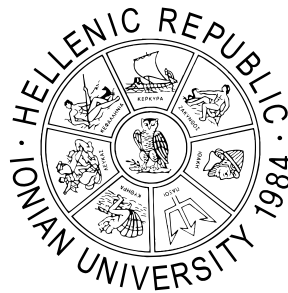


IONIAN UNIVERSITY

**Faculty of Information Science
and Informatics**

Department of Informatics



Research Directions in Informatics

M.Sc. Thesis

Resource Orchestration in Dynamic
Software-Defined Networking Environments

Alexandros Zervopoulos – ΠΙΜΕ201905

September, 2021

Supervisor

Konstantinos Oikonomou, *Professor*,
Ionian University

Advisory Committee Members

Konstantinos Oikonomou, *Professor*,
Ionian University

Christoforos Ntantogian, *Assistant Professor*,
Ionian University

Theodore Andronikos, *Associate Professor*,
Ionian University

To the challenges that keep us growing.
To the people that keep us going.

Abstract

Network management has traditionally been a complex task, due to the various interconnected components of a distributed system, which need to operate harmoniously with each other. The paradigm of Software-Defined Networking (SDN) promises to significantly simplify network management by providing centralized control of the entire network through an open and programmable interface. This aspect of centralized control is enabled by pushing intelligence away from the distributed data plane (e.g., switches) and onto the control plane, which is governed by the controller.

The first instantiation of SDN was the OpenFlow specification, which defines the various components of switches and controllers that facilitate the interactions between them. Perhaps the most notable interaction among these is the flow-setup process, by which a switch requests forwarding rules for a specific flow from the controller in the form of a `PACKET_IN` message. The controller responds to this request with a `PACKET_OUT` message, containing the necessary forwarding rules which are installed on the switch for future use. The time elapsed between the switch sending the `PACKET_IN` message and receiving the corresponding `PACKET_OUT` message is called the flow-setup time.

Despite its success and early adoption, OpenFlow has been criticized for not being scalable, in part due to this flow-setup process. In practice, it is not feasible to utilize a single controller for large enough networks, due to the large number of `PACKET_IN` messages that are generated from the switches. Furthermore, even if processing power is not the bottleneck, switches are likely to end up far away from the controller, resulting in long flow-setup times.

To address these concerns, distributing the control plane has been the primary solution, partitioning the network into multiple domains. In this way, each controller is generally only responsible for its own domain, providing better load balancing, flow-setup times and reliability. Nonetheless, much like any distributed system,

this solution does not come without its own set of challenges, such as the need synchronization, which needs to take place between controllers in order to provide a logically centralized view of the network.

Another such challenge is the task of appropriately placing the distributed controllers in the network, which has been referred to as the Controller Placement Problem (CPP). Fundamentally, this problem is similar to facility location problems, such as the k -median, although it somewhat differs, partly due to intricacies in the distributed flow-setup process present in protocols like OpenFlow. The CPP involves: (i) finding the optimal number of controllers, k ; (ii) optimally placing the k controllers within the network; and (iii) optimally assigning each switch to a controller.

This Thesis is concerned with addressing the CPP that arises in the distributed SDN context, assuming the number of controllers is known in advance, drawing similarities to the k -median problem. In particular, the CPP is formulated in terms of cost, which is further broken down to identify the cost that each switch contributes towards the final solution. It is shown that, unlike the k -median problem, reassigning (migrating) a switch to a different controller can potentially impact other switches. Thus, the switch assignment sub-problem of the CPP is by itself a hard combinatorial problem.

To provide a scalable solution to the CPP, a migration-based approach is proposed, incorporating controllers migrating from one virtual machine to another to relocate in the network, as well as switches migrating from one controller to another. Two kinds of policies inform the decision-making of these migrations: (i) the controller migration policy and (ii) the switch migration policy. In this way, the proposed approach involves a single migration of the employed controllers, followed by one or more successive rounds of switch migrations. These operations are complementary, as relocating the controllers may enable a more effective switch assignment, and vice versa.

The employed controller migration policy is based on an existing service migration approach that provides performance guarantees for the k -median problem, while requiring purely local information. It is shown that some of these performance guarantees still hold in the context of the CPP. As for the switch migration policy, two alternatives are proposed: (i) the distance-based policy, where switches are assigned to the closest controller; and (ii) the cost-based policy, which migrates

a switch if it leads to a cost reduction.

These policies are experimentally evaluated in a simulation setting to assess their effectiveness in reducing flow-setup time and overhead, in the form of control packets. The approach is initially evaluated in tree topologies using the OMNeT++ network simulator, along with the INET and OpenFlow frameworks, which are extended to accommodate the CPP. For general topologies, a custom Python-based simulator is constructed, providing higher-level abstractions to evaluate the proposed approach under more diverse conditions.

The acquired results indicate that the proposed approach is capable of achieving close to optimal solutions in tree topologies, despite of the initial controller allocation. However, it is shown that the switch migration policies perform identically, due to limited opportunities for migration being available, because of the tree structure. In general topologies, the proposed approach is also shown significantly improve upon the initial allocation. In this case, the cost-based switch migration policy outperforms the distance-based policy for small values of k , whereas the opposite is true for larger values of k .

Through these results, it is shown that the proposed approach can provide solutions to the CPP in a scalable manner, while also being capable of handling diverse conditions. In particular, although it is not verified experimentally, the proposed approach is expected to handle dynamic traffic scenarios well, due to the aspect of incrementally improving upon its current allocation. This, among other possible directions, are discussed upon and left for future work.

Περίληψη

Η διαχείριση ενός δικτύου αποτελούσε πάντοτε μία περίπλοκη διαδικασία, λόγω των διαφόρων διασυνδεδεμένων στοιχείων ενός κατακεντρωμένου συστήματος, τα οποία πρέπει να λειτουργούν αρμονικά μεταξύ τους. Η ανερχόμενη αρχιτεκτονική Software-Defined Networking (SDN) υπόσχεται να απλοποιήσει σημαντικά τη διαχείριση ενός δικτύου παρέχοντας κεντρικό έλεγχο ολόκληρου του δικτύου μέσω μιας ανοιχτής και προγραμματιζόμενης διεπαφής. Αυτή η δυνατότητα κεντρικού ελέγχου εξασφαλίζεται μετακινώντας την ‘ευφυΐα’ από το data plane (π.χ. switches) στο control plane, το οποίο ελέγχεται από τον επονομαζόμενο controller.

Η πρώτη εμφάνιση του SDN υπήρξε στο OpenFlow, το οποίο ορίζει τα απαραίτητα στοιχεία που ενσωματώνονται σε switches και controllers για την υποστήριξη των απαραίτητων αλληλεπιδράσεών τους. Πιθανώς η πιο χαρακτηριστική αλληλεπίδραση μεταξύ αυτών είναι η διαδικασία flow-setup, με την οποία ένα switch ζητά κανόνες για τη δρομολόγηση μιας συγκεκριμένης ροής από τον controller μέσω ενός μηνύματος `PACKET_IN`. Ο controller απαντά σε αυτό το αίτημα με ένα μήνυμα `PACKET_OUT`, το οποίο περιέχει τους απαραίτητους κανόνες δρομολόγησης, οι οποίοι εγκαθίστανται στη συνέχεια στο switch, σε περίπτωση που χρειαστούν στο μέλλον. Ο χρόνος που μεσολαβεί μεταξύ της αποστολής του μηνύματος `PACKET_IN` και της λήψης του αντίστοιχου μηνύματος `PACKET_OUT` ονομάζεται flow-setup time.

Παρά την επιτυχία και την υιοθέτησή του, το OpenFlow έχει επικριθεί ως προς την ικανότητα του να ανταπεξέλθει σε δίκτυα μεγάλης κλίμακας, εν μέρει λόγω της flow-setup διαδικασίας. Πρακτικά, δεν είναι εφικτή η χρήση ενός μόνο controller για αρκετά μεγάλα δίκτυα, λόγω του μεγάλου αριθμού μηνυμάτων `PACKET_IN` που δημιουργούνται από τα switches. Επιπλέον, ακόμη και αν η επεξεργαστική ισχύς δεν αποτελέσει εμπόδιο, τα switches είναι πιθανό να βρεθούν σε μεγάλη απόσταση από τον controller, με αποτέλεσμα να αυξηθούν δραματικά τα flow-setup times.

Για την αντιμετώπιση αυτών των προκλήσεων, έχει προταθεί ως η κύρια λύση στη

βιβλιογραφία ο κατανεμημένος έλεγχος του control plane από πολλαπλούς controllers, διαχωρίζοντας με αυτόν τον τρόπο το δίκτυο σε πολλά domains. Με αυτόν τον τρόπο, κάθε controller είναι συνήθως υπεύθυνος μόνο για το δικό του domain, παρέχοντας καλύτερη εξισορρόπηση του φορτίου, flow-setup times και αξιοπιστία. Παρ' όλ' αυτά, όπως κάθε άλλο κατανεμημένο σύστημα, έτσι και τα κατανεμημένα control planes συνοδεύονται από διάφορες προκλήσεις, όπως η ανάγκη για συγχρονισμό, ο οποίος πρέπει να πραγματοποιηθεί μεταξύ των controllers για τη σύνθεσή μιας έγκυρης λογικά κεντροποιημένης εικόνας του δικτύου.

Μια αντίστοιχα σημαντική πρόκληση είναι ο προσδιορισμός της κατάλληλης τοποθέτησης των controllers στο δίκτυο, το οποίο αναφέρεται ως Controller Placement Problem (CPP). Ουσιαστικά, αυτό το πρόβλημα είναι παρόμοιο με τα κλασικά προβλήματα facility location, όπως το k -median, αν και διαφέρει από αυτό, λόγω των ιδιαιτεροτήτων που παρουσιάζει το flow-setup process σε κατανεμημένα control planes, σύμφωνα με πρωτόκολλα, όπως το OpenFlow. Το CPP περιλαμβάνει: (α) την εύρεση του βέλτιστου αριθμού controllers, k , (β) τη βέλτιστη τοποθέτηση των controllers στο δίκτυο, και (γ) τη βέλτιστη ανάθεση κάθε switch σε έναν controller.

Η παρούσα Διπλωματική Εργασία ασχολείται με την αντιμετώπιση του CPP που προκύπτει στα κατανεμημένα SDN control planes, με την προϋπόθεση ότι ο αριθμός των controllers είναι γνωστός εκ των προτέρων, γεγονός που παρουσιάζει ομοιότητες με το k -median. Συγκεκριμένα, το CPP μοντελοποιείται ως προς το επιμέρους κόστος, το οποίο αναλύεται περαιτέρω για να προσδιοριστεί το κόστος που συνεισφέρει κάθε switch στην τελική λύση. Αποδεικνύεται ότι, σε αντίθεση με το k -median, η εκ νέου ανάθεση (migration) ενός switch σε διαφορετικό controller μπορεί με τη σειρά του να επηρεάσει το κόστος κι άλλων switch. Επομένως, το πρόβλημα της ανάθεσης switch που συμπεριλαμβάνεται στο CPP είναι από μόνο του ένα δύσκολο συνδυαστικό πρόβλημα.

Για την παροχή μιας αποδοτικής λύσης στο CPP σε μεγάλη κλίμακα, προτείνεται μια προσέγγιση βασισμένη στο migration, η οποία ενσωματώνει το migration των controllers από ένα εικονικό μηχάνημα (Virtual Machine) σε ένα άλλο, ώστε να μετακινηθούν σε άλλη τοποθεσία στο δίκτυο, καθώς και το migration των switches από έναν controller σε έναν άλλον. Δύο είδη πολιτικών καθορίζουν τη λήψη αποφάσεων για αυτά τα migrations: (α) η πολιτική για controller migration και (β) η πολιτική για switch migration. Με αυτόν τον τρόπο, η προτεινόμενη προσέγγιση περιλαμβάνει το migration των ενεργών controllers, ακολουθούμενο από έναν ή περισσότερους διαδο-

χικούς γύρους migrations των switches. Αυτές οι λειτουργίες αλληλοσυμπληρώνουν η μία την άλλη, καθώς το migration των controllers μπορεί να επιτρέψει μια πιο αποτελεσματική ανάθεση των switches και αντίστροφα.

Η πολιτική για controller migration βασίζεται σε μια υπάρχουσα προσέγγιση για migration υπηρεσιών, που παρέχει εγγυήσεις ως προς την απόδοσή του για το πρόβλημα k -median, ενώ απαιτεί μονάχα τοπική πληροφορία από το δίκτυο. Αποδεικνύεται ότι ορισμένες από αυτές τις εγγυήσεις απόδοσης εξακολουθούν να ισχύουν στο πλαίσιο του CPP. Όσον αφορά την πολιτική για switch migration, προτείνονται δύο εναλλακτικές λύσεις: (α) η πολιτική βάσει απόστασης, όπου τα switches ανατίθενται στον πλησιέστερο controller και (β) η πολιτική βάσει κόστους, η οποία αναθέτει εκ νέου ένα switch, εάν αυτό συνεπάγεται σε μείωση του κόστους.

Αυτές οι πολιτικές αξιολογούνται πειραματικά σε ένα περιβάλλον προσομοίωσης για να εκτιμηθεί η αποτελεσματικότητά τους στη μείωση του flow-setup time και την ελαχιστόποίηση των μηνυμάτων ελέγχου. Η προσέγγιση αρχικά αξιολογείται σε τοπολογίες δέντρων χρησιμοποιώντας τον προσομοιωτή δικτύου OMNeT++, μαζί με τα frameworks INET και OpenFlow, τα οποία επεκτείνονται για να επιτρέψουν την προσομοίωση του CPP. Για γενικές τοπολογίες, κατασκευάζεται ένας προσομοιωτής βασισμένος στη γλώσσα προγραμματισμού Python, ο οποίος παρέχει μία υψηλότερου επιπέδου πλατφόρμα για την αξιολόγηση της προτεινόμενης προσέγγισης υπό ποικιλία συνθηκών.

Τα αποτελέσματα που προκύπτουν υποδεικνύουν ότι η προτεινόμενη προσέγγιση είναι ικανή να επιτύχει σχεδόν βέλτιστες λύσεις σε τοπολογίες δέντρων, παρά την αρχική κατανομή των controllers. Ωστόσο, φαίνεται πως οι πολιτικές για switch migration λειτουργούν πανομοιότυπα, λόγω των περιορισμένων δυνατοτήτων για migration, γεγονός που αποδίδεται στη δενδρική τους δομή. Σε γενικές τοπολογίες, η προτεινόμενη προσέγγιση φαίνεται επίσης ότι παρέχει σημαντικά βελτιωμένες λύσεις σε σχέση με την αρχική κατανομή των controllers. Σε αυτήν την περίπτωση, η πολιτική για switch migration με βάση το κόστος υπερτερεί της πολιτικής βάσει απόστασης για μικρές τιμές του k , ενώ το αντίθετο ισχύει για μεγαλύτερες τιμές του k .

Βάσει αυτών των αποτελεσμάτων, συμπεραίνεται ότι η προτεινόμενη προσέγγιση μπορεί να παρέχει λύσεις στο CPP με αποδοτικό τρόπο σε μεγάλη κλίμακα, ενώ είναι επίσης ικανή να αποδίδει υπό διαφορετικές συνθήκες. Ειδικότερα, αν και δεν επαληθεύεται πειραματικά, η προτεινόμενη προσέγγιση αναμένεται να χειρίζεται καλά τα σενάρια δυναμικού traffic, λόγω της σταδιακής βελτίωσης που παρέχει σε σχέση με την

αρχική κατανομή των controllers. Αυτή, καθώς και άλλες ιδέες, προσδιορίζονται ως πιθανές μελλοντικές κατευθύνσεις, με στόχο την περαιτέρω βελτίωση της προτεινόμενης προσέγγισης.

Acknowledgments

First and foremost, I would like to express my gratitude towards my supervisor, Professor Konstantinos Oikonomou, for providing guidance, without which I would not be able to complete this thesis. Primarily, though, I would like to thank him for providing plenty of opportunities and experiences that have allowed me to grow and mature. Additionally, I would like to thank Prof. Oikonomou for always setting high standards and pushing me to accomplish my goals. I would also like to thank Assistant Professor Christoforos Ntantogian and Associate Professor Theodore Andronikos for their valuable comments over the course of completing this thesis.

I am also extremely grateful to my good friends and colleagues: Korina, Kostas, Stelios, Andreana, Spiros, and Giannis, with whom I have spent countless hours together. Thank you all for letting me play my part in our wonderful group, while encouraging me to grow as a person and improve as a friend. I am certain I would not have enjoyed this journey nearly as much without you and hope to share many more memories with you.

Last but certainly not least, I would like to sincerely thank my parents, Joanna and Panos, and my sister, Nasia, for always supporting me. They were always there for me in times of hardship and I will be forever grateful for that. I would not be able to get here, without their care and nonstop efforts. I can only hope to make them proud and reciprocate in the future.

Contents

A	Introduction	1
A.1	Scalability and The Distributed Control Plane	2
A.2	Related Work	4
A.3	Contribution	5
A.4	Thesis Structure	8
B	Software-Defined Networking	11
B.1	Software-Defined Networking Architecture	12
B.2	OpenFlow	14
B.3	Physically Distributed—Logically Centralized Control Plane	18
B.4	The Controller Placement Problem	20
B.4.1	Dynamic Variant	22
C	Formulation of the Controller Placement Problem	25
C.1	Control Plane Assumptions	25
C.2	System Model	28
C.3	Cost Formulation	29
C.3.1	Switch Cost	30
C.3.2	Switch Migration Costs	31
D	Migration-Based Approach to Controller Placement	35
D.1	Description	35
D.2	Controller Migration Policy	37
D.3	Switch Migration	39

E	Evaluation	45
E.1	Tree Topologies	45
E.1.1	Setup	46
E.1.2	Results	47
E.2	General Topologies	50
E.2.1	Setup	50
E.2.2	Results	51
F	Conclusions	55
I	Publication Table	65
	Publication Table	65
II	Glossary	66
III	Notations	68
IV	Abbreviations	70
	Abbreviations	70
	Index	71

List of Figures

B.1	The components of the SDN architecture as defined by ONF.	13
B.2	The components of an OpenFlow-enabled switch.	15
B.3	The OpenFlow handshake used to establish a connection between a controller and a switch.	16
B.4	The typical steps of packet processing performed by an OpenFlow switch.	17
B.5	Example of an The flow-setup process in an OpenFlow switch.	18
B.6	The two types of distributed controller architectures: flat(a) and hierarchical (b). The switches are assigned to controllers of the same color.	20
C.1	In-band (a) and out-of-band (b) control communication. In the former case, control communication takes place over the same links that data is forwarded through, requiring bootstrapping for the initial connection establishment. In the latter case, the switches communicate with the controller through different links, forming a separate network.	26
C.2	The flow-setup process for a flow from s_3 to s_5 : (a) Initial PACKET_IN message; (b) PACKET_OUT and FLOW_MOD messages within domain; (c) process repeats for next domain.	28
C.3	The case of switch migration in a network with multiple controllers: switch s_5 migrates from controller c_1 (a) to c_2 (b).	32
C.4	The two potential alternatives for migrating switch s from controller f to either g or h	33

D.1	Example of the two kinds of migrations that the proposed approach utilizes to alter the state of the network from (a) to (b).	36
E.1	Acquired results for $k = 1$: (a) The placement cost and mean flow-setup time. (b) The number of OpenFlow-related packets captured. .	48
E.2	Acquired results for $k = 2$: (a) The placement cost and mean flow-setup time. (b) The number of OpenFlow-related packets captured. .	49
E.3	Acquired results for $k = 3$: (a) The placement cost and mean flow-setup time. (b) The number of OpenFlow-related packets captured. .	50
E.4	The normalized placement cost calculated for the balanced initial placement across different values of k	52
E.5	The normalized placement cost calculated for the random initial placement across different values of k	53

List of Algorithms

1	Cost-Based Switch Migration Policy	42
2	Outline of Proposed Migration-Based Approach	43

Chapter A

Introduction

THE advent of Software-Defined Networking (SDN) has brought about a significant shift in the domain of network management, with major market players, such as Google [1], adopting this new paradigm for their infrastructures. This shift has occurred as SDN promises more centralized and efficient network management [2], which has traditionally been a fundamentally complex task in distributed networking systems. Given the trend of modern networks to continuously evolve and grow increasingly larger and more complex, which is evident in paradigms like cloud computing and multi-access edge computing [3], the need for more effective management approaches is undeniable.

Fundamentally, SDN aims to decouple the *control plane* and the *data plane*, so that the operation of forwarding elements, such as *switches* and routers, remains as simple as possible. This decoupling is achieved through the use of one or more *controllers*, which provide open and programmable interfaces to handle complex control plane operations. By replacing the traditionally distributed protocols operating over switches with programmable software, through the use of protocols like OpenFlow [4] and “logically centralized but physically distributed” controllers [5], operators are also more easily capable of parameterizing networks according to their needs, without being prohibited by the proprietary nature of traditional switches.

In practice, the “dumb” forwarding devices are instructed by the controller as to how to forward packets and perform other tasks related to the control plane. This is accomplished through interactions with the controller specified by protocols, such as OpenFlow [4], where switches effectively request forwarding rules from the

controller, which are then retained for future use. This process is known as the *flow-setup process*, which allows for controllers to respond to different situations, as specified by business requirements that are expressed through *applications* running on top of the controllers.

In this way, SDN allows for increased flexibility when managing networks, while serving as a complementary technology to various emerging areas, such as Network Function Virtualization [6]. More extensively, examples of network management, which are facilitated through the use of SDN include: network access control [7], traffic shaping and engineering [8], seamless virtual service migration and mobility [9], and load balancing [10, 11]. The commercial successes by Google [1] and the fact that movements, such as the Open Networking Foundation (ONF)¹, are backed up by a large semblance of networking companies showcases the potential of the SDN paradigm. Furthermore, much research effort has been devoted to extending the benefits of SDN to other environments, including the Internet of Things [12], Next-Generation Networks [13], Vehicular Networks [14], etc.

While this centralized element of control is certainly appealing, scalability concerns arise [15, 16], partly due to the fact that controllers require significant compute power to handle incoming requests from switches. Nonetheless, there are some characteristics, such as switch-to-controller latency, that cannot be improved by scaling vertically, but horizontally. Thus, many approaches have been proposed to address these shortcomings [16]. A major leap in improving SDN scalability, however, has been the idea of distributing the control plane, partitioning the network into multiple *domains*, each being governed by one or more controllers, to enable scalability, fault-tolerance, and responsiveness.

A.1 Scalability and The Distributed Control Plane

A *distributed control plane* seems like a straightforward way to provide scalability, but it does not come without its own set of challenges. Generally speaking, when multiple controllers are involved, each controller is only concerned with the requests originating from the switches in their domain. Due to this, controllers initially only hold a fraction of the network's state in their view, which needs to be shared to

¹<https://opennetworking.org/sdn-definition/>

the rest to form a *logically centralized* view. This is typically accomplished through some kind of synchronization scheme.

Therefore, even though introducing multiple controllers can enhance scalability, it also necessarily introduces additional overhead in terms of control messages, so a balance needs to be stricken. This also holds true when considering the flow-setup times between switches and controllers, as controllers are only responsible for altering the forwarding rules of the switches within their domains [17], according to OpenFlow, the most prominent SDN protocol. Therefore, installing forwarding rules along a flow that crosses multiple domains is potentially more costly than a flow that remains within a single domain.

To overcome such challenges, multiple approaches can be traced in the literature, each providing its own set of solutions. Among the first attempts towards a distributed control plane include Onix [18] and HyperFlow [5], while others have much commercial push behind them, such as ONOS [19], and constant advancements are being made to improve performance at scale, primarily exemplified by Orion [1]. The approaches generally differ in the employed synchronization scheme, the architecture structure (flat or distributed), and various mechanisms to provide fault-tolerance and reliability.

Regardless of each control plane's intricacies, some problems, like synchronization, are ever-present and need to be addressed. Perhaps the most prominent and typical problem that naturally arises in the distributed SDN control plane is the Controller Placement Problem (CPP), which was formulated by Heller et al. [20] relatively early on. This problem can be traced back to the k -median and facility location problems and involves: (i) finding the optimal number of controllers, k ; (ii) optimally placing the k controllers within the network; and (iii) optimally assigning each switch to a controller. Different solutions to this problem can have a large impact on flow-setup times (the time taken to complete the flow-setup process), fault-tolerance, etc [21].

Since its formulation by Heller et al. [20], the CPP has received much attention under different conditions and assumptions, partly because its initial formulation considers a rather static environment. For instance, Bari et al. [22] address the CPP in cases where traffic conditions fluctuate, requiring periodic reallocation of the controllers, which are assumed to operate over Virtual Machines (VMs). Thus, the reallocation may occur through the migration of the controller VMs from one

physical machine to another. It should be noted that, generally speaking, the CPP is more complex than the classic k -median and facility location problems, as it still involves variants of NP-Hard problems [20, 22, 23, 24], rendering optimal solutions intractable and making approximate solutions the more compelling option.

A.2 Related Work

Despite the potential benefits of SDN, scalability has been the primary concern for deploying it over large networks [15], especially for the first instantiation of SDN, the OpenFlow specification. However, over the various advancements in control plane design that have been developed with the passage of time, such concerns have been largely alleviated. To accomplish this, a major advancement was moving to a physically distributed control plane, deploying multiple controllers, while maintaining a logically centralized view of the network to facilitate network management.

Since then, various distributed control plane architectures have been proposed and even put to practice, with Google’s Orion [1] being deployed at extremely large scale. In any distributed control plane, the CPP naturally arises, playing an important role in a variety of relevant network performance metrics. As such, Heller et al. [20] initially formulated the CPP for SDN on 2012, relatively soon after Onix [18], the first approach towards a distributed control plane, was proposed (2010).

Heller et al. [20] primarily studied the CPP with regards to latency, particularly in the case of average and worst case switch-to-controller latency. Furthermore, they primarily focus on identifying the number and placement of the controllers in a Wide Area Network (WAN) setting, assuming that propagation latencies are known and that the switches are assigned to the controller with the minimum propagation latency, regardless of traffic.

However, the CPP has been extensively studied since then [25]. To start with, various objectives, aside from latency, have been investigated as alternatives for the CPP, including fault tolerance, load-balancing, etc. In fact, formulating the objective can vary significantly from setting to setting and across different control plane architectures, due to the fact a large number of factors can affect the control plane’s performance. For instance, higher distance between the controllers can increase the cost of synchronization that is needed to maintain a consistent global view of the network [22]. The network’s traffic also plays an important role and, though

most research focuses on static traffic, dynamic traffic is more realistic and makes the problem harder to solve, as periodic adjustments to the placement and switch assignment need to be made [22, 26].

Similarly, various different constraints may be imposed, such as the maximum latency or distance between a switch and its controller, and the controllers' limited capacity to process incoming requests, among others [27]. These constraints can provide a guarantee regarding the quality of the solution, at the cost of significantly reducing the feasible solution space and, as a result, potentially increasing the time to find a feasible solution. Thus, in practice, the values selected for each constraint need to be carefully considered and may even need to change over time, if an adequate solution cannot be found.

Regarding switch assignment, much like the work of Heller et al. [20], it is typically assumed that switches are assigned to the closest controllers. However, this is not necessarily optimal, as distributed control planes have a more complicated flow-setup process, where a portion of the load has to traverse multiple domains. Due to this, more intelligent switch assignment policies have been considered in the literature that achieve better load balancing [28] and lower flow-setup times [17].

The work by He et al. [17] could be considered closest to the approach proposed in this Thesis. More specifically, they consider the dynamic multi-controller scenario, where traffic spanning multiple domains is explicitly modeled as a mixed-Integer Linear Program. Two separate models are also derived, one dealing solely with the placement of the controllers assuming static domains, and another where the controllers are statically placed and only the switch-to-controller mapping is optimized. In this way, the impact of each sub-problem is investigated under different traffic conditions, revealing that the second model can often yield a better solution than the first, at the cost of longer runtime. Nonetheless, the original model addressing both sub-problems turns out to be more effective than either, though it also takes longer to converge. However, the authors rely on the Integer Programming formulation, rather than proposing an algorithm, which is more practical in realistic scenarios.

A.3 Contribution

In the present Thesis, the CPP is addressed in the context of a distributed SDN control plane. In particular, in the considered variant of the CPP, no additional con-

straints are imposed, like enforcing a capacity on the controllers, while the number of controllers is considered to be static and known in advance. Therefore, the challenge primarily lies in determining the controllers' location and optimally assigning the network's switches to them, assuming each domain is governed by a single controller. This variant of the CPP is closest to the traditional k -median problem; the primary difference lies in the switch assignment process, due to intricacies present in SDN protocols like OpenFlow, where each controller is only responsible for managing the switches in its own domain.

To quantify the difference from the k -median problem, the CPP cost is formulated in terms of each switch's contribution to the final solution. In this way, it is shown that reassigning (migrating) one switch to a different controller can affect the cost of others, which is not the case in the k -median problem. In the latter problem, nodes are typically assigned to the closest facility, so long as capacity concerns are not imposed, without regarding others' assignments. Thus, the switch assignment problem turns into a hard combinatorial problem by itself.

Furthermore, it is argued that VM migrations are a fundamental part of any approach addressing the CPP over a cloud-based network architecture. This motivates the development of a migration-centric approach, where controllers are capable of migrating to different locations in the network, whereas switches may migrate from one controller to another. As such, two kinds of migration policies are incorporated into this approach, describing the conditions under which a migration takes place: (i) the controller migration policy and (ii) the switch migration policy.

The employed controller migration policy allows the controllers to migrate on a hop-by-hop basis, adopting a service migration policy previously proposed by Oikonomou and Stavrakakis [29, 30] in the context of autonomic networks. According to this policy, controllers monitor and react to fluctuations in traffic by utilizing local information, while guarantees for its performance have been analytically derived. In the current work, this migration policy is shown to still maintain some of these guarantees in the SDN environment, producing a better controller allocation, if one is available.

Once a controller migrates, the set of its assigned switches may become sub-optimal and switch migrations can be used to arrive at a better solution. Two distinct switch migration policies are proposed and studied in this work: (i) the distance-based policy, which assigns switches to the closest controller, similarly to

the k -median problem; and (ii) the cost-based policy, which assigns switches to a controller if a cost reduction is achieved. The latter policy, while not as computationally efficient in general topologies, is compared to the former to examine whether improved CPP solutions can be derived, using more sophisticated switch migration policies than the ones commonly used for the k -median problem.

In particular, the cost-based switch migration policy is guaranteed to produce a switch assignment that is just as good or better than the current one. In an attempt to reduce its computational complexity, it is shown that migrating a switch only affects the switches that are directly adjacent to it; thus, to determine whether a migration reduces total cost, only the cost of the subset of affected switches has to be recomputed.

Putting things together, the proposed approach involves periodically migrating all controllers, followed by one or more rounds of switch migrations. Intuitively, the controller migration policy can improve the placement of a controller, but the switch migration policy may be used to optimize its overall placement, by altering the domains. These two aspects are somewhat complementary, a better switch assignment may allow for better controller migrations in the future, and vice versa.

In order to evaluate the proposed approach, experiments are conducted in a simulation environment. More specifically, the OMNeT++ simulator is used, in conjunction with the INET and OpenFlow frameworks, which are extended to provide a fairly realistic CPP environment for tree topologies. The approach is evaluated with respect to its ability to reduce flow-setup times as well as overhead, in terms of control packets. It is shown that the approach is capable of very closely approximating the optimal solution, which is achieved in only a few rounds of migrations, despite initially placing the controllers far away from the optimal locations. However, it is shown that in tree topologies the switch migration policy does not impact the cost much, with the two considered policies producing identical results. This is attributed to the fact that tree topologies offer very few opportunities for switch migrations, if domains are assumed to be connected.

Experiments are also carried out over general topologies, although in a more abstract simulation environment. In particular, a custom simulator is built in Python to assess the effectiveness of the proposed approach in solving the CPP over random graphs and a real-world network topology. It is shown that the cost-based migration policy performs well for fewer controllers, but does not offer much benefit over the

distance-based policy, when more controllers are considered. This may be due to the fact that only uniform traffic is considered, which may not allow for more complex switch assignments to take place.

A.4 Thesis Structure

The structure of the rest of this Thesis is outlined in the sequel. In Chapter B, a brief high-level introduction to the field of SDN is presented. In particular, the architecture and general principles behind SDN are illustrated, while highlighting the technologies and other enablers that led to its adoption. In particular, the OpenFlow specification is described extensively, including the OpenFlow communication protocol. Various mechanisms are illustrated, such as the OpenFlow handshake and the processing of packets performed by switches, so as to set the stage for the later Chapters. Furthermore, an overview is provided of the various approaches that have been proposed over the years towards a distributed control plane, in order to address scalability concerns surrounding the OpenFlow specification and SDN in general. In the distributed context, it is shown how the CPP arises and how solving it can improve the network's performance.

Chapter C provides a definition of the considered system model and formalizes the CPP in this context. In more detail, assumptions regarding the considered SDN architecture are initially laid out, such as its distributed nature, as well as the various operations that are expected of its components, including controller applications, etc. Then, the system is more formally studied, introducing notations and definitions to describe its various aspects, before the considered variant of the CPP is fully laid out. Furthermore, the contribution of each individual component of the system (switches, domains, etc.) to the total cost is analyzed, as well as how different operations (e.g., the migration of a switch) impact this cost.

In Chapter D, the approach proposed in the context of this Thesis is presented. Initially, some reasons are given that motivate migration as being considered a fundamental operation in the context of modern data center-like networks. It is through migrations that controllers operating over VMs may relocate, which are inherently considered in most approaches addressing the CPP. Thus, the migration-based approach, describing in full detail its two components, the controller and switch migration policies, that drive the decisions behind migrations. Some analytical results

describing the performance of these policies are derived, while contrasting how previous work in the k -median problem can be adapted to accommodate the SDN context.

In Chapter E, the proposed approach is experimentally evaluated through simulations to assess its effectiveness in lowering flow-setup time and control traffic. The migration-based approach is initially assessed in the context of tree topologies using a fairly realistic simulator that incorporates the OpenFlow protocol stack. For general topologies, a higher-level simulation environment is utilized, showcasing how the proposed approach performs under a variety of conditions. The acquired results are discussed upon, explaining the reasons behind the performance of the proposed approach across the different scenarios.

Finally, a summary of the work is made in Chapter F with respect to what was accomplished in the context of this Thesis. Afterwards, conclusions are drawn regarding the effectiveness of the proposed approach, while possible directions are discussed towards extending some of the presented ideas in future work.

Chapter B

Software-Defined Networking

THE emerging technology of SDN promises to facilitate the programmatic and efficient configuration of networking elements, such as switches. The primary motive behind SDN is the decoupling of the control plane from the data plane, the latter being handled by a centralized entity, the controller. This enables not only easier configuration of the network, but also the monitoring and orchestration of more complex control plane tasks.

First, it is important to distinguish what is meant by the terms control plane and data plane, as well as highlight the need for an open, programmable, and centralized control plane. The data plane, often called the forwarding plane, is concerned with merely forwarding network traffic from its source to its destination. The control plane, on the other hand, revolves around *how* to forward network traffic, with the primary example being routing, i.e., finding the paths that traffic will follow. Traditionally, both of these aspects are integrated into a single forwarding device (a switch or a router), with the control plane tasks being achieved through distributed protocols. Additionally, many of the operations these devices carry out are hardware-based, making them difficult to update and configure. Combined with the fact that these devices feature many vendor-specific and proprietary functions, network operators often end up being “locked” into a single vendor, as interoperability with other vendors is not common.

These aforementioned pain points initially pushed the SDN movement, with the Open Networking Foundation (ONF) helping advance technologies and bring together open-source projects, building towards the open and programmable vision

of SDN [31]. Nowadays, major enterprises, vendors, and operators have integrated SDN technologies into their infrastructure, showcasing its capabilities, even at extremely large scales [1]. It is, therefore, hard to doubt the success of SDN and its impact in facilitating network management and configuration.

Thus, it is important to discuss the key enablers of SDN as well as challenges that need to be addressed, before its promised benefits can be fully realized. In this Chapter, the structural components of the SDN architecture are outlined, along with the primary technologies and protocols that come alongside it. Furthermore, concerns regarding scalability are discussed, highlighting the need for a physically distributed control plane that still retains the benefits of a logically centralized controller. To this end, various approaches towards a more scalable and decentralized SDN control plane are briefly described. Finally, in any physically distributed SDN architecture, where multiple controllers are deployed, the issue of appropriately placing these controllers in the network needs to be addressed, giving rise to the CPP. Along with the CPP, a few of its variants that commonly occur in practice are discussed in the sequel.

B.1 Software-Defined Networking Architecture

The SDN architecture, as defined by ONF¹ and depicted in Fig. B.1, is divided into three layers: (i) the Infrastructure Layer; (ii) the Control Layer; and (iii) the Application Layer. These layers are described in detail in the sequel:

- The *Infrastructure Layer* primarily consists of programmable forwarding elements, such as switches and routers, which perform data plane operations, including forwarding, monitoring and statistics collection. These forwarding elements inform the controllers of their capabilities through the Southbound Interface (SBI), the most typical implementation being the OpenFlow protocol.
- The *Control Layer* comprises the controllers, which are responsible for control plane operations. A controller may be hosted on one server or more servers. The controllers are responsible for the decision-making process in an SDN

¹<https://opennetworking.org/sdn-definition/>

network, instructing and coordinating the forwarding elements of the Infrastructure Layer through the SBI.

- The *Application Layer* defines the high-level requirements and policies which determine the controllers' decision-making in the form of applications. These applications communicate with the controller through the Northbound Interface (NBI). So far, there is no NBI technology that is as prominent as OpenFlow is for the SBI, so general-purpose Application Programming Interfaces (APIs) are often used instead, which depend on the particular implementation of the controller.

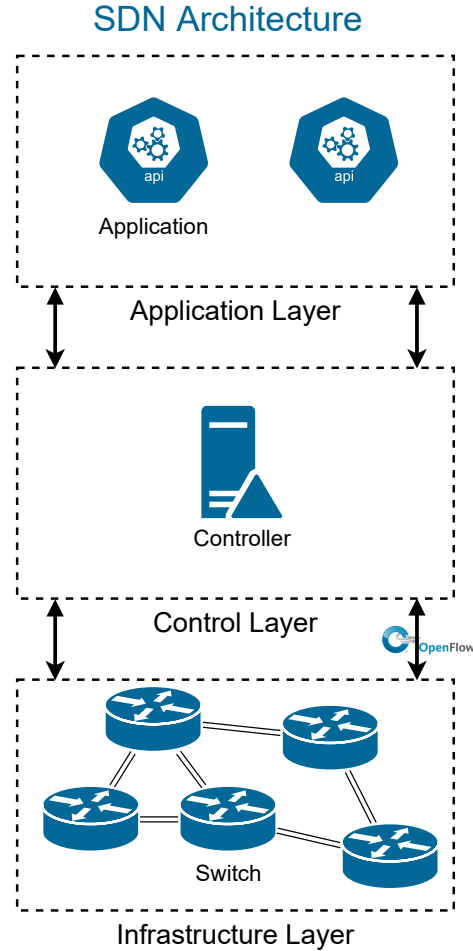


Figure B.1: *The components of the SDN architecture as defined by ONF.*

The SDN architecture essentially provides an easier way to translate high-level business requirements to the low-level infrastructure equipment through the use of the controller. Thus, the operator is concerned with developing and adjusting the high-level applications that suit their needs, which are then translated by the controller to rules that are propagated to the forwarding elements in the form of low-level instructions. As such, the switches or routers simply operate based on these instructions and do not need to participate in any of the decision-making processes related to the control plane, as they simply need to communicate with the controller. This allows for the removal of sophisticated protocols and complex state from switches, leaving them with the sole responsibility of forwarding packets according to the installed forwarding rules.

Instead, much of the sophistication is now moved to the controller, which needs to maintain a central and unified view of the entire network, including topology, traffic monitoring and statistics collection. This global view allows for easier troubleshooting, while also facilitating the operation of more complex high-level applications, such as Network Function Virtualization, malware-detection, and network access control, providing greater value to the network operator.

B.2 OpenFlow

OpenFlow is one of the most prominent specifications for the SBI, as it was initially considered one of the primary enablers of SDN. The OpenFlow specification lays out the main components of a switch, while also specifying its interactions with remote controllers in the form of the OpenFlow protocol. Even though it is no longer necessarily considered the de facto SBI specification, OpenFlow laid the foundations that most of the newer specifications build on top of. As such, this Section outlines the basics of the OpenFlow specification, particularly version 1.3, as it is currently the most widely supported version by commercial switches.

The components of an OpenFlow-enabled switch are illustrated in Fig. B.2. Most importantly, a switch is composed of a group table and at least one flow table. These tables contain the forwarding rules, which are installed by the controller. The controller manages the switch through packets specified by the OpenFlow protocol, which are transferred through a secure channel. These OpenFlow packets can be used to add, update or delete *flow entries* in the flow tables. It is worth noting that

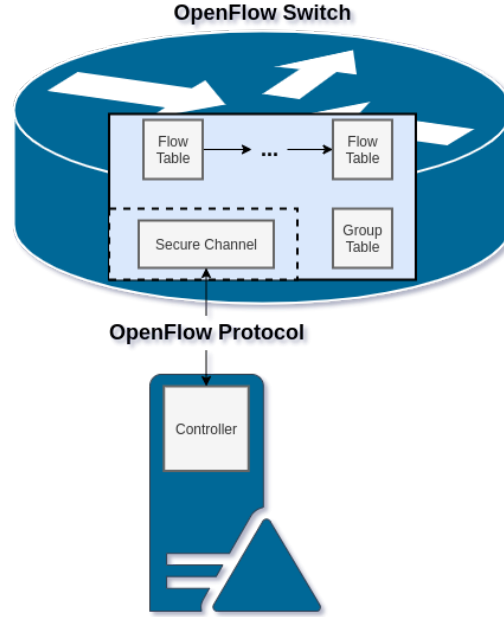


Figure B.2: *The components of an OpenFlow-enabled switch.*

Match Fields	Priority	Counters	Instructions	Timeouts	Cookie
--------------	----------	----------	--------------	----------	--------

Table B.1: *Fields contained in a flow entry.*

these operations can take place both proactively (e.g., during specified intervals) or reactively (i.e., in response to a message or event).

There are a few important points to be made regarding flow entries. The main information contained in a flow entry is specified by match fields, which are used when the switch performs a lookup to forward an incoming data packet and include information regarding its header and the port through which the packet arrived (ingress port). If the lookup is successful, the actions that a switch performs are specified by the entry's instructions, which may involve forwarding the data packet to a specific port, flooding it to all ports, or even dropping it. If the lookup is not successful, the switch will react according to the table-miss flow entry; usually, the switch will forward the packet to the controller, awaiting for further instructions. Furthermore, a flow entry may contain additional information, as shown in Table. B.1, such as the entry's priority to indicate precedence, counters for statistics collection, or timeouts for determining the entry's lifetime.

When more than one flow tables are used, they can form a pipeline, allowing for more complex packet processing and matching patterns. The group table, on the other hand, contains group entries, rather than flow entries. These can also be used for additional packet processing, although the details are spared here. It is worth noting that it is possible for a switch to operate according to its “traditional”, non-OpenFlow configuration, usually specified by its vendor, allowing for traditional and OpenFlow-enabled switches to co-exist, facilitating the incremental upgrade to SDN infrastructures.

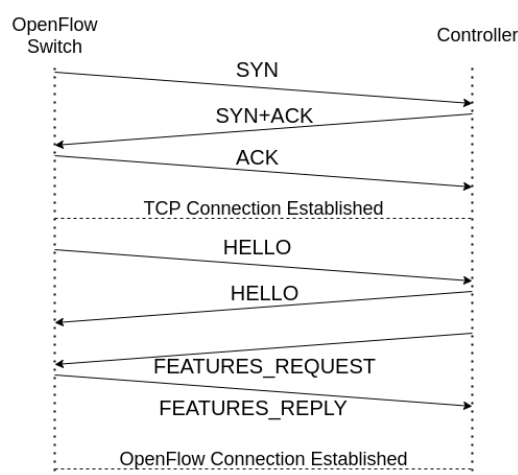


Figure B.3: *The OpenFlow handshake used to establish a connection between a controller and a switch.*

It is worth outlining the process that is used to establish a connection between a switch and a controller. The switch maintains a static Internet Protocol (IP) address entry for the controller that is configured by the user. Having been configured with a valid controller IP address, the switch may proceed to initiate a secure Transmission Control Protocol (TCP) connection with the controller. Afterwards, a sequence of messages, often termed the OpenFlow handshake, ensues, which consists of a HELLO message from the switch, to which the controller also responds with a HELLO message. Afterwards, the controller requests the switch’s capabilities and supported OpenFlow version through a FEATURES_REQUEST message. The switch then supplies the requested information using a FEATURES_REPLY message. Once all these steps have been completed successfully, which are outlined in Fig. B.3, the connection is established.

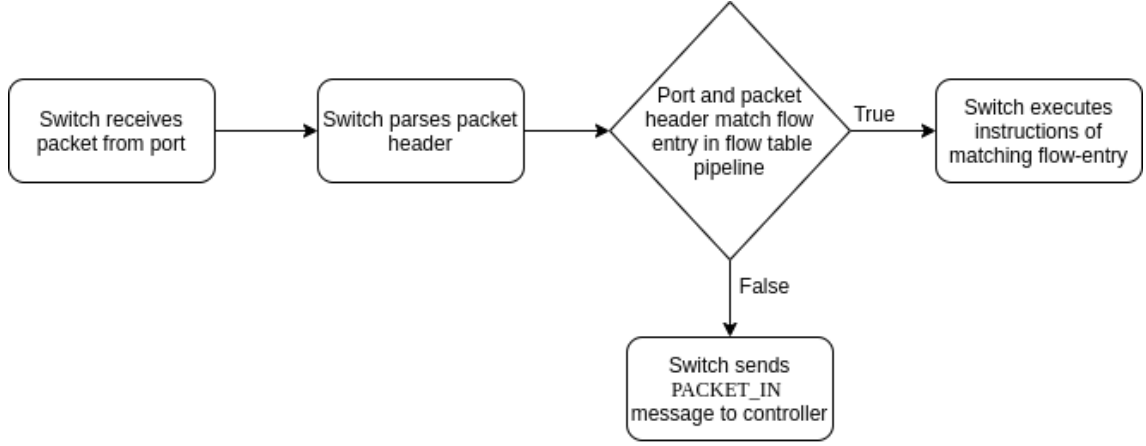


Figure B.4: *The typical steps of packet processing performed by an OpenFlow switch.*

The packet-matching process is of particular interest here, which is outlined in Fig. B.4. As previously mentioned, when a switch receives a packet and performs a lookup on its flow tables and no flow entry matches the packet, the switch typically contacts the controller using a `PACKET_IN` message that is sent through the secure channel. The controller, upon receiving this message, computes a path from the packet’s source to its destination and then installs flow entries to the flow tables of the switches involved in the path. The flow entries are added using `FLOW_MOD` messages. At the same time, the controller instructs the switch that originally sent the `PACKET_IN` message to forward the packet to one of its ports, according to the computed path using a `PACKET_OUT` message. The rest of the switches along the path, having hopefully received the `FLOW_MOD` messages in time, can forward the packet directly, without needing a `PACKET_OUT` message from the controller. This process, often referred to as the flow-setup process, is illustrated in Fig. B.5.

This flow-setup process allows for centralized control over control plane-related tasks, which provides great flexibility in the operation of modern networks. In particular, the reactive aspect of the flow-setup process provides flexibility in adapting to dynamic conditions, facilitating various tasks [32], such as routing, traffic shaping, especially in virtualization environments [33], such as data centers [34], although the fundamental ideas behind SDN and OpenFlow are quickly expanding to other areas of networking, including the Internet of Things [12], Next-Generation Networks [13], Vehicular Networks [14], etc.

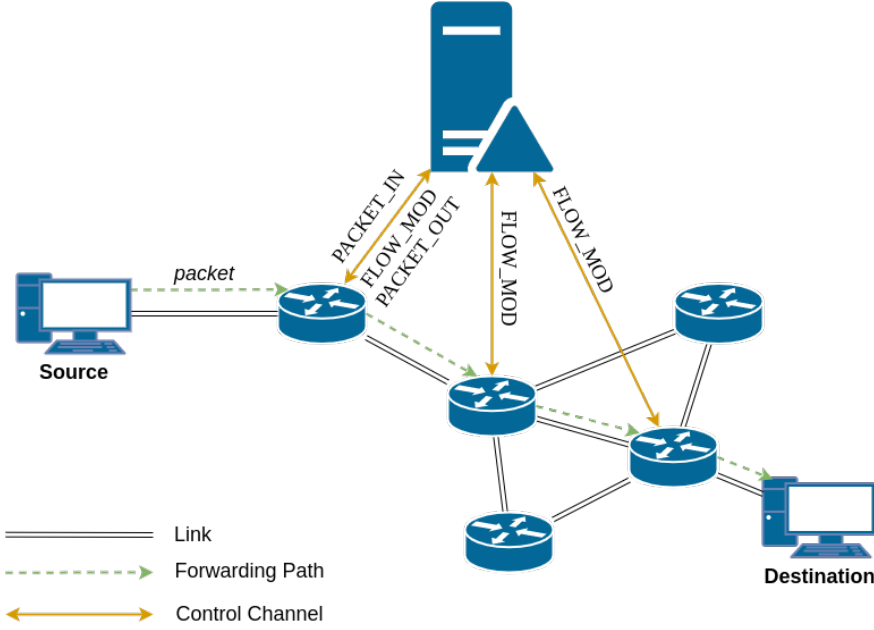


Figure B.5: Example of an The flow-setup process in an OpenFlow switch.

However, the centralized aspect of the flow-setup process in OpenFlow results in a significant number of control packets being generated and sent towards a single controller. In fact, the flow-setup process is often considered one of the primary issues of SDNs [35], as a single controller may become overwhelmed with the number of `PACKET_IN` messages generated in large-scale networks, causing a significant increase in latency until the `PACKET_OUT` and `FLOW_MOD` responses are delivered. This can hinder the network's performance, both in terms of the Quality of Service (QoS) that can be provided and Quality of Experience (QoE) perceived by the end users. Additionally, concerns are often raised about the reliability and security of a controller in networks utilizing OpenFlow.

B.3 Physically Distributed–Logically Centralized Control Plane

To address the aforementioned issues arising from the use of a single controller, researchers quickly turned to distributing the control plane over multiple controllers, without losing the benefits of centralized control. In this way, the load of each

B.3. *Physically Distributed–Logically Centralized Control Plane* 19

controller is constrained to its domain, i.e., the set of switches that are assigned to it. However, new issues arise when considering distributing the control plane.

Perhaps, the main issue that needs to be addressed, before a distributed approach is even possible, is how to provide a logically centralized global view of the network, when each controller maintains locally only a small part of it. After all, routing traffic that spans multiple domains would not be possible, unless all controllers along the way had adequate information to direct the traffic to its destination. Thus, to provide a global view of the network, synchronization is typically utilized, with controllers sharing their local views with each other to eventually reach a consensus on a global view.

In fact, controllers may need to communicate with each other for various reasons, besides synchronization. This communication is defined by the east-westbound interface (E-WBI), which is usually dependent on the controller’s implementation. Various issues arise from this inter-controller communication, as control traffic should be kept to a minimum, while providing reliability, a consistent global view, and load balancing [21].

Another important aspect regarding the E-WBI and the architecture of a distributed control plane is whether controllers communicate in a flat or hierarchical fashion. These different types of architectures are depicted in Fig. B.6. When a flat structure is imposed, controllers are considered “equals” and they all partake in synchronization and other control plane tasks. In a hierarchical structure, on the other hand, controllers have varying levels of responsibilities depending on their position in the hierarchy.

A flat structure is perhaps easier to conceptualize, as the distributed controllers operate much like a single centralized controller, so long as they have a global view of the network. In order to acquire this global view of the network, some kind of distributed synchronization is typically used to provide eventual, but scalable, consistency, which can be categorized [35] into: (i) distributed data stores (e.g., ONOS [19], Orion [1]); (ii) distributed hash-tables (e.g., Onix [18]); and (iii) distributed publish-subscribe messaging (e.g., HyperFlow [5]).

In a hierarchical structure, though, the layers of the hierarchy and the differences from layer to layer are largely dependent on the implementation. Generally speaking, the lower-layer controllers have a smaller view of the network, making mostly local decisions, which are typically assumed to represent the majority of the load, whereas

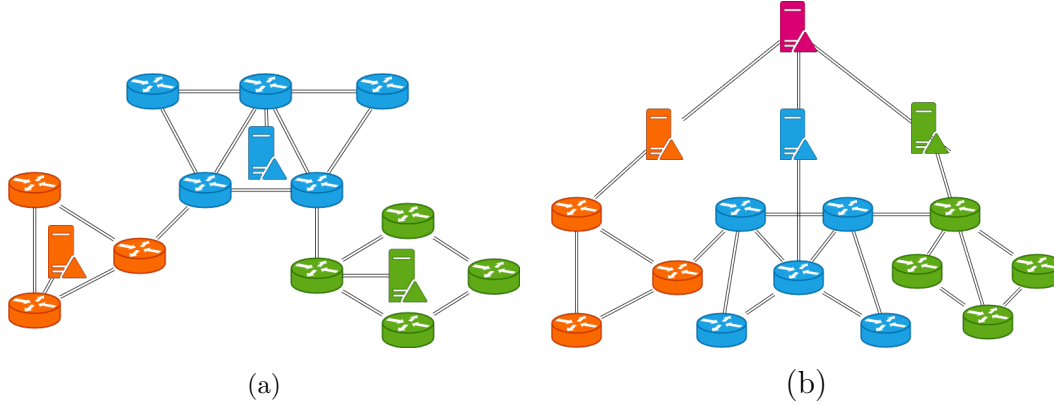


Figure B.6: *The two types of distributed controller architectures: flat(a) and hierarchical (b). The switches are assigned to controllers of the same color.*

the upper-layer controllers make the less common, global decisions. In this case, distributed synchronization is not necessary, as the lower-layer controllers are not necessarily interconnected and they can handle requests within their domain utilizing merely their local view. But, they do need to provide the required information to the higher-layer controllers, so they can make the global decisions.

B.4 The Controller Placement Problem

Overall, the success and necessity of distributed control planes is hard to argue against, with controllers, such as Orion, providing improved performance and easier management at massive scales [1]. However, there are still challenges that arise in the distributed context, which need to be addressed. These challenges generally revolve around the main themes discussed so far, as is also argued by Hu et al. [21], and they include: (i) scalability; (ii) reliability; and (iii) load balancing.

Perhaps the first issue that arises when considering the deployment of multiple controllers concerns their placement. The issue of appropriately placing controllers in the network, termed the CPP and initially formulated by Heller et al. [20], is fundamental in the distributed setting and incorporates three primary sub-problems: (i) determining the number of controllers to deploy; (ii) placing these controllers in the network; and (iii) assigning switches to each controller.

Solutions to each of these three sub-problems can significantly affect the net-

work's performance, either negatively, if the solutions are inadequate, or positively, if they are appropriate for the considered network. For instance, consider two ends of the spectrum for the first sub-problem. In one extreme, one may deploy only a single controller in the network, which would be insufficient at large scale, or one may deploy a controller for every switch in the network, which would significantly increase costs, communication overhead, and complexity. The answer to this, much like all three sub-problems, depends on many factors: traffic volume, the network's topology, requirements as to the network's performance (e.g., response times), availability of computational resources, and so on. For instance, Heller et al. [20] concluded that for many Internet2 topologies, a single controller suffices to provide reasonable response times.

Furthermore, finding an optimal solution to the CPP (or even any of its subproblems [20, 22, 23, 24]) is NP-hard and different solutions can be provided, depending on the objective being optimized. The CPP is interesting in that it has been studied to address all three aforementioned challenges (scalability, reliability, load balancing [21]) of the distributed SDN control-plane [25]. For instance, providing more controllers and placing them strategically can provide better reliability, in case of controller failure. Similarly, the switch-to-controller assignment is particularly important for load-balancing, while the controllers being placed closed to the switches reduces overhead and improves scalability. Through these examples, it should be clear that all three sub-problems, along with their solutions, are closely intertwined and can affect different aspects of the network's performance.

While many variants of the CPP exist, which differ in the set of imposed constraints and set out objectives, the primary formulation aims to minimize the total switch-to-controller latency, so as to minimize the total time to complete the flow-setup process, i.e., the flow-setup time. The most typical constraints include limiting the controllers' capacity in terms of the number of requests they can process, and enforcing an upper bound on the tolerable latency between any switch and its assigned controller. Furthermore, it is common to assume that controllers are co-located with a switch in the network, with controllers running over VMs that are hosted on a physical machine that is connected to that switch. Thus, the set of potential locations for a controller is the set or a certain subset of the switches. The latter is preferred in cases where some locations cannot support the deployment of an SDN controller, either because there are not enough resources at that location or because

the switch at that location does not support OpenFlow or similar SDN technologies.

B.4.1 Dynamic Variant

Although the classic formulation of the CPP has been widely studied, it fails to capture some of the inherently dynamic conditions that are present in real-world networks. The primary example of this is traffic, which fluctuates over time, exhibiting behavior like the time-of-day and day-of-week effects [36]. These dynamic traffic patterns may cause an initially optimal controller placement to become sub-optimal over time.

The problem of placing controllers under dynamic traffic was originally proposed and addressed by Bari et al. [22] and termed the Dynamic Controller Placement Problem. In particular, they propose a system that continually monitors the network's traffic patterns and periodically adjusts the controllers' placement. Additionally, they dynamically adjust the number of controllers to maintain low flow-setup times, regardless of the traffic volume.

It is worth noting that despite all the literature revolving around the CPP, not many works address the dynamic setting, perhaps because it is inherently more difficult. On the other hand, it could be argued that any static solution for the CPP can be executed periodically, integrated into an architecture that continuously monitors and collects traffic statistics, similar to the one proposed by Bari et al. [22], providing an adequate solution to the dynamic variant. Nevertheless, the formulation of this dynamic variant of the CPP highlights the need for continuous adjustment of the allocation of controllers.

However, the work of Bari et al. [22] is not without its shortcomings. Fundamentally, this approach assumes that the traffic patterns observed in a given time interval can be used to predict the traffic in the following time interval. As such, it is assumed that traffic patterns between intervals must be highly correlated for the provided solution to be effective, which may not necessarily be the case. It is also perhaps worth noting that the algorithms proposed by Bari et al. [22] do not consider how different a new controller allocation may be to the current one. It can be argued, that making small incremental adjustments to the current allocation may be more advantageous than coming up with an entirely new one, as the latter is likely to result in more data being transferred to accommodate the transfer and

synchronization of state among the newly deployed controllers.

Chapter C

Formulation of the Controller Placement Problem

HAVING provided an overview of SDN and the relevant technologies surrounding it, along with a brief overview of the CPP and its key properties, a more thorough and analytical formulation follows in order to lay the foundation for the description of the proposed migration-based approach towards solving the CPP. At first, the assumptions regarding the control plane are laid out to provide a clear view of the system's capabilities and limitations. Afterwards, the considered system is analytically modeled, introducing the necessary notation for the formulation of the CPP. Finally, the cost that each switch contributes towards the final solution is formulated and a few useful results are derived to aid in the design of the proposed approach.

C.1 Control Plane Assumptions

The considered network employs SDN technologies across all switches. The switches are assigned to a single controller at a time, which communicate in an in-band fashion (the difference between in-band and out-of-band communication is illustrated in Fig. C.1). In particular, switches utilize bootstrapping, using their traditional distributed protocols to initially establish communication and setup a connection with their assigned controllers.

The switch to controller communication follows the guidelines of the OpenFlow

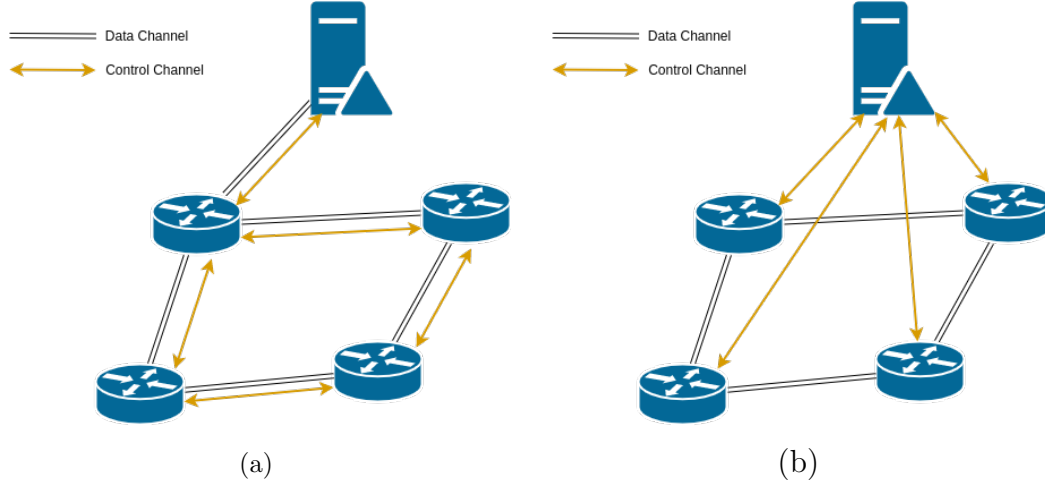


Figure C.1: *In-band (a) and out-of-band (b) control communication. In the former case, control communication takes place over the same links that data is forwarded through, requiring bootstrapping for the initial connection establishment. In the latter case, the switches communicate with the controller through different links, forming a separate network.*

protocol (v.1.3). Each switch is assigned to a single controller. However, a switch may migrate to a different controller under certain conditions. The switch migration takes place in the following manner: the switch initiates a connection with its newly assigned controller using a different TCP socket than the one being used for the connection with its previous controller. Once the OpenFlow connection is established, the switch closes the TCP socket to its previous controller, which in turn deletes any stored information associated with that switch. It should be noted that more complicated mechanisms exist, guaranteeing that no errors will occur during migration [37], but the described mechanism suffices and is selected due to its simplicity.

The control plane of the considered network is distributed and its structure is flat, with controllers communicating with each other through the E-WBI. Thus, the controllers need to assemble a global view from their individual local views through synchronization, but the particular method used to achieve this and its impact on controller placement is outside the scope of this work.

The application layer of the controllers in the network comprise several applications to perform the necessary control plane operations, as outlined in the sequel.

Topology Discovery In order to discover the topology, this application periodically sends Link-Layer Discovery Protocol (LLDP) messages to their corresponding switches. Through these messages, each controller maintains a local view of the network, which can then be used to form the global view through synchronization.

Routing This application handles `PACKET_IN` messages that arrive to the controller by calculating the shortest path between the source and destination of the `PACKET_IN` message and responding with a `PACKET_OUT` message to the switch and installing the appropriate flow entries on all involved switches through `FLOW_MOD` messages.

Statistics Utilizing this application, the controllers periodically collect statistics from their corresponding switches regarding port-level usage to measure control and data traffic.

ARP Caching This application is utilized to increase performance, as it enables controllers to cache IP-to-address information and respond to Address Resolution Protocol (ARP) requests from the switches, reducing the unnecessary propagation of control messages across domains.

Due to the switches being partitioned into multiple domains, the flow-setup process becomes more complex, as illustrated in Fig. C.2. This happens because of the fact that, according to OpenFlow, each switch may only install flow entries received from the controller it has been assigned to. Thus, more than one controllers need to participate in the installation of flow entries for flows (traffic associated with a certain source-destination pair) that span multiple domains. In particular, the first switch of each domain that the flow crosses sends a `PACKET_IN` message to its assigned controller.

The controllers are assumed to be operating over a VM hosted on a physical machine. It is assumed that all switch locations can host a controller. As a VM, the controller can migrate from one physical machine to another. By doing so, all open connections (TCP/OpenFlow) are broken and the switches assigned to a migrating controller need to reconnect to the controller after migration, repeating the OpenFlow handshake.

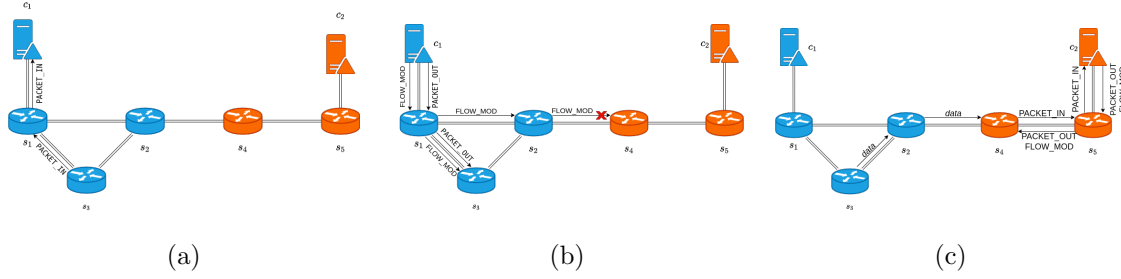


Figure C.2: The flow-setup process for a flow from s_3 to s_5 : (a) Initial **PACKET_IN** message; (b) **PACKET_OUT** and **FLOW_MOD** messages within domain; (c) process repeats for next domain.

C.2 System Model

The considered SDN network is modeled as a graph $G = (S, L)$, comprising a set of switches S and the links between them L . Specific locations in the network $F \subseteq S$ can be utilized to place controllers, which may be enabled or disabled dynamically, depending on the network's conditions. The number of enabled controllers in the network at any given time is assumed to be constant and equal to k . Each switch is assigned to a single (enabled) controller, which is responsible for handling the switch's routing decisions. Let $A : S \rightarrow F$ denote the function that returns the controller f that switch s is assigned to, e.g., $A_{s_3} = c_1$ in Fig. C.2. The controller assigned to switch i is denoted by A_i . The set of switches assigned to a controller $f \in F$ is called domain $D_f = \{i \in S \mid A_i = f\}$. It is assumed that domains are connected, i.e., there exists a path between any two switches of a domain that stays entirely within the domain. Let $\mathcal{A} = (F, A)$ denote a given *allocation* of controllers, determined by the set of active controllers F and the assignments of switches to these controllers A .

Each switch serves a number of different hosts through a subset of its ports, which generate traffic in the network. The traffic generated by the hosts has to pass through the switch, so it is naturally aggregated at the switch. In this way, the network's traffic forms an $|S| \times |S|$ matrix: τ^{ij} , such that the traffic flowing from switch $i \in S$ to switch $j \in S$, with $(i \neq j)$ is denoted by τ^{ij} . The diagonal of this matrix is set to $\tau_{ii} = 0$ for all $i \in S$. In particular, τ represents the number of new flows (resulting in the delivery of a **PACKET_IN** message) originating from switch i that are headed towards switch j . It should be noted that the traffic considered here

consists entirely of `PACKET_IN` and `PACKET_OUT` messages, not of data packets.

A `PACKET_IN` message is sent through the shortest path to the controller assigned to a switch, when the forwarding table of a switch does not contain an entry that matches the flow that arrives at that switch. The controller, upon receiving the `PACKET_IN` messages, computes the shortest path P^{ij} starting at switch i and ending at the switch of the flow's destination j , sending `FLOW_MOD` messages to switches along the way. However, a switch will discard any messages that originate from a controller that is different from the one it is currently assigned to. In this case, where a path traverses multiple domains, the first switch of each domain encountered along the computed path will have to send another `PACKET_IN` message to its assigned controller until the original data packet arrives at its destination. This process is illustrated in Fig. C.2.

The distance between switches i and j is denoted by $d(i, j) = d(j, i)$, which is measured in terms of hops (although the formulation would remain identical for time or distance). Since a given controller f is considered to reside in the same location as a switch i in the network, the distance between the controller f and the switch i it co-resides with is $d(f, i) = 0$. By extension, the distance between controller f , co-residing with switch i , and any other switch j is $d(f, j) = d(i, j)$.

The set of switches that are adjacent to any given switch $i \in S$ is denoted by $N(i)$. The concept of adjacency is also extended to apply to a domain D_f by considering the set of switches that are one hop away from any switch in D_f , but do not belong in D_f , more formally defined as

$$\mathcal{N}(D_f) = \bigcup_{i \in D_f} N(i) \setminus D_f. \quad (\text{C.1})$$

C.3 Cost Formulation

The rest of the analysis focuses on formulating the path setup process as a facility location problem. Consider a flow originating from switch i and headed towards switch j . The shortest path between these switches is denoted by $P^{ij} = \{i, a, \dots, b, j\}$. The cost of the flow-setup process can be broken down into three separate costs: (i) `PACKET_IN` cost; (ii) `PACKET_OUT` cost; and (iii) `FLOW_MOD` cost. However, note that the `FLOW_MOD` cost is largely ignored for the rest of this work.

C.3.1 Switch Cost

The cost is defined in terms of the total distance `PACKET_IN` and `PACKET_OUT` messages traverse, indicating the control overhead and flow-setup time for the network. Each switch in the network contributes towards the total cost, depending on the number of flows for which the switch generates `PACKET_IN` messages as well as the distance from its assigned controller. To simplify notation for the former, an additional variable is introduced, I_s^{ij} , that indicates whether switch s is the first in its domain to receive a data packet along the path P^{ij} ; i.e., $I_s^{ij} = 1$, if switch s generates a `PACKET_IN` message for the flow moving along P^{ij} , and $I_s^{ij} = 0$, otherwise. It should be noted that $I_j^{ij} = 0$, even if it is the first of its domain along P^{ij} , as j is the flow's destination and the switch will not require any forwarding rules from its controller.

Using this notation, the cost of switch s is formulated as:

$$C_s^{\mathcal{A}} = 2d(s, A_s) \sum_{i \in S} \sum_{j \in S} \tau^{ij} I_s^{ij}. \quad (\text{C.2})$$

Constant 2 from (C.2) corresponds to the round trip that takes place for a `PACKET_IN` and its corresponding `PACKET_OUT` messages, while the double summation indicates the frequency of generating `PACKET_IN` messages. Similarly, cost C_f for the domain of controller f may be expressed by simply summing the costs of all switches in its domain, as in:

$$\begin{aligned} C_f^{\mathcal{A}} &= \sum_{s \in D_f} C_s^{\mathcal{A}} = 2 \sum_{s \in D_f} \sum_{i \in S} \sum_{j \in S} \tau^{ij} d(s, f) I_s^{ij} \\ &= 2 \sum_{s \in S} d(s, f) \sum_{i \in S} \sum_{j \in S} \tau^{ij} I_s^{ij}. \end{aligned} \quad (\text{C.3})$$

Clearly, the total cost of a given allocation $\mathcal{A}$ will be the sum of the costs of all switches:

$$C^{\mathcal{A}} = \sum_{s \in S} C_s^{\mathcal{A}} = 2 \sum_{i \in S} \sum_{j \in S} \tau^{ij} \sum_{s \in S} d(s, A_s) I_s^{ij}. \quad (\text{C.4})$$

The cost given by (C.4) is the objective to be minimized. Minimizing this cost is equivalent to minimizing the average switch-to-controller latency, weighted by the frequency that `PACKET_IN` messages are sent to the controller.

Integer Linear Programming Formulation

In the sequel, the objective and constraints described previously are formulated as an Integer Linear Program, as is commonly done in the literature [25].

To formulate this, the necessary decision variables are defined. Let y_{ij} denote the binary decision variable that corresponds to switch $j \in S$ being assigned to the controller co-residing with switch $i \in S$. Furthermore, let x_i denote the binary decision variable that indicates whether controller co-residing with switch $i \in S$ is enabled. Thus, the Integer Linear Program is formulated as follows:

$$\begin{aligned}
& \text{minimize:} && 2 \sum_{i \in S} \sum_{j \in S} \tau^{ij} \sum_{s \in S} \sum_{f \in S} d(s, f) \cdot y_{fs} \cdot I_s^{ij} \\
& \text{subject to:} && \sum_{j \in F} y_{ij} = 1, \quad \forall i \in S \\
& && \sum_{j \in F} y_{ij} \leq x_i, \quad \forall i \in S \\
& && \sum_{i \in F} x_i = k \\
& && x_i, y_{ij} \in \{0, 1\}, \quad \forall i, j \in S
\end{aligned}$$

In contrast with some other approaches found in the literature [25], no constraints regarding the controllers' capacity or the maximum allowed latency between any switch and its assigned controller are imposed. Instead, the first constraint indicates that a switch must be assigned to exactly one controller, the second constraint that a switch can only be assigned to an enabled controller, the third constraint that exactly k controllers must be enabled, and the fourth constraint that variables x, y are binary.

C.3.2 Switch Migration Costs

Consider switch s migrating from the domain of controller f to an adjacent domain of controller f' . However, by migrating the switch to a different domain, the costs of other switches may be affected as well, since they may have to generate some of the `PACKET_IN` messages that s would have otherwise generated. Due to this, to determine whether a switch migration is worthwhile, the cost of all affected switches needs to be considered.

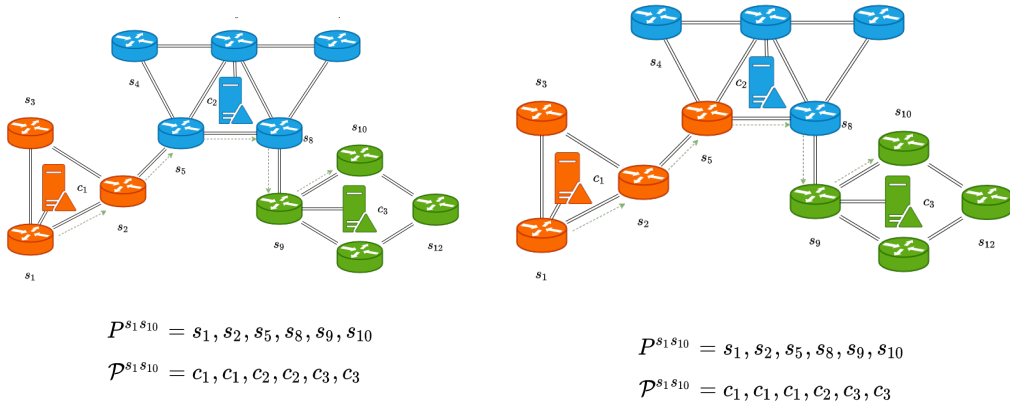


Figure C.3: The case of switch migration in a network with multiple controllers: switch s_5 migrates from controller c_1 (a) to c_2 (b).

To better understand this, consider a path P^{ij} going through switch s , i.e., $P^{ij} = i, \dots, x, s, y, \dots, j$. This sequence of switches may be mapped to a sequence of controllers $\mathcal{P}^{ij}$ through A , as in: $\mathcal{P}^{ij} = A_i, \dots, A_x, A_s, A_y, \dots, A_j$. The switches (and the controllers they are assigned to) that will be involved in the flow-setup process can be determined in this sequence, by identifying the first appearance of each controller in $\mathcal{P}^{ij}$. For example, consider the network shown in Fig. C.3, where s_5 migrates from c_1 to c_3 . For a flow from s_1 to s_{10} , the switches that generate **PACKET_IN** messages before the switch migration will be s_1, s_5, s_9 , as indicated by the first time each controller appears in $\mathcal{P}^{s_1 s_{10}}$. However, after migration, switch s_5 no longer generates **PACKET_IN** messages for the same flow, as flow-entries should have been installed from the **PACKET_IN** message of s_1 . Instead, s_8 will now generate the **PACKET_IN** messages for the domain of c_2 , as it is now the first switch in its domain, taking the place of s_5 .

Notice, however, that in tree topologies, any given path may cross a domain only once. More formally, in a sequence $\mathcal{P}^{ij}$, there cannot be a subsequence of the form $f, \dots, g, \dots, f$, because if it did, then D_f would not be connected. Therefore, A'_s cannot take a value that appears after A_y in $\mathcal{P}^{ij}$ (other than A_y).

Due to this fact, an important result is derived for trees: if switch s migrates, then only the cost of switches in $N(s)$ may be affected. To better understand this, consider an instance, where $A_x = A_s = f$ and $A_y = g$, then $\mathcal{P}^{ij} = A_i, \dots, f, f, g, \dots, A_j$. In this case, s can migrate to either g or any other controller, say h (so long as h

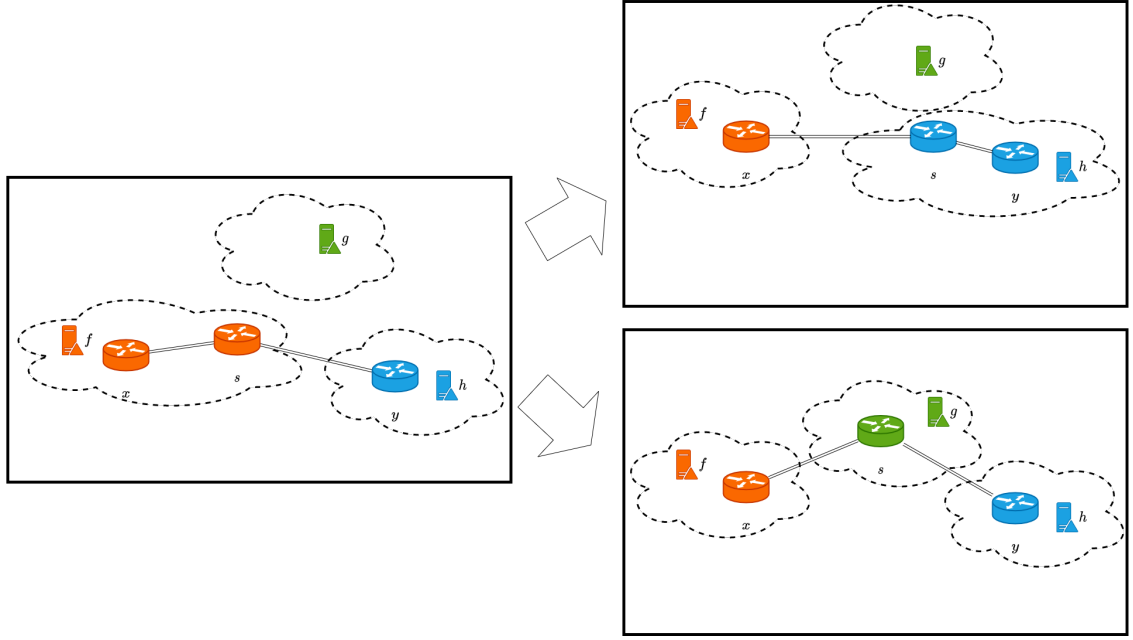


Figure C.4: The two potential alternatives for migrating switch s from controller f to either g or h .

does not appear in the subsequences $A_i, \dots, f$ and $g, \dots, A_j$). These two cases are illustrated in Fig. C.4. If s were to migrate to g , then $\mathcal{P}^{ij} = A_i, \dots, f, g, g, \dots, A_j$ and s would be the first switch of its domain, instead of y ; thus, s would generate more PACKET_IN messages, potentially resulting in $C_s^{A'} > C_s^A$ and $C_j^{A'} < C_j^A$. On the other hand, if s were to migrate to h , then an additional intermediate domain would be introduced in the path, potentially resulting in $C_s^{A'} > C_s^A$ and no reductions for the other switches.

Chapter D

Migration-Based Approach to Controller Placement

THE proposed approach addresses the CPP through an iterative algorithm that employs migrations. Two kinds of migration policies are proposed: (i) the controller migration policy, where controllers move from one physical machine to another in the network; and (ii) the switch migration policy, where the assignment of switches to controllers is altered. This approach is thoroughly described in this Chapter, along with a few analytical results that capture some aspects of its performance.

D.1 Description

In the CPP, since the controller is typically assumed to be operating over a VM or similar virtualization technology, the act of placing a controller in the network is fundamentally tied to the logistics of operating a VM. Furthermore, under realistic conditions, it would be necessary to require the migration of the controller VM to another physical machine, eventually. This situation is sure to come up naturally, as hardware failure is to be expected, especially in data center networks, where VM migrations are necessary to provide fault-tolerance, in case one physical machine goes down. Furthermore, addressing the CPP in a realistic network scenario, where conditions are sure to fluctuate, would necessarily involve reallocating the controllers [22], which likely would entail their migration, as well.

Due to these reasons, there is sufficient motivation to consider migrations as the basic operation through which controllers may be placed in the network. Furthermore, previous research [29, 30, 38], focusing on service placement in autonomic networks, has showcased the effectiveness of intelligent migration-based approaches in optimizing the allocation of services to respond to client demands. These approaches can be utilized to some extent in the context of SDN to address the CPP.

Following this idea, the proposed approach employs two kinds of migration policies to alter the controllers' location along with their domains. These policies, utilizing topological information and statistics collected from the forwarding elements, allows for the minimization of the CPP cost.

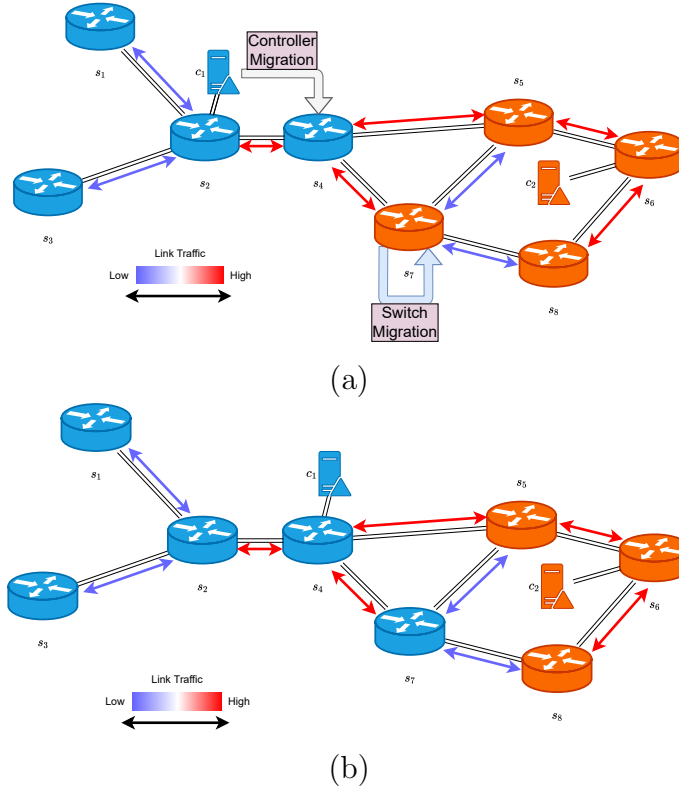


Figure D.1: Example of the two kinds of migrations that the proposed approach utilizes to alter the state of the network from (a) to (b).

An illustration of this approach is illustrated in Fig. D.1, visualizing the necessary components in the network along with migration operations that can take place. The depicted network is partitioned into two domains governed by controllers c_1 and c_2 .

The color associated with each link indicates the traffic volume that the link carries. In the initial configuration, the traffic between s_2 and s_4 is significantly higher than the traffic carried by the other links of s_2 . Thus, migrating the controller to the location of s_4 can result in a cost reduction, as the traffic would not have to travel as far. Similarly, the traffic that s_7 is involved with is primarily exchanged with s_4 , crossing to a different domain. To address this, the switch can join this domain by migrating to controller c_1 . The final state of the network after these two operations take place is depicted in the bottom part of the figure.

Fundamentally, the two kinds of migrations are largely interlinked, as the placement of the controllers affects the switch assignment and vice versa: a particular switch assignment may lead the controller to migrate to a more efficient location, which in turn may lead to a better switch assignment. Therefore, a balance needs to be stricken between these two operations. In the case of the proposed approach, a controller migration occurs first, immediately followed by a round of switch migrations. Variations of this approach may work just as well, e.g., a round of switch migrations followed by controller migrations. Also depending on the particular policy that is employed, performing more switch migration rounds may be beneficial under certain conditions. The migration policies proposed and studied in this Thesis are described in the sequel.

D.2 Controller Migration Policy

The controller migration policy describes the rules and conditions under which a controller decides to migrate to a different physical machine. This policy's main goal should be to bring the controller to a better location in the network, so as to minimize the CPP cost given by (C.4). It should be noted that this can be accomplished somewhat indirectly, as a larger cost reduction may be achieved by accommodating the switch migration policy, rather than greedily picking the best position purely for the controller.

In this case, the controller migration policy considered is an adapted version of the one presented in [29], which was initially proposed in the context of automatic networks. This particular policy has the benefit of requiring purely local information, making it practical even at very large scales. Furthermore, it has been thoroughly studied analytically in the context of service placement, providing per-

formance guarantees under specific conditions. However, service placement, which is typically modeled as a k -median problem, differs from the CPP. Thus, this migration policy is reexamined in the context of the CPP in SDN to determine whether it is suitable in this environment.

First of all, a more formal description of the migration policy is provided. According to this policy, each controller f monitors the traffic arriving through each link from its adjacent switches $j \in N(f)$ over an arbitrarily set time interval. In particular, the controller keeps track of the number of control messages (`PACKET_IN` and/or `PACKET_OUT`) that arrive through each link jf , denoted by Λ^{jf} . This traffic is equivalent to the sum of control packets that are sent to the controller from the switches in its domain, which may potentially originate from a different domain.

Definition 1 (Controller Migration Policy). *According to the controller migration policy, controller f , co-residing with switch i , migrates to adjacent switch j , if $\sum_{s \in N(i) \setminus \{j\}} \Lambda^{sf} < \Lambda^{jf}$.*

Assuming equal-weight links, this migration policy is proven to eventually lead to an allocation that is superior to the initial one in the context of service placement [29]. Intuitively, the controller decides to migrate to a neighboring switch if the control traffic originating from that switch is higher than the aggregate traffic originating from the rest of the controller's neighbors. In this way, the controller follows a trajectory of monotonically decreasing cost, until it eventually stops, if one of its neighbors does not satisfy the aforementioned traffic condition. The rest of this Section focuses on translating some of the original policy's properties to the CPP.

Proposition 1. *Minimizing the cost C_f^A of a single controller, assuming its domain is static, is an instance of the 1-median problem.*

Proof. The actual cost that each switch s causes towards controller f may be expressed as:

$$\mathcal{T}_s = \sum_{i \in S} \sum_{j \in S} \tau_{ij} I_s^{ij}.$$

Thus, the controller cost C_f^A may be reformulated as follows:

$$C_f^A = 2 \sum_{i \in D_f} d(f, i) \mathcal{T}_i. \quad (\text{D.1})$$

This cost is twice the cost of the 1-median problem, which implies that they share a global minimum. $\square$

Due to Proposition 1, assuming static domains, the properties of the employed controller migration policy [29] can still apply in the case of SDNs, as shown next.

Proposition 2. *Consider a network G with a single controller f and an initial allocation $\mathcal{A}$. If the topology of G is a tree and controller f migrates under the controller migration policy, then f will arrive at the optimal allocation $\mathcal{A}'$.*

Proof. Since there is only one controller, its domain is inherently the set of all switches, which is static. In light of Proposition 1, the proof follows from Theorem 3 in [29]. $\square$

Proposition 3. *From any initial allocation $\mathcal{A}$, a migration of all controllers to allocation $\mathcal{A}'$, according to the controller migration policy, will yield a lower cost, i.e., $C^{\mathcal{A}'} < C^{\mathcal{A}}$.*

Proof. If all k controllers migrate under *Policy S* from $\mathcal{A}$ to $\mathcal{A}'$, then, according to Proposition 2 cost $C_f^{\mathcal{A}'} < C_f^{\mathcal{A}}$. Since each controller f operates independently from the rest and the domains remain static, then $C_F^{\mathcal{A}'} < C_F^{\mathcal{A}}$. $\square$

Based on Propositions 1, 2, and 3, it is guaranteed that, regardless of topology and the number of controllers k , the controller migration policy will yield a better controller placement. In the case of tree topologies, the controller placement will be optimal. However, to guarantee this the domains are assumed to remain static. Nonetheless, it is shown next that this constraint does not significantly affect the effectiveness of the approach.

D.3 Switch Migration

While the controller migration policy suffices to reduce costs across the network, if the domains remain static, further cost reduction can potentially be achieved by migrating a switch from one controller to another under certain conditions. In contrast to the k -median problem, where the mapping of nodes to facilities is accomplished based on distance and each node is assigned to its closest facility, in SDN the cost of any switch is dependent on other switches' assignment along a path, as given by

(C.2). Eventually, the migration of a switch has the potential of impacting other switches, as shown in Section C.3.2.

Due to these reasons, a more sophisticated switch migration policy may be more suitable. To study the impact of the switch migration policy in the CPP, two distinct switch migration policies are proposed. Much like the controller migration policy, the switch migration policy is concerned with describing the conditions under which a switch may migrate from one controller to another. Both switch migration policies are employed in a similar manner: each time controller f migrates, it considers expanding its domain D_f to a subset of the domain's adjacent switches $\mathcal{N}(D_f)$, according to the policy's rules.

Definition 2 (Distance-Based Switch Migration Policy). *According to the distance-based policy, switch i , currently assigned to controller f , migrates to controller g , if $d(g, i) < d(f, i)$.*

The distance-based switch migration policy is essentially identical to the way nodes are assigned to facilities in the k -median problem. This policy only considers the distance between switches and their controller, which is perhaps the most significant factor contributing to the cost of that switch, as given by (C.2). However, it does not take into account the traffic patterns formed between different domains. At the same time, the distance-based policy provides a useful guarantee: all domains are certain to remain connected.

Definition 3 (Cost-Based Switch Migration Policy). *According to the cost-based policy, switch i , currently assigned to controller f (allocation $\mathcal{A}$), migrates to controller g (allocation $\mathcal{A}'$), if $C^{\mathcal{A}'} < C^{\mathcal{A}}$.*

The cost-based policy guarantees that a migration will only occur if it results in reduction of the total cost. Thus, similarly to the controller migration policy, repeated applications of the cost-based policy results in the cost following a monotonically decreasing trajectory. In conjunction with the controller migration policy guaranteeing a cost reduction, alternating between applications of the controller and the cost-based switch migration policy guarantee a total cost reduction. In other words, from an initial allocation, an application of the controller migration policy across all controllers reduces total cost, then the domains are altered so as to drop the cost again. Then, another controller migration guarantees another cost reduction, and so on, until the cost cannot be reduced further through some migration.

Unlike the distance-based policy, though, the domains are not guaranteed to be connected at first glance. However, if a switch is left “stranded” in a foreign domain, all the traffic passing through it will result in additional `PACKET_IN` messages, which would result in a significant cost increase. Furthermore, since each controller f only considers expanding its domain to its neighborhood $\mathcal{N}(D_f)$, domains can only become disconnected in unexpected cases, where controllers do not coordinate when migrating switches. Thus, despite no analytical result being derived to guarantee it, domains are expected to remain connected by applying the cost-based policy, unless extreme scenarios are considered.

It is worth pointing out that the cost-based policy guarantees a reduction of the total cost in each step. Thus, given static traffic, the migration-based approach converges within a finite number of steps, since it follows a monotonically decreasing trajectory. Note that with each controller migration, its domain may be expanded to its directly adjacent switches, although it is possible that switches further away could result in additional cost reduction. To address this, it is possible to repeatedly attempt to expand the domain of each controller, until no switch migrations may take place. However, calculating the total cost $C^{\mathcal{A}}$ of a given placement is a costly process utilizing global information; thus, they should be used sparingly.

Another interesting aspect of the cost-based policy is the fact that, unlike the distance-based policy, there may be benefit in repeatedly applying it in successive rounds. After the first round, the initial domain may be expanded to nodes that are 1 hop away. Then, the ensuing applications of the policy can expand the domain further, which may be useful in providing more options, with respect to the switches that the domain may be expanded to. This is not the case for the distance-based policy, as repeated applications of this policy will not yield more options, unless the controller migrates to a different location. Nonetheless, a parameter m can be introduced to indicate the number of successive switch migration rounds, depending on the considered policy. Setting $m = 0$ would be equivalent to assuming that domains always remain static, which is not advisable and, thus, not considered here. Additionally, the case of $m = 1$ is primarily considered here.

Algorithm 1 describes the proposed cost-based migration policy. There are a few details that affect the algorithm’s performance and effectiveness that should be pointed out. Firstly, if a better mapping is found in the nested for-loop, the current allocation $\mathcal{A}$ is not updated with the new switch. This choice has been made

to reduce the policy's computational complexity, as the total cost C^A would then have to be recomputed and updated for all other switches, i.e., the cost calculation would have to be made within the nested for-loop. This design decision makes the policy somewhat greedy, as it does not consider all possible options, but it simply takes any option that leads to an improvement over the initial assignment, even if this decision is somewhat ill-informed. Secondly, the total cost calculation C^A that occurs when a switch migrates can be optimized depending on the considered topology. For instance, in tree topologies, a switch migration will only impact the cost of neighboring switches, so only these costs need be updated.

Algorithm 1 Cost-Based Switch Migration Policy

```

 $\mathcal{C}_s$  : holds the best cost-controller tuple  $(c, f)$  found for each switch
// Initialize  $\mathcal{C}_s$  with current cost-controller tuples
 $C^A \leftarrow$  compute total cost according to (C.4)
for  $s \in S$  do
     $\mathcal{C}_s \leftarrow (C^A, A_s)$ 
end for
// Find best cost-controller tuple for each switch
for  $f \in F$  do
    for  $s \in \mathcal{N}(D_f)$  do
         $C^{A'} \leftarrow$  calculate according to (C.4), as if  $A_s = f$ 
        // If new cost better than previously stored cost
        if  $C^{A'} < \text{cost}(\mathcal{C}_s)$  then
             $\mathcal{C}_s \leftarrow (C^{A'}, f)$ 
        end if
    end for
end for
// Derive new switch-to-controller mapping
for  $s \in S$  do
    // Get the controller from the tuple
     $A'_s \leftarrow \text{controller}(\mathcal{C}_s)$ 
end for
return  $(F', A')$ 

```

It could be argued that the cost-based policy effectively describes a family of

switch migration policies that guarantee cost reduction. This is certainly true to some extent, as there is no particular mechanism that is being used to expedite the total cost calculation in general topologies. However, in the case of tree topologies, a switch migration will only impact the cost of that switch's neighborhood, as shown in Section C.3.2, significantly reducing the computational complexity of calculating the total cost. Perhaps other such properties can be identified that would help pinpoint a more efficient mechanism, but this is left for future work. In the context of this study, the cost-migration policy is selected so that a comparison against the more traditional distance-migration policy can be made, with the purpose of indicating the impact of switch migration policies on the CPP.

To summarize the migration-based approach proposed in this work, is outlined in Algorithm 2, in a manner that is generic enough to be policy-ignorant. To reiterate, this algorithm is executed periodically, with the controllers collecting traffic-related statistics between executions. Upon execution, all controllers will have migrated at most one hop away from their initial location and their domains will have likely been altered at least once through the switch migration policies. Due to the periodic execution for this approach, it is expected that it would be able to cope with the challenges set by dynamic traffic, as was showcased for the original migration policy by Oikonomou in [30]; this setting corresponds to the dynamic CPP variant proposed by Bari et al. [22].

Algorithm 2 Outline of Proposed Migration-Based Approach

F : current controllers

A : switch to controller assignment

Λ : the relevant control traffic measured across all links

m : the number of successive switch migration rounds

τ : the traffic matrix

$F' \leftarrow \text{controllerMigrationPolicy}(\mathcal{A}, \Lambda)$

$A' \leftarrow \text{Update } A \text{ based on new controller locations } F'$

for $i = 1, \dots, m$ **do**

$A' \leftarrow \text{switchMigrationPolicy}(F', A', \tau)$

end for

return (F', A')

Chapter E

Evaluation

IN this Chapter, the migration-based approach towards the CPP is evaluated with respect to the reduction in flow-setup times and control overhead. Initially, realistic simulations are carried out studying tree topologies in particular, considering a realistic TCP/OpenFlow network stack. For general topologies, on the other hand, a custom-made simulator that captures the system at a more abstract level is constructed and used for experimentation.

E.1 Tree Topologies

The proposed approach is evaluated in the OMNeT++ simulation environment (v.5.5.1) [39], utilizing the INET¹ framework (v.3.6.6), which provides a suite of Internet protocols. Additionally, a fork of the OpenFlow extension² for INET (originally developed by Gray et al. [40]) is extended to allow for the migration of controllers and the migration of switches to different controllers during runtime. The simulations are carried out on a machine equipped with an Intel Core i7 10750H and 16 GB of RAM. The primary objective of the evaluation is to assess how effective the migration-based approach to the CPP is, particularly in terms of minimizing flow-setup times for the case of tree topologies.

The new features added to the OpenFlow extension can be summarized as follows:

¹<https://inet.omnetpp.org/>

²<https://github.com/CoRE-RG/OpenFlow>

- Switches may migrate to a controller. While a migration is being performed, the switch maintains connection with its original controller and initiates the new connection using a new TCP socket. Once the new connection is established through the OpenFlow handshake, the old connection is closed.
- Controllers may switch between disabled and enabled states upon command, and they are hosted at all switch locations. Upon switch migration, an enabled controller awaits until all its switches have connected to the new controller, before it switches to the disabled state. When a disabled controller receives an activation command, it immediately requests a synchronization, before accepting new connections from switches.
- Various controller applications are modified to support controllers alternating between disabled and enables states. For instance, applications that perform an operation periodically (e.g., sending LLDP messages for topology discovery) are halted while the controller remains disabled.
- Various modules are added that support and enable the orchestration of controllers, including statistics collection from switches, controller management, result collection, etc.

E.1.1 Setup

The considered topology is a randomly generated tree, consisting of 30 switches, which remains the same across all experiments. The diameter of this tree, corresponding to the longest shortest path, is equal to 9. There are 4 branches rooted at the median of this tree, the leaves of which have distance 4, 4, 4, and 5 from the median, respectively. The switches and controllers in this topology are connected with links featuring a data rate of 40 Gbps and a delay equal to 1 ms.

Across all simulations, the controller is equipped with an LLDP and an ARPResponder agent, used for topology discovery purposes. An LLDPForwarding agent is also utilized, which computes shortest paths and responds to `PACKET_IN` messages. The control plane architecture is flat, following the principles of the HyperFlow implementation [40], where the controllers synchronize their global state to a synchronizer that is statically placed in the network’s median.

Experiments are conducted for $k \in \{1, 2, 3\}$ controllers operating in the network. The switches are initially assigned to controllers using the distance-based policy. The controllers start migrating at 140 seconds and then migrate once every 20 seconds, following the controller migration policy, and a round of switch migrations follow immediately afterwards, based on the switch migration policy employed. If the controllers do not migrate for three consecutive 20-second intervals, the simulation ends. Regarding traffic, each switch hosts a single client, initially generating a ping message to a random host, then consecutively to all other hosts, so that in any 20-second interval $\tau_{ij} = 1$, for all $i, j \in S$. Rules installed on switches are deleted based on an idle timeout, where the rule is deleted once a countdown finished, but the countdown is refreshed each time the rule is used.

It is worth pointing out that, while the optimal placement of the controllers is calculated using a Python script, the optimal assignment of switches to controllers has a much larger search space. For instance, the brute-force approach to optimal switch assignment for 2 controllers requires searching through 2^S possible assignments for each of the $\binom{|S|}{2}$ controller placements. In practice, even for networks with 30 switches, finding an optimal assignment took more than 10 minutes for a single placement of two controllers. Therefore, the switch assignment is not compared to the theoretically optimal one.

Instead, the optimal cost is calculated using the distance-based policy, where each switch is assigned to one of the controllers that have been optimally placed. During simulation, the total placement cost is calculated according to (C.4), whenever the controller migrates to a new location. Additionally, the flow-setup time is recorded for all flows and averaged out over 5-second windows. Last but not least, all OpenFlow-related control packets that cross each link are aggregated, including the `PACKET_HELLO` messages used, when a switch initiates a connection with its controller.

E.1.2 Results

In the case of $k = 1$, the controller is initially placed on the switch farthest away (5 hops) from the median, in order to examine the worst case scenario and showcase the effectiveness of the controller migration policy. In this scenario, all switches are mapped to a single controller, so no switch migrations occur, other than reassigning

the switches to the newly enabled controller, when a controller migration takes place.

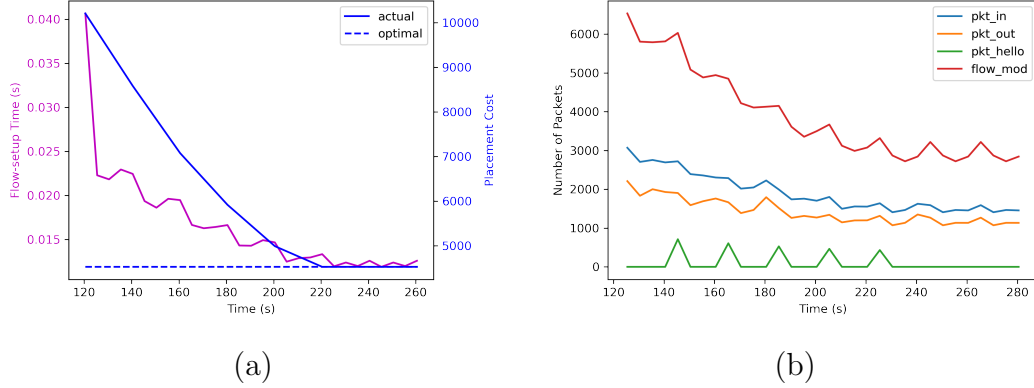


Figure E.1: Acquired results for $k = 1$: (a) The placement cost and mean flow-setup time. (b) The number of OpenFlow-related packets captured.

The results for $k = 1$ are showcased in Fig. E.1. It is evident that as the controller migrates and the placement cost is reduced, the flow-setup times drop as well, starting out at 40 ms and dropping to less than 0.15 ms. The controller eventually reaches the optimal location, as expected by Proposition 2. As for the number of OpenFlow-related packets that are captured, it is clear that, regardless of the packet's type, there is a reduction in their numbers. Since the displayed counts have been captured from all links, this indicates that, as the controller migrates closer to the optimal location, the messages do not have to travel as far across the network, which is to be expected. This indicates the significance of the controller placement, not just in reducing the flow-setup times, which are observed by the end-user, but also in the control overhead, improving the network's efficiency.

For $k = 2$, one controller is initially placed 5 hops away from the median, whereas the second one is placed on a switch of a different branch, 4 hops away from the median. The results for this scenario are depicted in Fig. E.2. It is worth pointing out that the cost-based and distance-based policies did not exhibit different behavior, producing identical results; thus, they are jointly presented. Similarly to $k = 1$, as the two controllers migrate to their final locations and the placement cost is reduced, the flow-setup times drop and the total number of packets sent is reduced as well. In comparison to Fig. E.1, the flow-setup times are slightly shorter, as is to be expected, due to the controllers being closer to their switches, even though the optimal location is not reached by all controllers.

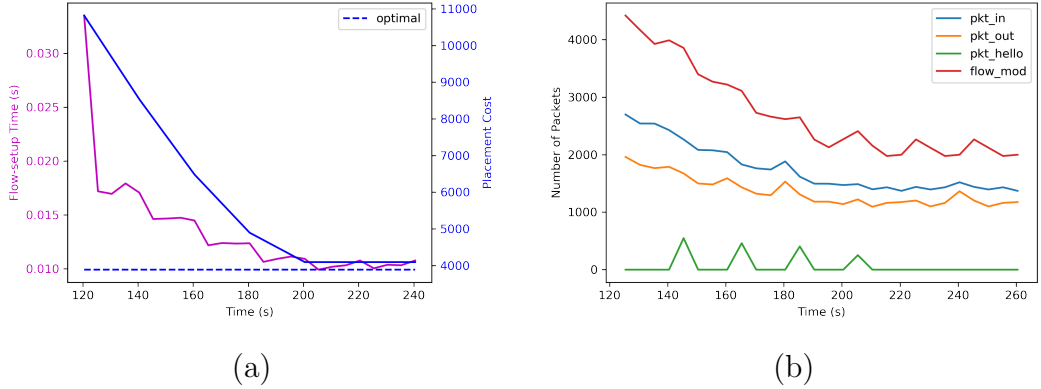


Figure E.2: *Acquired results for $k = 2$: (a) The placement cost and mean flow-setup time. (b) The number of OpenFlow-related packets captured.*

For $k = 3$, the results are depicted in Fig. E.3. In this scenario, the two controllers are initially placed at the same positions as before, while the third controller is placed on a different branch of the median, at a distance of 4 hops away from the median. Once again, as the controllers are migrating to better locations, the flow-setup times drop and the total distance the control packets traverse is reduced. However, an interesting observation can be made 200 seconds into the simulation, as the controllers end up migrating away from the optimal location. This occurs due to some of the switches producing less `PACKET_IN` and `PACKET_OUT` messages than expected, which is likely attributed to the idle timeout used in the simulation, which is not accounted for by the analysis. It is evident that the flow-setup times do not increase, even though the theoretical placement cost has slightly increased after this point in time.

Across all scenarios, the switch migration policies did not produce different results. This is attributed to the fact that tree topologies offer very few opportunities for switch migration. In particular, it is almost always the case that switches only need to migrate to the controller they were assigned to, after it migrates to a new location. Furthermore, when the controllers converge to their final locations, they are often adjacent to other controllers; in this way, controllers block access to the switches in their domain and switch migrations cannot occur. This clustering of controllers occurs because, unlike the traditional k -median problem, the demands to controllers are triggered when a flow reaches a new domain, due to the intricacies of OpenFlow. Therefore, in the tree topology that is studied, domains take up at

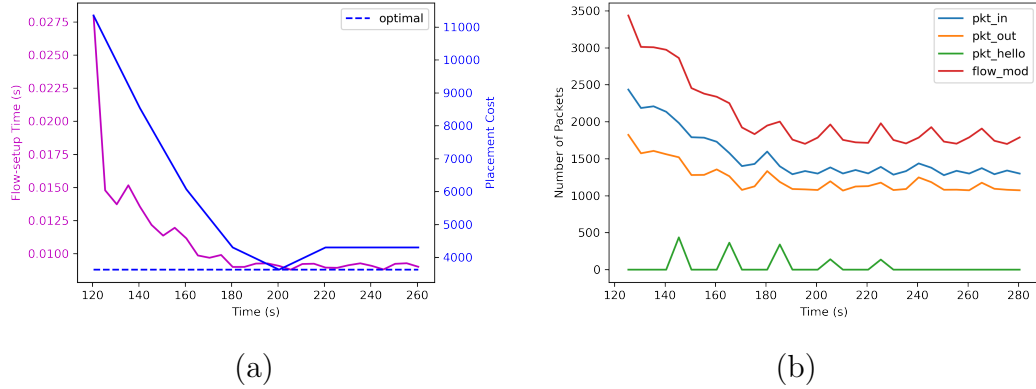


Figure E.3: Acquired results for $k = 3$: (a) The placement cost and mean flow-setup time. (b) The number of OpenFlow-related packets captured.

least one branch rooted at the median and each domain ends up being “pulled” towards the median by the other domains.

E.2 General Topologies

To evaluate the proposed approach in general topologies, a custom simulator is built using Python 3.8. In this way, the controller and switch migration policies are applied in a more abstract manner than the OMNeT++ simulations carried out in the previous section.

E.2.1 Setup

Initially, the considered topologies are randomly generated Erdős-Rényi graphs, consisting of 50 switches and the probability for creating an edge is set to $p = 0.05$, so that the resulting graph is relatively sparse and its diameter high. Due to the larger variance in random graph generation and the fact that it is not as simple to identify the worst initial controller placement, when compared to trees, two different initial placement strategies are considered: the balanced strategy and the random strategy. In particular, the balanced strategy places the first controller at the node farthest away from the graph’s median. All consecutive controllers are placed as far away as possible from the median, while maintaining a distance of at least $\delta/2$ hops from all other controllers. The random placement strategy initially places the k controllers

randomly around the graph. Both placement strategies are performed once for each graph and the mean and confidence intervals are derived for the cost.

Experiments are conducted for $k \in \{2, 3, 4, 5\}$ controllers operating in the network. The switches are initially assigned to controllers using the distance-based policy. In this set of experiments, the controllers operate according to discrete time-steps, i.e., the controllers perform one migration at each time-step, immediately followed by a round of switch migrations within the same time-step. The simulation ends once no controller migrations are performed. Regarding traffic, it still holds that $\tau_{ij} = 1$, for all $i, j \in S$, with $i \neq j$. It is worth pointing out that, since this is a higher-level simulation, the flow-setup process is not directly simulated, but the estimated cost is simply calculated according to (C.2) and (C.4).

Furthermore, the total cost is normalized based on the cost of the initial placement by dividing the cost at each time step with the initial cost. This normalization allows for the comparison of costs across different topologies, focusing on the cost improvement over time rather than its absolute value. It should be noted that the initial cost is identical for both switch migration policies, so the results are directly comparable; if the normalized placement cost of one policy at a given time-step is smaller than that of another policy, then the absolute cost is smaller as well. Finally, 20 experiments are conducted for each combination of parameter values, with the mean value and 95% confidence intervals being shown. Due to the large number of experiments, the optimal allocation cost is not calculated.

E.2.2 Results

The results acquired for the balanced initial placement across different values of k are depicted in Fig. E.4. From a first glance, it is evident that for $k = 2$, the cost-based policy slightly outperforms the distance-based policy across all time-steps. However, for larger values of k , the distance-based policy eventually achieves lower cost than the cost-based policy, which is even more evident for $k = 5$. In general, it is observed that the cost-based policy terminates earlier than the distance-based policy. In some cases, this is not optimal as the controllers stop too early, resulting in an overall smaller cost reduction than the distance-based switch migration policy can provide. Nonetheless, the migration-based approach is capable of reaching around 40% cost reduction compared to the initial placement in all cases.

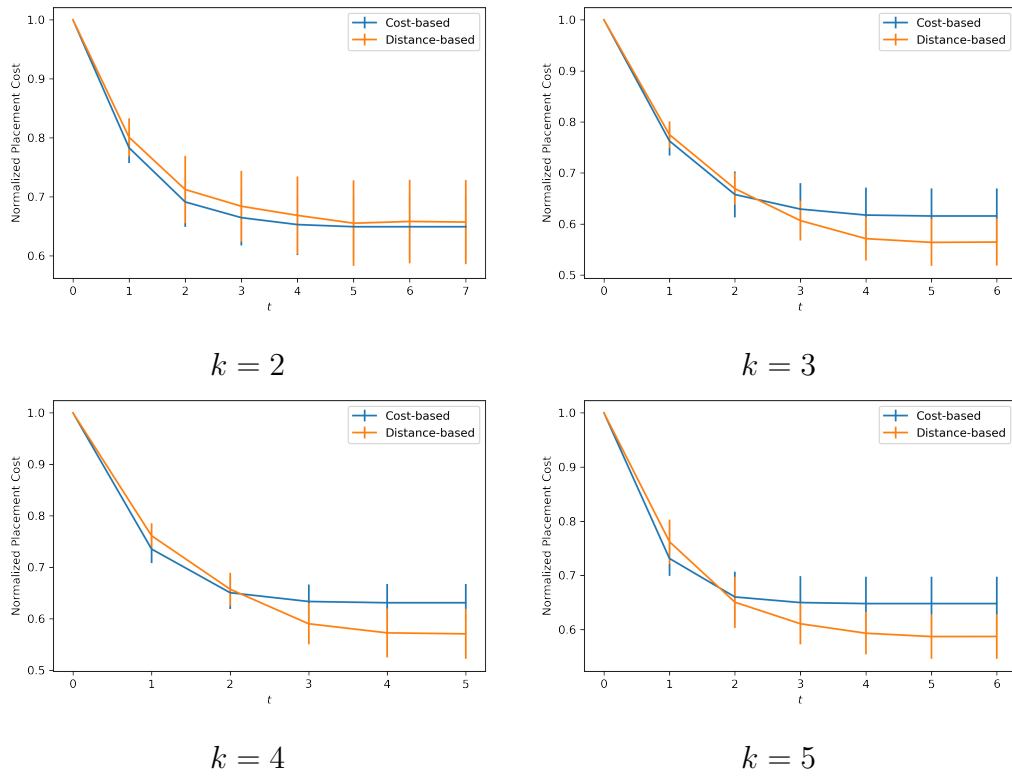


Figure E.4: The normalized placement cost calculated for the balanced initial placement across different values of k .

Regarding the random initial placement strategy, the acquired results are depicted in Fig. E.5. It is observed that the performance of the two switch migration policies is much closer in this scenario, with the cost-based approach even achieving the highest reduction through fewer controller migrations when $k = 5$. Compared to the balanced strategy, a smaller reduction in cost is achieved, capped at about 20%, which is to be expected as the balanced strategy is closer to the worst-case scenario placement considered in tree topologies.

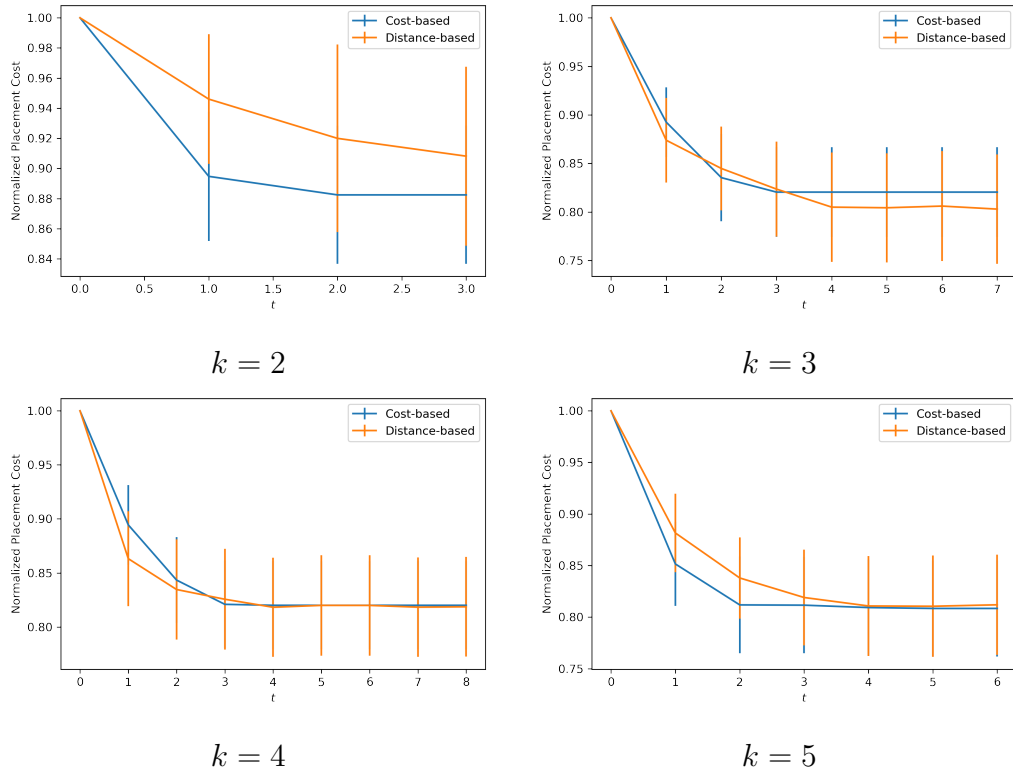


Figure E.5: The normalized placement cost calculated for the random initial placement across different values of k .

Chapter F

Conclusions

In this Thesis, the CPP that arises in distributed SDN control planes was studied, pointing out similarities and differences with traditional facility location problems, and in particular the k -median problem. With this in mind, previously proposed approaches that can adequately address the k -median problem can potentially be adapted to the SDN context.

To this end, a migration-based approach was proposed that incorporates controllers migrating from one physical machine to another and switches migrating from one controller to another. One controller and two switch migration policies were considered that guide these migration decisions under different conditions. In essence, this approach involves an application of the controller migration policy, immediately followed by one or more rounds of switch migrations.

This migration-based approach was evaluated over various topologies, including random tree and random graph network topologies. The evaluation focuses on the effectiveness of the approach to address the CPP, reducing flow-setup time and control overhead. To facilitate these simulations, an existing simulation framework that provides part of OpenFlow functionality was extended to allow for the dynamic provisioning of controllers and assignment of switches. These features were not present in any freely available and general-purpose networking simulator during the undertaking of this Thesis, to the best of the author's knowledge.

The acquired results indicate that the migration-based approach performs fairly well, with the performance of the switch migration policies varying based on the conditions. Most importantly, it was shown that the switch migration policy does

not affect performance in tree topologies, whereas in general topologies the cost-based policy performs better when k is small, while the distance-based policy is better when k is relatively large. Across all experiments, the proposed approach is capable of significantly improving upon the initial placement, regardless of how controllers were initially allocated.

These results indicate that the proposed approach is fairly promising, though its capabilities were not fully explored over the course of the presented work. For instance, in its general form, the migration-based approach may employ more than one successive rounds of the switch migration policy, determined by parameter m , expanding the initial domain to nodes further than 1 hop away, but no experiments were made to assess how effective this may be. Furthermore, much like the original version of the controller migration policy [30], the migration-based approach is expected to be adept at dynamic traffic scenarios, but, once again, this was not assessed experimentally.

Furthermore, less uniform traffic patterns would give rise to more complex situations, impacting both switch and controller migration decisions. To this end, the aspect of timeouts on installed rules produced slightly unexpected results that are difficult to model analytically and need to be studied experimentally. Finally, more efficient switch migration policy alternatives may be derived from the cost-based policy, likely through more extensive analysis.

Bibliography

- [1] A. D. Ferguson, S. Gribble, C.-Y. Hong, C. Killian, W. Mohsin, H. Muehe, J. Ong, L. Poutievski, A. Singh, L. Vicisano, R. Alimi, S. S. Chen, M. Conley, S. Mandal, K. Nagaraj, K. N. Bollineni, A. Sabaa, S. Zhang, M. Zhu, and A. Vahdat, “Orion: Google’s software-defined networking control plane,” in *18th USENIX Symposium on Networked Systems Design and Implementation (NSDI 21)*, pp. 83–98, USENIX Association, Apr. 2021.
- [2] H. Kim and N. Feamster, “Improving network management with software defined networking,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 114–119, 2013.
- [3] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, “On multi-access edge computing: A survey of the emerging 5g network edge cloud architecture and orchestration,” *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1657–1681, 2017.
- [4] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, “Openflow: enabling innovation in campus networks,” *ACM SIGCOMM computer communication review*, vol. 38, no. 2, pp. 69–74, 2008.
- [5] A. Tootoonchian and Y. Ganjali, “Hyperflow: A distributed control plane for openflow,” in *Proceedings of the 2010 internet network management conference on Research on enterprise networking*, vol. 3, 2010.
- [6] B. Han, V. Gopalakrishnan, L. Ji, and S. Lee, “Network function virtualization: Challenges and opportunities for innovations,” *IEEE Communications Magazine*, vol. 53, no. 2, pp. 90–97, 2015.

- [7] V. Dangovas and F. Kuliesius, “Sdn-driven authentication and access control system,” in *The International Conference on Digital Information, Networking, and Wireless Communications (DINWC2014)*, pp. 20–23, 2014.
- [8] I. F. Akyildiz, A. Lee, P. Wang, M. Luo, and W. Chou, “A roadmap for traffic engineering in sdn-openflow networks,” *Computer Networks*, vol. 71, pp. 1–30, 2014.
- [9] D. Erickson, G. Gibb, B. Heller, D. Underhill, J. Naous, G. Appenzeller, G. Parulkar, N. McKeown, M. Rosenblum, M. Lam, *et al.*, “A demonstration of virtual machine mobility in an openflow network.”
- [10] M. Qilin and S. Weikang, “A load balancing method based on sdn,” in *2015 Seventh International Conference on Measuring Technology and Mechatronics Automation*, pp. 18–21, IEEE, 2015.
- [11] U. Zakia and H. B. Yedder, “Dynamic load balancing in sdn-based data center networks,” in *2017 8th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pp. 242–247, IEEE, 2017.
- [12] I. Farris, T. Taleb, Y. Khettab, and J. Song, “A survey on emerging sdn and nfv security mechanisms for iot systems,” *IEEE Communications Surveys & Tutorials*, vol. 21, no. 1, pp. 812–837, 2018.
- [13] K. Wang, Y. Wang, D. Zeng, and S. Guo, “An sdn-based architecture for next-generation wireless networks,” *IEEE Wireless Communications*, vol. 24, no. 1, pp. 25–31, 2017.
- [14] I. Ku, Y. Lu, M. Gerla, R. L. Gomes, F. Ongaro, and E. Cerqueira, “Towards software-defined vanet: Architecture and services,” in *2014 13th Annual Mediterranean Ad Hoc Networking Workshop (MED-HOC-NET)*, pp. 103–110, 2014.
- [15] S. H. Yeganeh, A. Tootoonchian, and Y. Ganjali, “On scalability of software-defined networking,” *IEEE Communications Magazine*, vol. 51, no. 2, pp. 136–141, 2013.

- [16] M. Karakus and A. Durresi, “A survey: Control plane scalability issues and approaches in software-defined networking (sdn),” *Computer Networks*, vol. 112, pp. 279–293, 2017.
- [17] M. He, A. Basta, A. Blenk, and W. Kellerer, “Modeling flow setup time for controller placement in sdn: Evaluation for dynamic flows,” in *2017 IEEE International Conference on Communications (ICC)*, pp. 1–7, IEEE, 2017.
- [18] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama, *et al.*, “Onix: A distributed control platform for large-scale production networks.,” in *OSDI*, vol. 10, pp. 1–6, 2010.
- [19] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O’Connor, P. Radoslavov, W. Snow, *et al.*, “Onos: towards an open, distributed sdn os,” in *Proceedings of the third workshop on Hot topics in software defined networking*, pp. 1–6, 2014.
- [20] B. Heller, R. Sherwood, and N. McKeown, “The controller placement problem,” *SIGCOMM Comput. Commun. Rev.*, vol. 42, p. 473–478, Sept. 2012.
- [21] T. Hu, Z. Guo, P. Yi, T. Baker, and J. Lan, “Multi-controller based software-defined networking: A survey,” *IEEE Access*, vol. 6, pp. 15980–15996, 2018.
- [22] M. F. Bari, A. R. Roy, S. R. Chowdhury, Q. Zhang, M. F. Zhani, R. Ahmed, and R. Boutaba, “Dynamic controller provisioning in software defined networks,” in *Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013)*, pp. 18–25, IEEE, 2013.
- [23] M. Cello, Y. Xu, A. Walid, G. Wilfong, H. J. Chao, and M. Marchese, “Balcon: A distributed elastic sdn control via efficient switch migration,” in *2017 IEEE International Conference on Cloud Engineering (IC2E)*, pp. 40–50, 2017.
- [24] Z. Guo, S. Zhang, W. Feng, W. Wu, and J. Lan, “Exploring the role of paths for dynamic switch assignment in software-defined networks,” *Future Generation Computer Systems*, vol. 107, pp. 238–246, 2020.
- [25] T. Das, V. Sridharan, and M. Gurusamy, “A survey on controller placement in sdn,” *IEEE communications surveys & tutorials*, vol. 22, no. 1, pp. 472–503, 2019.

- [26] M. T. I. ul Huque, W. Si, G. Jourjon, and V. Gramoli, "Large-scale dynamic controller placement," *IEEE Transactions on Network and Service Management*, vol. 14, no. 1, pp. 63–76, 2017.
- [27] J. Lu, Z. Zhang, T. Hu, P. Yi, and J. Lan, "A survey of controller placement problem in software-defined networking," *IEEE Access*, vol. 7, pp. 24290–24307, 2019.
- [28] G. Yao, J. Bi, Y. Li, and L. Guo, "On the capacitated controller placement problem in software defined networks," *IEEE Communications Letters*, vol. 18, no. 8, pp. 1339–1342, 2014.
- [29] K. Oikonomou and I. Stavrakakis, "Scalable Service Migration: The Tree Topology Case," in *5th Annual Mediterranean Ad Hoc Networking Workshop*, 2006.
- [30] K. Oikonomou and I. Stavrakakis, "Scalable service migration in autonomic network environments," *IEEE Journal on Selected Areas in Communications*, vol. 28, no. 1, pp. 84–94, 2009.
- [31] N. Feamster, J. Rexford, and E. Zegura, "The road to sdn: an intellectual history of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 2, pp. 87–98, 2014.
- [32] D. Kreutz, F. M. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proceedings of the IEEE*, vol. 103, no. 1, pp. 14–76, 2014.
- [33] Y. Li and M. Chen, "Software-defined network function virtualization: A survey," *IEEE Access*, vol. 3, pp. 2542–2553, 2015.
- [34] M. Gharbaoui, B. Martini, D. Adami, G. Antichi, S. Giordano, and P. Castoldi, "On virtualization-aware traffic engineering in openflow data centers networks," in *2014 IEEE Network Operations and Management Symposium (NOMS)*, pp. 1–8, IEEE, 2014.
- [35] M. Alsaeedi, M. M. Mohamad, and A. A. Al-Roubaiey, "Toward adaptive and scalable openflow-sdn flow control: A survey," *IEEE Access*, vol. 7, pp. 107346–107379, 2019.

- [36] T. Benson, A. Akella, and D. A. Maltz, “Network traffic characteristics of data centers in the wild,” in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pp. 267–280, 2010.
- [37] A. Dixit, F. Hao, S. Mukherjee, T. Lakshman, and R. R. Kompella, “Elasticon; an elastic distributed sdn controller,” in *2014 ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, pp. 17–27, IEEE, 2014.
- [38] N. Laoutaris, G. Smaragdakis, K. Oikonomou, I. Stavrakakis, and A. Bestavros, “Distributed placement of service facilities in large-scale networks,” in *IEEE INFOCOM 2007 - 26th IEEE International Conference on Computer Communications*, pp. 2144–2152, 2007.
- [39] A. Varga, *OMNeT++*, pp. 35–59. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010.
- [40] N. Gray, T. Zinner, S. Gebert, and P. Tran-Gia, “Simulation framework for distributed sdn-controller architectures in omnet++,” in *Mobile Networks and Management* (R. Agüero, Y. Zaki, B.-L. Wenning, A. Förster, and A. Timm-Giel, eds.), (Cham), pp. 3–18, Springer International Publishing, 2017.

Appendices

Appendix I

Publication Table

Refereed Conferences

- A. Zervopoulos and K. Oikonomou, “A Migration-Based Approach for the SDN Controller Placement Problem in Tree Topologies”, in *Proceedings of the 2021 International Balkan Conference on Communications and Networking (BalkanCom’21)*, Novi Sad, Serbia, Sep. 2021.

Appendix II

Glossary

Software-Defined Networking	An emerging networking architecture that decouples the control plane and the data plane
Control Plane	The part of the network concerned with how to forward traffic from its source to its destination
Data Plane	The part of the network concerned with merely forwarding traffic according to nodes' forwarding tables.
Controller	The entity responsible for managing the control plane in a Software-Defined Networking architecture
OpenFlow	Specification that defines the main components of switches and controllers, along with the possible interactions between them
Southbound Interface	The interface used for communication between switches and controllers
Northbound Interface	The interface used for communication between controllers and applications
East-westbound Interface	The interface used for communication between controllers
Controller Domain	The set of switches assigned to a controller
Controller Placement Problem	The problem of appropriately placing controllers in the network, which arises in distributed Software-Defined Networking control planes

Flow-setup Process	The process by which switches request forwarding rules from controllers, exchanging <code>PACKET_IN</code> and <code>PACKET_OUT</code> messages
Flow-setup Time	The time taken to complete the flow-setup process
Controller Migration	Migrating the virtual machine, over which a controller is operating, to a different physical machine than its current one
Switch Migration	Reassigning a switch from its current controller to a different one

Appendix III

Notations

k	Number of enabled SDN controllers in network
G	Graph representing considered network
S	Set of switches in network
L	Set of links in network
F	Set of enabled controllers
A_s	Function indicating the controller that switch $s \in S$ is assigned to
D_f	The set of switches in the domain of $f \in F$
$\mathcal{A}$	Representation of controller allocation, comprising F and $A_s, \forall s \in S$
τ^{ij}	Number of PACKET_IN messages generated by traffic flowing from i to j , $i, j \in S$
$d(i, j)$	Distance in terms of hops between switches i and j , $i, j \in S$
$N(s)$	Set of switches adjacent to switch $s \in S$
$\mathcal{N}(D_f)$	Set of switches adjacent to, but not included in, domain $D_f, f \in F$
P^i	The shortest path between switches $i, j \in S$
I_s^{ij}	Binary function indicating whether switch $s \in S$ generates PACKET_IN messages for traffic flowing from i to j , $i, j \in S$
$C_s^{\mathcal{A}}$	Cost that switch $s \in S$ contributes, given allocation $\mathcal{A}$
$C_f^{\mathcal{A}}$	Cost that domain of $f \in F$ contributes, given allocation $\mathcal{A}$
$C^{\mathcal{A}}$	Total cost of allocation $\mathcal{A}$
y_{ij}	Binary decision variable indicating whether switch $j \in S$ is assigned to controller $i \in F$

x_i	Binary decision variable indicating whether an enabled controler co- resides with switch $i \in S$
$\mathcal{P}^{ij}$	Sequence of controllers that switches in P^{ij} are assigned to, $i, j \in S$
Λ^{sf}	Control traffic flowing between controller $f \in F$ and adjacent switch $s \in N(f)$.

Appendix IV

Abbreviations

SDN	Software-Defined Networking
VM	Virtual Machine
ONF	Open Networking Foundation
CPP	Controller Placement Problem
WAN	Wide Area Network
SBI	Southbound Interface
NBI	Northbound Interface
API	Application Programming Interface
TCP	Transmission Control Protocol
IP	Internet Protocol
QoS	Quality of Service
QoE	Quality of Experience
E-WBI	East-westbound Interface
LLDP	Link-Layer Discovery Protocol
ARP	Address Resolution Protocol

Index

- k*-median, 3, 6, 29
- Adjacency, 29
- Constraints, 5, 21, 31
- Control Overhead, 7
- Control Plane, 1, 2, 11, 26
- Controller, 1, 12, 19, 21, 26–28, 30, 37
- Controller Allocation, 28
- Controller Application, 2, 26
- Controller Migration, 27
- Controller Migration Policy, 6, 38
- Controller Placement, 3, 5, 6, 20, 28
- Controller Placement Problem, 3–5, 20, 29, 35
- Cost, 6, 29–31
- Data Plane, 1, 11
- Domain, 2, 19, 28, 30
- Dynamic Controller Placement Problem, 22, 35
- East-westbound Interface, 19
- Facility Location, 3, 29
- Flat Architecture, 19, 26
- Flow, 29
- Flow Entry, 15
- Flow-setup Process, 2, 17, 27
- Flow-setup Time, 3, 4, 7, 21
- Forwarding Device, 11
- Hierarchical Architecture, 19
- HyperFlow, 3, 46
- Integer Linear Program, 5
- Integer Linear Programming, 31
- Logically Centralized, 3, 18
- Migration, 6, 27, 31, 35, 37, 40
- Network Function Virtualization, 2, 14
- Northbound Interface, 13
- OMNeT++, 7, 45
- Onix, 3, 4
- ONOS, 3
- Open Networking Foundation, 2, 12
- OpenFlow, 1, 3, 6, 14, 26, 45
- OpenFlow `FLOW_MOD` Message, 17, 29
- OpenFlow `PACKET_IN` Message, 17, 29
- OpenFlow `PACKET_OUT` Message, 17, 29
- OpenFlow Handshake, 16, 46

Orion, 3, 4, 12

Shortest Path, 29

Software-Defined Networking, 1, 6, 11

Southbound Interface, 12, 14

Switch, 1, 12, 14, 28, 30, 31, 40

Switch Assignment, 3, 5, 6, 20, 28, 31

Switch Migration Policy, 6, 40

Synchronization, 3, 19, 26

Virtual Machine, 3, 21, 27