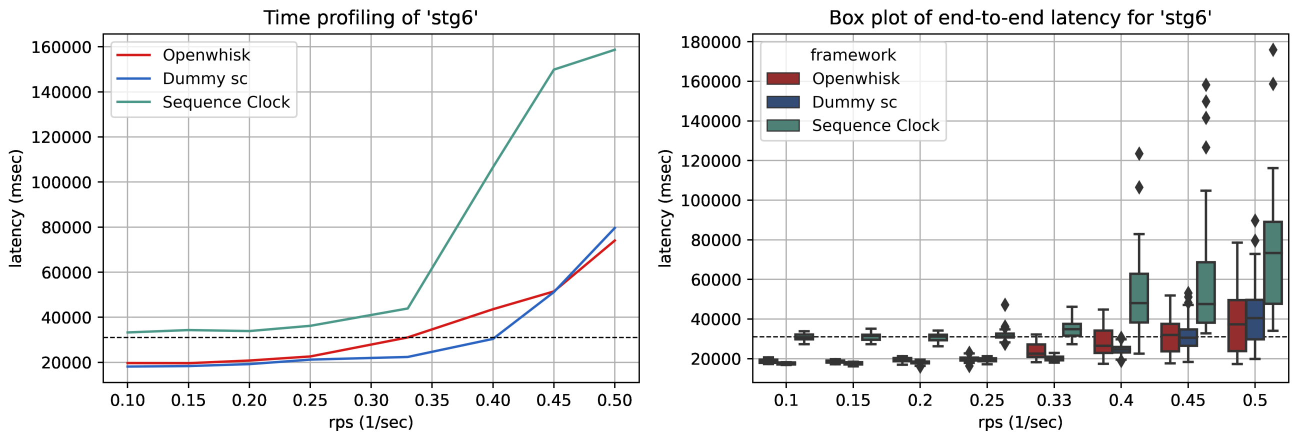
6.3 Comparing Dummy SC & SC with OpenWhisk

In the following lines, an in depth study for the behaviour of OpenWhisk in comparison with Sequence Clock and its two algorithms. The Dummy algorithm was added, in order to reinsure us that sequences do not suffer from a network delay offset due to *sequence controllers'* mechanism. For our analysis, we perform a plethora of tests by utilizing the pipelines *stg3, stg6, mtg3, mtg6, mpg3, mpg6* from the generic suite and *p021, p643, p024879, p051463* created using the SeB Suite.

6.3.1 Generic Sequences's Analysis

Single-Threaded Pipelines

Two single threaded pipelines are used, which are *stg3* with a target latency of 8917 mseconds and *stg6* with a target latency of 31064 mseconds. Their difference is the length of the sequence, which plays a crucial role on time latency behaviour.



(a) 99-th Percentile for *stg6*. (b) Time Latency Distribution for *stg6*.

Figure 6.11: Comparing OpenWhisk, Dummy SC, and SC with *stg6* pipeline.

From Figure 6.11a, one can conclude the following:

- As Dummy algorithm's performance indicates, the mechanism of *sequence controllers* does not result to any additional network delay.

- The Sequence Clock Greedy algorithm and the PID controller result in a 99-th percentile of time latency to be extremely near target latency for *rps* values that are less than 0.33. Unfortunately, for higher values, SC not only fails to regulate the specified quantity but ends up to be worse than the default configuration of OpenWhisk.

The above arguments are confirmed also by Figure 6.11b, where Sequence Clock seems to be more stable than OpenWhisk for the values that achieve a time latency with a distribution near the target.

Having said this, one might rush to say that the analysis is over and regulation happens for some *rps* levels. Unfortunately, by exploring the additional plots and observations, a different opinion appears. From Figure 6.12, Sequence Clock appears to have the highest violations' percentage for all *rps* levels. For the values that are beneath 0.33, as the boxplot in 6.11b and Figure 6.13 imply, these violations are not severe. However, OpenWhisk originally never had violations in these levels of system's pressure.

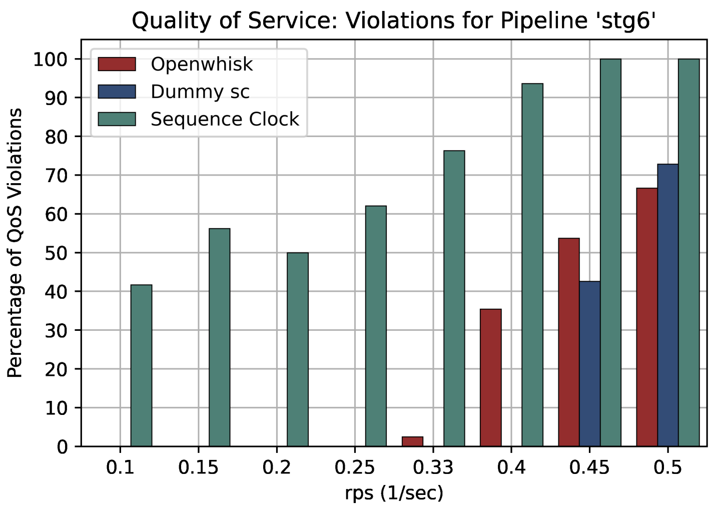


Figure 6.12: Violations' Percentage for *stg6*.

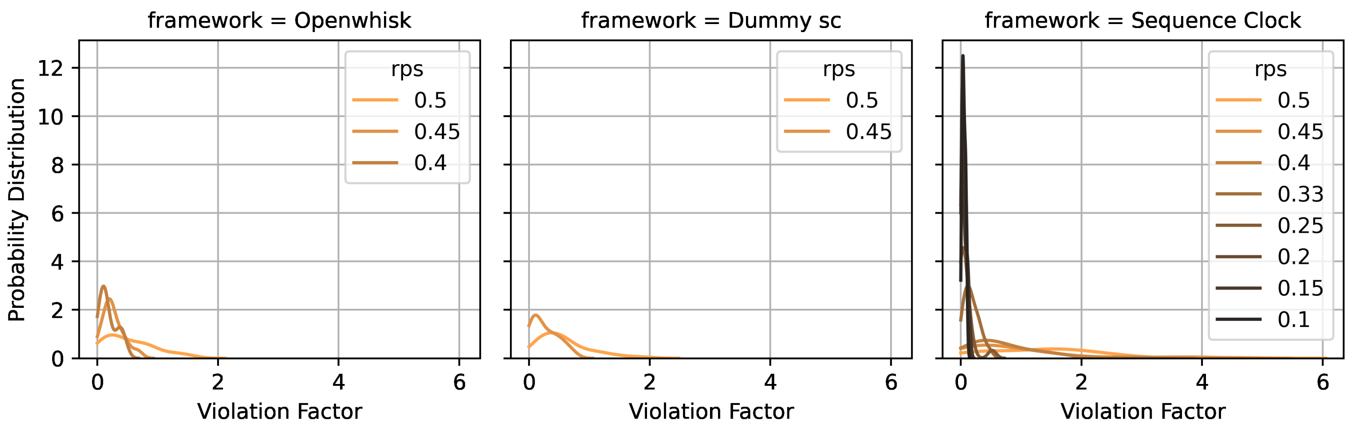


Figure 6.13: Violation Factor KDE plot for *stg6*.

Furthermore, a look at the CPU usage is useful like the one present in Figure 6.14. From this, the conclusion derived is that for weaker systems like *node2*, Sequence Clock results into lower values of CPU usage, but with the drawback of occupying CPU for a higher time interval. This can be explained by the fact that, for such nodes, the λ factor is being decreased rather quickly (due to the numerator's lower value in equation 5.4)

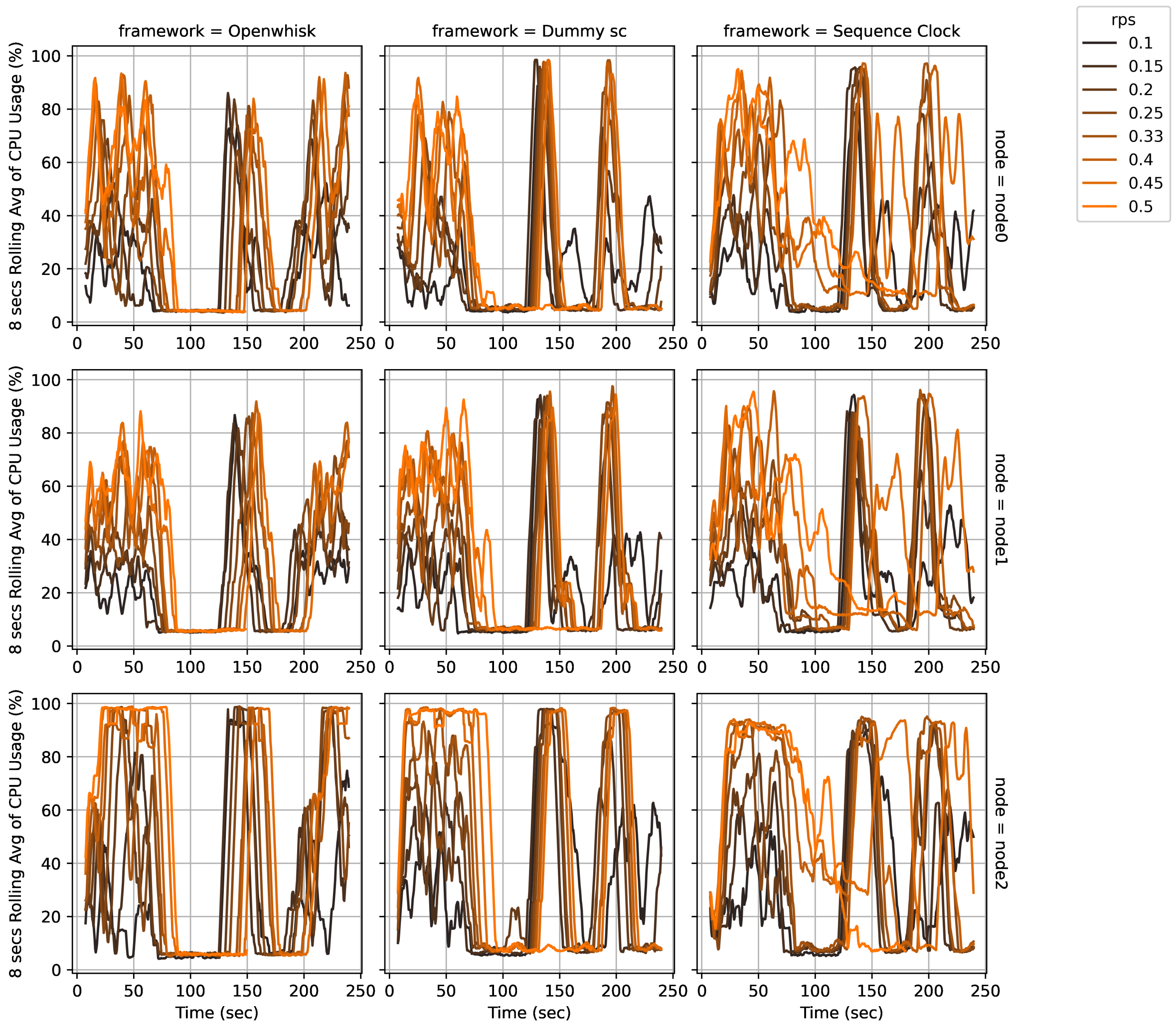
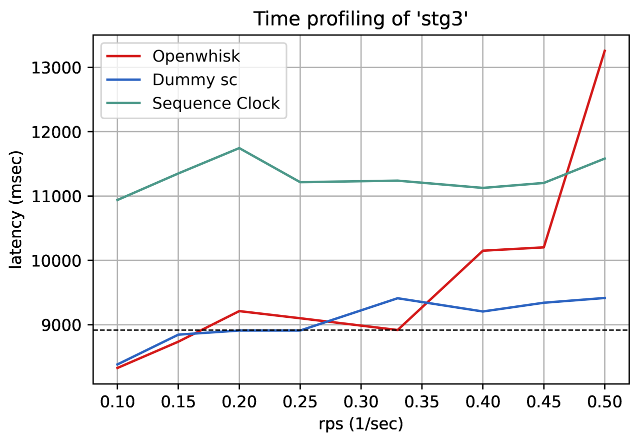
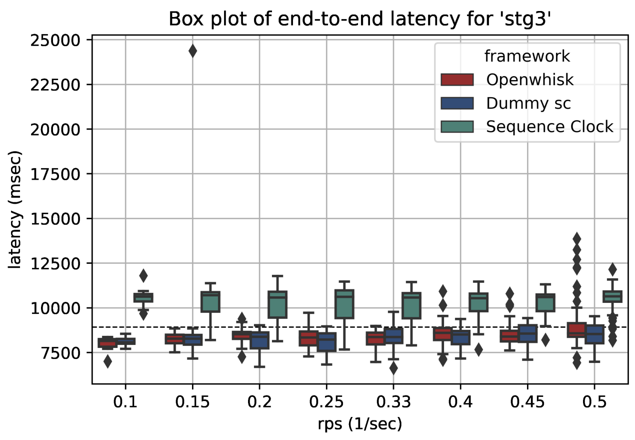


Figure 6.14: CPU Percentage (8-seconds Rolling Average) for *stg6*.

Moving on with *stg3*, it is being clear that sequences with less functions suffer less from latency elongation. Furthermore, Figure 6.15 reveals an additional disadvantage of Sequence Clock. The PID controller's gains cannot be optimized and tuned for all the pipelines, regardless their structure and inner functions, as the values that resulted in well behaved regulation in *stg6*, now result into a "steady state error" in *stg3*.



(a) 99-th Percentile for *stg3*.



(b) Time Latency Distribution for *stg3*.

Figure 6.15: Comparing OpenWhisk, Dummy SC, and SC with *stg3* pipeline.

As far as violations are concerned, the situation is worse than *stg6* as both the violation factor and violations' percentage remain high for all the *rps* levels.

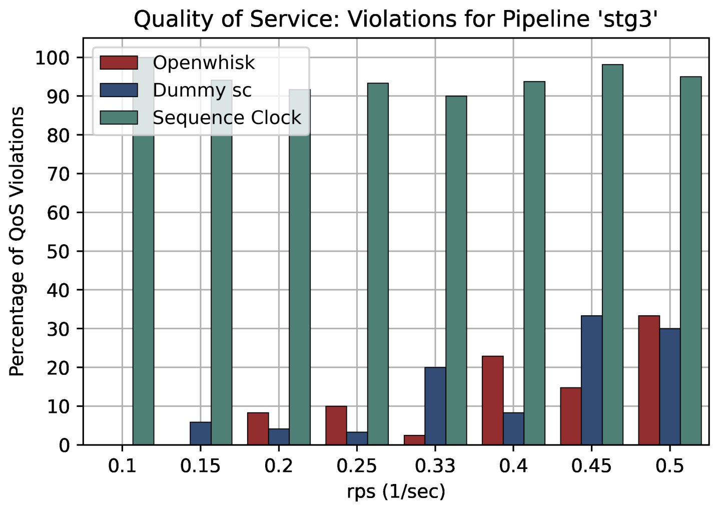


Figure 6.16: Violations' Percentage for *stg3*.

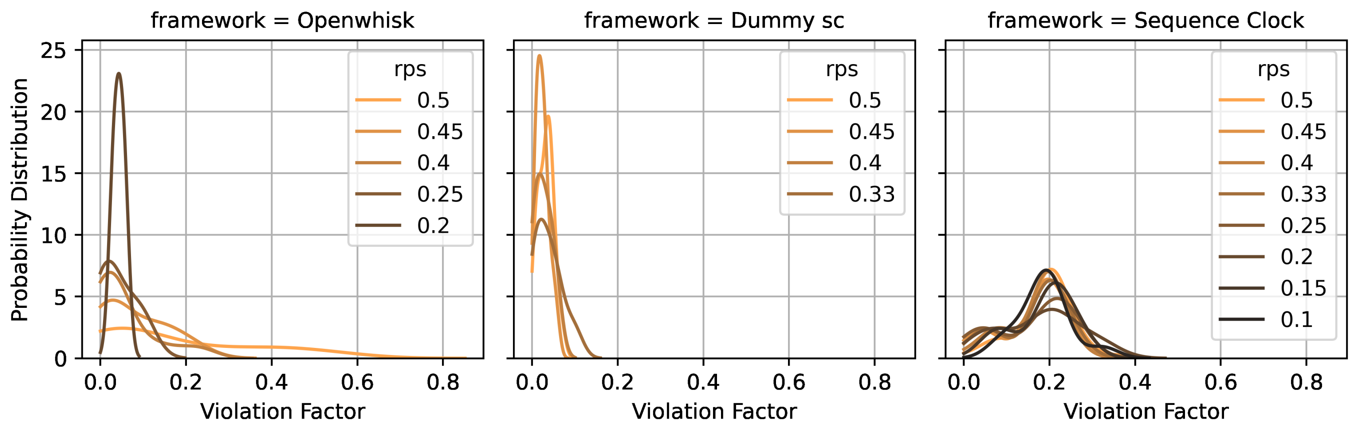


Figure 6.17: Violation Factor KDE plot for *stg3*.

However, concerning the CPU percentage usage, again Sequence Clock requires the same or less (for weaker nodes) of CPU as one can observe from Figure 6.18.

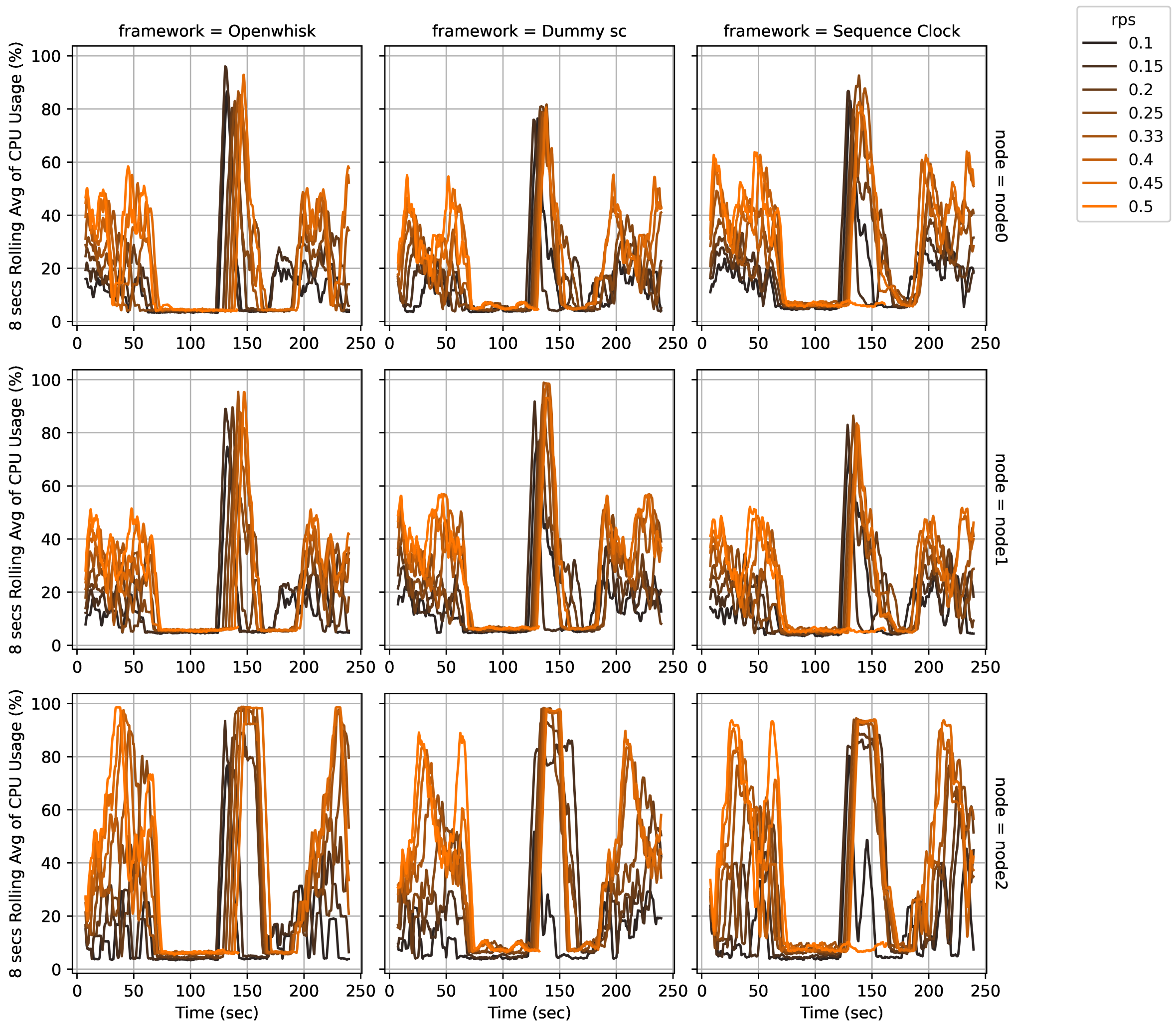
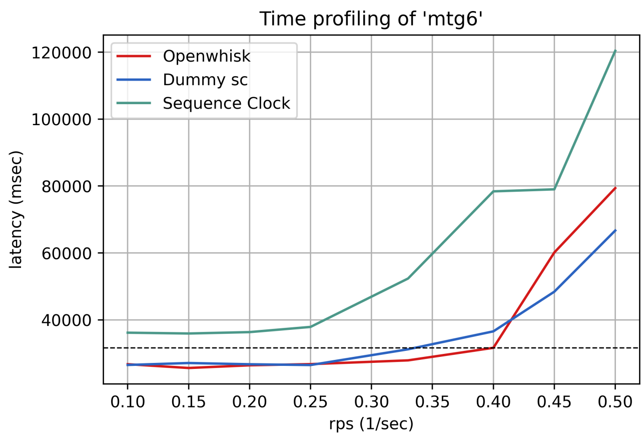


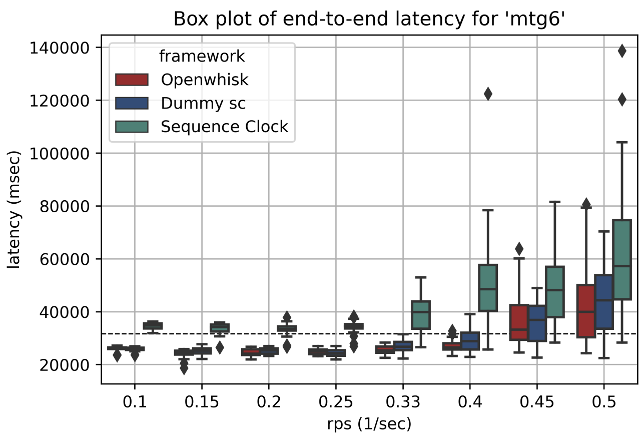
Figure 6.18: CPU Percentage (8-seconds Rolling Average) for *stg3*.

Multi-Threaded Pipelines

This category includes *mtg3* with a target latency equal to 12526 mseconds and *mtg6* with a target latency equal to 31648 mseconds. According to the following Figures, a similar situation with single threaded applications takes place. The steady state error is higher in the 3 functions pipeline, while the regulation mechanisms seems to be able to handle *rps* value lower than 0.33.

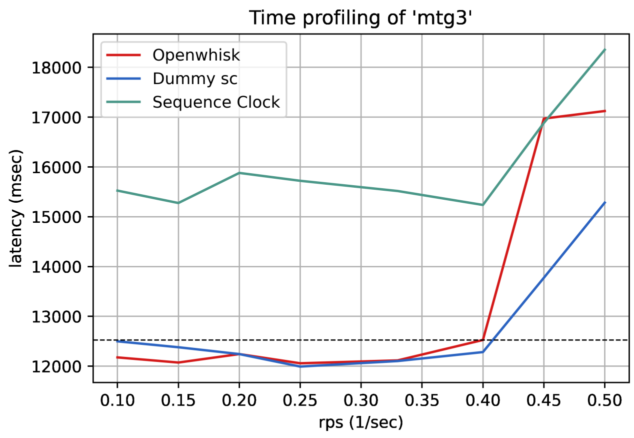


(a) 99-th Percentile for *mtg6*.

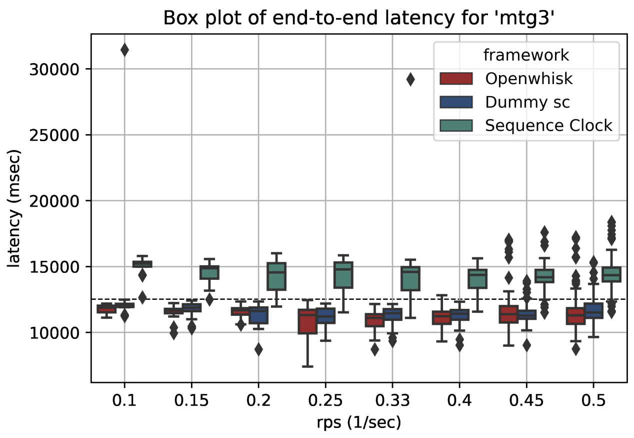


(b) Time Latency Distribution for *mtg6*.

Figure 6.19: Comparing OpenWhisk, Dummy SC, and SC with *mtg6* pipeline.



(a) 99-th Percentile for *mtg3*.



(b) Time Latency Distribution for *mtg3*.

Figure 6.20: Comparing OpenWhisk, Dummy SC, and SC with *mtg3* pipeline.

Multi-Processed Pipelines

Before moving into the analysis of multi-processed pipelines, i.e., *mpg3* with target latency 19389 mseconds and *mpg6* with target latency 26758 mseconds, it is important to mention a significant problem of OpenWhisk's API and OpenWhisk's Go client[3]. Specifically, there are two modes for invoking an action. The first one is the blocking mode, where main's program blocks (as the name implies), until OpenWhisk replies. Respectively, the second one allows asynchronous calls where OpenWhisk replies immediately only with the activation ID. This can be used later for querying *CouchDB* for the action's records.

The problem lies on the first one, where in high pressure situations OpenWhisk replies as in the async mode. Such behaviour prevents *Sequence Controller* from invoking the next function as it has no data to feed it. It would be extremely easy for us to implement a polling mechanism inside it in such

[3]We have not yet located if the problem lies on OpenWhisk or just the Go client.

cases. However, this approach would have created unnecessary network traffic, while *Sequence Controller* would have been waiting for the activation record to be written inside *CouchDB* rather than the actual action to be terminated.

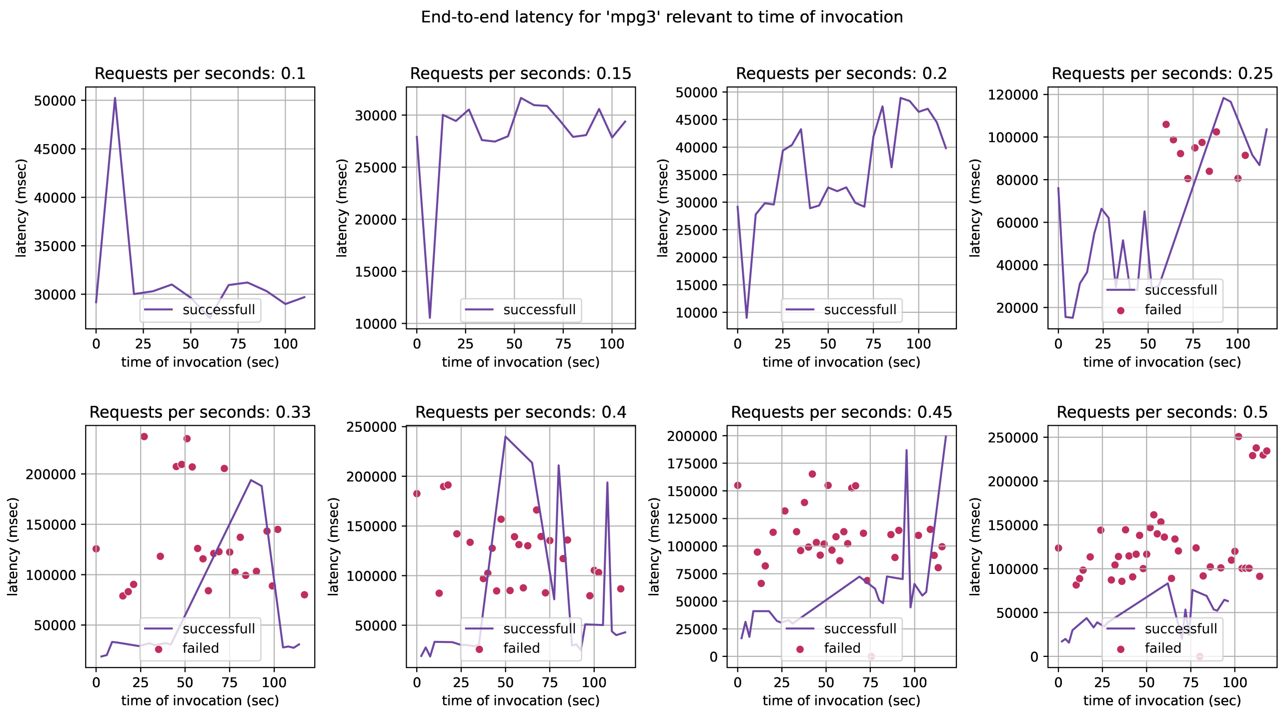
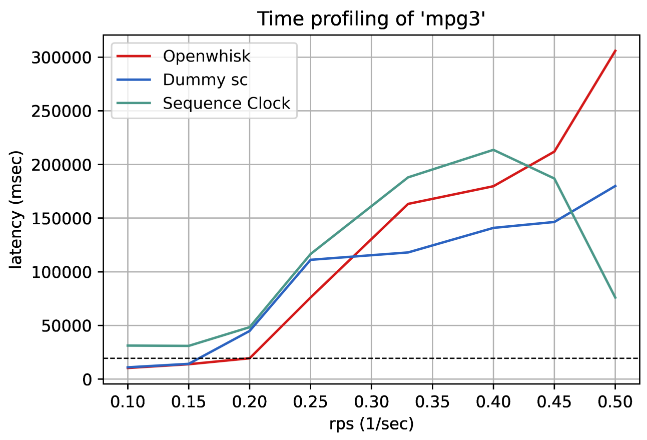


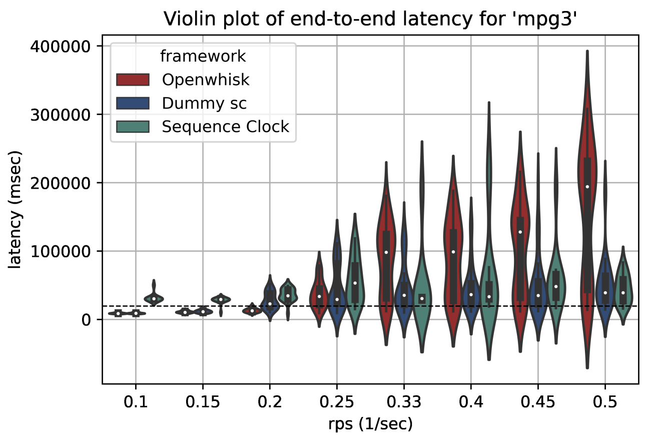
Figure 6.21: SequenceClock: Time Latency per *rps* value for *mtg3*. Failed invocations are marked as dots.

Consequently, the process of interpreting the results should be done with extreme caution whenever SC *sequences* terminate with an *action developer error*[4]. Such thing can be observed in Figure 6.21, where in higher *rps* values most of sequence invocations experience this, resulting in lack of measurements. Having mentioned this, we present the results for *mpg3* in Figure 6.22 and for *mpg6* in Figure 6.23.

[4]OpenWhisk Go client replies with an error value "Request accepted, process not finished yet." whenever this happen. *Sequence Controllers* read this error and panic, thus the *action developer error*.

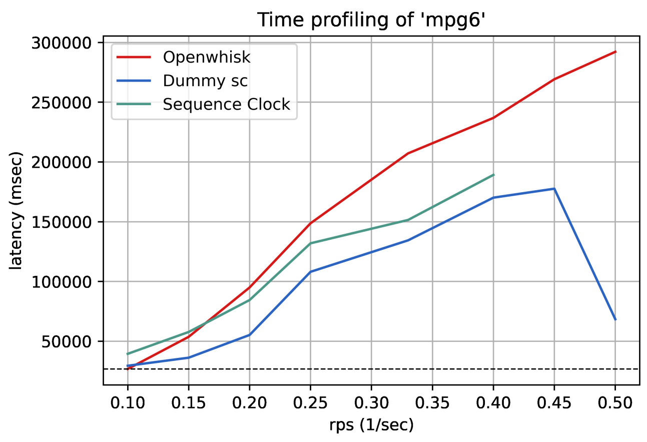


(a) 99-th Percentile for *mpg3*.

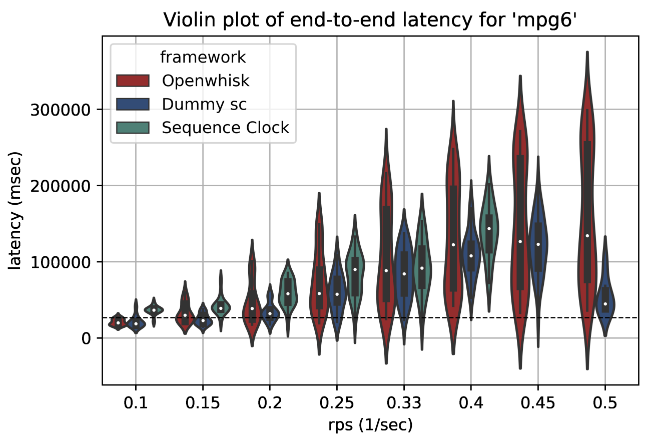


(b) Time Latency Distribution for *mpg3*.

Figure 6.22: Comparing OpenWhisk, Dummy SC, and SC with *mpg3* pipeline.



(a) 99-th Percentile for *mpg6*.



(b) Time Latency Distribution for *mpg6*.

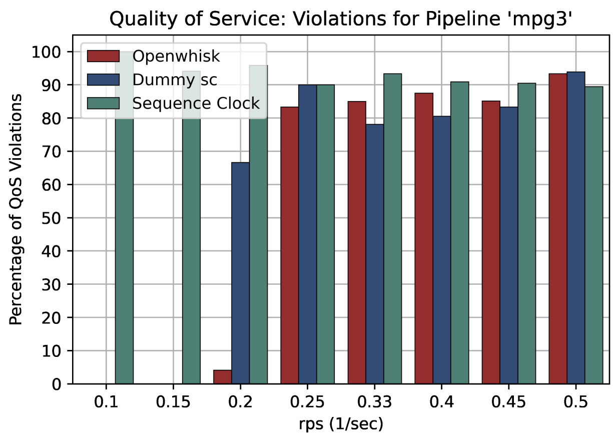
Figure 6.23: Comparing OpenWhisk, Dummy SC, and SC with *mpg6* pipeline.

Concerning the above graphs, two points should be made:

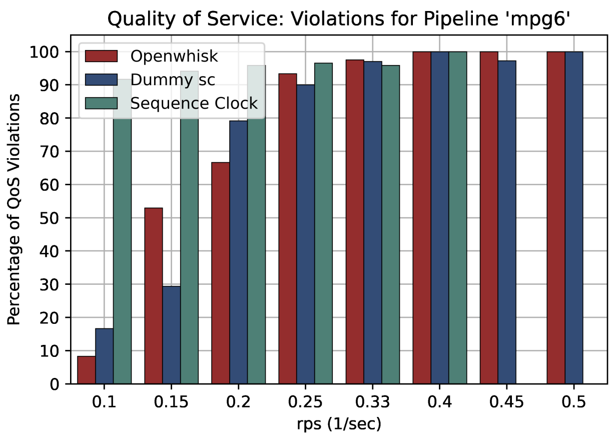
1. Multi-processed functions, and thus pipelines, do not follow the exponential like curve when plotted against *rps*. Instead, a linear like behaviour seems to occur, which actual is shown clearly in Figure 6.23a.

2. It would be disorienting to express that SC outperforms OpenWhisk, as for *rps* values higher than 0.33 failures start to occur with the last *rps* value not having a single successful invocation.

As far as violations are concerned, we observe the same performance, with violations to occur in all *rps* levels (see 6.24a, 6.24b) even in relative low pressure states.



(a) Violations' Percentage for *mpg3*.



(b) Violations' Percentage for *mpg6*.

Figure 6.24: Comparing OpenWhisk, Dummy SC, and SC with *mpg3* & *mpg6* pipelines.

6.3.2 SeBS Sequences' Analysis

As explained, in order to have a comprehensive evaluation we experiment with the additional pipelines created with the more general applications of SeBS.

Pipeline *p643*

Similarly with the generic ones, like *p643* with a target latency of 6023 mseconds, 3-function pipelines do not achieve the desired time latency and lead the system to violations as Figure 6.25 shows.

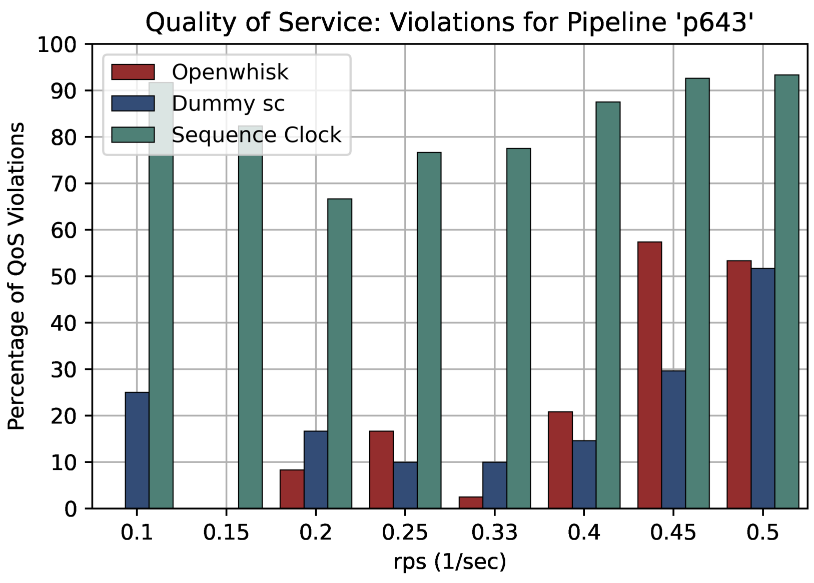


Figure 6.25: Violations' Percentage for *p643*.

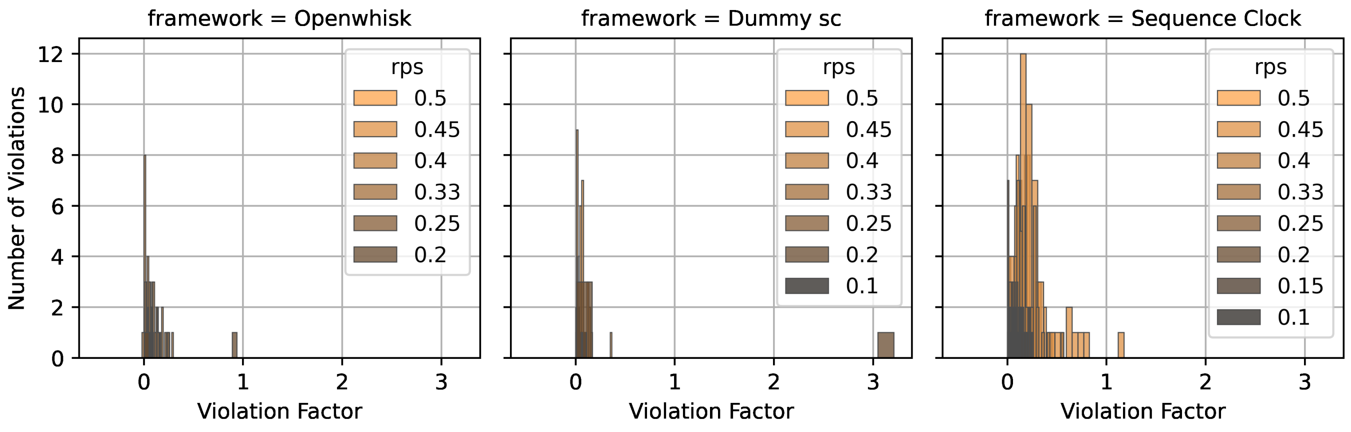


Figure 6.26: Violation Factor Histogram for *p643*.

Pipeline *p051463*

Lastly, we compare the performance of the three different frameworks, when used for the pipeline *p051463*. It is important to note that in this pipeline a behaviour similar the one we described in 6.3.1 occurs. Thus, unfortunately, there are less of measured values for higher *rps* levels as showned from Figure 6.27.

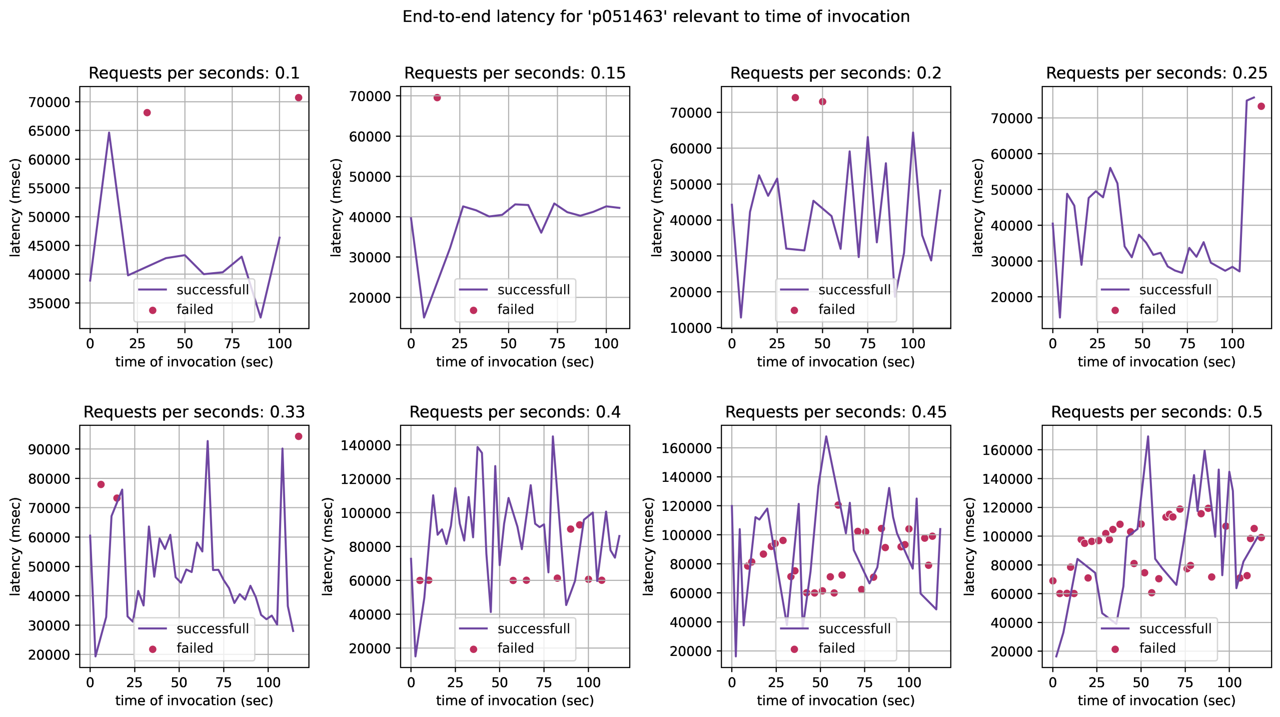
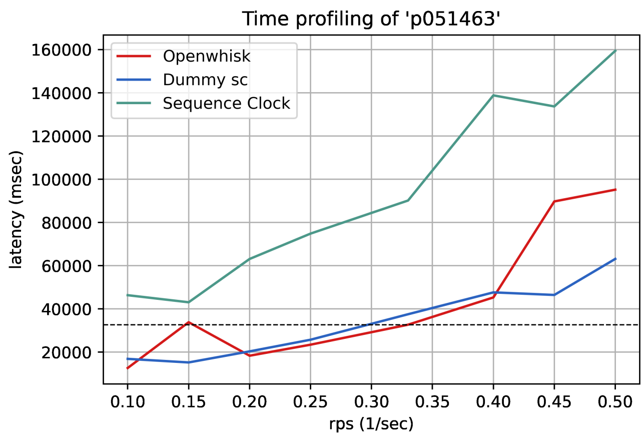


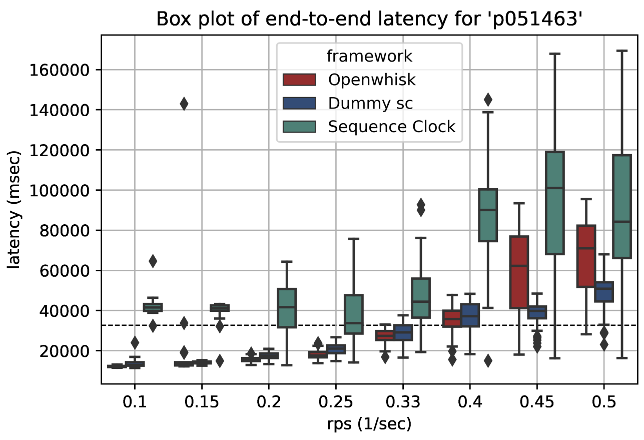
Figure 6.27: Time Latency per *rps* value for *p051463*. Failed invocations are marked as dots.

Nevertheless for the lower values of *rps*, this phenomenon is negligible, so we can draw a safe conclusion. In the case of *p051463*, as shown in Figure 6.28, the mean value of time latency is closer to the target, but the entire distribution in each level of *rps* experiences a significant variance (in contrast with *stg6*).

This is similar to the "steady state error" that *stg3* experiences, where the PID controller functions as un-optimised and un-tuned component.



(a) 99th-Percentile of Time Latency for *p051463*.



(b) Time Latency Box Plot for *p051463*.

Figure 6.28: Comparing OpenWhisk, Dummy SC, and SC with *p051463* pipeline.

Violations' analysis of the aforementioned observations is placed below, where both the violation factor's density (see Figure 6.30) and violations' percentage (see 6.29) are being plotted.

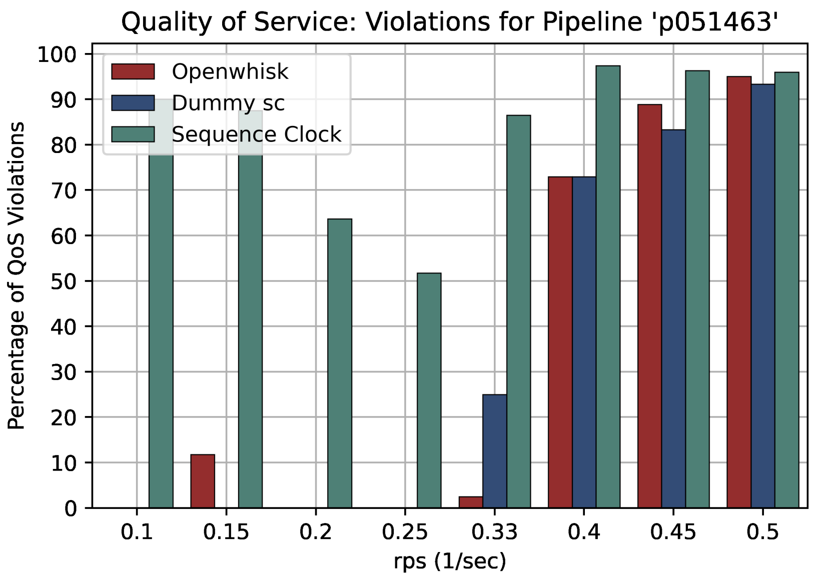


Figure 6.29: Violations' Percentage for *p051463*.

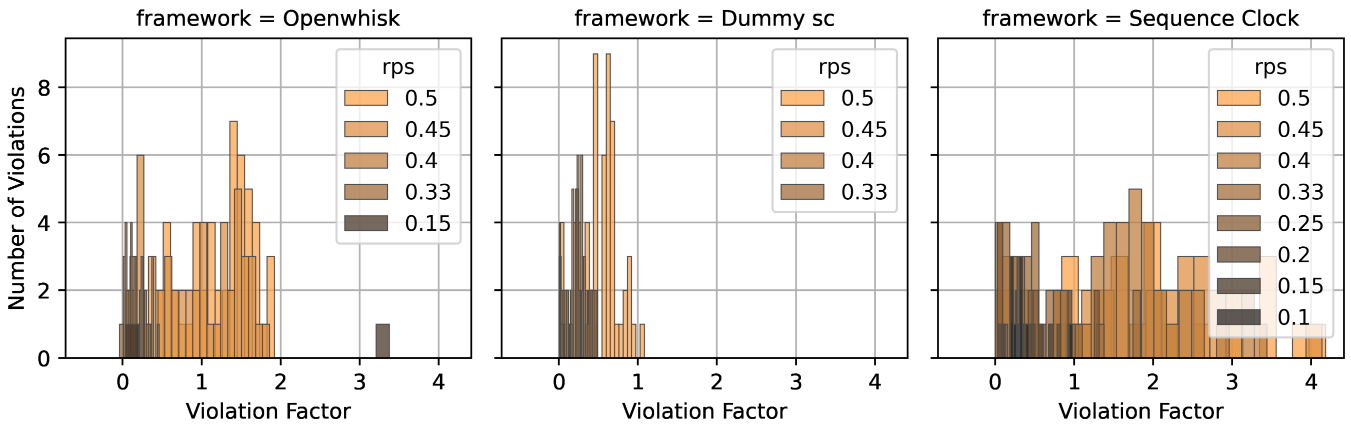


Figure 6.30: Violation Factor's Probability Density for *p051463*.

6.3.3 Causes of Failure

It is really important to look for the reasons that lead Sequence Clock to perform significantly worse than OpenWhisk and justify what exactly went wrong. First of all, we must point out that measurements showed that time latency behaves like a random variable, especially in high levels of *rps*. We emphasize that, as every form of conventional regulation (resembling the CPU quotas PID controller) will lead to a shift to distribution's mean and a scaling to distribution's variance. Thus, violations will appear even in situations, where they were absence, i.e., during lower values of *rps*. This would not have been a problem in SC's case, if these violations had remained between a reasonable interval (eg. with a violation factor less than 0.1) and a reasonable percentage.

The real problem becomes clear while studying the behaviour in Figures like 6.11a. One can observe latency to suffer from decelerating during *rps* levels that are higher than the value which specified each target. For example, target latency for the pipeline *stg6* was set as the 99-th percentile of the latency distribution during $rps = 0.33$, while SC seems to function worse for $rps > 0.33$. The reason behind this is the fact that quotas changes seem to achieve a descent result in decelerating actions and a terrible job in accelerating them. As CPU Quotas indicate amount on the CPU during a certain CPU period, a quantity larger than that will remain unexploited by the specified action if this action does not have a need of that extra time.

Additionally, Sequence Clock is built with the perspective of trying to regulate action sequences without a prior knowledge of the functions' code[5]. This results the PID controller to perform as expected for certain pipelines and behave as non optimised for others. To put it simply, the gains which result to

[5]This is an oversimplification. Of course pipelines' time profiling is needed, in order for targets to be extracted. Still, Sequence Clock does not set limitations to the developer concerning programming languages, actions' code implementation, etc.

a responsive system for 6-function pipelines, at the same time lead 3-function pipelines to a shifted latency distribution with action execution ending before slack settles into the desired value of zero. Other pipelines ended up with a latency that had a mean value near the desired one, but appeared with large values of variance, which implies less stability.

6.3.4 Dummy SC Superiority Paradox

If one pays a close attention on Figures 6.12, 6.17, and 6.23b, they will notice a strange and unexpected characteristic of Sequence Clock. Dummy algorithm performs better than OpenWhisk not only concerning time latency, as it suffers from less violations, but also concerning the violation factor density which is closer to zero. The latter means that in cases, where the percentage of violations is similar with the one of OpenWhisk, these violations are less severe.

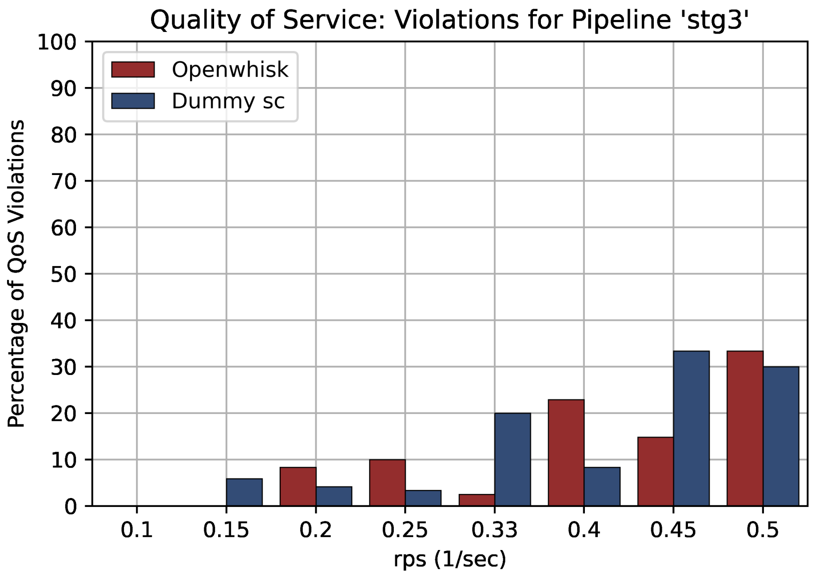
At this point an explanation is desirable on how such thing is possible. How an "unsupervised" system handles invocations in high pressure situations better than the supervised proposal and the default configuration of OpenWhisk? First things first, superiority over the greedy algorithm of Sequence Clock is explained by the aforementioned arguments of 6.3.3. However, this does not still provides a solid answer on the outperformance over OpenWhisk. The argument that we propose is simple. OpenWhisk handles sequences' invocation in a centralized way. No external or separated component is deployed inside the cluster with the purpose of orchestrating invocations of a running pipeline. On the contrary, Sequence Clock and thus Dummy Sequence Clock, deploys a separated action for this purpose and thus a separated pod/container inside the cluster. This decentralized way of invoking the internal functions of a pipeline offers more scalability and achieves both goals that we discussed in section 5.1.

Moreover, if we accept the above argument as true, then we can also explain the absence of network delay into Dummy execution even with the additional transfer of data from an action to another. It is no secret that this approach is not network optimised, as *sequence controllers* need to fetch each function's output and redirect it to next function's input[6]. Yet, the absence of such component (*Sequence Controllers*) in OpenWhisk's sequences indicates that a similar process occurs. In any case, data showed that if cluster nodes utilized with a more clever way, scalability and stability is achieved more easily.

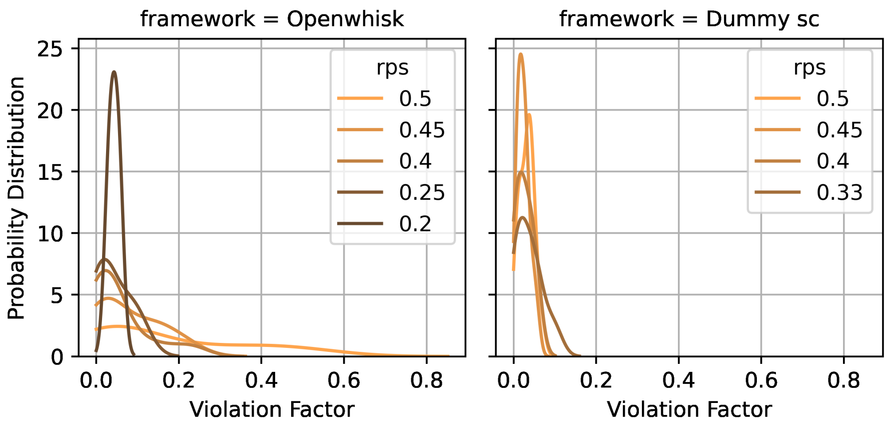
In conclusion, the cases where the above holds true is mainly in single threaded pipelines, as Figures 6.31a & 6.31b indicate. Also, realistic SeBS sequences experience similar behaviour (see Figures 6.32a & 6.32b). In multi-threaded and multi-processed pipelines as the more intensive ones, Dummy SC

[6]For a sequence of n functions $s = (f_0, f_1, ..., f_{n-1})$, Dummy *Sequence Controller* performs $2n + 2$ data transfers (including initial input and final output), while the actual needed data transfers are n.

suffers from approximately the same amount of violations' percentages, but it manages to maintain the violation factor lower than the values that OpenWhisk allows. The latter is visible in Figure 6.33, where the maximum of violation factor's distribution shifts by 73% (it drops from ~ 10 to ~ 6) and in Figure 6.31b, where a similar shift occurs by a factor of 82%.

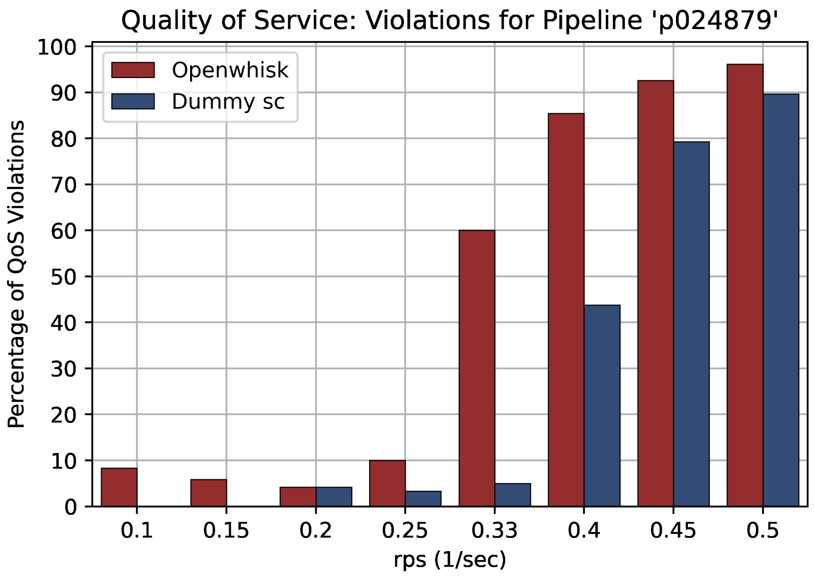


(a) Violations' Percentage for *stg3*.

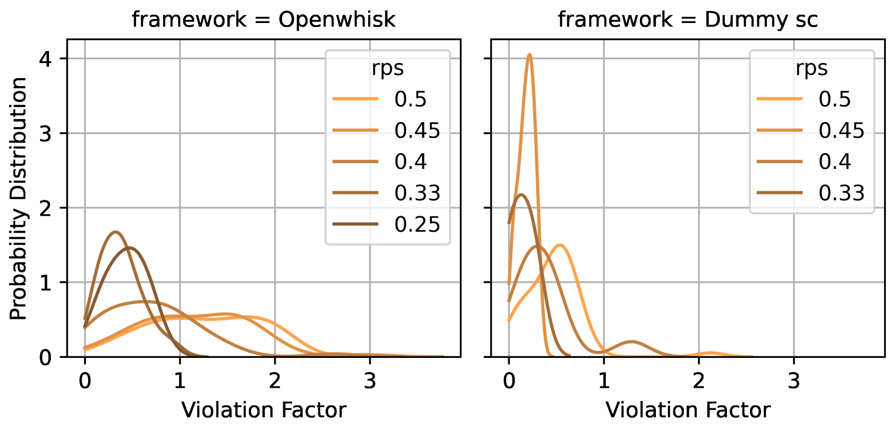


(b) Violation Factor's Probability Density for *stg3*.

Figure 6.31: Comparing OpenWhisk's with Dummy SC's Violations for *stg3*.



(a) Violations' Percentage for *p024879*.



(b) Violation Factor's Probability Density for *p024879*.

Figure 6.32: Comparing OpenWhisk's with Dummy SC's Violations for *p024879*.

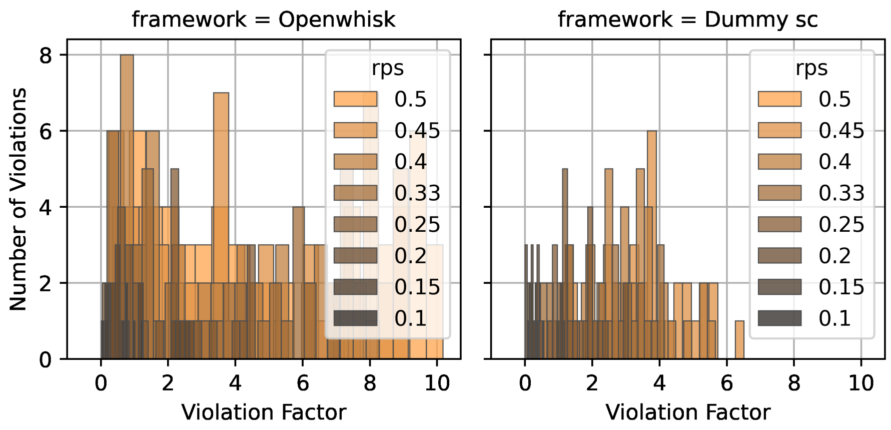


Figure 6.33: Violations' Percentage for *mpg6*.

Chapter 7

Conclusion and Future Work

7.1 Summary

In this thesis, we investigated the challenge of offering Quality of Service (QoS) in Serverless Architectures. The FaaS cloud computing model was presented with objective criticism concerning both its advantages and disadvantages over its predecessors. Additionally, in order to achieve a thorough understanding of this concept, we studied extensively various FaaS frameworks, like OpenFaas & OpenWhisk, their architectures and the usage of containerization technologies inside them.

Time latency constrains for serverless pipelines by this day remain a grey area for all the known Cloud Providers and open-source frameworks. Attempting to observe the outcomes of this problem, we performed a plethora of experiments at the function and at the sequence level by testing their performance response during variations of different variables and configurations. We identified a vast catalog of factors that have an impact on the pipeline's end-to-end latency and we concentrated on resource contention and system's pressure. The experiments showed, on average, an almost exponential growth in the time latency of a pipeline when plotted against increasing invocations rates, with the stability and predictability of the system following a similar fate. These observations make clear the necessity of a dynamic runtime resource management system in serverless platforms.

The next step to the development process was the design and the creation of Sequence Clock, a target latency tool for serverless pipelines, which tries to distribute fairly CPU resources into the hosted actions' containers of each node. Both the greedy version and the unsupervised/dummy version of Sequence Clock were evaluated by a set of runs, where the cluster was set under various stress conditions of increasing rate of invocations. These runs were later compared with the default performance of OpenWhisk under the same conditions. This process indicated the capability of Sequence Clock's control loop to decelerate serverless pipelines and achieve in certain situations more stable distributions of end-to-end latency near the specified target. The inability of greedy Sequence Clock to reduce time violations' percentages and to confine the severity of these violations was also shown by the detailed analysis of system's metrics. However, the same experiments proved that dummy Sequence Clock, an unsupervised, distributed and simpler approach was able to achieve the same or even better results than OpenWhisk. In more depth, it reduced the percentage of violations (in some cases even down to zero) and restricted the maximum of violation factor up to 82% for 3-function pipelines and up to 73% for 6-function pipelines. Finally, an exhaustive explanation was provided on the reasons that led Sequence Clock into this under-performed behaviour, whenever the CPU quotas control is enabled and its unexpected superiority over OpenWhisk with the usage of the dummy mode.

7.2 Future Work

In this thesis, although the extensive analysis that was presented concerning the time violations problem, the proposals described were an immature attempt to potentially offer a well-defined solution. Thus the absence of QoS in serverless architectures remains an open issue.

In general, the direction where this study needs to be headed to depends on the principles that one wants to build their framework/tool on. Before the design of a new solution, a fundamental question needs to be answered, i.e, should the framework be function aware demanding a prior knowledge of actions' requirements or not? The experiments taught us that more general and adaptive practises come at a price of misshits between different situations, while on the other hand they offer versatility and ease. Regardless the answer that one wants to provide to the aforementioned question, in the following lines they will find some interesting thoughts and proposals for future work for both development changes and research opportunities.

7.2.1 Development Scope

As far as Sequence Clock and its regulation mechanism based on CPU quotas are concerned, an interesting approach would be a combination of the greedy algorithm and the dummy method. Evaluation showed that CPU quotas regulation achieved descent results on low levels of rps by forcing latency distribution to fluctuate around the target with small variance. On the other hand, Dummy Sequence Clock achieved lower execution times than OpenWhisk even in situations of high invocations rates, where on average pipelines' runtimes needed to be accelerated. By these manners, our proposal is to test a configuration of the *Watchers* that will use CPU quotas regulation for decelerating pipelines, while in high contention situations and for pipelines with a negative slack (or pipelines with a history of negative slack) it will let the function containers to run uncontrollably, i.e. with the CPU quotas set to -1. A way of measuring the contention levels would be nothing more than the value of the λ factor.

Moreover, Sequence Clock is designed with the assumption that function containers can run concurrently multiple requests. However, runtimes like the Python's one do not offer concurrency and instead use a queue mechanism. An interesting change would be one regarding the *Conflicts Policy*, discussed in 5.2.7, where a queue aware approach and its performance could be tested in related experiments. For example, the *Active* data structure could be modified to include a timestamp for each request and use that for re-calculation of the actual slack and the requested resources (quotas).

7.2.2 Research Scope

Although the latter optimizations and modifications might offer further information into the phenomenon of time violations, a more crucial concept should be studied and that is the parameters and configurations that could result in an actual function's latency decrease. Had these variables found, it would set the foundations for future systems that could try to dynamically distribute, allocate or control system's resources for achieving the desirable result. Compeling parameters include CPU clock (if a way of controlling host's CPU frequency & clock from inside the containerized runtime is found), network bandwidth (for network intensive functions), memory bandwidth, and so on and so forth.

Lastly, further experimentation is required concerning a possible and dynamic decision process of cold starts, or a horizontal scaling mechanism in general. Logic states that there might be a certain turning point where the penalty of deploying a new container inside the cluster is less than the total function's execution time, if the system decides to reuse existing containers (where resource contention between other concurrent requests or a possible waiting queue might result into a significant time delay).

Bibliography

[1] "Pcmag encyclopedia: Cloud computing," https://www.pcmag.com/encyclopedia/term/cloud-computing, accessed: October 28th, 2021.

[2] "What is cloud computing: A beginner's guide," https://azure.microsoft.com/en-us/overview/what-is-cloud-computing/#benefits, accessed: October 28th, 2021.

[3] N. Suryavanshi, "What are cloud computing services [iaas, caas, paas, faas, saas]," https://medium.com/@nnilesh7756/what-are-cloud-computing-services-iaas-caas-paas-faas-saas-ac0f6022d36e, nov 2017, accessed: October 25th, 2021.

[4] A. Tzenetopoulos, D. Masouros, S. Xydis, and D. Soudris, *Interference-Aware Orchestration in Kubernetes*, 10 2020, pp. 321–330.

[5] T. Hou, "Iaas vs paas vs saas enter the ecommerce vernacular: What you need to know, examples more," https://www.bigcommerce.com/blog/saas-vs-paas-vs-iaas/#the-key-differences-between-on-premise-saas-paas-iaas, accessed: October 29th, 2021.

[6] "What is serverless," https://www.ibm.com/cloud/learn/faas?utm_medium=OSocial&utm_source=Youtube&utm_content=000023UA&utm_term=10010608&utm_id=YTDescription-101-What-is-Serverless-LH-Functions-as-a-Service-Guide, July 2019, accessed: October 25th, 2021.

[7] L. Wang, M. Li, Y. Zhang, T. Ristenpart, and M. Swift, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. Boston, MA: USENIX Association, Jul. 2018, pp. 133–146. [Online]. Available: https://www.usenix.org/conference/atc18/presentation/wang-liang

[8] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky,

M. Espinosa, R. Lin, Z. Liu, J. Padilla, and C. Delimitrou, “An open-source benchmark suite for microservices and their hardware-software implications for cloud amp; edge systems,” in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS ’19. New York, NY, USA: Association for Computing Machinery, 2019, p. 3–18. [Online]. Available: https://doi.org/10.1145/3297858.3304013

[9] “What is kubernetes?” https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/, accessed: October 30th, 2021.

[10] “Kubernetes secrets,” https://kubernetes.io/docs/concepts/configuration/secret/, accessed: October 30th, 2021.

[11] “Kubernetes components,” https://kubernetes.io/docs/concepts/overview/components/, accessed: October 30th, 2021.

[12] “Understanding kubernetes objects,” https://kubernetes.io/docs/concepts/overview/working-with-objects/kubernetes-objects/, accessed: October 30th, 2021.

[13] “Kubernetes namespaces,” https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/, accessed: October 30th, 2021.

[14] “Kubernetes pods,” https://kubernetes.io/docs/concepts/workloads/pods/, accessed: October 30th, 2021.

[15] “Kubernetes deployment,” https://kubernetes.io/docs/concepts/workloads/controllers/deployment/, accessed: October 30th, 2021.

[16] “Kubernetes deamonsets,” https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/, accessed: October 30th, 2021.

[17] “Kubernetes services,” https://kubernetes.io/docs/concepts/services-networking/service/, accessed: October 30th, 2021.

[18] “Kubernetes volumes,” https://kubernetes.io/docs/concepts/storage/volumes/, accessed: October 30th, 2021.

[19] “Kubernetes persistent volumes,” https://kubernetes.io/docs/concepts/storage/persistent-volumes/, accessed: October 30th, 2021.

[20] “Apache openwhisk: Open source serverless cloud platform,” https://openwhisk.apache.org/, accessed: October 31th, 2021.

[21] "Openwhisk deployment on kubernetes," https://github.com/apache/openwhisk-deploy-kube, accessed: October 31th, 2021.

[22] "What is nginx," https://www.nginx.com/resources/glossary/nginx/, accessed: October 31th, 2021.

[23] "Openwhisk: System overview," https://github.com/apache/openwhisk/blob/master/docs/about.md, accessed: October 31th, 2021.

[24] "Openwhisk helm chart: Configration choices," https://github.com/apache/openwhisk-deploy-kube/blob/master/docs/configurationChoices.md, accessed: October 31th, 2021.

[25] M. Copik, G. Kwasniewski, M. Besta, M. Podstawski, and T. Hoefler, "Sebs: A serverless benchmark suite for function-as-a-service computing," in *Proceedings of the 22nd International Middleware Conference*, ser. Middleware '21. Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3464298.3476133

[26] "Sebs: Serverless benchmark suite," https://github.com/spcl/serverless-benchmarks, accessed: February 4th, 2022.

[27] C. Tozzi, "Microservices vs. serverless architecture," https://www.sumologic.com/blog/microservices-vs-serverless-architecture/, march 2021, accessed: October 27th, 2021.

[28] F. Romero, M. Zhao, N. J. Yadwadkar, and C. Kozyrakis, "Llama: A heterogeneous & serverless framework for auto-tuning video analytics pipelines," *CoRR*, vol. abs/2102.01887, 2021. [Online]. Available: https://arxiv.org/abs/2102.01887

[29] "Ibm cloud docs: Cloud functions - system details and limits," https://github.com/ibm-cloud-docs/openwhisk/blob/master/limits.md, accessed: October 25th, 2021.

[30] S. Kotni, A. Nayak, V. Ganapathy, and A. Basu, "Faastlane: Accelerating function-as-a-service workflows," in *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, Jul. 2021, pp. 805–820. [Online]. Available: https://www.usenix.org/conference/atc21/presentation/kotni

[31] "Memory protection keys," https://www.kernel.org/doc/html/latest/core-api/protection-keys.html, accessed: October 27th, 2021.

[32] “Top 10 container orchestration tools,” https://appfleet.com/blog/top-10-container-orchestration-tools/, march 2021, accessed: October 30th, 2021.

[33] “etcd: A distributed, reliable key-value store for the most critical data of a distributed system,” https://etcd.io/, accessed: October 30th, 2021.

[34] “Kubernetes statefulsets,” https://kubernetes.io/docs/concepts/workloads/controllers/statefulset/, accessed: October 30th, 2021.

[35] K. Wuestkamp, “Kubernetes services simply visually explained,” https://medium.com/swlh/kubernetes-services-simply-visually-explained-2d84e58d70e5, october 2019, accessed: October 30th, 2021.

[36] “Kubernetes storage class,” https://kubernetes.io/docs/concepts/storage/storage-classes/, accessed: October 30th, 2021.

[37] “Kubernetes config maps,” https://kubernetes.io/docs/concepts/configuration/configmap/, accessed: October 30th, 2021.

[38] “Helm: The package manager for kubernetes,” https://helm.sh/, accessed: October 30th, 2021.

[39] “The apache incubator,” https://incubator.apache.org/, accessed: October 31th, 2021.

[40] “Couchdb,” http://couchdb.apache.org/#about, accessed: October 31th, 2021.

[41] L. P. Warner Onstine, “What is couchdb and why should i care?” https://www.infoq.com/articles/warner-couchdb/, jul 2012, accessed: October 31th, 2021.

[42] “Apache kafka,” https://kafka.apache.org/, accessed: October 31th, 2021.

[43] B. Marr, “What is kafka? a super-simple explanation of this important data analytics tool,” https://bernardmarr.com/what-is-kafka-a-super-simple-explanation-of-this-important-data-analytics-tool/, accessed: October 31th, 2021.

[44] M. Thömmes, “Squeezing the milliseconds: How to make serverless platforms blazing fast!” https://medium.com/openwhisk/squeezing-the-milliseconds-how-to-make-serverless-platforms-blazing-fast-aea0e995 april 2017, accessed: October 31th, 2021.

[45] "Creating and invoking docker actions," https://github.com/apache/openwhisk/blob/master/docs/actions-docker.md, accessed: October 31th, 2021.

[46] "Creating and invoking python actions," https://github.com/apache/openwhisk/blob/master/docs/actions-python.md, accessed: October 31th, 2021.

[47] "Creating and invoking go actions," https://github.com/apache/openwhisk/blob/master/docs/actions-go.md, accessed: October 31th, 2021.

[48] "Openfaas: Serverless functions, made simple." https://www.openfaas.com/, accessed: October 30th, 2021.

[49] "Introduction to openfaas pro," https://docs.openfaas.com/openfaas-pro/introduction/, accessed: October 30th, 2021.

[50] "Faas-netes: Serverless kubernetes controller for openfaas," https://github.com/openfaas/faas-netes, accessed: October 30th, 2021.

[51] "Openfaas: Auto-scaling," https://docs.openfaas.com/architecture/autoscaling/, accessed: October 30th, 2021.

[52] "The go programming language," https://golang.org/, accessed: October 30th, 2021.

[53] "Error loading shared library ld-linux-x86-64.so.2: on alpine linux," https://dustri.org/b/error-loading-shared-library-ld-linux-x86-64so2-on-alpine-linux.html, accessed: February 5th, 2022.

[54] A.-L. Barabási and R. Albert, "Emergence of scaling in random networks," *Science*, vol. 286, no. 5439, pp. 509–512, 1999. [Online]. Available: https://www.science.org/doi/abs/10.1126/science.286.5439.509

[55] S. Brin and L. Page, "The anatomy of a large-scale hypertextual web search engine," *Computer Networks and ISDN Systems*, vol. 30, no. 1, pp. 107–117, 1998, proceedings of the Seventh International World Wide Web Conference. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016975529800110X

[56] "The academic paper that started google," https://blogs.cornell.edu/info2040/2019/10/28/the-academic-paper-that-started-google/, october 2019, accessed: February 5th, 2022.

[57] "Squiggle library," https://github.com/Benjamin-Lee/squiggle, accessed: February 5th, 2022.

[58] "Gnu operating system: Gnu bash," https://www.gnu.org/software/bash/, September 2020, accessed: February 6th, 2022.

[59] "awk — linux manual page," https://man7.org/linux/man-pages/man1/awk.1p.html, accessed: February 6th, 2022.

[60] "top — linux manual page," https://man7.org/linux/man-pages/man1/top.1.html, accessed: February 6th, 2022.

[61] M. Galarnyk, "Understanding boxplots," https://towardsdatascience.com/understanding-boxplots-5e2df7bcbd51, september 2018, accessed: February 6th, 2022.

[62] "Gin web framework," https://github.com/gin-gonic/gin, accessed: November 1st, 2021.

[63] "The go blog: Defer, panic, and recover," https://go.dev/blog/defer-panic-and-recover, accessed: February 2nd, 2022.

[64] "Openwhisk command-line interface wsk," https://github.com/apache/openwhisk-cli, accessed: February 3rd, 2022.

[65] "Viper: Go configurations with fang," https://github.com/spf13/viper, accessed: February 3rd, 2022.

[66] "Github repository of cobra," https://github.com/spf13/cobra, accessed: February 3rd, 2022.

[67] S. Chen, C. Delimitrou, and J. F. Martínez, "Parties: Qos-aware resource partitioning for multiple interactive services," in *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, 2019, p. 107–120. [Online]. Available: https://doi.org/10.1145/3297858.3304005

[68] V. Satopaa, J. Albrecht, D. Irwin, and B. Raghavan, "Finding a "kneedle" in a haystack: Detecting knee points in system behavior," in *2011 31st International Conference on Distributed Computing Systems Workshops*, 2011, pp. 166–171.

Chapter 8

Appendix

1 PID Controllers

A PID Controller or propotional-integral-derivative controller is a control loop mechanism employing feedback that is widely used in industrial control systems and a variety of other applications requiring continuously modulated control.

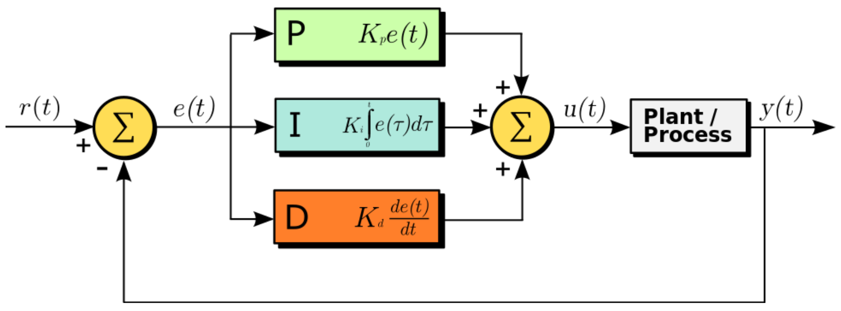


Figure 8.1: Block Diagram of a PID controller[1]

Let $y(t)$ be a measured process variable of a process (output) and $r(t)$ be the desired setpoint (input). Then, the open-loop control function in the time domain can be described by the following formula:

$$u(t) = K_p \cdot e(t) + K_I \cdot \int_0^t e(\tau)\, d\tau + K_P \cdot \frac{de(t)}{dt}$$

, where $e(t) = r(t) - y(t)$. The parameters K_I, K_P, K_D are called gains and each one has a specific purpose:

- K_P: Proportional gain.
- K_I: Intergal gain. It uses, of course an integral, which in practise is a summation of past values (the history of the system). Its results are sensible when there is a continuous error for a significant amount of time.

[1]https://commons.wikimedia.org/wiki/File:PID_en.svg

When the error is eliminated, the integral term will cease to grow. This will result in the proportional effect diminishing as the error decreases.

- K_D: With the derivative of the error, it tries to predict the future of the system by watching any tendency for change, the rate of change.The more rapid the change, the greater the controlling or damping effect.

Similarly, the transfer function (open loop) in the Laplace domain can be written as follows (initial error value is considered zero $e(0) = 0$):

$$U(s) = K_P + \frac{K_I}{s} + K_D \cdot s$$

Thus, the close-loop transfer function can be written as:

$$H(s) = \frac{U(S) \cdot G(s)}{1 + U(s) \cdot G(s)} = \frac{Y(s)}{R(s)}$$

, where $G(s)$ is the plant transfer function (the natural function of the system), $Y(s)$ and $R(s)$ the laplacian transformation of the output and input respectively.

1.1 Proportional Term

More or less, it defines the responsiveness of the system. The output that it produces is proportional to the error values. This means that wrongly large values of K_P may result in instability, while wrongly small values may result to a weak system sensitive to external noise.

1.2 Integral Term

Usage of this type of control aims accelerates the movement of the process towards setpoint and to eliminate an error called *steady state error*, which is nothing more than the difference between the desired final output and the actual one:

$$\lim_{t \to \infty} e(t) = e_{steadystate}$$

The physical meaning of the reason why the *integral term* is best suited for this is simple. Imagine a test case where the system has reached a point very close to the desired state. The error is small and thus the proportional contribution is small. In addition, if the rate of change of the error is zero, the *derivative term* also is unable to contribute for the correction.

Potential drawbacks to its usage is overshooting the desired state as it sees only the past and ignores the present.If this behaviour is dangerous for the system, the integral term must used with caution.

1.3 Derivative Term

The derivative term predicts system behavior and thus improves settling time and stability of the system. Maybe the most difficult gain to tune and it is never used by itself. However, it is the only one that decreases overshooting effect.