ΕΘΝΙΚΟ ΜΕΤΣΟΒΙΟ ΠΟΛΥΤΕΧΝΕΙΟ

ΣΧΟΛΗ ΗΛΕΚΤΡΟΛΟΓΩΝ ΜΗΧΑΝΙΚΩΝ ΚΑΙ ΜΗΧΑΝΙΚΩΝ ΥΠΟΛΟΓΙΣΤΩΝ

ΤΟΜΕΑΣ ΕΠΙΚΟΙΝΩΝΙΩΝ ΗΛΕΚΤΡΟΝΙΚΗΣ ΚΑΙ ΣΥΣΤΗΜΑΤΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Χρήση Τεχνολογίας SDN για Δυναμική Αναδιάρθρωση Δικτύων Data Center

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

ΑΝΑΣΤΑΣΙΑΣ ΚΑΡΑΤΡΑΝΤΟΥ

Επιβλέπων : Ευστάθιος Συκάς
Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2016

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

Χρήση Τεχνολογίας SDN για Δυναμική Αναδιάρθρωση Δικτύων Data Center

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

της

ΑΝΑΣΤΑΣΙΑΣ ΚΑΡΑΤΡΑΝΤΟΥ

Επιβλέπων : Ευστάθιος Συκάς
Καθηγητής Ε.Μ.Π.

Εγκρίθηκε από την τριμελή εξεταστική επιτροπή την.

(Υπογραφή)	*(Υπογραφή)*	*(Υπογραφή)*
.....	.....	.....
Ευστάθιος Συκάς	Μιχαήλ Θεολόγου	Γεώργιος Στασινόπουλος
Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.	Καθηγητής Ε.Μ.Π.

Αθήνα, Μάρτιος 2016

(Υπογραφή)

.....

ΑΝΑΣΤΑΣΙΑ ΚΑΡΑΤΡΑΝΤΟΥ

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

Περιεχόμενα

Περιεχόμενα Σχημάτων

Περιεχόμενα Πινάκων

Περίληψη

Το μέγεθος των Data center μεγαλώνει εκθετικά, έτσι ώστε να ικανοποιήσει τις ανάγκες των εφαρμογών και των χρηστών. Το γεγονός αυτό εγείρει ανησυχία σχετικά με τις μεγάλες ενεργειακές ανάγκες που προκύπτουν και τον περιβαλλοντικό αντίκτυπο που αυτές έχουν. Η παρατήρηση αυτή επιβάλλει να ληφθεί υπόψη, ως βασικός σχεδιαστικός παράγοντας, η ενεργειακή αποτελεσματικότητα στο σχεδιασμό των Δικτύων Data Center. Επιπλέον, παρατηρείται πως για μεγάλα χρονικά διαστήματα η κίνηση στα δίκτυα Data Center απέχει σημαντικά από τη μέγιστη τιμή της. Αυτό οδηγεί στην ύπαρξη αδρανών δικτυακών συσκευών, που όμως σπαταλούν μεγάλες ποσότητες ενέργειας. Για τον μετριασμό αυτού του ζητήματος οι έρευνες υποδεικνύουν πώς να τεθούν εκτός λειτουργίας κάποιες συσκευές ή κάποιες θύρες αυτών σύμφωνα με την εφαρμογή κάποιου «πράσινου αλγορίθμου».

Ο σκοπός της παρούσας διπλωματικής εργασίας είναι η ανάπτυξη μίας εφαρμογής που στοχεύει στην τροποποίηση της κατάστασης των θυρών κάποιων συγκεκριμένων προωθητικών συσκευών σύμφωνα με τα πορίσματα κάποιας σουίτας βελτιστοποίησης. Ο βασικός στόχος αφορά την δυναμική και παράλληλα ενεργειακά αποδοτική δρομολόγηση σε Δίκτυα Data Center με την εκμετάλλευση της τεχνολογίας των Δικτύων Καθοριζόμενων από Λογισμικό (*SDN*) και την αποσύνδεση των επιπέδων ελέγχου και δεδομένων.

Η ανάπτυξη των τμημάτων λογισμικού σε αυτό το εγχείρημα βασίστηκε στην χρήση του προγράμματος εξομοίωσης Mininet, του πρωτοκόλλου επικοινωνιών OpenFlow και του ελεγκτή ελεύθερου λογισμικού Ryu, που είναι αναπτυγμένος εξ ολοκλήρου σε Python.

Εν κατακλείδι, η παρούσα εργασία συμβάλλει στη δημιουργία του απαραίτητου δικτυακού πλαισίου SDN για την εφαρμογή και τον έλεγχο των πορισμάτων οποιουδήποτε ενεργειακού αλγορίθμου.

Λέξεις Κλειδιά: δίκτυα Data Center, δυναμική παρακολούθηση, τροποποίηση θυρών, Fat Tree, Ryu ελεγκτής, πρωτόκολλο OpenFlow, Ενεργειακή αποτελεσματικότητα

Abstract

Data centers are growing exponentially to meet user and application demands. This fact raises concern about the vast energy needs and consequent environmental impact. This observation mandates energy efficiency to be considered as a major design parameter in Data Center Networks. Furthermore, most of the time data center traffic is far below the peak value. This leads to idle network devices which waste a significant amount of energy. To mitigate this issue studies show how to switch off network devices or devices' ports with the implementation of a green algorithm.

The purpose of this thesis is the development of an application aiming to modify the port state of some particular devices according to the indications of an optimization suite. The main goal concerns the dynamic, energy efficient routing in data center networks by leveraging software defined networking and the separation of control and data planes.

The development of the software components in this project was based on the usage of the emulation program called Mininet, communications protocol OpenFlow and the Python-based, open source SDN controller Ryu.

In conclusion, this project intends to contribute to the creation of the required framework in order to implement and test the outcomes of any green algorithm.

Keywords: data center networks, dynamic monitoring, port modification, fat tree, Ryu controller, OpenFlow protocol, energy efficiency

Ευχαριστίες

Η παρούσα διπλωματική εργασία εκπονήθηκε στο Εθνικό Μετσόβιο Πολυτεχνείο και σηματοδοτεί την ολοκλήρωση των σπουδών μου σε αυτό. Θα ήθελα να ευχαριστήσω θερμά τον επιβλέποντα καθηγητή κ. Ευστάθιο Συκά για την εμπιστοσύνη που μου έδειξε αναθέτοντάς μου την εκπόνηση της, καθώς και για τη δυνατότητα να ασχοληθώ με ενδιαφέρουσα και καινοτόμα θεματολογία. Παράλληλα, θα ήθελα να ευχαριστήσω τον κ. Πάρι Χαραλάμπου για την καθοδήγηση, τις συμβουλές και τη συμβολή του κατά τη διάρκεια της χρονιάς που πέρασε. Τέλος, θα ήθελα να ευχαριστήσω θερμά την οικογένεια και τους φίλους μου για την απόλυτη στήριξη τους κατά τη διάρκεια αυτού του εγχειρήματος.

1

Εισαγωγή

1.1 Τα Δίκτυα Data Center και τα Ζητήματα στη Διαχείριση τους

Το Data Center είναι φυσική ή εικονική υποδομή που χρησιμοποιείται από επιχειρήσεις για στέγαση των υπολογιστών, των εξυπηρετητών, των δικτυακών συστημάτων καθώς και των απαιτούμενων στοιχείων από το τμήμα IT (*Information Technology*). Οι ανάγκες που εξυπηρετεί συνήθως συνοψίζονται σε αποθήκευση και επεξεργασία καθώς και εξυπηρέτηση μεγάλων ποσοτήτων, κριτικής σημασίας δεδομένων για τους πελάτες σε αρχιτεκτονικές πελάτη-εξυπηρετητή (*client-server architectures*).

Ένα τέτοιο δίκτυο απαιτεί εκτεταμένα και εφεδρικά συστήματα παροχής ενέργειας, συστήματα ψύξης, υψηλού επιπέδου ασφάλεια και πλεονάζουσες δικτυακές συνδέσεις.

Η διαχείριση του δικτύου Data Center (*DCN*) περιλαμβάνει την εξασφάλιση της αξιοπιστίας στις συνδέσεις που αυτό περιλαμβάνει αλλά και την προσεκτική φύλαξη των αποθηκευμένων σε αυτό πληροφοριών. Ιδιαίτερα σημαντική παράμετρος είναι, όμως, και η διαχείριση του φορτίου κατά τον πιο οικονομικό τρόπο.

Για την εκπόνηση αυτής της εργασίας αφορμή αποτέλεσε η τελευταία παράμετρος και συγκεκριμένα η μείωση του ενεργειακού κόστους στη διαχείριση της κίνησης εντός ενός DCN. Για την εξυπηρέτηση του σκοπού αυτού, αναπτύχθηκε η κατάλληλη υποδομή έτσι ώστε να είναι δυνατή η εφαρμογή και η αξιοποίηση των πορισμάτων, οποιουδήποτε ενεργειακού αλγορίθμου.

1.2 Αντικείμενο διπλωματικής

Συγκεκριμένα, το αντικείμενο αυτής της διπλωματικής εργασίας είναι η ανάπτυξη των εργαλείων που θα επιτρέπουν να γίνεται δρομολόγηση με ενεργειακά κριτήρια σε ένα DCN. Οι προσεγγίσεις που μελετήθηκαν για το σκοπό αυτό και αποτελούν άξονα για αυτό το έργο αφορούν την απενεργοποίηση ενεργειακά δαπανηρών και παροδικά αδρανών δικτυακών συσκευών με την εκμετάλλευση των πλεονεκτημάτων των Δικτύων Καθοριζόμενων από Λογισμικό (*SDN*). Διευκρινίζεται πως, η εργασία αυτή δεν μελετά την αποτελεσματικότητα ενός ενεργειακού αλγορίθμου, αλλά καθιστά εφικτή την εφαρμογή οποιουδήποτε αλγορίθμου στοχεύει σε ενεργειακή αναδιάρθρωση, παρέχοντας: (α) την δικτυακή τοπολογία, (β) τα εργαλεία άντλησης όλων των απαραίτητων πληροφοριών των δικτυακών συσκευών και (γ) την μέθοδο εφαρμογής των υποδείξεων της βελτιστοποίησης, δηλαδή την εφαρμογή των κανόνων αναδιάρθρωσης του δικτύου με ενεργειακά κριτήρια.

1.2.1 Συνεισφορά

Η συνεισφορά της διπλωματικής συνοψίζεται ως εξής:

1. Μελέτη ελεγκτών SDN με στόχο την εύρεση του καταλληλότερου ανάλογα με τη σκοπιμότητα της εφαρμογής που αναπτύσσεται ή χρησιμοποιείται.
2. Υλοποίηση τμημάτων κώδικα που μπορούν να χρησιμοποιηθούν από οποιαδήποτε άλλη εφαρμογή προκειμένου να παρέχουν στον χρήστη ή προγραμματιστή πληροφορίες και στατιστικά στοιχεία από δικτυακές τοπολογίες που μελετώνται με στόχο την περαιτέρω εκμετάλλευση τους.
3. Υλοποίηση τμήματος κώδικα που επιτρέπει την αλλαγή από ενεργή σε ανενεργή και αντίστροφα, στην κατάσταση κάποιων θυρών που υποδεικνύονται με χρήση οποιουδήποτε αλγορίθμου ή προγράμματος βελτιστοποίησης.
4. Μελέτη των επιδόσεων του πιο διαδεδομένου για δίκτυα SDN προσομοιωτή δικτύων και εξομοίωση σε αυτόν της πιο διαδεδομένης για δίκτυα data center τοπολογίας (*Fat Tree*).
5. Παρουσίαση ενός ολοκληρωμένου πειράματος για την συνολική κατανόηση των όσων δημιουργήθηκαν.

1.3 Οργάνωση κειμένου

Η παρούσα διπλωματική εργασία αποτελείται συνολικά από εννέα (9) κεφάλαια.
Στο παρόν κεφάλαιο παρουσιάζονται οι προβληματισμοί και οι ανάγκες που οδήγησαν στην ανάπτυξη της εφαρμογής που αποτελεί αντικείμενο της εργασίας καθώς και η συνεισφορά της.

Στο δεύτερο (2) κεφάλαιο αναλύεται η τεχνολογία των Δικτύων Καθοριζόμενων από Λογισμικό (*SDN*) [16]. Ειδικότερα, παρουσιάζονται: (α) η ανάγκη δημιουργίας αυτής της καινοτομίας [17], (β) τα στοιχεία που απαρτίζουν ένα *SDN* δίκτυο, (γ) η πιο διαδεδομένη Νότια Διεπαφή ενός τέτοιου δικτύου, δηλαδή το πρωτόκολλο OpenFlow του οποίου γίνεται εκτενής χρήση κατά την ανάπτυξη της εφαρμογής [1], (δ) το βασικότερο στοιχείο της αρχιτεκτονικής *SDN*, δηλαδή ο ελεγκτής, (στ) οι ελεγκτές που δοκιμάστηκαν για να χρησιμοποιηθούν στην παρούσα εργασία (Opendaylight, ONOS, Ryu), (ε) μία μελέτη για τις συγκριτικές επιδόσεις των προαναφερθέντων ελεγκτών και τέλος [11] (ζ) τα τρωτά σημεία της αρχιτεκτονικής *SDN* αναφορικά με την ασφάλεια των δικτύων.

Στο τρίτο (3) κεφάλαιο γίνεται παρουσίαση του ελεγκτή που επιλέχθηκε να χρησιμοποιηθεί, δηλαδή του Ryu [14]. Αναλυτικότερα, σε αυτό περιλαμβάνονται: (α) η ανάλυση της αρχιτεκτονικής του, (β) η δημιουργία εφαρμογών σε αυτόν, (γ) η εφαρμογή *Switching Hub* που αποτελεί το πιο βασικό τμήμα λογισμικού του ελεγκτή καθώς και (δ) τα μηνύματα OpenFlow που αποστέλλονται σε διάφορες περιπτώσεις μεταξύ αυτού του ελεγκτή και των δικτυακών συσκευών, πολλά από τα οποία αποτέλεσαν δομικό στοιχείο στην ανάπτυξη της εφαρμογής.

Στο τέταρτο κεφάλαιο παρουσιάζεται το λογισμικό Mininet [7]. Αναφέρονται διάφορα χρηστικά στοιχεία αυτού και αναλύεται η επίδοση του σύμφωνα με ερευνητικές μελέτες. Ακόμη, παρουσιάζεται η τοπολογία που χρησιμοποιήθηκε για τις ανάγκες του παρόντος εγχειρήματος.

Το πέμπτο κεφάλαιο αφορά αποκλειστικά τα δίκτυα Data Center [19]. Συγκεκριμένα, παρουσιάζονται: (α) οι στόχοι στη σχεδίαση τους και (β) οι διάφορες αρχιτεκτονικές που συναντώνται σε αυτά με ειδική αναφορά στην πιο διαδεδομένη, κάποιες σπανιότερες που εξυπηρετούν εξειδικευμένες ανάγκες και την καινοτομία της εταιρείας Cisco.

Στο έκτο (6) κεφάλαιο προσεγγίζεται το πρόβλημα της ενεργειακής σπατάλης σε Δίκτυα Data Center και ο τρόπος να εφαρμοστεί η βιβλιογραφική λύση την οποία λαμβάνει ως αφορμή η εργασία. Η λύση αυτή πραγματοποιείται σε τέσσερα (4) επίπεδα: την ανίχνευση της υφιστάμενης τοπολογίας, τη συλλογή στατιστικών δεδομένων, την βελτιστοποίηση κατανάλωσης ενέργειας και την τροποποίηση της κατάστασης των θυρών των δικτυακών συσκευών έτσι ώστε να επιτραπεί εκ νέου η ενεργειακή δρομολόγηση.

Στο έβδομο κεφάλαιο γίνεται αναφορά στα τεχνικά ζητήματα που αντιμετωπίστηκαν κατά την εκπόνηση της εργασίας και παρουσιάζονται τα τμήματα κώδικα που αναπτύχθηκαν και χρησιμοποιήθηκαν.

Το όγδοο κεφάλαιο αποτελεί τη σύνοψη όσων μελετήθηκαν και αναπτύχθηκαν στα πλαίσια της διπλωματικής εργασίας. Αρχικά, σε αυτό, παρουσιάζονται εκτενώς τα πειράματα που διεξήχθησαν με στόχο την κατανόηση από τον αναγνώστη της λειτουργικότητας της εφαρμογής. Επιπλέον, παρουσιάζονται τα συμπεράσματα και

οι σκέψεις που αφορούν μελλοντικές επεκτάσεις και προτάσεις με αφορμή τη συμβολή της εργασίας αυτής.
Τέλος, στο ένατο κεφάλαιο συγκεντρώνεται η βιβλιογραφία που μελετήθηκε για την εκπόνηση της εργασίας.

2

Δίκτυα Καθοριζόμενα από Λογισμικό

2.1 Η ιδέα των Δικτύων Καθοριζόμενων από Λογισμικό

Εξαιτίας των πρόσφατων τεχνολογικών αλλαγών σε όλα τα πεδία γενικά, αλλά κυρίως στους κλάδους της Πληροφορικής και των Τηλεπικοινωνιών, προέκυψε η ανάγκη επαναπροσδιορισμού των δικτυακών αρχιτεκτονικών, καθώς οι παραδοσιακές άρχισαν να εμφανίζουν σοβαρούς περιορισμούς. Σήμερα, η τάση είναι η διασύνδεση των πάντων με χρήση κατάλληλων τεχνολογιών όπως το “Cloud Computing” ή το “Internet of Things”. Οι καινοτόμες αυτές προσεγγίσεις απαιτούν εκτός από μεγαλύτερο εύρος ζώνης, απλούστερα και πιο ευκίνητα δίκτυα που να διευκολύνουν την καινοτομία. Τα Δίκτυα Καθοριζόμενα από Λογισμικό (Software Defined Networks – SDN) αποτελούν τη νέα δομή που θα αντιμετωπίσει τα αποδεδειγμένα μειονεκτήματα των παραδοσιακών δικτυακών δομών.

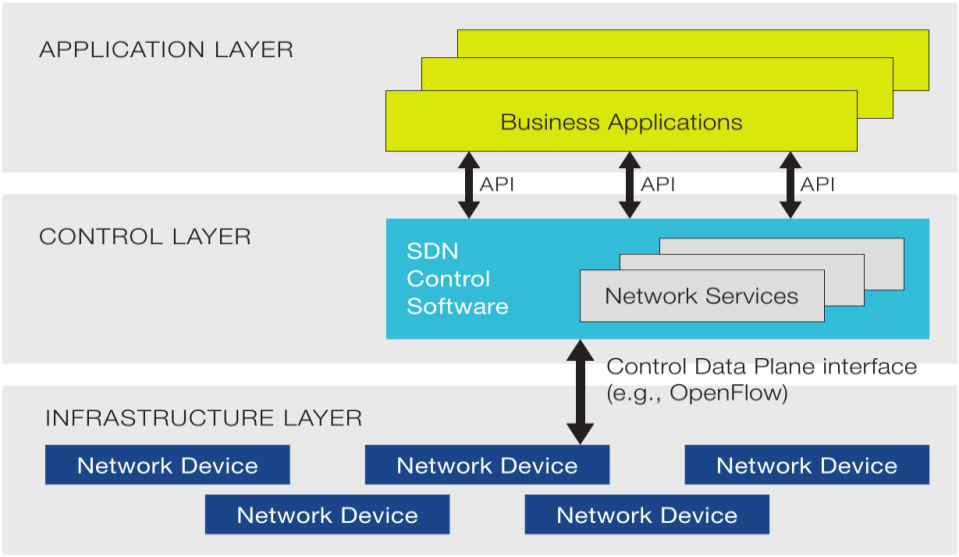
Οι «πυλώνες» στους οποίους δομείται αυτή η νέα ιδέα είναι: (α) ο χωρισμός των επιπέδων δεδομένων και ελέγχου, (β) η χρήση ροών δεδομένων αντί για διευθύνσεις προορισμού για τη λήψη αποφάσεων προώθησης, (γ) η λογική του ελέγχου μεταφέρεται σε μία εξωτερική οντότητα, τον ελεγκτή (SDN controller) που τρέχει ένα δικτυακό λειτουργικό σύστημα και (δ) το δίκτυο είναι προγραμματιζόμενο μέσα από εφαρμογές λογισμικού που τρέχουν στο λειτουργικό σύστημα του δικτύου και αλληλοεπιδρούν με τις συσκευές του δικτύου, που πλέον χωρίς καμία «ευφυία» αποτελούν απλές συσκευές προώθησης.

Οι ιδέες πίσω από τα Δίκτυα Καθοριζόμενα από Λογισμικό δεν είναι νέες, αλλά αποτέλεσμα προηγούμενων ερευνητικών θεμάτων, όπως τα ενεργά δίκτυα ή η ανάπτυξη του πρωτοκόλλου OpenFlow που προηγήθηκε στο Πανεπιστήμιο του Stanford.

Πλέον τα δίκτυα έχουν αποκτήσει πολυπλοκότητα και κατά συνέπεια παρουσιάζουν δυσκολίες στην παραμετροποίηση και τη διαχείριση. Επιπλέον, έχει αυξηθεί ραγδαία ο αριθμός των κινητών συσκευών, όπως και το περιεχόμενο στο οποίο αυτές έχουν πρόσβαση. Στόχος, λοιπόν, της νέας αυτής αρχιτεκτονικής είναι να μετριάσει περιορισμούς όπως πολυπλοκότητα, εξάρτηση από τα ιδιαίτερα κατασκευαστικά χαρακτηριστικά των δικτυακών συσκευών, ασυνέπειες στις δικτυακές πολιτικές που εφαρμόζονται ή θέματα επεκτασιμότητας.
Συνοπτικά, η τεχνολογία SDN προσφέρει:

- Δυνατότητα άμεσου προγραμματισμού των συσκευών.
- Ευελιξία, καθώς υπάρχει η δυνατότητα δυναμικής προσαρμογής των ροών κίνησης ανάλογα με τις υπάρχουσες απαιτήσεις.
- Κεντρική διαχείριση, καθώς η «ευφυία» του δικτύου συγκεντρώνεται σε συγκεκριμένες συσκευές, τους ελεγκτές (SDN controllers), που διατηρούν εποπτεία ολόκληρου του υφιστάμενου δικτύου
- Παραμετροποίηση των δικτυακών συσκευών μέσω λογισμικού γρήγορα, δυναμικά, με αυτοματοποιημένα SDN προγράμματα που δημιουργούν οι διαχειριστές των δικτύων ανάλογα με τις απαιτήσεις του καθενός
- Ανεξαρτησία από τα πρότυπα των κατασκευαστών, καθώς πρόκειται για ελεύθερο λογισμικό με καθολική εφαρμογή.

Για την κατανόηση των SDN δικτύων παρουσιάζεται η τυπική αρχιτεκτονική τους:



Σχήμα 2.1 Αρχιτεκτονική SDN

Όλα τα μοντέλα SDN περιλαμβάνουν τα ακόλουθα στοιχεία:

- ***Ελεγκτής (Controller):*** Εφαρμογή που λειτουργεί ως στρατηγικό σημείο ελέγχου στο δίκτυο. Παρέχουν κεντρικοποιημένη εικόνα του δικτύου και επιτρέπουν στον

διαχειριστή αυτού να επιβάλει στις υποκείμενες συσκευές το πως θα διαχειριστούν τις ροές κίνησης.

- ***Southbound APIs:*** Χρησιμοποιούνται για να προωθήσουν τις πληροφορίες στους δικτυακούς κόμβους (μεταγωγείς και δρομολογητές). Πρόκειται για πρωτόκολλα, με πιο καθιερωμένο το OpenFlow.
- ***Northbound APIs:*** Χρησιμοποιούνται για την επικοινωνία με στοιχεία ανώτερου επιπέδου, δηλαδή τις εφαρμογές.

2.2 Το πρωτόκολλο OpenFlow

Το πρωτόκολλο OpenFlow είναι το πρώτο πρότυπο διεπαφής που ορίστηκε ανάμεσα στα στρώματα ελέγχου και προώθησης στην αρχιτεκτονική SDN. Αυτό επιτρέπει άμεση πρόσβαση και χειρισμό του επιπέδου προώθησης των δικτυακών συσκευών, μεταγωγέων και δρομολογητών, τόσο φυσικών όσο και εικονικών.
Συγκεκριμένα, το OpenFlow επιτρέπει στους ελεγκτές να καθορίσουν τη διαδρομή των δικτυακών πακέτων κατά μήκος των μεταγωγέων του δικτύου. Κατά συνέπεια είναι εφικτή μια πιο εξελιγμένη διαχείριση κίνησης απ' ότι με προηγούμενες τεχνολογίες, δηλαδή τις Λίστες Ελέγχου Πρόσβασης (ACLs) και τα πρωτόκολλα δρομολόγησης. Επιπλέον, το πρωτόκολλο αυτό επιτρέπει τη διαχείριση μεταγωγέων διαφορετικών κατασκευαστών απομακρυσμένα και με χρήση ενός ελεύθερου λογισμικού. Η λειτουργία του συνοψίζεται στη δημιουργία πινάκων προώθησης στους μεταγωγείς με την πρόσθεση και την αφαίρεση, περιοδικά, κανόνων για ενέργειες χειρισμού των εισερχόμενων πακέτων σε αυτούς.
Το OpenFlow λειτουργεί πάνω από το πρωτόκολλο TCP (Transmission Control Protocol) και υπαγορεύει την ανάγκη για χρήση του πρωτοκόλλου TLS (Transport Layer Security). Οι ελεγκτές «ακούν» στη θύρα TCP 6653 για μεταγωγείς που θέλουν να δημιουργήσουν σύνδεση με μαζί τους.
Το πρωτόκολλο αυτό επιτρέπει τρεις τύπους μηνυμάτων:

- ***Ελεγκτή προς μεταγωγέα:*** ο ελεγκτής δημιουργεί μηνύματα διαφόρων τύπων (Feature, Configuration, Modify state, Read state, Packet out, Barrier, Role-request, Asynchronous configuration) για να τροποποιήσει ή να επιθεωρήσει την κατάσταση του μεταγωγέα.
- ***Ασύγχρονα:*** αυτά δημιουργούνται από τον μεταγωγέα για να ενημερώσει τον ελεγκτή για γεγονότα που συμβαίνουν στο δίκτυο καθώς τις αλλαγές στην κατάσταση του (Packet in, Flow removed, Port status).
- ***Συμμετρικά:*** μπορούν να δημιουργηθούν και από τις δύο πλευρές χωρίς να προηγηθεί αίτημα επικοινωνίας (Hello, Echo, Error, Experimental).

2.2.1 Εξερεύνηση ενός Δικτύου OpenFlow

Η εξερεύνηση ενός δικτύου OpenFlow περιλαμβάνει τρείς δραστηριότητες:

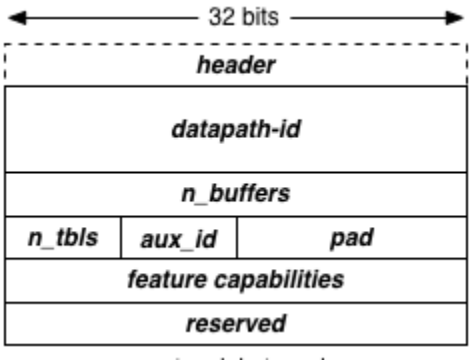
- ***Ανίχνευση μεταγωγέων:*** αυτό επιτυγχάνεται με την ανταλλαγή μηνυμάτων OpenFlow τύπου *feature request/reply*. Οι πιο πρόσφατες εκδόσεις του πρωτοκόλλου επιτρέπουν στις δικτυακές συσκευές να συνδεθούν ταυτόχρονα με περισσότερους από έναν ελεγκτές, έναν κύριο (*master*) και οσουσδήποτε δευτερεύοντες (*slaves*). Τα μηνύματα Feature Request/Reply είναι η πρώτη δραστηριότητα που λαμβάνει χώρα κατά τη δημιουργία καναλιού επικοινωνίας (*TCP, SCTP, TLS*) μεταξύ του ελεγκτή και ενός μεταγωγέα. Το μήνυμα Feature Request αποτελείται, απλά, από μία επικεφαλίδα OpenFlow με την τιμή FeatureReq να έχει οριστεί στο πεδίο «*type*». Σε αυτό ο μεταγωγέας απαντά αναφέροντας τις δυνατότητες του με το μήνυμα FeatureRes. Στο τελευταίο, περιλαμβάνεται το πεδίο *datapath_id* (64 bit) που θεωρείται ανάλογο της διεύθυνσης MAC ενός Ethernet switch, με τη διαφορά ότι ένας φυσικός μεταγωγέας μπορεί να έχει περισσότερα από ένα τέτοια αναγνωριστικά (ένα για κάθε Ethernet διεπαφή). Ακόμη, συναντάται το πεδίο *n_buffers* που υποδεικνύει το αριθμό των πακέτων που μπορεί ο μεταγωγέας να αποθηκεύσει σε ουρά για να προωθήσει αργότερα στον ελεγκτή. Αξίζει να αναφερθεί επιπλέον, το πεδίο *auxiliary_id* που υποδηλώνει τον τρόπο με τον οποίο η συσκευή αντιμετωπίζει το υποκείμενο κανάλι (ως κύριο ή επικουρικό ελεγκτή) και το πεδίο *pad* που χρησιμοποιείται για διατήρηση της ευθυγράμμισης των byte. Οι δυνατότητες που υποστηρίζει ο μεταγωγέας εμφανίζονται ως μάσκα bit. Αναλυτικά τα όσα αναφέρθηκαν παρουσιάζονται στους πίνακες 2.1 και 2.2 που ακολουθούν:

Πίνακας 2.1 Η δομή του μηνύματος Feature Request του OpenFlow

Name	Bits	Byte Ordering	Constrains
datapath_id	64	MSBF (Most Significant Byte First)	None
n_buffers	32	MSBF	None
n_tables	8	-	None
auxiliary_id	8	-	None
Pad	16	-	None
capabilities	32	-	Στον πίνακα που ακολουθεί
Reserved	32	MSBF	None

Πίνακας 2.2 Περιορισμοί στο μήνυμα Feature Request του OpenFlow

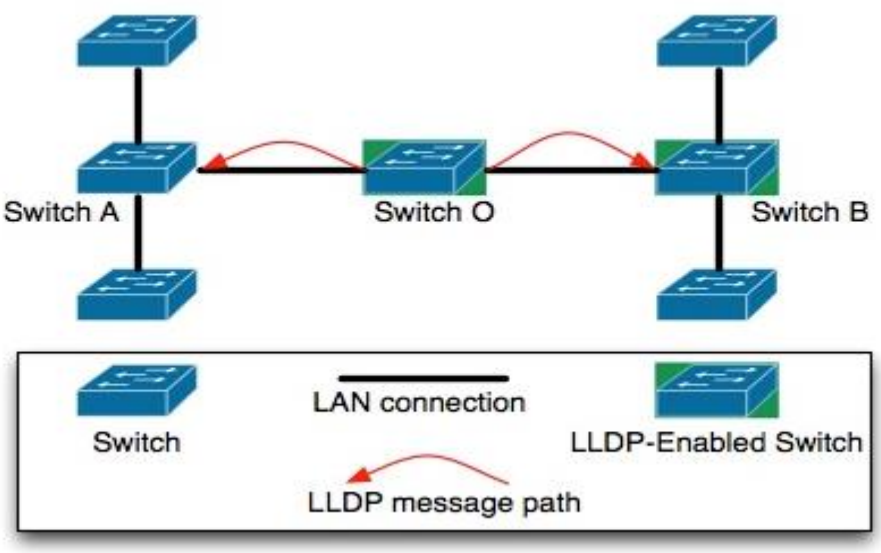
Field	Name	Value
Capabillities	FLOW_STATS	0x00000001
	TABLE_STATS	0x00000002
	PORT_STATS	0x00000004
	GROUP_STATS	0x00000008
	IP_REASM	0x00000020
	QUEUE_STATS	0x00000040
	PORT_BLOCKED	0x00000100



Σχήμα 2.2 Η σειρά των Byte στο μήνυμα Feature request

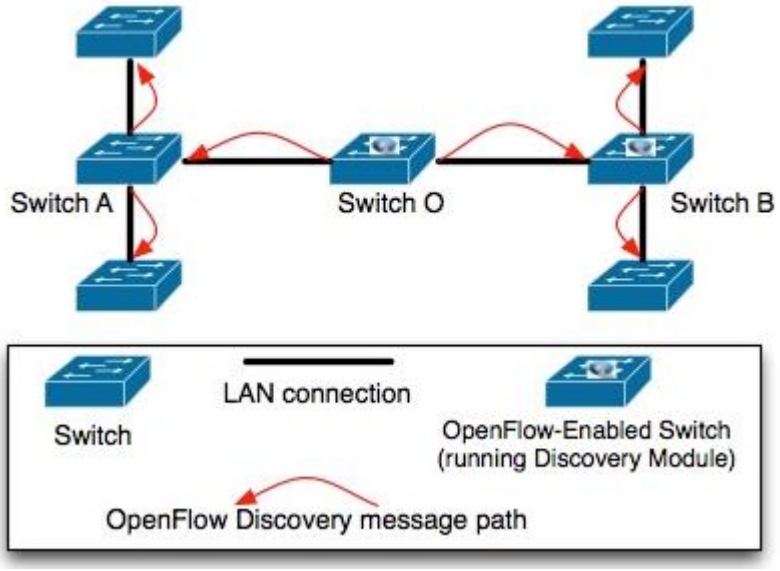
- ***Ανίχνευση συνδέσεων:*** όταν ένας μεταγωγέας συνδέεται με τον ελεγκτή, ο ελεγκτής περιοδικά (π.χ. κάθε 5 δευτερόλεπτα) ζητά από αυτόν να δημιουργήσει *πλημμύρα* μηνυμάτων LLDP (*Link Layer Discovery Protocol*) και BDDP (*Broadcast Doman Discovery Protocol*) από όλες τις θύρες του. Το LLDP είναι ένα πρωτόκολλο ουδέτερο ως προς τον σχεδιαστή των δικτυακών συσκευών, του επιπέδου ζεύξης στη σουίτα πρωτοκόλλων IP που επιτρέπει στις συσκευές να διαφημίζουν τις δυνατότητες που παρέχουν και τους γείτονες τους, συνήθως συνδεδεμένες Ethernet διεπαφές. Αξίζει να σημειωθεί, πως δεν πρόκειται για πρωτόκολλο που λειτουργεί υιοθετώντας την τεχνική ερώτηση/απάντηση (*request/response*) αλλά μόνο με διαφημίσεις. Κάθε συσκευή που λαμβάνει ένα διαφημιστικό μήνυμα πρέπει να ανανεώσει τον πίνακα

που διατηρεί και περιλαμβάνει τις δυνατότητες των γειτονικών συσκευών. Μία τρίτη οντότητα (*agent*) ζητά αυτούς τους πίνακες έτσι ώστε να σχηματίσει την εικόνα της τοπολογίας σε επίπεδο 2. Σημαντικό είναι ακόμη, πως κάθε μεταγωγέας δέχεται διαφημίσεις μόνο από τους άμεσα συνδεδεμένους γείτονες (στο επίπεδο 1) και δεν προωθεί αυτές τις διαφημίσεις.



Σχήμα 2.3 Διαδρομή μηνύματος LLDP

Στην περίπτωση που αναφέρεται (Δίκτυα SDN), όμως, αυτό που πραγματικά συμβαίνει είναι η χρήση του πρωτοκόλλου OpenFlow Discovery που εκμεταλλεύεται τη λειτουργικότητα του LLDP κάνοντας κάποιες ήπιες τροποποιήσεις προκειμένου να εξερευνήσει συγκεκριμένα ένα OpenFlow δίκτυο. Πρόκειται, ουσιαστικά, για τεχνικό όρο προκειμένου να καταστεί σαφής η χρήση του. Οι διαφορές του περιλαμβάνουν: (α) την αποστολή των διαφημίσεων με χρήση διεύθυνσης *multicast* και (β) την δυνατότητα προώθησης των διαφημίσεων από κάθε μεταγωγέα.



Σχήμα 2.4 Διαδρομή τροποποιημένου μηνύματος OFDP

- ***Ανίχνευση τερματικών:*** πρόκειται για πιο σύνθετη διαδικασία από τις προηγούμενες και για το κάθε ένα γίνεται με την αποστολή του πρώτου πακέτου προς κάποιον αποδέκτη στο δίκτυο (τερματικό που ακόμα θεωρείται άγνωστο). Ο πρώτος μεταγωγέας που θα το λάβει, που αποτελεί και το σημείο διασύνδεσης του τερματικού στο δίκτυο που εποπτεύει ο ελεγκτής, μην έχοντας εγγραφή για την εισερχόμενη ροή θα το προωθήσει στον ελεγκτή. Ο ελεγκτής τότε θα εντοπίσει το προαναφερθέν σημείο διασύνδεσης που αφορά τον αποστολέα και θα δώσει εντολή για πλημμύρα στον μεταγωγέα που το δέχτηκε. Η διαδικασία αυτή θα καταλήξει στην αναγνώριση του σημείου διασύνδεσης του αποδέκτη του μηνύματος και τελικά στην ανίχνευση του τερματικού.

2.3 Ο ελεγκτής στα δίκτυα SDN

Όπως προαναφέρθηκε πυρήνα ενός δικτύου SDN αποτελεί ο ελεγκτής, καθώς αυτός καθορίζει τη συμπεριφορά ολόκληρου του δικτύου. Οι ελεγκτές διακρίνονται σε ανοιχτού λογισμικού από κοινοτικές πρωτοβουλίες (Opendaylight, Project Floodlight, Beacon, NOX/POX κ.α.) και ιδιωτικών κατασκευαστών (Juniper Contrail, Nuage Virtualized Services Controller by Alcatel-Lucent, VortiQa Open Network Director by Freescale Semiconductor κ.α.). Επιπλέον, μπορούν να κατηγοριοποιηθούν ανάλογα με τη γλώσσα προγραμματισμού στην οποία έχουν υλοποιηθεί, την απόδοση που μπορούν να επιτύχουν, τον χρόνο που απαιτείται για τον προγραμματιστή να μάθει να αναπτύσσει εφαρμογές σε αυτούς, τον τύπο των νοτίων διεπαφών που υποστηρίζουν, τον σκοπό για τον οποίο δημιουργήθηκαν, την υποστήριξη για κατανεμημένο έλεγχο που μπορεί να παρέχουν, κ.α.
Η αυξανόμενη υιοθέτηση της τεχνολογίας SDN όχι μόνο για πειραματικούς σκοπούς αλλά και από τη βιομηχανία δημιουργεί την ανάγκη κατάλληλης επιλογής ελεγκτή για κάθε περίπτωση χρήσης. Η επιλογή αυτή μπορεί να βασιστεί στην ικανοποίηση ή μη των ακόλουθων παραμέτρων [12]:

- ***Επίδοση (performance)****:* Ο ελεγκτής ναι μεν δεν προορίζεται για προώθηση πακέτων (ροών) αλλά ελέγχει τον τρόπο με τον οποίο αυτά προωθούνται. Για αυτό πρέπει να είναι ικανός να ενημερώνει τους πίνακες ροών που διατηρούν οι μεταγωγείς εκ των προτέρων και να επεξεργάζεται έγκαιρα τις νέες εγγραφές έτσι ώστε να αποφεύγεται η συμφόρηση (*performance bottlenecks*).
- ***Υποστήριξη προτύπων ελεύθερου λογισμικού (open-source standards)****:* Απαραίτητη προϋπόθεση έτσι ώστε να είναι δυνατή η ενσωμάτωση συσκευών και λειτουργιών διαφορετικών κατασκευαστών.
- ***Αξιοπιστία (reliability)****:* Ο ελεγκτής πρέπει να είναι σχεδιασμένος έτσι ώστε να επιτρέπει εναλλακτικές λύσεις σε κάθε ζήτημα, όσον αφορά το λογισμικό και το

υλισμικό (*redundancy features*). Πρέπει να υποστηρίζει τον διαμοιρασμό της κίνησης σε διαφορετικές συνδέσεις. Ακόμη, κάποια δίκτυα απαιτούν τη χρήση περισσότερων του ενός ελεγκτών, οπότε είναι χρήσιμη η δυνατότητα δημιουργίας συστάδας πολλαπλών ελεγκτών (*cluster*).

- ***Λειτουργικότητα (functionality):*** Κάθε ελεγκτής παρέχει διαφορετικές κατασκευασμένες εφαρμογές (*built-in applications*).
- ***Ασφάλεια (security):*** Ο ελεγκτής είναι απαραίτητο να εφαρμόζει πιστοποίηση χρήστη και έγκυρα φίλτρα απομόνωσης όλων των εικονικών δικτύων.
- ***Δυνατότητα επέκτασης (scalability):*** Ένας μόνο ελεγκτής θεωρείται απαραίτητο να μπορεί να διαχειριστεί τουλάχιστον 100 μεταγωγείς και πλεονάζουσες εγγραφές στους πίνακες ροών χωρίς σημαντική πτώση των επιδόσεων του.

Στον πίνακα που ακολουθεί παρουσιάζονται συνοπτικά διάφορες εφαρμογές χρήσης ορισμένων δημοφιλών ελεγκτών:

Πίνακας 2.3 Σύνοψη κάποιων σημαντικών εφαρμογών χρήσης ελεγκτών

Περίπτωση / Ελεγκτής	Trema	Nox/Pox	Ryu	Floodlight	ODL	ONOS
Εικονοποίηση Δικτύου με Virtual Overlays	ΝΑΙ	ΝΑΙ	ΝΑΙ	ΕΝ ΜΕΡΕΙ	ΝΑΙ	ΟΧΙ
Εικονοποίηση Δικτύου Hop-by-hop	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ
Υποστήριξη OpenStack Neutron	ΟΧΙ	ΟΧΙ	ΝΑΙ	ΝΑΙ	ΝΑΙ	ΟΧΙ
Legacy Network Interoperability	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΝΑΙ	ΕΝ ΜΕΡΕΙ
Service Insertion and Chaining	ΟΧΙ	ΟΧΙ	ΕΝ ΜΕΡΕΙ	ΟΧΙ	ΝΑΙ	ΕΝ ΜΕΡΕΙ
Παρακολούθηση Δικτύου	ΕΝ ΜΕΡΕΙ	ΕΝ ΜΕΡΕΙ	ΝΑΙ	ΝΑΙ	ΝΑΙ	ΝΑΙ
Επιβολή πολιτικών	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΕΝ ΜΕΡΕΙ	ΝΑΙ	ΕΝ ΜΕΡΕΙ
Εξισορρόπηση φορτίου	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΝΑΙ	ΟΧΙ
Traffic Engineering	ΕΝ ΜΕΡΕΙ	ΕΝ ΜΕΡΕΙ	ΕΝ ΜΕΡΕΙ	ΕΝ ΜΕΡΕΙ	ΝΑΙ	ΕΝ ΜΕΡΕΙ
Dynamic Network Taps	ΟΧΙ	ΟΧΙ	ΝΑΙ	ΝΑΙ	ΝΑΙ	ΟΧΙ
Multi-Layer Network Optimization	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΟΧΙ	ΝΑΙ	ΕΝ ΜΕΡΕΙ
Transport Networks - NV, Traffic-rerouting, Interconneting DCs, etc	ΟΧΙ	ΟΧΙ	ΕΝ ΜΕΡΕΙ	ΟΧΙ	ΕΝ ΜΕΡΕΙ	ΕΝ ΜΕΡΕΙ
Δίκτυα τύπου *Campus*	ΕΝ ΜΕΡΕΙ	ΕΝ ΜΕΡΕΙ	ΕΝ ΜΕΡΕΙ	ΕΝ ΜΕΡΕΙ	ΕΝ ΜΕΡΕΙ	ΟΧΙ
Δρομολόγηση	ΝΑΙ	ΟΧΙ	ΝΑΙ	ΝΑΙ	ΝΑΙ	ΝΑΙ

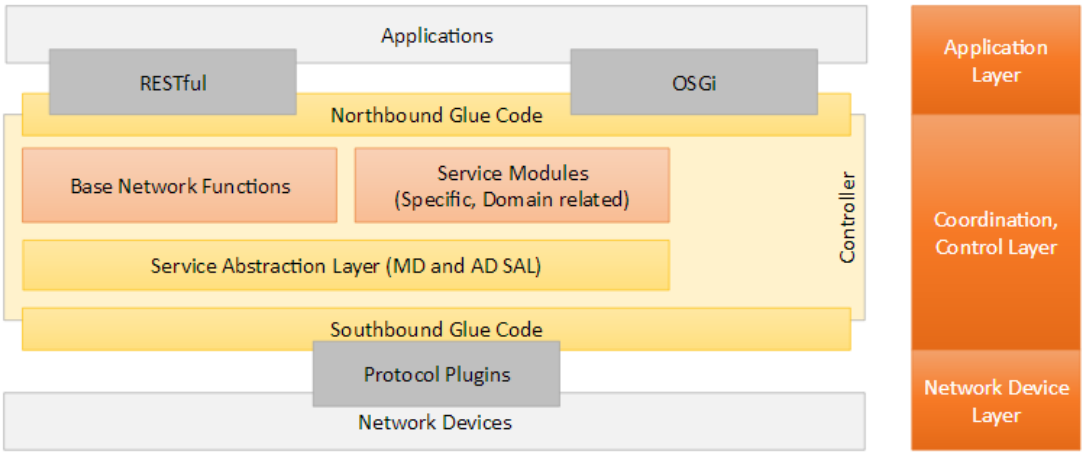
Τέλος, στον ακόλουθο Πίνακα εμφανίζονται ορισμένες σημαντικές ιδιότητες που έχουν κάποιοι ευρέως διαδεδομένοι ελεγκτές:

Πίνακας 2.4 Ιδιότητες SDN ελεγκτών

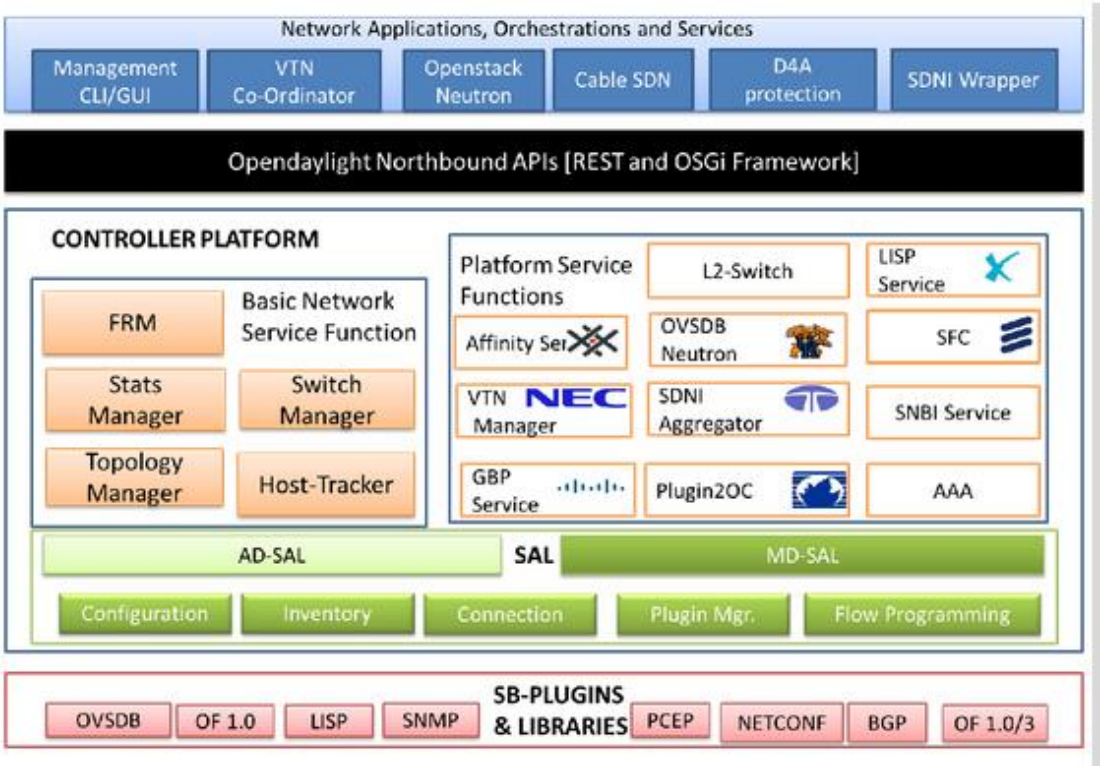
Ιδιότητα	NOX	POX	Beacon	Floodlight	OpenDaylight
Γλώσσες που υποστηρίζει ο ελεγκτής	C, C++, Python	Python	Java	Java, Python	Java
Υπό περαιτέρω ανάπτυξη	Όχι	Ναι	Υπό συντήρηση	Ναι	Ναι
Ενεργή κοινότητα	Όχι	Ναι	Ναι	Ναι	Ναι
Ευκολία στην εγκατάσταση	Όχι	Ναι	Ναι	Ναι	Ναι
Ευκολία στην ανάπτυξη εφαρμογών	Όχι	Ναι	Ναι	Ναι	Όχι
Παροχή τεκμηρίωσης	΄Όχι	Ναι	Ναι	Ναι	Εν μέρει
REST API	Όχι	Ναι - περιορισμένη	Ναι	Ναι	Ναι
User Interface	Python+QT4	Python+QT4, Web	Web	Java, Web	Web
Υποστήριξη τερματικών με πολλαπλά σημεία διασύνδεσης	Όχι	Όχι	Όχι	Ναι	Ναι
Υποστήριξη τοπολογιών με βρόχους	Όχι	Ναι, με χρήση Spanning Tree	Όχι	Ναι	Ναι
Υποστήριξη συνδέσεων που δεν χρησιμοποιούν/υποστηρίζουν το OpenFlow	Όχι	Όχι	Όχι	Ναι	Ναι
Υποστήριξη επιπέδου πάνω από τα πρωτόκολλα του νότιου στρώματος	Όχι	Όχι	Όχι	Όχι	Ναι
Υποστήριξη OpenStack Quantum	Όχι	Όχι	Όχι	Ναι	Ναι

2.3.1 Το Opendaylight

Το Project Opendaylight είναι ένα συνεργατικό εγχείρημα ελεύθερου λογισμικού υπό την αιγίδα του συνδέσμου *Linux Foundation* και την υποστήριξη περισσότερων από 40 εταιρειών με στόχο την ενθάρρυνση της υιοθέτησης της τεχνολογίας SDN [10]. Το λογισμικό είναι γραμμένο εξ ολοκλήρου σε Java. Η πρώτη έκδοση αυτού (Hydrogen) κυκλοφόρησε στις αρχές του 2014 ενώ αναμένεται η τέταρτη (Beryllium) στις αρχές του 2016. Πρόκειται για μία στιβαρή κατασκευή που παρέχει επιδόσεις επιπέδου παραγωγής με κύριο, όμως, μειονέκτημα την πολυπλοκότητα του και τον υψηλό χρόνο εκμάθησης για τη δημιουργία νέων εφαρμογών. Αξίζει να σημειωθεί πως υποστηρίζει τις εκδόσεις 1.0 και 1.3 του πρωτοκόλλου OpenFlow.
Η απλοποιημένη εικόνα της αρχιτεκτονικής του αποτελείται κυρίως από: (α) πρωτόκολλα και *plugins* νοτίων διεπαφών που δημιουργούν το στρώμα δικτύου της συσκευής, (β) το στρώμα ελέγχου και συντονισμού, (γ) βόρειες διεπαφές (APIs) και εφαρμογές που ορίζουν το στρώμα εφαρμογής.



Σχήμα 2.5 Απλοποιημένη αρχιτεκτονική ODL



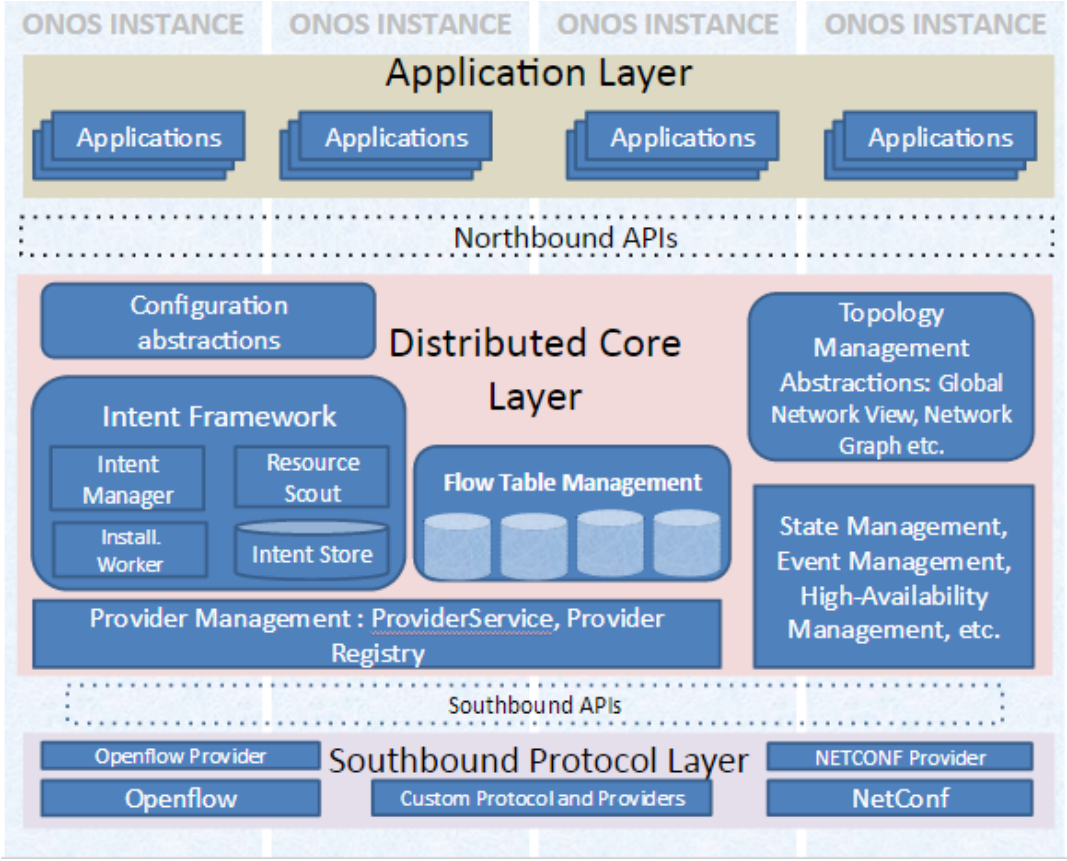
Σχήμα 2.6 Πλήρης αρχιτεκτονική ODL

Στο περιβάλλον του Opendaylight ο ελεγκτής λειτουργεί σαν ενδιάμεσο λογισμικό (middleware). Είναι το λογισμικό που ενώνει τις εφαρμογές που απαιτούν τις υπηρεσίες των δικτυακών συσκευών και των πρωτοκόλλων που «μιλούν» στις δικτυακές συσκευές για να εξάγουν τις υπηρεσίες αυτές. Ο ελεγκτής επιτρέπει στις εφαρμογές να μην χρειάζεται να γνωρίζουν τις προδιαγραφές των συσκευών του δικτύου. Ως εκ τούτου, οι προγραμματιστές μπορούν να επικεντρωθούν στην ανάπτυξη της λειτουργικότητας των εφαρμογών.

2.3.2 Το ONOS

Το Project ONOS (Open Network Operating System) [18] είναι και αυτό ένα εγχείρημα ελεύθερου λογισμικού υπό την αιγίδα του συνδέσμου *Linux Foundation* με στόχο τη δημιουργία ενός λειτουργικού συστήματος για δίκτυα καθοριζόμενα από λογισμικό (software defined networks OS) που θα προσφέρει δυνατότητα δημιουργίας κλιμακούμενων εφαρμογών, υψηλές αποδόσεις και μεγάλη διαθεσιμότητα. Η δημοσίευση του πηγαίου κώδικα του ONOS έγινε στα τέλη του 2013 και μέχρι σήμερα μετρά πέντε εκδόσεις. Η πλατφόρμα του ONOS είναι επεκτάσιμη και κατανεμημένη στοχεύοντας στη δημιουργία φιλικού περιβάλλοντος για τον χρήστη. Το λογισμικό είναι γραμμένο εξ ολοκλήρου σε Java. Παρουσιάζει καλές επιδόσεις όπως και το προαναφερθέν Opendaylight αλλά και παρόμοιες δυσκολίες εκμάθησης. Στις νότιες διεπαφές του συμπεριλαμβάνεται το OpenFlow, εκδόσεις 1.0 και 1.3, και άλλα πρωτόκολλα όπως OVS-DB και OF config, καθώς και πρωτόκολλα διαχείρισης όπως το Netconf και το PCEP.

Η αρχιτεκτονική του ONOS μπορεί να ερμηνευθεί ως συλλογή πολλαπλών υποσυστημάτων τριών σειρών (three-tiered), όπου κάθε υποσύστημα εκτελεί μία υπηρεσία και έχει υλοποιηθεί ως συνδυασμός συνιστωσών σε τρία διαφορετικά στρώματα, εφαρμογής, πυρήνα και νοτίων πρωτοκόλλων.



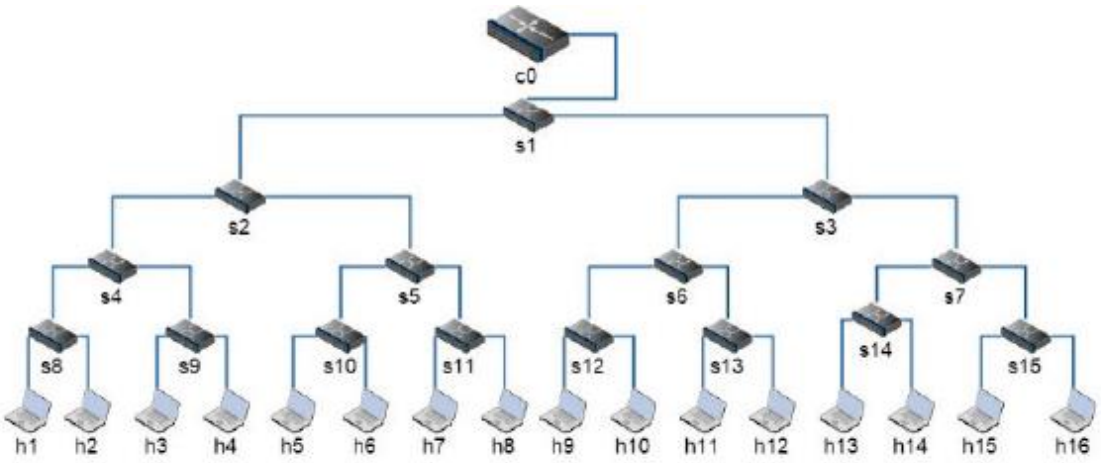
Σχήμα 2.7 Η αρχιτεκτονική του ONOS

Όπως παρατηρείται στην παραπάνω εικόνα, οι πολλαπλοί κλώνοι του ONOS (instances) φανερώνουν τον κατανεμημένο χαρακτήρα του ελεγκτή αυτού. Στην κορυφή της αρχιτεκτονικής, τοποθετούνται οι εφαρμογές που μεταξύ άλλων ξεχωρίζουν οι εξής: (α) Segment Routing, (β) multi-layer SDN control, (γ) topology viewer, (δ) path computation, (ε) SDP-IP peering.

2.3.3 Σύγκριση επιδόσεων των ελεγκτών

Για να γίνουν κατανοητές οι διαφορές μεταξύ κάποιων αρκετά δημοφιλών ελεγκτών γίνεται σύγκριση των επιδόσεων τους με τη χρήση του λογισμικού προσομοίωσης mininet [6]. Οι ελεγκτές που χρησιμοποιήθηκαν είναι οι προαναφερθέντες (α) Opendaylight και (β) ONOS, καθώς και οι (γ) POX: γραμμένος σε Python, κατάλληλος για ερευνητικούς και πειραματικούς σκοπούς, με μειονεκτήματα την μη υποστήριξη νεότερων εκδόσεων του πρωτοκόλλου OpenFlow και εφαρμογών κατανεμημένου χαρακτήρα και (δ) ο Ryu: επίσης γραμμένος σε Python που όμως καλύπτει τα προαναφερθέντα κενά του POX. Ακόμη, επιλέχθηκε δενδρική τοπολογία τεσσάρων επιπέδων, με 16 ξενιστές (hosts), 2 αντιστοιχούν σε κάθε μεταγωγέα τελευταίου

στρώματος. Τα στοιχεία προώθησης της τοπολογίας είναι απλοί μεταγωγείς ovsk (Open Virtual Switches) που υποστηρίζουν το πρωτόκολλο OpenFlow ως διεπαφή για τροποποίηση των πινάκων ροής (flow tables).



Σχήμα 2.8 Δενδρική τοπολογία για την προσομοίωση στο mininet

Σύμφωνα, λοιπόν με τη βιβλιογραφία, στην πρώτη φάση, οι μεταγωγείς λειτουργούν απλά σαν διανομείς (hub), δηλαδή κάθε εισερχόμενο πακέτο μεταφέρεται σε όλες τις θύρες (εκτός αυτής στην οποία εισήλθε) με τη μέθοδο της πλημμύρας. Στη δεύτερη φάση, η λειτουργία των συσκευών αναβαθμίζεται σε αυτό που αποκαλείται στη βιβλιογραφία «L2 learning switch». Αυτό σημαίνει, πως ο ελεγκτής, όταν δέχεται κάποιο πακέτο, συσχετίζει την διεύθυνση MAC της πηγής με τη θύρα του μεταγωγέα από την οποία έφτασε το πακέτο. Ακολούθως, ο πίνακας ροής του μεταγωγέα αποκτά έναν καινούργιο κανόνα που υποδεικνύει την προώθηση πακέτων με τη συγκεκριμένη διεύθυνση MAC στη θύρα που μόλις έμαθε.
Σε κάθε φάση της προσομοίωσης έγιναν δύο τεστ επιδόσεων: ping μεταξύ των h1 και h16, με χρήση συνολικά 10 πακέτων ICMP για καθορισμό του μέσου χρόνου RTT και του κατά πόσο γρήγορα μειώνεται στη περίπτωση του L2 learning switch, όπου οι μεταγωγείς έχουν τους δικούς τους πίνακες ροής και παύουν να χρησιμοποιούν σταδιακά τη μέθοδο της πλημμύρας. Επιλέχθηκαν τα συγκεκριμένα τερματικά καθώς η διαδρομή μεταξύ τους είναι η μεγαλύτερη στο δίκτυο. Ακόμη, διεξάγεται τεστ εκτελώντας την εντολή iperf μεταξύ των προαναφερθέντων τερματικών. Το εργαλείο αυτό παρέχουν τα Linux για μέτρηση των επιδόσεων του δικτύου ανάμεσα σε δύο κόμβους, συγκεκριμένα για μέτρηση του εύρους ζώνης TCP.
Ακολουθούν οι πίνακες των αποτελεσμάτων της βιβλιογραφίας:

Πίνακας 2.5 Αποτελέσματα του test ping στην πρώτη φάση (λειτουργία διανομέα)

SDN Controller	Min. RTT [ms]	Max. RTT [ms]	Avg. RTT [ms]
POX	63.45	125.24	105.2
Ryu	56.59	87.84	76.26
ONOS	42.21	74.25	69.4
Odl	43.67	73.55	63.28

Πίνακας 2.6 Αποτελέσματα του test ping στη δεύτερη φάση (λειτουργία μεταγωγέα)

SDN Controller	Min. RTT [ms]	Max. RTT [ms]	Avg. RTT [ms]	Sec until RTT < ms
POX	0.29	103.23	20.76	2
Ryu	0.11	61.05	11.86	2
ONOS	0.14	177.92	21.71	1
Odl	0.17	156.24-	22.65-	1

Πίνακας 2.7 Αποτελέσματα μέτρησης με iperf στην πρώτη φάση (λειτουργία διανομέα)

SDN Controller	TCP Bandwidth (h1 -> h16)	TCP Bandwidth (h16 -> h1)
POX	3.81 Mbps	4.64 Mbps
Ryu	1.38 Mbps	1.68 Mbps
ONOS	780 Kbps	805 Kbps
Odl	533 Kbps	580 Kbps

Πίνακας 2.5 Αποτελέσματα μέτρησης με iperf στη δεύτερη φάση (λειτουργία μεταγωγέα)

SDN Controller	TCP Bandwidth (h1 -> h16)	TCP Bandwidth (h16 -> h1)
POX	3.93 Gbps	3.93 Gbps
Ryu	8.39 Gbps	8.40 Gbps
ONOS	9.23 Gbps	9.23 Gbps
Odl	8.64 Gbps	8.64 Gbps

Από τους πίνακες 2.7 και 2.8 γίνεται αντιληπτό πως οι τιμές για το RTT σε λειτουργία διανομέα δεν διαφέρουν σημαντικά. Αυτό οφείλεται στην απλοποιημένη τοπολογία δικτύου που επιλέχθηκε συγκριτικά με κάποια πιο ρεαλιστική. Εντούτοις, σε λειτουργία μεταγωγέα μπορεί να παρατηρηθεί διαφορά στο χρόνο σύγκλισης του δικτύου, καθώς τα προωθητικά στοιχεία διαφημίζουν τους δικούς τους πίνακες ροών με τις σωστές εγγραφές έτσι ώστε η τιμή του RTT να είναι η ελάχιστη δυνατή. Ο ελεγκτής με τον ελάχιστο χρόνο σύγκλισης φαίνεται να είναι το ONOS, όπως ήταν αναμενόμενο καθώς αποτελεί ένα ολοκληρωμένο προϊόν παραγωγής σε αντίθεση με τον POX ή τον Ryu. Αξίζει να σημειωθεί και πάλι, πως οι διαφορές είναι μικρές λόγω της εξαιρετικά απλουστευμένης τοπολογίας.
Παρατηρώντας τα αποτελέσματα του τεστ iperf φαίνονται τεράστιες αποκλίσεις. Και πάλι το ONOS φαίνεται να έχει την καλύτερη απόδοση, με εύρος ζώνης που υπερβαίνει τα 9Gbps.

2.3.4 Ζητήματα ασφάλειας στα Δίκτυα Καθοριζόμενα από Λογισμικό

Τρωτά σημεία στον ελεγκτή

Τα τρωτά σημεία των ελεγκτών καθώς και οι απευθείας επιθέσεις σε αυτούς αποτελούν την πιο σοβαρή απειλή για τα SDN δίκτυα [3]. Ένας ελαττωματικός ελεγκτής ή ένας με κακόβουλο λογισμικό θα μπορούσε να θέσει σε κίνδυνο ολόκληρο το δίκτυο. Η χρήση συμβατικών συστημάτων ανίχνευσης εισβολής δεν φαίνεται να είναι επαρκής λύση, καθώς θα ήταν δύσκολο να εντοπιστεί ο συνδυασμός των γεγονότων που πυροδοτούν μία συγκεκριμένη συμπεριφορά αλλά ακόμη πιο δύσκολο θα ήταν να ονομαστεί αυτή κακόβουλη. Επιπλέον, μία κακόβουλη εφαρμογή θα μπορούσε να λειτουργήσει ελεύθερα στο δίκτυο, καθώς οι ελεγκτές παρέχουν μόνο τρόπο δημιουργίας εντολών παραμετροποίησης των δικτυακών συσκευών.

Ως πιθανές λύσεις θα μπορούσαν να εφαρμοστούν αρκετές τεχνικές όπως η αντιγραφή του ελεγκτή για τον εντοπισμό και το φιλτράρισμα της ασυνήθιστης συμπεριφοράς, η χρήση διαφορετικών ελεγκτών, πρωτοκόλλων, λογισμικών και η περιοδική ανάκτηση του συστήματος σε μία αξιόπιστη κατάσταση.

Τρωτά σημεία στα επίπεδα ελέγχου και δεδομένων

Είναι γνωστό, πως η χρήση των πρωτοκόλλων TSL/SSL δεν εξασφαλίζει την ασφάλεια στην επικοινωνία και αυτό κατά συνέπεια θέτει σε κίνδυνο τη ζεύξη ελεγκτή – συσκευής. Η ασφάλεια στα τηλεπικοινωνιακά συστήματα που χρησιμοποιούν αυτά τα πρωτόκολλα είναι τόσο ισχυρή όσο η πιο αδύναμη ζεύξη τους. Για παράδειγμα, παγκοσμίως γίνονται πολλές επιθέσεις τύπου «*man in the middle*» με το πρωτόκολλο SSL. Επιπλέον, αυτό το μοντέλο δεν αρκεί για την εγκαθίδρυση ασφαλούς σύνδεσης ελεγκτή και μεταγωγέων. Όταν κάποιος κάνει επίθεση αποκτώντας πρόσβαση στο επίπεδο ελέγχου, μπορεί να συγκεντρώσει αρκετή δύναμη, δηλαδή μεγάλο αριθμό μεταγωγέων υπό τον ελεγκτή, έτσι ώστε να ξεκινήσει Επιθέσεις Άρνησης Υπηρεσιών (*Denial of Service attack –DDoS*). Αυτή η έλλειψη πιστοποιητικών εμπιστοσύνης μπορεί να οδηγήσει ακόμα και στη δημιουργία εικονικού δικτύου - μαύρης τρύπας που θα επιτρέπει τη διαρροή δεδομένων παράλληλα με τη κανονική ροή της κίνησης.

3

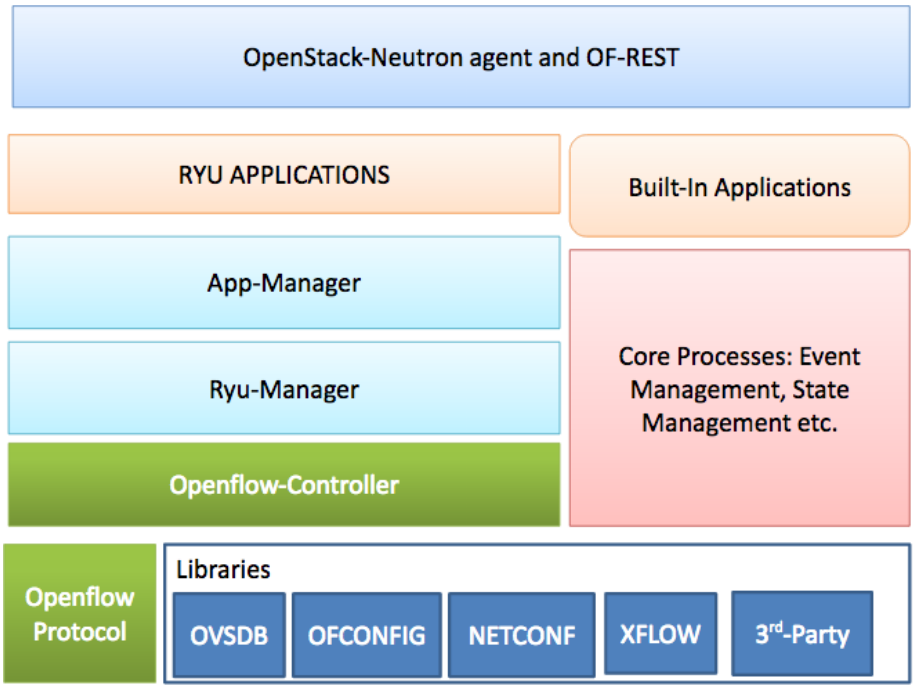
Ο Ελεγκτής Ryu

Ο RYU αποτελεί έναν αρκετά διαδεδομένο ελεγκτή ανοιχτού κώδικα σε εφαρμογές Δικτύων Καθοριζόμενων από Λογισμικό (*SDN*). Είναι σχεδιασμένος για να διευκολύνει τη διαχείριση του δικτύου και της κίνησης σε αυτό καθώς και την ανάπτυξη λογισμικού παρέχοντας τμήματα λογισμικού (*components*) με πλήρως ορισμένες Διεπαφές Προγραμματισμού Εφαρμογών (*APIs*). Ο RYU είναι γραμμένος εξ ολοκλήρου σε Python. Ένα από τα ισχυρά του πλεονεκτήματα είναι ότι υποστηρίζει πολλαπλά πρωτόκολλα για τη διαχείριση συσκευών (*southbound protocols*), όπως OpenFlow, Network Configuration Protocol (NETCONF), OpenFlow Management and Configuration Protocol (OF-Config) και άλλα.

Ο συγκεκριμένος ελεγκτής επιλέχθηκε μεταξύ των OpenDaylight και ONOS, που περιεγράφηκαν στις παραγράφους 2.3.1 και 2.3.2 αντίστοιχα, για την υλοποίηση της παρούσας εργασίας. Η επιλογή αυτή έγινε αξιολογώντας τα κριτήρια κατηγοριοποίησης των ελεγκτών που αναφέρονται στην παράγραφο 2.3, δίνοντας έμφαση στην ανάγκη χρήσης ενός ελεγκτή που απαιτεί λίγο χρόνο εκμάθησης και εμφανίζει καταλληλότητα για ερευνητικές/πειραματικές εφαρμογές. Ακόμη, λήφθηκε υπόψη η ικανότητα του RYU να διευκολύνει τον προγραμματιστή στην ανάπτυξη εφαρμογών στους τομείς της Δρομολόγησης (*Routing*) και της Εποπτείας των Δικτύων (*Network Monitoring*) (Πίνακας 2.3). Υπενθυμίζεται, πως σύμφωνα με την μελέτη των επιδόσεων των ελεγκτών OpenDaylight, ONOS και RYU που παρουσιάστηκε στην παράγραφο 2.3.3 [6], οι προαναφερθέντες ελεγκτές σε μία απλοποιημένη τοπολογία δεν εμφανίζουν σημαντικές διαφορές επιδόσεων. Δόθηκε έμφαση, λοιπόν, στην επιλογή ενός ελεγκτή που θα διευκολύνει την ανάπτυξη της εφαρμογής χωρίς να παρουσιάζει περιττή πολυπλοκότητα.

3.1 Η αρχιτεκτονική του Ryu

Όπως κάθε SDN ελεγκτής, ο RYU μπορεί να δημιουργήσει και να στείλει μηνύματα OpenFlow, να ακούσει ασύγχρονα γεγονότα (π.χ. αφαίρεση κάποιας ροής – flow), να αναλύσει και να διαχειριστεί εισερχόμενα πακέτα. Στο παρακάτω σχήμα απεικονίζεται η αρχιτεκτονική του πλαισίου του ελεγκτή RYU.



Σχήμα 3.1 Η αρχιτεκτονική του Ryu

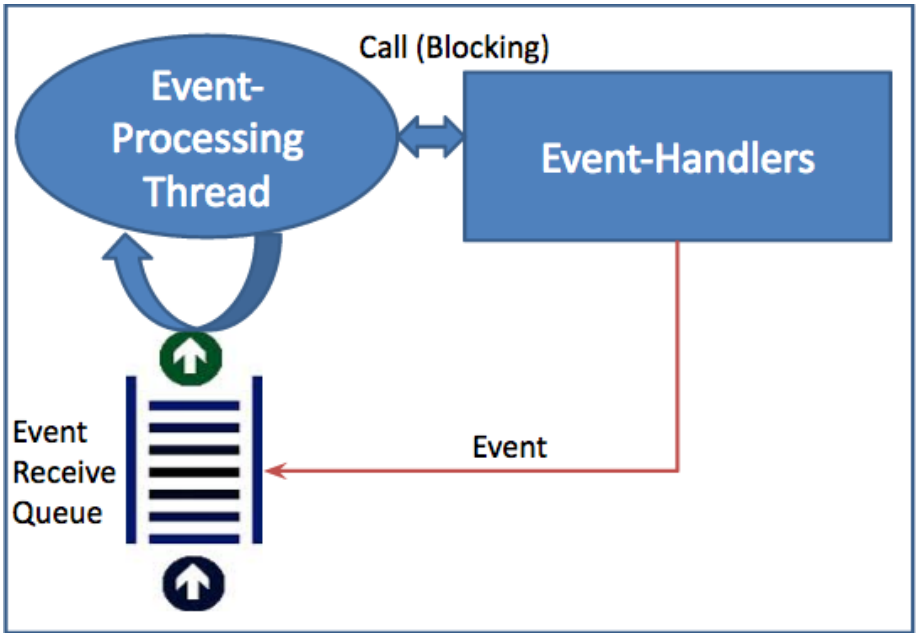
Στοιχεία της παραπάνω αρχιτεκτονικής:

- ***Βιβλιοθήκες (Libraries)***: Περιέχει μια εντυπωσιακή συλλογή από βιβλιοθήκες που αφορούν από την υποστήριξη των «southbound» πρωτοκόλλων μέχρι τη λειτουργία της επεξεργασίας των δικτυακών πακέτων.
- ***Openflow Protocol & Openflow-Controller***: Ο RYU υποστηρίζει έως και την τελευταία έκδοση (1.4) του πρωτοκόλλου OpenFlow και περιλαμβάνει βιβλιοθήκη κωδικοποίησης και αποκωδικοποίησης του πρωτοκόλλου αυτού. Επιπλέον, ένα από τα κύρια στοιχεία της αρχιτεκτονικής αυτής είναι ο ελεγκτής OpenFlow. Αυτός είναι υπεύθυνος για τη διαχείριση των OpenFlow μηνυμάτων που χρησιμοποιούνται για την παραμετροποίηση των ροών (*flows*), τη διαχείριση των γεγονότων κ.ο.κ. Στον παρακάτω πίνακα, παρουσιάζεται η σύνοψη των μηνυμάτων OpenFlow του ελεγκτή RYU.

Controller to Switch Messages	Asynchronous Messages	Symmetric Messages	Structures
Handshake, switch-config, flow-table-config, modify/read state, queue-config, packet-out, barrier, role-request	Packet-in, flow-removed, port-satus, and Error.	Hello, Echo-Request & Reply, Error, experimenter	Flow-match
send_msg API and packet builder APIs	set_ev_cls API and packet parser APIs	Both Send and Event APIs	

Σχήμα 3.2 Μηνύματα OpenFlow

- ***Managers & Core – Processes***: Το κύριο εκτελέσιμο πρόγραμμα καλείται ryu-manager. Όταν καλείται ακούει σε συγκεκριμένη διεύθυνση IP και σε συγκεκριμένη θύρα, την 6633 όπως είναι προκαθορισμένο. Κατά συνέπεια, κάθε συμβατός με το πρωτόκολλο OpenFlow μεταγωγέας μπορεί να συνδεθεί στον ελεγκτή. Η εφαρμογή manager είναι το θεμελιώδες τμήμα λογισμικού (*component*) για όλες τις εφαρμογές του Ryu, όλες κληρονομούν την κλάση RyuApp από την εφαρμογή manager.
- ***RYU Northbound***: Στο στρώμα της Διεπαφής Προγραμματισμού εφαρμογών (*API layer*) ο Ryu περιλαμβάνει μία επέκταση λογισμικού (*plug-in*) που ονομάζεται OpenStack Neutron και υποστηρίζει την παραμετροποίηση για τη δημιουργία VLAN και GRE τοπολογίας. Επιπλέον, ο ελεγκτής Ryu υποστηρίζει τη δημιουργία διεπαφής REST (*Representational State Transfer*).
- ***RYU Applications***: Ο ελεγκτής Ryu αποτελεί κατανεμημένο λογισμικό με πολλαπλές εφαρμογές όπως είναι οι ακόλουθες, “simple_switch”, “router”, “isolation”, “firewall”, “GRE tunnel”, “topology”, “VLAN”. Οι εφαρμογές του Ryu κατά τη λειτουργία τους στέλνουν ασύγχρονα μηνύματα μεταξύ τους. Η λειτουργία μιας εφαρμογής του Ryu παρουσιάζεται αναλυτικά στην παρακάτω εικόνα:



Σχήμα 3.3 Λειτουργία εφαρμογής Ryu

Κάθε εφαρμογή διατηρεί μία ουρά για τα γεγονότα, το πρώτο που συμβαίνει, εξυπηρετείται και πρώτο (*First in First Out, FIFO*). Κάθε γεγονός που εξέρχεται από την ουρά υφίσταται την κατάλληλη επεξεργασία και στη συνέχεια καλείται ο κατάλληλος διαχειριστής γεγονότων (*event handler*) ο οποίος μπλοκάρει την υπόλοιπη διαδικασία μέχρι να ολοκληρωθεί η εργασία του. Η εκτέλεση κάθε εφαρμογής στον Ryu γίνεται δίνοντας στον ryu – manager το αντίστοιχο αρχείο παραμετροποίησης της (python script):

```
Ryu-manager [--flagfile <path to configuration file>]
[generic/application specific options… ]
```

Μία από τις πιο σημαντικές επιλογές που μπορεί να δοθεί στην προαναφερθείσα εντολή είναι η **app_lists**, η οποία περιλαμβάνει τα ονόματα των δομοστοιχείων (*modules*) προς εκτέλεση.

3.2 Δημιουργία εφαρμογών στον Ryu

Μία εφαρμογή RYU αποτελεί ένα δομοστοιχείο (*module*) Python. Πρόκειται για μονού-νήματος (*single-threaded*) οντότητες που μπορούν να δώσουν ποικίλες λειτουργικότητες στον Ryu, που από μόνος του δεν έχει καμία και περιορίζει τη λειτουργία του υφιστάμενου δικτύου στην προώθηση οποιουδήποτε πακέτου στον ελεγκτή με τη δημιουργία γεγονότος PacketIn. Σημειώνεται, πως ως γεγονός ορίζεται ένα μήνυμα μεταξύ εφαρμογής και ελεγκτή.

3.2.1 Προγραμματιστικό Μοντέλο Ryu

Οι εφαρμογές του Ryu στέλνουν μεταξύ τους ασύγχρονα γεγονότα. Πηγή γεγονότων μπορεί, όμως, να υπάρξει και εσωτερικά στον ελεγκτή, για παράδειγμα από το πρωτόκολλο OpenFlow.
Κάθε εφαρμογή διατηρεί ουρά τύπου FIFO (*First In First Out*) για τα γεγονότα, προκειμένου να τηρήσει την προτεραιότητα εξυπηρέτησης σε αυτά που προηγούνται χρονικά. Επιπλέον, η εκάστοτε εφαρμογή δημιουργεί ένα νήμα (*thread*) επεξεργασίας των γεγονότων. Ρόλος του νήματος είναι η αποδέσμευση της ουράς σταδιακά από τα γεγονότα με την κλήση του αρμόδιου διαχειριστή (*event handler*).
Ένα δομοστοιχείο-εφαρμογή αποτελεί υποκλάση της κλάσης ryu.base.app_manager. Ryu.App. Εάν οριστούν δύο ή περισσότερες τέτοιες υπό-κλάσεις, τότε επιλέγεται αυτή που προηγείται αλφαβητικά.
Η παρατήρηση συγκεκριμένων γεγονότων από μία εφαρμογή και η ενεργοποίηση κάποιας μεθόδου σε συγκεκριμένες περιπτώσεις γίνεται με τη χρήση ενός διακοσμητή Python (*decorator*), του *@set_ev_cls*, στη μέθοδο *ryu.controller.handler*. Ενώ, η δημιουργία γεγονότος γίνεται με τη χρήση της κατάλληλης μεθόδου της κλάσης *ryu.base.app_manager.RyuApp* (π.χ. *send_event* ή *send_event_to_observers*). Διευκρινίζεται, πως μία μέθοδος, τροποποιημένη από κάποιο διακοσμητή, γίνεται διαχειριστής γεγονότος αν δεχθεί ως όρισμα κάποιον ελεγκτή κυκλοφορίας (*dispatcher*) που ορίζει το στάδιο της διαπραγμάτευσης και μπορεί να είναι ένα από τα ακόλουθα:

- ***HANDSHAKE_DISPATCHER:*** Αποστολή και αναμονή μηνύματος *Hello* (έναρξη διαπραγμάτευσης).
- ***CONFIG_DISPATCHER:*** Αποστολή αιτήματος για λήψη χαρακτηριστικών από την δικτυακή συσκευή.
- ***MAIN_DISPATCHER:*** Λήψη από τον ελεγκτή του μηνύματος χαρακτηριστικών ή αποστολή παραμετροποίησης.
- ***DEAD_DISPATCHER:*** Αποσύνδεση της συσκευής.

3.2.2 Εφαρμογή Switching Hub

Μία από τις πιο βασικές εφαρμογές, που χρησιμοποιείται από πολλές άλλες, αποτελεί η εφαρμογή Switching Hub. Παρακάτω αναλύεται προκειμένου να γίνει σαφής ο τρόπος δημιουργίας τέτοιων εφαρμογών σε συνδυασμό με όσα αναφέρθηκαν στην προηγούμενη παράγραφο.
Μία Switching Hub εφαρμογή περιλαμβάνει τις ακόλουθες λειτουργίες:

- Μαθαίνει τη διεύθυνση MAC ενός host που είναι συνδεδεμένος σε κάποια θύρα του μεταγωγέα και τη διατηρεί σε σχετικό πίνακα.

- Όταν λαμβάνει πακέτο με προορισμό κάποιο συνδεδεμένο host του οποίου τη διεύθυνση έχει αποθηκευμένη στον προαναφερθέντα πίνακα, προωθεί το πακέτο σε εκείνον.
- Όταν λαμβάνει πακέτο με άγνωστη διεύθυνση προορισμού αντιγράφει και προωθεί το πακέτο προς όλες τις θύρες (τεχνική flooding).

Ένας μεταγωγέας OpenFlow μπορεί να εκτελέσει τα ακόλουθα λαμβάνοντας οδηγίες από κάποιον ελεγκτή σαν τον Ryu:

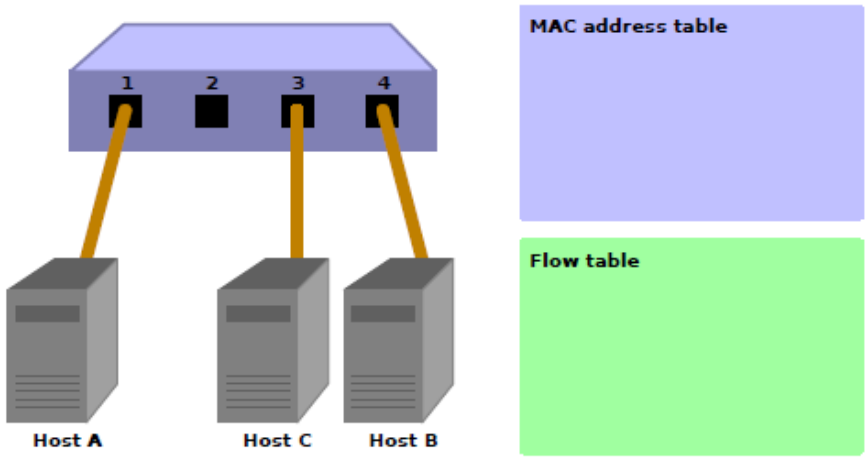
- Μπορεί να ξανά γράψει τη διεύθυνση του εισερχόμενου πακέτου ή να το προωθήσει.
- Μπορεί να μεταφέρει το εισερχόμενο πακέτο στον ελεγκτή (Packet In).
- Μπορεί να μεταφέρει το προωθημένο από τον ελεγκτή πακέτο (Packet Out).

Οι παραπάνω λειτουργίες μπορούν να συνδυαστούν σε μια εφαρμογή.
Αρχικά, χρησιμοποιείται η συνάρτηση packet-in για να γίνει γνωστή η διεύθυνση MAC. Αυτή η συνάρτηση χρησιμοποιείται, επίσης, από τον ελεγκτή για να δεχθεί πακέτα από τον μεταγωγέα. Ο μεταγωγέας αναλύει τα εισερχόμενα πακέτα για να μάθει τη διεύθυνση MAC του host και άλλες πληροφορίες για τη συνδεδεμένη θύρα. Στη συνέχεια, ο μεταγωγέας μεταφέρει τα ληφθέντα πακέτα. Ο μεταγωγέας διερευνά αν η διεύθυνση MAC του προορισμού ανήκει σε μία από αυτές που γνωρίζει και αναλόγως κάνει μία από τις ακόλουθες διαδικασίες:

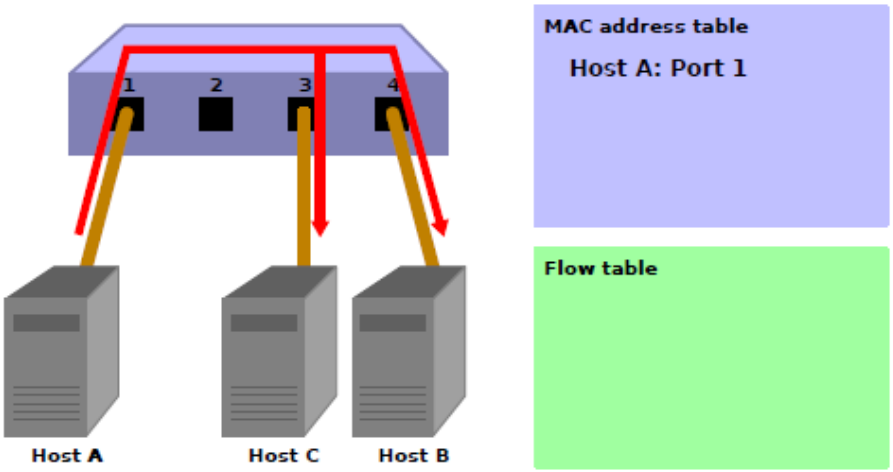
- Αν η διεύθυνση είναι γνωστή, τότε χρησιμοποιώντας τη συνάρτηση packet-out προωθεί το πακέτο μέσω της διασυνδεδεμένης θύρας
- Αν είναι άγνωστή με την προαναφερθείσα συνάρτηση πραγματοποιεί πλημμύρα.

Η παραπάνω διαδικασία παρουσιάζεται ακολούθως και σχηματικά:

1ο Στάδιο – Άδειος πίνακας διευθύνσεων MAC:



2ο Στάδιο – Ο Host A στέλνει στον Host B:



Packet-In:

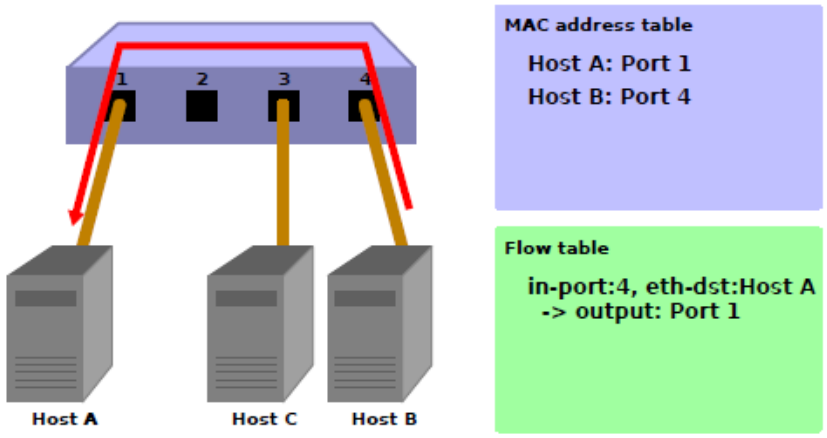
```
in-port: 1
eth-dst: Host B
eth-src: Host A
```

Packet-Out:

```
action: OUTPUT:Flooding
```

Όταν ο Α στέλνει στον Β αποστέλλεται μήνυμα packet-in και γίνεται γνωστή η διεύθυνση του Α από την θύρα 1. Καθώς ο πίνακας είναι ακόμη κενός, τα πακέτα αποστέλλονται με πλημμύρα στους C και B.

3ο Στάδιο – Ο Host B στέλνει στον Host A:



Packet-In:

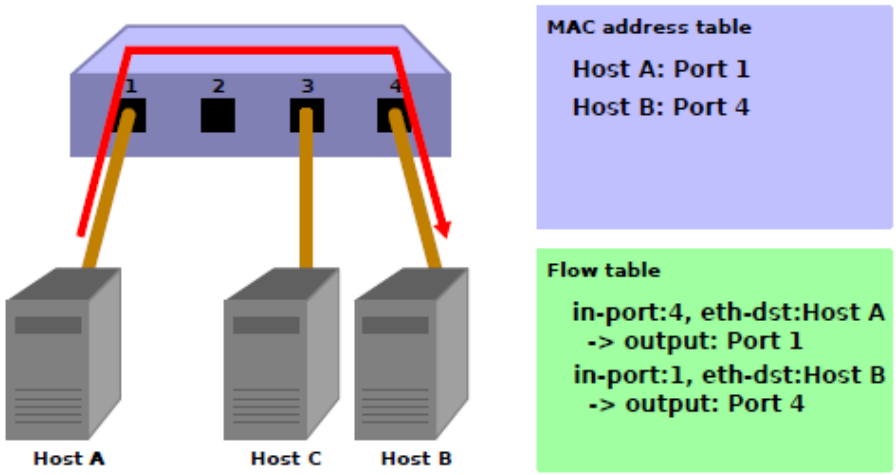
```
in-port: 4
eth-dst: Host A
eth-src: Host B
```

Packet-Out:

```
action: OUTPUT:Port 1
```

Όταν τα πακέτα επιστρέφονται από τον B στον A, γίνεται προσθήκη στον πίνακα και τα πακέτα μεταφέρονται στη θύρα 1. Για αυτό το λόγο τα πακέτα δεν λαμβάνονται από τον C.

4ο στάδιο – Ο Host A στέλνει στον Host B:



Packet-In:

```
in-port: 1
eth-dst: Host B
eth-src: Host A
```

Packet-Out:

```
action: OUTPUT:Port 4
```

3.2.2.1 Υλοποίηση της εφαρμογής Switching Hub με τον ελεγκτή Ryu

Ορισμός κλάσεων και αρχικοποίηση

Προκειμένου να υλοποιηθεί η εφαρμογή σαν μία εφαρμογή Ryu κληρονομείται η κλάση ryu.base.app_manager.RyuApp. Ακόμη, για να χρησιμοποιηθεί η επιθυμητή έκδοση του πρωτοκόλλου OpenFlow, συνηθέστερα η έκδοση 1.3, ορίζεται η μεταβλητή OFP_VERSIONS. Επιπλέον, ορίζεται ο πίνακας των διευθύνσεων MAC mac_to_port.

```
class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    # ...
```

Διαχειριστής Γεγονότων

Η εφαρμογή Ryu, όταν κάποια συσκευή δέχεται μήνυμα OpenFlow, εφαρμόζει έναν διαχειριστή γεγονότων (event handler) που ανταποκρίνεται στο μήνυμα που πρέπει

να ληφθεί. Ο διαχειριστής αυτός ορίζει μία συνάρτηση που έχει ως όρισμα το αντικείμενο του γεγονότος και χρησιμοποιεί τον decorator ryu.controller.handler.set_ev_cls. Αυτός προσδιορίζει την κλάση που υποστηρίζει το ληφθέν μήνυμα και την κατάσταση του OpenFlow μεταγωγέα για το όρισμα.
Το όνομα της κλάσης είναι ryu.controller.ofp_event.EventOFP + *<OpenFlow message name>* . Για παράδειγμα, όταν το μήνυμα είναι εισερχόμενο το προαναφερθέν όνομα γίνεται ryu.controller.ofp_event.EventOFPPacketIn.

Προσθήκη εγγραφής που δεν υπάρχει στον πίνακα

Όταν ολοκληρώνεται η χειραψία (handshake) με τον OpenFlow μεταγωγέα, γίνεται προσθήκη στον πίνακα της εγγραφής που λείπει προκειμένου να γίνει αποδεκτό το εισερχόμενο μήνυμα. Συγκεκριμένα, η προσθήκη γίνεται μόλις ληφθεί το μήνυμα Switch Features (Features Reply).

```
@set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
def switch_features_handler(self, ev):
    datapath = ev.msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser
```

Η κλάση Datapath εκτελεί σημαντική επεξεργασία όπως πραγματική επικοινωνία με τον OpenFlow μεταγωγέα και έκδοση του γεγονότος που ανταποκρίνεται στο εισερχόμενο μήνυμα.
Ακολούθως παρουσιάζονται τα κύρια ορίσματα (*attributes*) που χρησιμοποιούνται από την εφαρμογή του Ryu:

Πίνακας 3.1 Attributes

Όρισμα	Επεξήγηση
Id	ID (datapath ID) του συνδεδεμένου OF μεταγωγέα
Ofproto	Υποδεικνύει το ofproto δομοστοιχείο που υποστηρίζει την έκδοση του OF πρωτοκόλλου που χρησιμοποιείται. Θα μπορούσε να είναι ένα από τα ακόλουθα: ryu.ofproto.ofproto_v1_0 ryu.ofproto.ofproto_v1_2 ryu.ofproto.ofproto_v1_3
ofproto_parser	Όπως και το ofproto, υποδεικνύει το δομοστοιχείο ofproto_parser. Θα μπορούσε να είναι ένα από τα ακόλουθα: ryu.ofproto.ofproto_v1_0_parser ryu.ofproto.ofproto_v1_2_parser ryu.ofproto.ofproto_v1_3_parser

Μήνυμα εισερχόμενου πακέτου (Packet-in message)

Δημιουργία χειριστή του γεγονότος Packet-In προκειμένου να γίνονται δεκτά πακέτα με άγνωστο προορισμό.

```
@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
def _packet_in_handler(self, ev):
    msg = ev.msg
    datapath = msg.datapath
    ofproto = datapath.ofproto
    parser = datapath.ofproto_parser

    # ...
```

Ακολούθως παρουσιάζονται ορίσματα της κλάσης OFPPacketIn που χρησιμοποιούνται συχνά:

Πίνακας 3.2 Attributes

Όρισμα	Επεξήγηση
match	ryu.ofproto.ofproto_v1_3_parser.OFPMatch κλάση στην οποία ορίζονται οι πληροφορίες από τα εισερχόμενα πακέτα
data	δυαδικά δεδομένα που υποδεικνύουν τα εισερχόμενα πακέτα
total_len	μήκος δεδομένων στα εισερχόμενα πακέτα
buffer_id	όταν εισερχόμενα πακέτα φορτώνονται σε προσωρινή μνήμη του μεταγωγέα, υποδεικνύει το ID του. Αν δεν συμβεί κάτι τέτοιο τότε αυτό γίνεται ryu.ofproto.ofproto_v1_3.OFP_NO_BUFFER

Ενημέρωση του πίνακα διευθύνσεων MAC

```
def _packet_in_handler(self, ev):
    # ...

    in_port = msg.match['in_port']

    pkt = packet.Packet(msg.data)
    eth = pkt.get_protocols(ethernet.ethernet)[0]

    dst = eth.dst
    src = eth.src

    dpid = datapath.id
    self.mac_to_port.setdefault(dpid, {})

    self.logger.info("packet in %s %s %s %s", dpid, src, dst, in_port)

    # learn a mac address to avoid FLOOD next time.
    self.mac_to_port[dpid][src] = in_port

    # ...
```

Λαμβάνεται η θύρα στην οποία εισέρχεται το πακέτο (*in_port*) από το πεδίο OFPPacketIn. Με χρήση της βιβλιοθήκης του Ryu αποκτώνται οι διευθύνσεις MAC αποστολέα και παραλήπτη από την επικεφαλίδα Ethernet. Βάσει της λαμβανόμενης

διεύθυνσης αποστολέα και της εισερχόμενης θύρας γίνεται η ανανέωση του πίνακα διευθύνσεων MAC.
Προκειμένου να είναι εφικτή η σύνδεση με πολλούς OpenFlow μεταγωγείς, σχεδιάζεται έτσι ο πίνακας διευθύνσεων MAC ώστε να είναι διαχειρίσιμος για κάθε μεταγωγέα ξεχωριστά. Για την διάκριση των μεταγωγέων γίνεται χρήση του ID του μονοπατιού δεδομένων (*datapath ID*).

Καθορισμός της θύρας αποστολής για τη μεταφορά του πακέτου

Όταν υπάρχει η διεύθυνση αποστολής στον πίνακα διευθύνσεων, γίνεται χρήση της θύρας που υποδεικνύεται, αλλιώς η τιμή του OUTPUN υποδεικνύει ενέργεια πλημμύρας (*OFPP_FLOOD*).

```
def _packet_in_handler(self, ev):
    # ...

    if dst in self.mac_to_port[dpid]:
        out_port = self.mac_to_port[dpid][dst]
    else:
        out_port = ofproto.OFPP_FLOOD

    actions = [parser.OFPActionOutput(out_port)]

    # install a flow to avoid packet_in next time
    if out_port != ofproto.OFPP_FLOOD:
        match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
        self.add_flow(datapath, 1, match, actions)

    # ...
```

Εάν βρεθεί η διεύθυνση αποστολής, γίνεται προσθήκη στον πίνακα ροής του *OpenFlow* μεταγωγέα. Όπως και στην περίπτωση που λείπει κάποια εγγραφή (όπως έχει προαναφερθεί) εκτελείται η συνάρτηση add_flow() για να γίνει προσθήκη ροής. Τελικά, για την προσθήκη ροής γίνεται χρήση του μηνύματος Flow Mod.

```
def add_flow(self, datapath, port, dst, actions):
    # ...

    mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                            match=match, instructions=inst)
    datapath.send_msg(mod)
```

Η κλάση που ανταποκρίνεται στον προαναφερθέντα τύπο μηνύματος είναι η OFPFlowMod. Με την δημιουργία αυτής της κλάσης γίνεται και αποστολή του μηνύματος στον OpenFlow μεταγωγέα με χρήση της μεθόδου datapath.send_msg().

Η εκτέλεση της προαναφερθείσας εφαρμογής γίνεται με χρήση της εντολής:

```
Ryu-manager - -verbose ryu.app.simple_switch_13
```

3.2.3 Εφαρμογή Spanning Tree

Η λειτουργία της συνάρτησης *Spanning Tree* αφορά την αποτροπή των κρουσμάτων πακέτων *broadcast* σε ένα δίκτυο που περιλαμβάνει βρόχους (*loops*). Με άλλα λόγια η εφαρμογή αυτής καταργεί τους βρόχους αποκλείοντας την ύπαρξη πλεοναζόντων διαδρομών (*redundancy*) διατηρώντας, όμως, τη δυνατότητα να χρησιμοποιηθούν οι πλεονάζουσες ζεύξεις σε περίπτωση που υπάρξει κάποιο πρόβλημα σε αυτές που έχουν προτιμηθεί και παραμείνει διαθέσιμες εξ αρχής. Το πρωτόκολλο *Spanning Tree* (*Spanning Tree Protocol - STP*) έχει αρκετούς τύπους (*STP, MSTP, RSTP, RVST+*) με πιο απλό το STP του οποίου η εφαρμογή προτιμάται στα πλαίσια αυτής της εργασίας. Προκειμένου, το *STP* να πραγματοποιήσει το στόχο του, αντιλαμβάνεται το δίκτυο ως ένα λογικό δέντρο και κατά συνέπεια αποκόβει λογικά μία ζεύξη και όχι φυσικά.
Σημειώνεται, πως σύμφωνα με την τεκμηρίωση του προαναφερθέντος πρωτοκόλλου, οι δικτυακές συσκευές στις οποίες αυτό εφαρμόζεται αναφέρονται ως γέφυρες (*bridges*) και όχι μεταγωγείς. Πρόκειται για σύμβαση που δεν συνεπάγεται απαραίτητα και την χρήση των συγκεκριμένων συσκευών στα δίκτυα που επιλέγεται η χρήση του STP. Υπενθυμίζεται, πως οι γέφυρες είναι δικτυακές προωθητικές συσκευές που διασυνδέουν δύο δικτυακές περιοχές που χρησιμοποιούν τα ίδια πρωτόκολλα. Η λειτουργία τους συνοψίζεται στη λήψη απόφασης που αφορά την κατεύθυνση στην οποία θα προωθηθεί το μήνυμα. Είναι απόγονοι των διανομέων (*hub*) και πρόγονοι των μεταγωγέων (*switch*).
Η λειτουργία του *STP* βασίζεται στην κυκλοφορία μηνυμάτων *BPDU* (*Bridge Protocol Data Unit*). Αυτά ανταλλάσσονται από τις γέφυρες έτσι ώστε να γίνει σύγκριση πληροφοριών που αφορούν τις γέφυρες και τις θύρες τους και να αποφασιστεί αν μπορεί η εκάστοτε ζεύξη να συμμετάσχει ενεργά στο δίκτυο. Πρόκειται για μία διαδικασία που περιλαμβάνει τα ακόλουθα στάδια:

1. Εκλογή της γέφυρας-ρίζας (*root-bridge*): Ορίζεται ως ρίζα αυτή με το μικρότερο αναγνωριστικό ID. Αυτή η γέφυρα, πλέον, είναι η μόνη που μπορεί να δημιουργήσει μήνυμα *BPDU*, το προωθεί στις υπόλοιπες και αυτές το αναπαράγουν με τη σειρά τους. Σημειώνεται, πως το ID της γέφυρας υπολογίζεται συνδυάζοντας στα πρώτα δύο Byte τον αριθμό προτεραιότητας της γέφυρας και στα 6 επόμενα τη διεύθυνση MAC της γέφυρας.

2. Ανάθεση ρόλων στις θύρες: Σύμφωνα με το κόστος της διαδρομής από την θύρα που εξετάζεται μέχρι τη ρίζα, αποφασίζεται ποιόν από τους παρακάτω ρόλους θα αναλάβει:

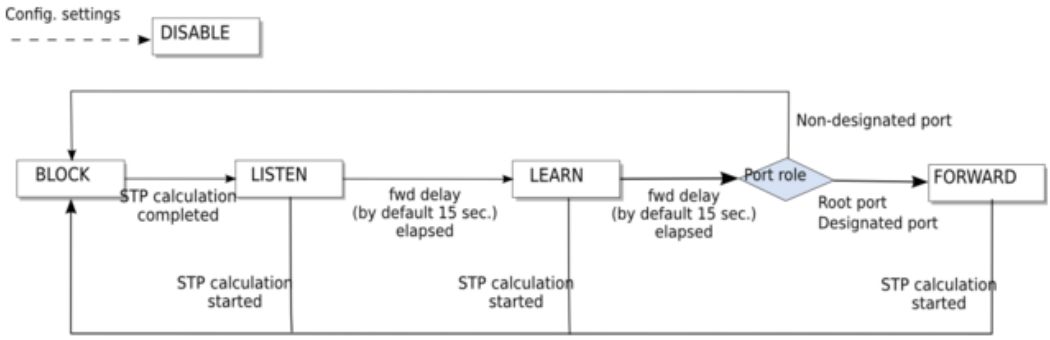
- Θύρα-ρίζα (*root port*): αυτή με το μικρότερο κόστος προς τη γέφυρα ρίζα. Αυτή λαμβάνει τα μηνύματα και τα προωθεί στις υπόλοιπες.

- Εφεδρική καθορισμένη (*designated port*): κάθε συσκευή εντοπίζει αυτή με το μικρότερο κόστος προς τη ρίζα. Η θύρα σε αυτή τη κατεύθυνση ορίζεται ως *designated*.
- Μη εφεδρική καθορισμένη (*not designated port*): όλες οι υπόλοιπες θύρες στο δίκτυο. Αυτές είναι οι μόνες που δεν προωθούν μηνύματα BPDU.

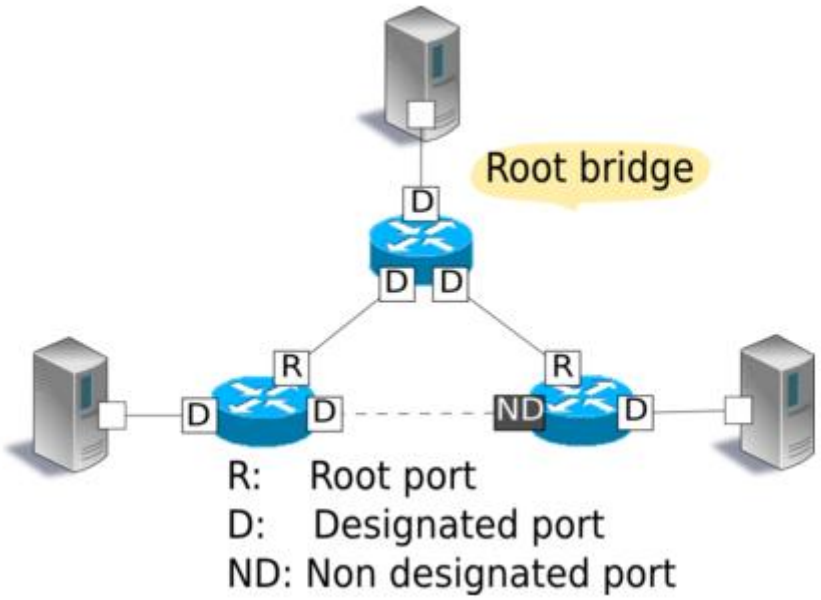
3. Αλλαγή της κατάστασης της θύρας: μόλις ολοκληρωθεί η ανάθεση ρόλων στις θύρες η κάθε μία εισέρχεται σε κατάσταση *Listen*. Στη συνέχεια η αλλαγή γίνεται όπως φαίνεται στο σχήμα 3.4 που ακολουθεί. Τελικά, η κατάσταση μπορεί να είναι *Forward*, *Block* ή *Disabled.* Η λειτουργία μία θύρας ανάλογα με την κατάσταση της μπορεί να είναι:

- Disable: Αγνοεί κάθε εισερχόμενο πακέτο.
- Block: Μόνο λαμβάνει BPDU μηνύματα.
- Listen: Στέλνει και λαμβάνει BPDU μηνύματα.
- Learn: Στέλνει και λαμβάνει BPDU μηνύματα και μαθαίνει διευθύνσεις MAC.
- Forward: Στέλνει και λαμβάνει BPDU μηνύματα, μαθαίνει διευθύνσεις MAC και μεταφέρει πλαίσια.

Κάθε φορά που εντοπίζεται διακοπή σε κάποια ζεύξη λόγω κάποια αποτυχίας ή δεν κυκλοφορούν μηνύματα BPDU για το μέγιστο διάστημα που έχει οριστεί, επαναλαμβάνονται τα βήματα 1 έως 3 για να οριστούν εκ νέου όλα τα παραπάνω.



Σχήμα 3.4 Διαδικασία ορισμού της κατάστασης των θυρών κατά τη λειτουργία του πρωτοκόλλου STP



Σχήμα 3.5 Οι ρόλοι των θυρών σε ένα δίκτυο όπου λειτουργεί το πρωτόκολλο STP

Η λειτουργία που περιεγράφηκε αποτελεί συνάρτηση που ενσωματώνεται στην εφαρμογή *Switching Hub*. Ο Ryu με την εγκατάσταση του παρέχει έτοιμη αυτή την εφαρμογή για την πρώτη έκδοση του πρωτοκόλλου OpenFlow και χρειάζεται επέκταση για να υποστηρίζει και τις νεότερες.

3.3 Διεπαφή Προγραμματισμού Εφαρμογών πρωτοκόλλου OpenFlow

Η κλάση ***ryu.ofproto.ofproto_parser.MsgBase(*args, **kwargs)*** είναι βασική για τις κλάσεις μηνυμάτων *OpenFlow*. Ένα στιγμιότυπο αυτής πρέπει να έχει τουλάχιστον τις ακόλουθες ιδιότητες:

- ***Datapath:*** ένα στιγμιότυπο ryu.controller.controller.Datapath
- ***Version:*** η έκδοση του πρωτοκόλλου OpenFlow
- ***Msg_type:*** ο τύπος του μηνύματος
- ***Msg_len:*** το μήκος του μηνύματος
- ***Xid:*** το ID της συναλλαγής
- ***Raw:*** ακατέργαστα δεδομένα

3.3.1 Μηνύματα OpenFlow και η δομή αυτών

Όπως αναφέρθηκε και προηγουμένως (Σχήμα 3.2) το πρωτόκολλο OpenFlow υποστηρίζει τρεις τύπους μηνυμάτων σε ένα δίκτυο SDN. Ακολούθως, παρουσιάζονται αναλυτικότερα τα σημαντικότερα από αυτά.

Μηνύματα με πρωτοβουλία του ελεγκτή

Αρχικό στάδιο της επικοινωνίας αυτής (Ελεγκτής ↔ Μεταγωγέας) είναι η χειραψία (*handshake*). Ο ελεγκτής στέλνει αίτημα αφού έχει ανοίξει κανάλι επικοινωνίας μεταξύ των δύο οντοτήτων:

Class ryu.ofproto.ofproto_v1_0_parser.OFPFeaturesRequest(*datapath*)

Κατόπιν, ανταποκρίνεται ο μεταγωγέας με την απάντηση του:

Class ryu.ofproto.ofproto_v1_0_parser.OFPSwitchFeatures(*datapath, datapath_id=None, n_buffers=None, n_tables=None, capabilities=None, actions=None, ports=None)*

Όπου τα ορίσματα έχουν την ακόλουθη ερμηνεία:

- ***Datapath_id:*** το μοναδικό ID του μονοπατιού δεδομένων
- ***N_buffers:*** ο μέγιστος αριθμός πακέτων που μπορούν να φορτωθούν στην προσωρινή μνήμη ταυτόχρονα
- ***N_tables:*** ο αριθμός των πινάκων που υποστηρίζει το μονοπάτι δεδομένων
- ***Capabilities:*** σειρά από bit της σημαίας ικανοτήτων (*OFPC_FLOW_STATS, OFPC_TABLE_STATS, OFPC_PORT_STATS, OFPC_STP, OFPC_RESERVED, OFPC_IP_REASM, OFPC_QUEUE_STATS, OFPC_ARP_MATCH_IP*)

Ο ελεγκτής μπορεί, ακόμη, να στείλει αίτημα για να προβεί σε παραμετροποίηση της συσκευής:

Class ryu.ofproto.ofproto_v1_0_parser.OFPSetConfig(*datapath, flags=None, miss_send_len=None)*

Όπου τα ορίσματα είναι τα ακόλουθα:

- ***Flags:*** *OFPC_FRAG_NORMAL, OFPC_FRAG_DROP, OFPC_FRAG_REASM, OFPC_FRAG_MASK*
- ***Miss_send_len:*** Μέγιστος αριθμός bytes της καινούργιας ροής που το μονοπάτι δεδομένων πρέπει να στείλει στον ελεγκτή

Ο ελεγκτής έχει επιπλέον τη δυνατότητα να αιτηθεί να λάβει την υπάρχουσα παραμετροποίηση της συσκευής:

Class ryu.ofproto.ofproto_v1_0_parserOFPGetConfigRequest(*datapath)*

Με τη σειρά του ο μεταγωγέας ανταποκρίνεται ως εξής:

Class ryu.ofproto.ofproto_v1_0.parserOFPGetConfigReply(*datapath)*

Προκειμένου να τροποποιήσει τον πίνακα ροής ο ελεγκτής στέλνει το ακόλουθο μήνυμα:

Class ryu.ofproto.ofproto_v1_0.parser.PFPFlowMod(*datapath, match, cookie, command, idle_timeout=0, hard_timeout=0, priority=32768, buffer_id= 4294967295, out_port=65535, flags=0, actions=None)*

Με τα ακόλουθα ορίσματα:

- ***Match:*** στιγμιότυπο του OFPMatch
- ***Cookie:*** αδιαφανές αναγνωριστικό που εκδίδεται από τον ελεγκτή
- ***Command:*** *OFPFC_ADD, OFPFC_MODIFY, OFPFC_MODIFY_STRICT, OFPFC_DELETE, OFPFC_DELETE_STRICT*
- ***Idle_timeout:*** διάστημα αδράνειας πριν την απόρριψη (sec)
- ***Hard_timeout:*** μέγιστο χρονικό διάστημα πριν την απόρριψη (sec)
- ***Priority:*** το επίπεδο προτεραιότητας της ροής που εισάγεται τον πίνακα
- ***Flags:*** *OFPFF_SEND_FLOW_REM, OFPFF_CHECK_OVERLAP, OFPFF_EMERG*

Η τροποποίηση της κατάστασης μίας θύρας γίνεται με το ακόλουθο μήνυμα:

Class ryu.ofproto.ofproto_v1_0_parser.OFPPortMod(*datapath, pont_no=0, hw_addr='00:00:00:00:00:00', config=0, mask=0, advertise=0)*

Με ορίσματα:

- ***Port_no:*** αριθμός πόρτας, της οποίας η κατάσταση τροποποιείται
- ***Hw_addr:*** η διεύθυνση του υλικού που πρέπει να ταυτίζεται με τη διεύθυνση OFPPhyPort και OFPSwitchFeature.
- ***Config:*** OFPPC_PORT_DOWN, OFPPC_NO_STP, OFPPC_NO_RECV, OFPPC_NO_RECV_STP, OFPPC_NO_FLOOD, OFPPC_NO_FWD, OFPPC_NO_PACKET_IN
- ***Mask:*** σειρά από bits για τις σημαίες παραμετροποίησης που θα αλλάξουν
- ***Advertise:*** OFPPF_10MB_HD, OFPPF_10MB_FD, OFPPF_100MB_HD, OFPPF_100MB_FD, OFPPF_1GB_HD, OFPPF_1GB_FD, OFPPF_10GB_FD, OFPPF_COPPER, OFPPF_FIBER, OFPPF_AUTONEG, OFPPF_PAUSE, OFPPF_PAUSE_ASYM

Υπάρχει, ακόμη, η δυνατότητα να αιτηθεί η περιγραφή της κατάστασης ενός μεταγωγέα και να επιστραφεί αυτή με αντίστοιχα μηνύματα:

Class ryu.ofproto.ofproto_v1_0.parser.OFPDescStatRequest(*datapath,flags)*

Class ryu.ofproto.ofproto_v1_0.parser.OFPDescStatReply(*datapath)*

Πιο συγκεκριμένα, ο ελεγκτής μπορεί να ζητήσει εξατομικευμένα στατιστικά δεδομένα και να του τα αποστείλει ο μεταγωγέας:

Class ryu.ofproto.ofproto_v1_0.parser.OFPStatsRequest(*datapath, flags, match, table_id, out_port)*

Class ryu.ofproto.ofproto_v1_0.parserOFPStatsReply(*datapath*)

Όπου τα παραπάνω ορίσματα αφορούν:

- ***Match:*** στιγμιότυπο του OFPMatch
- ***Table_id:*** ID του πίνακα προς ανάγνωση, 0xff για όλους του πίνακες ή 0xfe για επείγουσα περίπτωση

Υπάρχει η δυνατότητα αιτήματος συγκεντρωτικών στατιστικών:

class ryu.ofproto.ofproto_v1_0.parser.OFPAggregateStatsRequest(*datapath, flags, match, table_id, out_port*)

class ryu.ofproto.ofproto_v1_0.parser.OFPAggregateStatsReply(*datapath*)

Για τις πληροφορίες που περιλαμβάνει ο πίνακας ροής:

class ryu.ofproto.ofproto_v1_0.parser.OFPTableStatsRequest(*datapath,flags*)

Class ryu.ofproto.ofproto_v1_0.parser.OFPTableStatsReply (*datapath*)

Για τα στατιστικά στοιχεία που αφορούν κάποια θύρα τα μηνύματα είναι τα ακόλουθα:

class ryu.ofproto.ofproto_v1_0.parser.OFPPortStatsRequest(*datapath,flags, port_no*)

class ryu.ofproto.ofproto_v1_0.parser.OFPPortStatsReply(*datapath*)

Το προαναφερθέν απαντητικό μήνυμα περιλαμβάνει τα ακόλουθα:

- ***Port_no:*** αριθμός θύρας
- ***Rx_packets:*** αριθμός εισερχόμενων πακέτων
- ***Tx_packets:*** αριθμός απεσταλμένων πακέτων
- ***Rx_bytes:*** αριθμός εισερχόμενων bytes
- ***Tx_bytes:*** αριθμός απεσταλμένων bytes
- ***Rx_dropped:*** αριθμός εισερχόμενων πακέτων που έχουν χαθεί
- ***Tx_dropped:*** αριθμός απεσταλμένων πακέτων που έχουν χαθεί

- ***Rx_errors:*** αριθμός λαθών που έχουν ληφθεί
- ***Tx_errors:*** αριθμός λαθών που έχουν αποσταλεί
- ***Rx_frame_err:*** αριθμός λαθών στη στοίχιση των πλαισίων
- ***Rx_over_err:*** αριθμός εισερχόμενων πακέτων που απορρίπτονται λόγω υπερχείλισης στην προσωρινή μνήμη
- ***Rx_crc_err:*** εισερχόμενα πακέτα με λάθη στο πεδίο ελέγχου crc
- ***Collisions:*** αριθμός συγκρούσεων κατά τη μετάδοση πακέτων

Κατά τον ίδιο τρόπο δημιουργούνται μηνύματα με τις εξής σκοπιμότητες: (α)στατιστικά στοιχεία για τις ουρές αναμονής, (β)στατιστικά στοιχεία για τον κατασκευαστή της συσκευής, (γ) αποστολή μηνύματος δια μέσου κάποιου μεταγωγέα και (δ) επιβεβαίωση για τις ολοκληρωμένες λειτουργίες εκ μέρους του μεταγωγέα.

Ασύγχρονα μηνύματα

Ένας μεταγωγέας μπορεί να στείλει πακέτο το οποίο έχει λάβει με το ακόλουθο μήνυμα:

Class ryu.ofproto.ofproto_v1_0_parser.OFPPacketIn(*datapath,buffer_id=None, total_len=None, in_port=None, reason=None, data=None*)

Με ορίσματα:

- ***Reason:*** τον λόγο για τον οποίο στέλνεται το πακέτο (OFPR_NO_MATCH, OFPR_ACTION, OFPR_INVALID_TTL)
- ***Data:*** το πλαίσιο Ethernet

Ακόμη, μπορεί να στείλει μήνυμα για να ενημερώσει τον ελεγκτή για την αφαίρεση κάποιας ροής λόγω εκπνοής χρόνου ή λόγω διαγραφής:

class ryu.ofproto.ofproto_v1_0_parser.OFPFlowRemoved(*datapath*)

Το μήνυμα αυτό περιλαμβάνει τα ακόλουθα ορίσματα:

- ***Match:*** στιγμιότυπο του OFPMatch
- ***Cookie:*** αδιαφανές αναγνωριστικό που εκδίδεται από τον ελεγκτή
- ***Priority:*** επίπεδο προτεραιότητας της εγγραφής
- ***Reason:*** OFPRR_IDLE_TIMEOUT, OFPRR_HARD_TIMEOUT, OFPRR_DELETE
- ***Duration_sec:*** το χρονικό διάστημα για το οποίο υπήρχε η εγγραφή στον πίνακα (*seconds)*
- ***Duration_nsec:*** το χρονικό διάστημα για το οποίο υπήρχε η εγγραφή στον πίνακα σε δευτερόλεπτα (*nanoseconds)*
- ***Packet_count:*** αριθμός πακέτων που συσχετίστηκε με την εγγραφή

- ***Byte_count:*** αριθμός byte που συσχετίστηκε με την εγγραφή

Ένας μεταγωγέας μπορεί να αποστείλει μήνυμα προκειμένου να ενημερώσει για κάποια αλλαγή στην κατάσταση κάποιας θύρας του:

Class ryu.ofproto.ofproto_v1_0_parser.OFPPortStatus(*datapath, reason=None, desc=None*)

Όπου χρησιμοποιούνται τα παρακάτω ορίσματα:

- ***Reason:*** OFPPR_ADD, OFPPR_DELETE, OFPPR_MODIFY
- ***Desc:*** στιγμιότυπο του OFPPhyPort

Τέλος, μία συσκευή προώθησης στο OpenFlow δίκτυο μπορεί να στείλει μήνυμα για ενημέρωση του ελεγκτή για ενδεχόμενα λάθη σε κάποιο μήνυμα:

Class ryu.ofproto.ofproto_v1_0_parser.OFPErrorMsg(*datapath, type=None, code=None, data=None*)

Με ορίσματα:

- ***Type:*** τύπος του λάθους
- ***Code:*** Λεπτομέρειες για τον τύπο του λάθους
- ***Data:*** Δεδομένα μεταβλητού μήκους ανάλογα με τα δύο προαναφερθέντα ορίσματα

Συμμετρικά μηνύματα

Και οι δύο πλευρές μπορούν να στείλουν μηνύματα τύπου *Hello*, *Echo Request* και *Reply*, η ανταλλαγή των οποίων γίνεται αυτοματοποιημένα από το πλαίσιο του Ryu χωρίς να απασχολείται η εκάστοτε εφαρμογή. Επίσης, και από τις δύο πλευρές μπορούν να εκδοθούν μηνύματα που αφορούν πληροφορίες σχετικά με τον κατασκευαστή της συσκευής.

Class ryu.ofproto.ofproto_v1_3_parser.OFPHello(*datapath*)
Class ryu.ofproto.ofproto_v1_3_parser.OFPEchoRequest(*datapath, data=None*)
Class ryu.ofproto.ofproto_v1_3_parser.OFPEchoReply(*datapath, data=None*)
Class ryu.ofproto.ofproto_v1_3_parser.OFPVendor(*datapath*)

Σημειώνεται, πως τα μηνύματα που αναλύθηκαν παραπάνω, και των τριών κατηγοριών, ισχύουν και για χρήση νεότερων εκδόσεων του πρωτοκόλλου (1.2 έως 1.5) με κατάλληλη τροποποίηση του αριθμού της έκδοσης κατά τον ορισμό της εκάστοτε κλάσης,
π.χ

Class ryu.ofproto.ofproto_v1_5_parser.OFPFeaturesRequest(*datapath*).

4

Το περιβάλλον προσομοίωσης

4.1 Το λογισμικό Mininet

Για τις ανάγκες της παρούσας διπλωματικής εργασίας, καθώς και πολλών άλλων εφαρμογών Δικτύων Καθοριζόμενων από Λογισμικό, απαραίτητη είναι η δημιουργία δικτύου με πληθώρα συσκευών προώθησης και δρομολόγησης, όπως και αρκετούς τελικούς χρήστες. Λαμβάνοντας, όμως, υπόψη τους περιορισμένους πόρους, η ανάγκη αυτή ικανοποιείται με κάποιο πρόγραμμα προσομοίωσης, στην εν λόγω περίσταση το Mininet.
Η ιδέα πίσω από τη δημιουργία του Mininet, ήταν η δημιουργία ενός λογισμικού με τις ακόλουθες ιδιότητες [7]:

- **Ευέλικτο:** νέες τοπολογίες και νέες λειτουργικότητες πρέπει να μπορούν να οριστούν με χρήση του λογισμικού αυτού, με τη χρήση οικείων γλωσσών προγραμματισμού και λειτουργικών συστημάτων.
- **Δυνατότητα ανάπτυξης:** η ανάπτυξη ενός λειτουργικά σωστού πρωτοτύπου σε δίκτυα βασισμένα σε υλισμικό δεν πρέπει να απαιτεί αλλαγές στον κώδικα ή την παραμετροποίηση.
- **Διαδραστικό:** η διαχείριση και η λειτουργία του εικονικού δικτύου πρέπει να προσωμοιάζει αλληλεπίδραση διαχειριστή – φυσικού/πραγματικού δικτύου.
- **Επεκτασιμότητα:** το περιβάλλον προτυποποίησης πρέπει να μπορεί να κλιμακωθεί σε δίκτυα εκατοντάδων ή χιλιάδων κόμβων.
- **Ρεαλιστικό:** η συμπεριφορά του περιβάλλοντος προτυποποίησης πρέπει να αναπαριστά την πραγματική. Για παράδειγμα, οι εφαρμογές και τα πρωτόκολλα θα πρέπει να μπορούν να χρησιμοποιηθούν σε πραγματικές συνθήκες χωρίς σημαντικές προσαρμογές.
- **Δυνατότητα να χρησιμοποιηθεί από άλλους χρήστες** με στόχο την περαιτέρω ανάπτυξη κάποιας εφαρμογή σε αυτό το περιβάλλον.

Το Mininet, λοιπόν, υποστηρίζει τις προαναφερθείσες ιδιότητες χρησιμοποιώντας ελαφρά εικονοποίηση (*light virtualization*), επεκτάσιμη διεπαφή εντολών CLI

(*Command Line Interface*) και διεπαφή εφαρμογών API (*Application Program Interface*). Οι χρήστες μπορούν να αναπτύξουν μία νέα δικτυακή λειτουργία ή ακόμη μία νέα αρχιτεκτονική, να την ελέγξουν σε μεγάλες τοπολογίες με κίνηση από διάφορες εφαρμογές και στη συνέχεια να εφαρμόσουν ακριβώς τον ίδιο κώδικα σε ένα πραγματικό παραγωγικό δίκτυο. Το Mininet παρουσιάζει εξαιρετικές επιδόσεις όταν τρέχει σε ένα μόνο προσωπικό υπολογιστή εκμεταλλευόμενο τα πλεονεκτήματα του λειτουργικού συστήματος των Linux. Ολόκληρο το εικονικό δίκτυο «συσκευάζεται» σε ένα εικονικό μηχάνημα (*vm*), στο οποίο μπορεί να έχει πρόσβαση οποιοσδήποτε ενδιαφερόμενος για να το ελέγξει, τροποποιήσει ή απλά χρησιμοποιήσει.

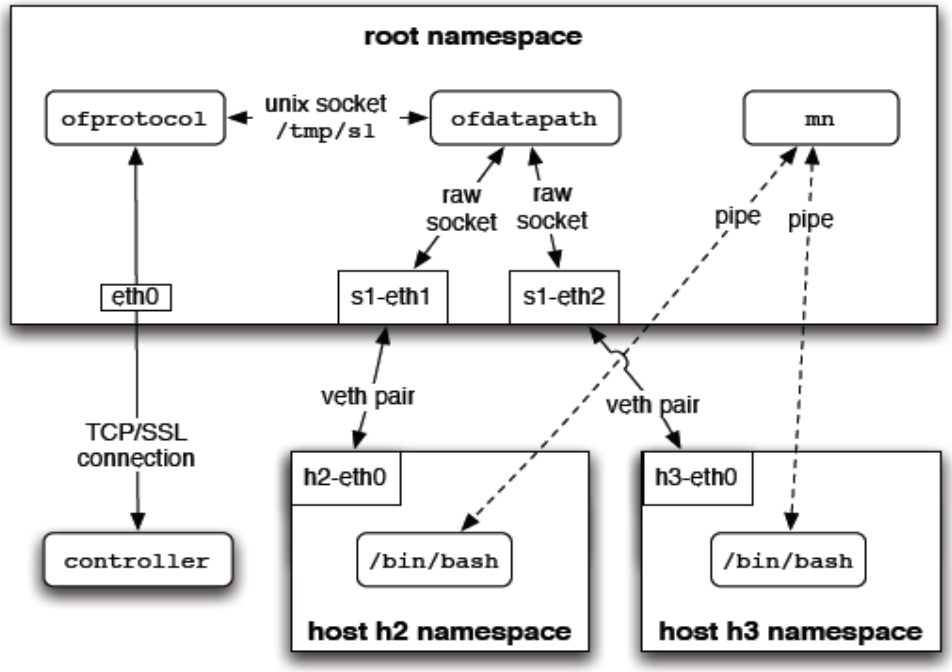
Τα δομικά στοιχεία της τοπολογίας του Mininet είναι τα ακόλουθα:

Ζεύξεις (Links): Ένα εικονικό ζεύγος Ethernet λειτουργεί σαν καλώδιο που συνδέει δύο εικονικές διεπαφές, τα πακέτα που στέλνονται δια μέσου της μίας οδεύουν στην άλλη και κάθε διεπαφή εμφανίζεται ως μία πλήρως λειτουργική θύρα Ethernet.

Τερματικά (Hosts): Οι χώροι ονομάτων (*namespaces*) του δικτύου είναι περιέχοντες (*containers*) για την κατάσταση του δικτύου. Παρέχουν διαδικασίες (*processes*) και ομάδες διαδικασιών με αποκλειστική ιδιοκτησία διεπαφών, θυρών και πινάκων δρομολόγησης (όπως τα πρωτόκολλα ARP και IP). Για παράδειγμα, δύο εξυπηρετητές δικτύου σε δύο χώρους ονομάτων μπορούν να συνυπάρχουν σε ένα σύστημα και να ακούν και οι δύο σε ιδιωτικές διεπαφές eth0 στη θύρα 80. Ένα τερματικό στο Mininet είναι απλά μια διαδικασία κελύφους (*shell process*) που έχει μετακινηθεί στο δικό της χώρο ονομάτων. Κάθε τερματικό έχει τις δικές του εικονικές διεπαφές και τη δική του διοχέτευση (*pipe*) σε μία διαδικασία Mininet γονιό (*parent mininet process*) mn, που στέλνει εντολές και ελέγχει το αποτέλεσμα (*output*).

Μεταγωγείς (Switches): Το λογισμικό των OpenFlow εικονικών μεταγωγέων παρέχει το ίδιο σημασιολογικό φάσμα που θα παρείχε και ένας πραγματικός μεταγωγέας ως υλικό.

Ελεγκτές (Controllers): Οι ελεγκτές μπορούν να βρίσκονται οπουδήποτε στο πραγματικό ή εικονικό δίκτυο με την προϋπόθεση πως το εικονικό μηχάνημα στο οποίο γίνεται η εξομοίωση του δικτύου έχει τη δυνατότητα σύνδεσης σε επίπεδο IP. Ο ελεγκτής μπορεί να τρέχει στο ίδιο εικονικό μηχάνημα με το Mininet, στο τερματικό που φιλοξενεί αυτό το εικονικό μηχάνημα ή στο σύννεφο.



Σχήμα 4.1 Δημιουργία εικονικού δικτύου από το Mininet

4.2 Δημιουργία τοπολογίας

Για τη δημιουργία μίας τοπολογίας στο Mininet το πρώτο βήμα είναι η χρήση της εντολής *mn*, παράδειγμα χρήσης αυτής, που δημιουργεί δίκτυο με OpenFlow μεταγωγείς, αποτελεί το ακόλουθο:

mn - -switch ovsk - -controller nox - -topo tree, depth=2, fanout=8 - -test pingall.

Σε αυτό το παράδειγμα, συνδέονται Open vSwitch μεταγωγείς σε δενδρική τοπολογία με βάθος 2 και εξάπλωση 8, υπό τον ελεγκτή Nox και ορίζεται δοκιμή ping για τον έλεγχο της σύνδεσης μεταξύ οποιουδήποτε ζεύγους κόμβων. Για τη δημιουργία του παραπάνω δικτύου το Mininet εξομοιώνει τις συνδέσεις (*links*), τα τερματικά (*hosts*), τους μεταγωγείς (*switches*) καθώς και τον ελεγκτή (*controller*). Για αυτό τον σκοπό γίνεται χρήση των μηχανισμών εικονοποίησης του λειτουργικού συστήματος των Linux: διαδικασίες που τρέχουν στους χώρους ονομάτων (*namespaces*) του δικτύου, εικονικά ζευγάρια Ethernet.

Μετά τη δημιουργία του δικτύου είναι σημαντική η αλληλεπίδραση με αυτό: η εκτέλεση εντολών στα τερματικά, η πιστοποίηση της λειτουργίας των μεταγωγέων, ενδεχομένως ο περιορισμός των αποτυχιών ή η προσαρμογή της συνδεσιμότητας των ζεύξεων. Το Mininet περιλαμβάνει διεπαφή γραμμής εντολών με επίγνωση του υφιστάμενου δικτύου (*network-aware CLI*) για να επιτρέπει στου προγραμματιστές να ελέγχουν και να διαχειρίζονται ολόκληρο το δίκτυο από μία μόνο κονσόλα. Καθώς το CLI έχει επίγνωση του δικτύου, γνωρίζει τα ονόματα των κόμβων και τις παραμετροποιήσεις που έχουν γίνει, μπορεί αυτόματα να αντικαταστήσει διευθύνσεις IP και ονόματα τερματικών. Για παράδειγμα η εντολή: h1 ping h2

υποχρεώνει το τερματικό h1 να κάνει ping στην IP διεύθυνση του h2. Η εντολή αυτή διοχετεύεται στη διαδικασία bash του εξομοιωμένου τερματικού h1 προκαλώντας τη δημιουργία ICMP echo αιτήματος από την ιδιωτική διεπαφή eth0 αυτού που θα εισέλθει στον πυρήνα μέσω των εικονικών ζεύξεων Ethernet. Το αίτημα αυτό επεξεργάζεται ο μεταγωγέας στο χώρο ονομάτων της βάσης (*root namespace*). Εάν το πακέτο έπρεπε να διασχίσει πολλαπλούς μεταγωγείς θα έμενε στον πυρήνα χωρίς επιπλέον αντίγραφα. Επιπλέον, το CLI δρα ως πολυπλέκτης τερματικών για τους ξενιστές και προσφέρει ποικιλία εντολών και εκφράσεων Python.

4.2.1 Εξατομικευμένα Δίκτυα

Το Mininet παρέχει μία API διεπαφή Python για τη δημιουργία εξατομικευμένων πειραμάτων, τοπολογιών και τύπων κόμβων. Σύντομα τμήματα κώδικα σε Python είναι αρκετά για τον ορισμό εξατομικευμένων αναδρομικών τεστ που μπορούν να δημιουργήσουν κάποιο δίκτυο, να εκτελέσουν εντολές σε πολλούς κόμβους ταυτόχρονα και να παρουσιάσουν τα αποτελέσματα. Παράδειγμα τέτοιου κώδικα αποτελεί το ακόλουθο:

```
from mininet.net import Mininet
from mininet.topolib import TreeTopo
tree4 = TreeTopo (depth=2,fanout=2)
net = Mininet(topo=tree4)
net.start()
h1, h4 = net.hosts[0], net.hosts[3]
print h1.cmd('ping -c1 %s' % h4.IP())
net.stop()
```

Το παραπάνω δημιουργεί ένα μικρό δίκτυο (4 τερματικά, 3 μεταγωγείς) και εκτελεί ping από το ένα τερματικό στο άλλο κάθε 4sec περίπου.

Το Mininet υποστηρίζει τη δημιουργία παραμετροποιήσιμων τοπολογιών καθιστώντας εφικτή τη δημιουργία ευέλικτων δικτύων που μπορούν να τροποποιηθούν ανάλογα με τις παραμέτρους που ορίζονται κατά την κλίση των προγραμμάτων και να επαναχρησιμοποιηθούν από νέες εφαρμογές. Παράδειγμα τέτοιας αποτελεί το ακόλουθο που περιλαμβάνει N τερματικά συνδεδεμένα σε ένα μοναδικό μεταγωγέα.

```
from mininet.topo import Topo
    from mininet.net import Mininet
    from mininet.util import dumpNodeConnections
    from mininet.log import setLogLevel

    class SingleSwitchTopo(Topo):
        "Single switch connected to n hosts."
        def build(self, n=2):
            switch = self.addSwitch('s1')
            # Python's range(N) generates 0..N-1
            for h in range(n):
                host = self.addHost('h%s' % (h + 1))
                self.addLink(host, switch)

    def simpleTest():
        "Create and test a simple network"
        topo = SingleSwitchTopo(n=4)
        net = Mininet(topo)
        net.start()
        print "Dumping host connections"
        dumpNodeConnections(net.hosts)
        print "Testing network connectivity"
        net.pingAll()
        net.stop()

    if __name__ == '__main__':
        # Tell mininet to print useful information
        setLogLevel('info')
        simpleTest()
```

Κάποιες σημαντικές κλάσεις, μέθοδοι, συναρτήσεις και μεταβλητές που χρησιμοποιούνται αναλύονται ακολούθως:

- ***Topo:*** η βασική κλάση για δημιουργία τοπολογίας
- ***Build():*** η μέθοδος για παράκαμψη στην κλάση της τοπολογίας. Οι παράμετροι κατασκευής περνούν αυτόματα μέσω του Topo.__init__()
- ***addSwitch:*** προσθέτει μεταγωγέα και επιστρέφει το όνομα του
- ***addHost:*** προσθέτει τερματικό και επιστρέφει το όνομα του
- ***addLink:*** προσθέτει μία αμφίδρομη ζεύξη σε κάποια τοπολογία, για να μην είναι αμφίδρομη πρέπει να οριστεί διαφορετικά
- ***Mininet:*** η κύρια κλάση για τη δημιουργία και τη διαχείριση ενός δικτύου
- ***Start():*** εκκινεί το δίκτυο
- ***Pingall():*** ελέγχει την επιτυχία των συνδέσεων προσπαθώντας να πραγματοποιήσει ping (ανταλλαγή μηνυμάτων ICMP) από και προς όλα τα τερματικά
- ***Stop():*** σταματά το δίκτυο
- ***Net.hosts:*** όλα τα τερματικά στο δίκτυο
- ***dumpNodeConnections:*** σταματά ορισμένες συνδέσεις

4.3 Διανομή ενός προσομοιωμένου δικτύου

Το Mininet ως εικονικό μηχάνημα είναι κατανεμημένου χαρακτήρα με όλες τις εξαρτήσεις προ-εγκατεστημένες σε αυτό, ικανό να τρέξει σε οποιοδήποτε επόπτη εικονικών μηχανημάτων (*virtual machine monitors/hypervisors*) όπως VMware, Xen, VirtualBox. Το εικονικό μηχάνημα παρέχει ένα βολικό περιέκτη για την κατανομή, όταν ένα πρότυπο έχει αναπτυχθεί η εικόνα του vm μπορεί να μοιραστεί σε άλλους χρήστες για να το τρέξουν, εξετάσουν ή τροποποιήσουν. Ένα πλήρες, συμπιεσμένο εικονικό μηχάνημα Mininet είναι περίπου 800MB. Ακόμη, το Mininet μπορεί να εγκατασταθεί απευθείας σε οποιαδήποτε διανομή Linux χωρίς να αντικατασταθεί ο πυρήνας.

4.4 Επεκτασιμότητα

Η ελαφρά εικονοποίηση (*lightweight virtualization*) είναι το κλειδί που επιτρέπει την επέκταση σε δίκτυα με εκατοντάδες κόμβους ενώ διατηρούνται οι διαδραστικές επιδόσεις. Παρακάτω παρουσιάζονται τα αποτελέσματα μετρήσεων του χρόνου δημιουργίας διαφόρων τοπολογιών, του διαθέσιμου εύρους ζώνης καθώς και κάποιων τιμών – ορόσημων για τη λειτουργία του δικτύου.

Πίνακας 4.1 Αποτελέσματα μετρήσεων iperf για το από άκρο σε άκρο εύρος ζώνης [7]

S (Switches)	User(Mbps)	Kernel(Mbps)
1	445	2120
10	49.9	940
20	25.7	573
40	12.6	315
60	6.2	267
80	4.15	217
100	2.96	167

Πίνακας 4.2 Πίνακας μετρήσεων χρόνου εγκατάστασης και τερματισμού καθώς και χρήση μνήμης για δίκτυα με H hosts και S switches. Οι δοκιμές έγιναν σε Debian 5/Linux 2.6.33.1 VM στο VMware Fusion 3.0 σε MacBook Pro (2.4GHz intel Core 2 Duo/6GB)[7]

Topology	H	S	Setup(s)	Stop(s)	Mem(MB)
Minimal	2	1	1.0	0.5	6
Linear(100)	100	100	70.7	70.0	112
VL2(4, 4)	80	10	31.7	14.9	73
FatTree(4)	16	20	17.2	22.3	66
FatTree(6)	54	45	54.3	56.3	102
Mesh(10, 10)	40	100	82.3	92.9	152
Tree(4^4)	256	85	168.4	83.9	233
Tree(16^2)	256	17	139.8	39.3	212
Tree(32^2)	1024	33	817.8	163.6	492

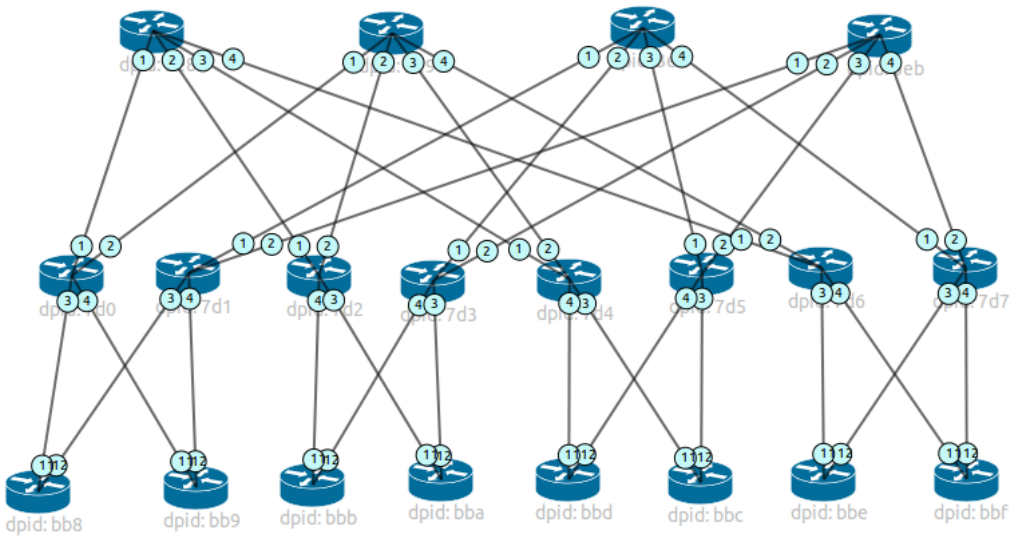
Πίνακας 4.3 Πίνακας με μετρήσεις για το χρόνο που απαιτούν οι βασικές λειτουργίες στο Mininet [7]

Operation	Time (ms)
Create a node (host/switch/controller)	10
Run command on a host ('echo hello')	0.3
Add link between two nodes	260
Delete link between two nodes	416
Start user space switch (OpenFlow reference)	29
Stop user space switch (OpenFlow reference)	290
Start kernel switch (Open vSwitch)	332
Stop kernel switch (Open vSwitch)	540

Το Mininet παρουσιάζει τη δυνατότητα να επεκταθεί στις μεγάλες τοπολογίες που παρουσιάστηκαν παραπάνω (περισσότερα από 1000 τερματικά) επειδή εικονοποιεί λίγα και μοιράζεται περισσότερα. Το σύστημα αρχείων, ο χώρος ID του χρήστη, ο χώρος ID των διεργασιών, ο πυρήνας, οι βιβλιοθήκες κ.α. μοιράζονται μεταξύ των διαδικασιών και διαχειρίζονται από το λειτουργικό σύστημα. Από τις τοπολογίες που παρουσιάζονται στο σχήμα 4.3 μόνο οι μικρότερες θα ταίριαζαν στη μνήμη ενός τυπικού προσωπικού υπολογιστή εάν γινόταν χρήση εικονοποίησης του συστήματος. Το Mininet παρέχει, επιπλέον, εύρος ζώνης ικανό να χρησιμοποιηθεί όπως παρουσιάζεται στο σχήμα 4.2. Τέλος, στο σχήμα 4.4 φαίνεται ο χρόνος που καταναλώνεται από συγκεκριμένες ενέργειες όταν δημιουργείται μία νέα τοπολογία. Αξίζει να σημειωθεί ότι ακριβές διεργασίες φαίνεται να είναι η προσθήκη και η διαγραφή κάποιας ζεύξης καθώς απαιτούν περίπου 250ms και 400ms αντίστοιχα.

4.5 Χρήση του Mininet στην παρούσα εργασία

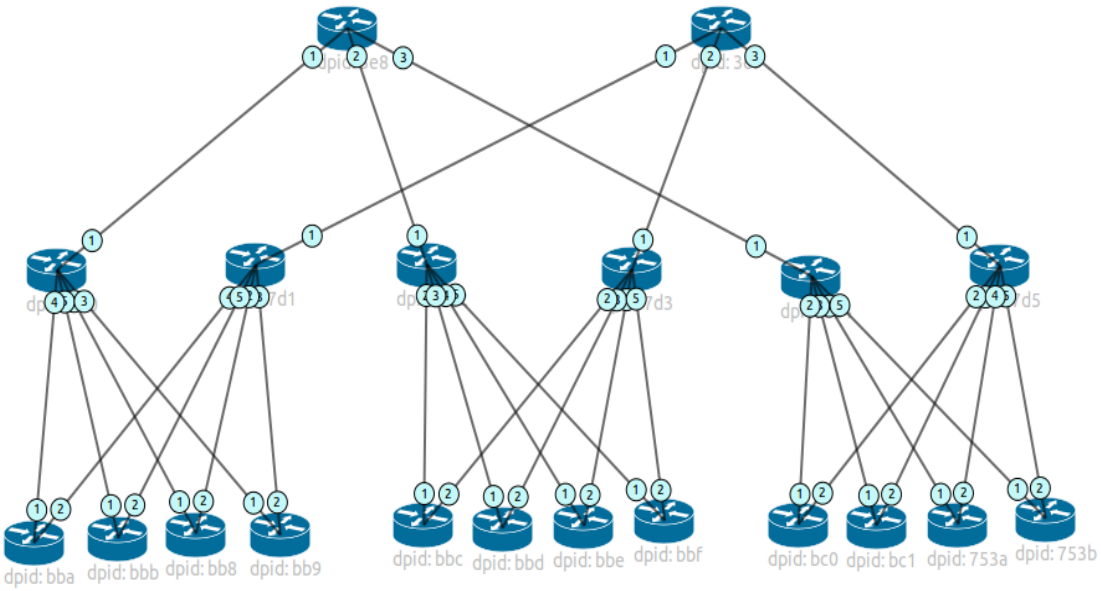
Για τις ανάγκες της παρούσας εργασίας επιλέχθηκε η εξομοίωση τοπολογίας *fat tree*. Πρόκειται για δενδρική αρχιτεκτονική που προσφέρει εναλλακτικές διαδρομές από τερματικό σε τερματικό (*redundancy*) και βασίζεται στην ιδέα των δικτύων Clos. Ακολούθως παρουσιάζεται το μοντέλο που δημιουργήθηκε σε γλώσσα Python και με χρήση του λογισμικού Mininet:



Σχήμα4.1 Αυθεντική τοπολογία fat tree με 4 μεταγωγείς στο επίπεδο πυρήνα

Το πρώτο επίπεδο στο δέντρο περιλαμβάνει 4 *core switches*, το δεύτερο από 8 *aggregate switches* και το τελευταίο από 8 *access/edge switches*. Οι μεταγωγείς του επιπέδου *core* συνδέονται εναλλάξ με 4 από το παρακάτω επίπεδο, ενώ οι μεταγωγείς του επιπέδου *aggregate* συνδέονται ένας προς έναν με αυτούς του επιπέδου που ακολουθεί. Τέλος, προστίθενται τερματικά με ζεύξεις προς το επίπεδο *access.*

Ωστόσο, η παραπάνω τοπολογία συναντάται κυρίως στη βιβλιογραφία και λιγότερο στην πράξη. Για το λόγο αυτό, αναπτύχθηκε ακόμη ένα μοντέλο τύπου fat tree, που όμως περιλαμβάνει *3 core switches, 6 aggregate switches* και *12 access switches*. Η τοπολογία αυτή φαίνεται στο σχήμα που ακολουθεί.



Σχήμα 4.2 Ρεαλιστική τοπολογία fat tree με 2 μεταγωγείς στο επίπεδο πυρήνα

Ο κώδικας που αναπτύχθηκε για τη προσομοίωση των δύο τοπολογιών που παρουσιάστηκαν, αναλύεται στο Κεφάλαιο 7 της παρούσας εργασίας.

5

Δίκτυα Data Center

Ένα Data center αποτελεί μία «πισίνα» πόρων, υπολογιστικών, αποθηκευτικών και δικτυακών, διασυνδεδεμένων με τη χρήση κάποιου τηλεπικοινωνιακού δικτύου. Ένα δίκτυο data center (*Data Center Network DCN*) διαδραματίζει κεντρικό ρόλο, καθώς ευθύνεται για τη διασύνδεση όλων των πόρων. Τα διάφορα DCN πρέπει να είναι επεκτάσιμα και αποτελεσματικά για να συνδέουν δεκάδες, εκατοντάδες ακόμα και χιλιάδες εξυπηρετητές έτσι ώστε να μπορούν να χειριστούν τις αυξανόμενες απαιτήσεις των υπηρεσιών νέφους.

5.1 Στόχοι στη σχεδίαση δικτύων Data Center

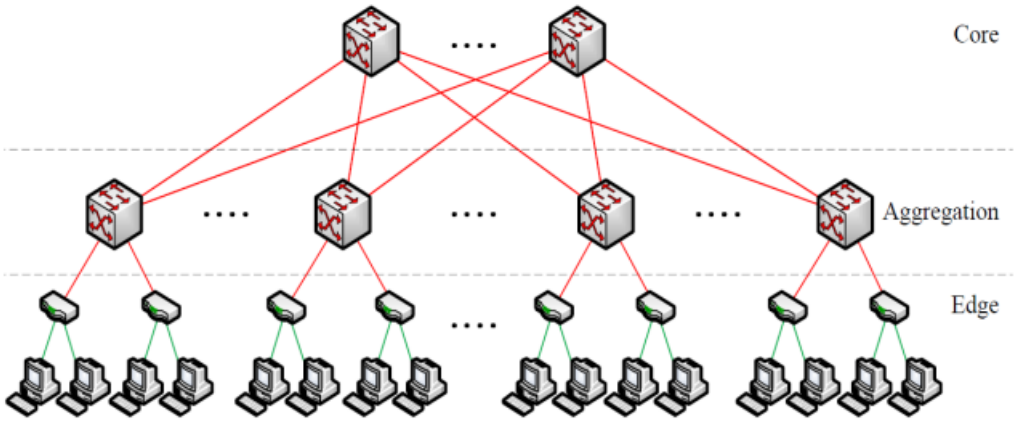
Η σημαντικότερη πρόκληση στη σχεδίαση ενός DCN είναι η άμεση δυνατότητα επέκτασης αυτού. Παράλληλα, κατά το σχεδιασμό απαιτείται να αντιμετωπιστούν οι περιορισμοί που αφορούν: (α) την ύπαρξη ενός μοναδικού σημείου αποτυχίας (*single point of failure*) και (β) τους υπερβολικά μεγάλους ρυθμούς δημιουργίας συνδέσεων ψηλά στην αρχιτεκτονική, δηλαδή κοντά στην ρίζα του δένδρου (*over-subscription*). Τα βασικότερα σημεία που λαμβάνονται υπόψη στο σχεδιασμό ενός δικτύου Data center περιλαμβάνουν: (α) το να επιτρέπεται η επικοινωνία μεταξύ των τερματικών στην ταχύτητα της γραμμής ανεξάρτητα από τα σημεία στα οποία βρίσκονται αυτά, (β) οι προσθήκες που γίνονται να είναι συμβατές με την προϋπάρχουσα τεχνολογία, έτσι ώστε να μην απαιτούνται αλλαγές στις εφαρμογές που φιλοξενούνται ή στο στρώμα Ethernet και (γ) ο περιορισμός του κόστους που ερμηνεύεται ως υιοθέτηση οικονομικής υποδομής και μειωμένη κατανάλωση ενέργειας και εκπομπή θερμότητας.

5.2 *Τύποι δικτύων Data Center*

Οι σύγχρονες αρχιτεκτονικές DCN είναι κυρίως τοπολογίες τριών στρωμάτων. Αποτελούνται από τους μεταγωγείς του πυρήνα (*core switches*) που συνδέονται μεταξύ τους αλλά και με εξωτερικούς δικτυακούς παρόχους, το στρώμα χρήστη ή πρόσβασης και τέλος το στρώμα συγκέντρωσης ανάμεσα στα δύο προαναφερθέντα που μετακινεί την πληροφορία βόρεια ή νότια. Αναλυτικά τα στρώματα που απαρτίζουν την αρχιτεκτονική τριών στρωμάτων είναι τα ακόλουθα:

- ***Στρώμα πρόσβασης (Access/Edge Layer):*** Παρέχει συνδεσιμότητα επιπέδου 2 ή 3 για τους εξυπηρετητές του Data Center με υψηλές επιδόσεις και χαμηλή καθυστέρηση. Σημειώνεται, πως προτιμάται η σύνδεση μεταξύ του επιπέδου αυτού και των εξυπηρετητών να γίνεται στο επίπεδο 2 (Ethernet) και όχι στο επίπεδο 3 (IP). Ο διαχωρισμός της κίνησης σε αυτή την περίπτωση, ανάλογα με την εφαρμογή από την οποία προέρχεται και τη σκοπιμότητα που εξυπηρετεί, γίνεται με τη χρήση της τεχνολογίας VLAN (*Virtual LAN*) και όχι κανόνων δρομολόγησης.
- ***Στρώμα συγκέντρωσης (Aggregation/Distribution Layer):*** Σε αυτό το στρώμα συγκεντρώνονται όλες οι ζεύξεις, επιπέδων 2 και 3 και προωθούνται στο στρώμα του πυρήνα. Πρόκειται για ένα σημείο με μεγάλη σημασία για την ασφάλεια και τις υπηρεσίες των εφαρμογών.
- ***Στρώμα πυρήνα (Core Layer):*** Σε αυτό πραγματοποιείται η σύνδεση του Data center με το εξωτερικό δίκτυο με χρήση ζεύξεων του στρώματος 3. Πρόκειται για ένα κεντρικοποιημένο στρώμα δρομολόγησης.

Για δίκτυα Data center που πλήττονται κυρίως από κίνηση ανατολική-δυτική (*east-west traffic*), δηλαδή κίνηση μεταξύ εξυπηρετητών, προτιμώνται τοπολογίες «leaf-spine». Σε αυτές τις τοπολογίες παρατηρείται ένα στρώμα με αυξημένο αριθμό μεταγωγέων ικανό να διαχειριστεί την αυξημένη εσωτερική κίνηση.



Σχήμα 5.1 Κοινή αρχιτεκτονική DCN

5.2.1 Εξειδικευμένες αρχιτεκτονικές DCN

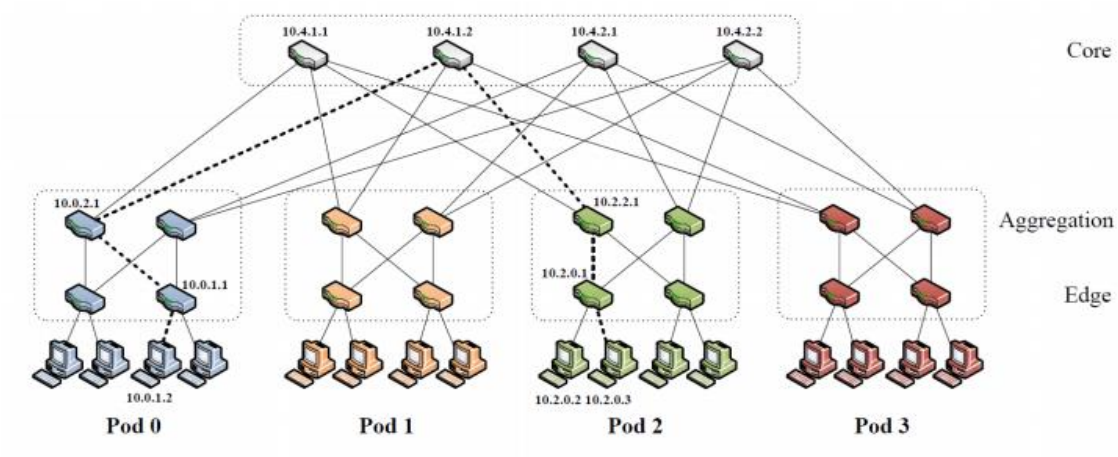
Ακολούθως παρουσιάζονται διαφορετικές τοπολογίες DCN:

- ***DCell:*** υβριδική αρχιτεκτονική με κεντρικό άξονα τον εξυπηρετητή, ο οποίος συνδέεται με πολλούς άλλους. Ο εξυπηρετητής σε αυτή την αρχιτεκτονική είναι εξοπλισμένος με πολλές Κάρτες Διεπαφών Δικτύου (*Network Interface Cards NICs*). Η DCell ακολουθεί μία αναδρομική ιεραρχία από κύτταρα (*cells*). Το $cell_0$ αποτελεί τη βασική μονάδα για τη δημιουργία της τοπολογίας και βρίσκεται σε πολλαπλά επίπεδα, περιλαμβάνει n εξυπηρετητές και έναν μεταγωγέα. Το $cell_1$ περιλαμβάνει k = n+1 $cell_0$ κύτταρα και αντίστοιχα το $cell_2$ k*n + 1 $cell_1$. Κύριο χαρακτηριστικό αυτής της αρχιτεκτονικής είναι η δυνατότητα επέκτασης καθώς και οι κακές επιδόσεις σχετικά με το εύρος ζώνης και την καθυστέρηση.
- ***FiConn:*** Παρόμοια αρχιτεκτονική με την DCell. Χρησιμοποιεί μία ιεραρχία διασυνδέσεων εξυπηρετητή προς εξυπηρετητή και κυττάρων, προϋποθέτει, όμως τη χρήση μόνο 2 NICs.
- ***BCube:*** Αρχιτεκτονική παρόμοιας φιλοσοφίας με τις DCell και FiConn, που αφορά κυρίως σπονδυλωτά Data center δίκτυα.
- ***Hybercube:*** Ένα τέτοιο 3D δίκτυο είναι απλά ένας κύβος, ένα κουτί δηλαδή, 6 πλευρών με μεταγωγείς σε κάθε γωνία. Ένα τέτοιο δίκτυο 4D είναι ένας κύβος εντός ενός άλλου κύβου, με μεταγωγείς στις γωνίες που συνδέουν τους κύβους μεταξύ τους (ο εσωτερικός συνδέεται με τον εξωτερικό στις γωνίες).
- ***Toroidal:*** Αυτός ο όρος αναφέρεται σε οποιαδήποτε τοπολογία δακτυλίου. Μία σπείρα (*torus*) 3D είναι μία καλά διασυνδεδεμένη τοπολογία δικτύου δακτυλίων. Τα δακτυλιοειδή αποτελούν δημοφιλή επιλογή σε υπολογιστικά περιβάλλοντα που απαιτούν υψηλές επιδόσεις και βασίζονται στους μεταγωγείς για τη διασύνδεση των υπολογιστικών κόμβων.
- ***Jellyfish:*** Πρόκειται για μία τοπολογία χωρίς συγκεκριμένη δομή που εξαρτάται από τις προτιμήσεις του σχεδιαστή. Έρευνες δείχνουν πως η χρήση αυτής οδηγεί σε 25% υψηλότερη χωρητικότητα έναντι των παραδοσιακών δικτύων.
- ***Scafida:*** Πρόκειται για μία ιδέα παρόμοια με της τοπολογία Jellyfish, καθώς περιλαμβάνει σε σημαντικό βαθμό τυχαίο σχεδιασμό. Εντούτοις, περιλαμβάνει μεταγωγείς πυκνά συνδεδεμένους που δημιουργούν περιοχές διανομέων (*hub sites*).
- ***Butterfly:*** Ειδική κατασκευή της Google, που θυμίζει σκακιέρα. Πρόκειται για ένα πλέγμα από μεταγωγείς, όπου η κίνηση μπορεί να κατευθυνθεί σε οποιοδήποτε μεταγωγέα σε συγκεκριμένες κατευθύνσεις. Κίνητρο που συντέλεσε στη δημιουργία της είναι η μείωση της ενέργειας που καταναλώνεται.

5.2.2 Η δημοφιλέστερη αρχιτεκτονική DCN

Η αρχιτεκτονική, που καλείται να αντιμετωπίσει τα προβλήματα της κλασικής αρχιτεκτονικής τριών επιπέδων που αφορούν το μειωμένο από άκρο σε άκρο εύρος ζώνης καθώς και την πληθώρα συνδέσεων στα υψηλότερα σημεία της

αρχιτεκτονικής, καλείται *Fat Tree.* Η δημιουργία της βασίζεται στην τοπολογία Clos και περιλαμβάνει, όπως και η κλασική αρχιτεκτονική τριών επιπέδων, ιεραρχική δομή των στρωμάτων πρόσβασης (*access*), συσσώρευσης (*aggregation*) και πυρήνα (*core*). Η διάκριση γίνεται στον αριθμό των μεταγωγέων του δικτύου που είναι σημαντικά μεγαλύτερος από ότι στην παραδοσιακή αρχιτεκτονική. Η αρχιτεκτονική αυτή αποτελείται από k «δεξαμενές», με την κάθε μία να περιλαμβάνει $(\frac{k}{2})^2$εξυπηρετητές, $\frac{k}{2}$ μεταγωγείς στο στρώμα πρόσβασης και $\frac{k}{2}$ μεταγωγείς στο στρώμα συσσώρευσης. Το στρώμα πυρήνα περιλαμβάνει $(\frac{k}{2})^2$ μεταγωγείς με τον καθένα να συνδέεται με έναν στο στρώμα συσσώρευσης σε κάθε μία από τις δεξαμενές.



Σχήμα 5.2 Απλή αρχιτεκτονική Fat tree με k=4

Τα πλεονεκτήματα που παρουσιάζει αυτή η αρχιτεκτονική περιλαμβάνουν:

- Κάθε διχοτόμηση του δέντρου (*bisection*) έχει το ίδιο εύρος ζώνης.
- Κάθε στρώμα έχει αθροιστικά το ίδιο εύρος ζώνης.
- Μπορεί να χτιστεί με χρήση φτηνών συσκευών που έχουν ομοιόμορφη χωρητικότητα.
- Όλες οι συσκευές μπορούν να μεταδώσουν με την ταχύτητα της γραμμής, αν τα πακέτα κατανεμηθούν ομοιόμορφα στα διαθέσιμα μονοπάτια.
- Καλή επεκτασιμότητα: k θύρες ανά μεταγωγέα υποστηρίζουν $\frac{k^3}{4}$ εξυπηρετητές.

Ωστόσο, παρουσιάζονται κάποια προβλήματα στην υιοθέτηση του *Fat tree*:

- Η δρομολόγηση στο επίπεδο 3 (*IP Layer routing*) θα επιλέγει κάθε φορά το ίδιο μονοπάτι, από τα διαθέσιμα ίσου κόστους μονοπάτια.
- Δημιουργία σημείων συμφόρησης (*bottlenecks*).
- Αναδιάταξη πακέτων εάν το επίπεδο IP εκμεταλλευτεί τυφλά την ύπαρξη ποικίλλων μονοπατιών ίσου κόστους.
- Πολυπλοκότητα στη σύνδεση της καλωδίωσης για μεγάλα δίκτυα.

Η πρόταση της Cisco

Τέλος, αξίζει να γίνει αναφορά στην πρόταση της εταιρείας Cisco αναφορικά με την σχεδίαση των Data Center Δικτύων.

Η ιεραρχική σχεδίαση τριών σειρών (*three tier hierarchical design*) μεγιστοποιεί τις επιδόσεις, την διαθεσιμότητα του δικτύου και τη δυνατότητα επέκτασης αυτού. Ωστόσο, οι περισσότερες επιχειρήσεις μικρού μεγέθους διαθέτουν δικτυακές υποδομές που δεν μεγαλώνουν σημαντικά με το πέρασμα του χρόνου και μπορούν να εξυπηρετηθούν ικανοποιητικά από μία ιεραρχική δομή δύο σειρών (*two tier hierarchical design*), όπου τα στρώματα πυρήνα και συσσώρευσης ενσωματώνονται σε ένα νέο. Το βασικότερο κίνητρο για αυτή την τροποποίηση είναι η μείωση του κόστους με την παράλληλη διατήρηση των πλεονεκτημάτων του γνωστού ιεραρχικού μοντέλου τριών σειρών. Η υλοποίηση ενός δικτύου συνεπτυγμένου πυρήνα (*collapsed core network*) οδηγεί στην ενσωμάτωση των λειτουργιών διανομής και πυρήνα σε μία συσκευή. Έτσι το νέο μοντέλο πρέπει να παρέχει τα ακόλουθα:

- Φυσικά ή λογικά μονοπάτια υψηλών ταχυτήτων προς το δίκτυο.
- Ένα σημείο οριοθέτησης και συγκέντρωσης στο επίπεδο 2.
- Ορισμό των πολιτικών δρομολόγησης και πρόσβασης στο δίκτυο.
- Έξυπνες δικτυακές υπηρεσίες, όπως Quality of Service (QoS), δικτυακή εικονοποίηση κ.α.

6

Η Υποδομή για Ενεργειακά Αποτελεσματική, Δυναμική Δρομολόγηση

6.1 Το πρόβλημα της ενεργειακής σπατάλης σε Δίκτυα Data Center

Καθώς εντείνονται οι ανησυχίες σχετικά με τις ενεργειακές ανάγκες και τον περιβαλλοντικό αντίκτυπο των Δικτύων Data Center, τα ενεργειακά αποτελεσματικά δίκτυα (*energy efficient networks*) αποτελούν μία από τις μεγαλύτερες προκλήσεις στον χώρο των Επικοινωνιών. Υπολογισμοί δείχνουν πως το δικτυακό μέρος των DCN καταναλώνει το 15% της συνολικής ενέργειας στον κυβερνοχώρο. Περίπου 15.6 δισεκατομμύρια kWh ενέργειας χρησιμοποιούνται παγκοσμίως από τη δικτυακή υποδομή εντός των DCN. Σύμφωνα με μελέτες, η κατανάλωση από τα DCN αναμένεται να αυξηθεί κατά 50% στο άμεσο μέλλον. Σε εφαρμογή βρίσκονται για τον περιορισμό της κατανάλωσης ενέργειας τόσο το πρότυπο της IEEE 802.3az, όσο και ενεργειακά-αποδοτικές αρχιτεκτονικές όπως η DCell και το Fat Tree. Επιπλέον, προτιμάται εξοπλισμός σχεδιασμένος για μειωμένο ενεργειακό κόστος.

Ωστόσο, η σημαντικότερη αιτία για την ενεργειακή σπατάλη είναι το γεγονός ότι οι περισσότερες δικτυακές συσκευές δεν καταναλώνουν ενέργεια ανάλογα με τις εργασίες που επιτελούν, αντιθέτως καταναλώνουν περισσότερο όταν υπό-χρησιμοποιούνται. Αντιθέτως, η χρησιμοποίηση του δικτύου και η κίνηση σε αυτό ακολουθούν το τυπικό μοτίβο ανθρώπινης δραστηριότητας, με σημαντικές διαφορές στις ακραίες τιμές και εμφανή περιοδικότητα [5]. Ως εκ τούτου, οι δικτυακές συσκευές καταλήγουν να εμφανίζουν σημαντικά μειωμένη χρήση για μακρά διαστήματα, με αποτέλεσμα να είναι κατ' ουσία αδρανείς αλλά να καταναλώνουν αρκετά υψηλά ποσοστά ενέργειας. Οι προτεινόμενες λύσεις για αυτό το ζήτημα αφορούν: (α) συμμετρική κατανάλωση ενέργειας από τις δικτυακές συσκευές με την προσαρμογή της ταχύτητας και της χωρητικότητας στο πραγματικό φορτίο που δέχονται και (β) μεταβολή της κατάστασης ορισμένων θυρών σε στάδιο *sleep*. Στη

δεύτερη περίπτωση, το δίκτυο καταλήγει να απαρτίζεται από συσκευές με υψηλή χρησιμοποίηση και αδρανείς συσκευές που όμως τίθενται σε κατάσταση *sleep* και δεν σπαταλούν ενέργεια. Συνεπώς, προκύπτει το ενδεχόμενο να τεθούν εκτός λειτουργίας τα μονοπάτια ελάχιστης διαδρομής που περιλαμβάνει η αρχική τοπολογία. Έτσι τα οφέλη από την «απενεργοποίηση» ορισμένων συσκευών μετριάζονται από τις αυξημένες ενεργειακές απαιτήσεις των υπολοίπων που έχουν να διαχειριστούν μεγαλύτερο πλέον φορτίο. Γεννάται έτσι η ανάγκη δημιουργίας ενός μοντέλου για την εύρεση της «χρυσής τομής» σε αυτή τη συμβιβαστική τακτική (*trade off*).
Λαμβάνοντας, λοιπόν, υπόψη τα προαναφερθέντα και με γνώμονα τις μελέτες που αφορούν την μετατροπή σε κατάσταση *sleep* των ανενεργών θυρών ενός δικτύου, δημιουργήθηκε μία εφαρμογή με στόχο να απαγορεύει τη ροή κίνησης μέσα από ορισμένες θύρες ελέγχοντας περιοδικά την κατάσταση του δικτύου και αλλάζοντας έτσι συνεχώς τους κανόνες δρομολόγησης στο υφιστάμενο δίκτυο.

6.2 Ανάπτυξη της εφαρμογής

Η εξυπηρέτηση του σκοπού της παρούσας εργασίας έγινε με την ανάπτυξη τμημάτων κώδικα (*components*), που το κάθε ένα υπηρετεί διαφορετική σκοπιμότητα και όχι με τη συνολική δημιουργία μίας εφαρμογής. Αναφέρεται συνοπτικά, πως προκειμένου να πραγματοποιηθεί αναδιάρθρωση του δικτύου, σύμφωνα με τις μελέτες και τα κριτήρια που αναφέρθηκαν προηγουμένως, απαιτείται η συλλογή πληροφοριών από το δίκτυο, γνώση της τοπολογίας του και τρόπος να εφαρμοστούν τα πορίσματα των μελετών που υιοθετούνται στις δικτυακές συσκευές.

Το πρώτο βήμα, λοιπόν, είναι η ανίχνευση της τοπολογίας του δικτύου που γίνεται με την ανάπτυξη του Python module ***get_topology_mini.py***. Συγκεκριμένα, το πρόγραμμα που δημιουργήθηκε ανιχνεύει δυναμικά την τοπολογία του OpenFlow δικτύου, έτσι ώστε ο προγραμματιστής να είναι σε θέση να γνωρίζει ποιοι μεταγωγείς υπάρχουν σε αυτό καθώς και το πώς αυτοί συνδέονται μεταξύ τους και μέσω ποιων θυρών. Συγκεκριμένα:

- Η ανίχνευση των μεταγωγέων γίνεται με χρήση των μηνυμάτων Feature Request και Feature Reply του OpenFlow (Κεφ. 3).
- Η ανίχνευση των συνδέσεων μεταξύ των μεταγωγέων γίνεται μέσω των πακέτων LLDP (*Link Layer Discovery Protocol*) (Κεφ. 3).

Το πρόγραμμα δημιουργεί την κλάση GetTopologyApp, που κληρονομεί από την κλάση app_manager.ryuApp. Η ανίχνευση γίνεται με τη συνάρτηση _topo_change_handler(), η οποία αναγνωρίζει αν πρέπει να γίνει ανίχνευση της

τοπολογίας, εκφράζει το λόγο και κάνει την εγγραφή των αποτελεσμάτων στο αρχείο εξόδου topology.txt. Η συνάρτηση αυτή εκτελείται όταν:

- Συμβεί κάποιο από τα γεγονότα: είσοδος, αφαίρεση, επανασύνδεση μεταγωγέα ή αλλαγή στην κατάσταση μίας θύρας ή ενός μεταγωγέα (event.EventSwitchEnter, event.EventSwitchLeave, event.EventSwitchReconnected, event.EventPortModify, ofp_event.EventOFPStateChange). Δηλαδή, όταν συμβεί κάποια αλλαγή στην τοπολογία του δικτύου.
- Ο timer ξεπεράσει την τιμή self.max_age όπου η συνάρτηση _monitor καλεί ρητά την _topo_change_handler(). Η προαναφερθείσα τιμή max_age αναπαριστά την μέγιστη τιμή του χρόνου σε δευτερόλεπτα που αν συμπληρωθεί πρέπει να γίνει έλεγχος και ενημέρωση για αλλαγές στην τοπολογία. Αυτό συμβάλει στην διατήρηση πρόσφατης ενημέρωσης για την τοπολογία του δικτύου στο αρχείο εξόδου.

Στη συνέχεια, αναπτύχθηκε το πρόγραμμα ***traffic_mon.py*** που παρακολουθεί δυναμικά την κίνηση που διέρχεται από κάθε θύρα και κάθε μεταγωγέα του δικτύου. Η εκτέλεση αυτού του προγράμματος σε συνδυασμό με το προαναφερθέν πρόγραμμα ανίχνευσης τοπολογίας δίνει τη δυνατότητα εκτίμησης του όγκου της κίνησης σε κάθε ζεύξη του δικτύου. Έτσι μπορεί να γίνει αντιληπτό από τον προγραμματιστή ποιες ζεύξεις και ποιες συσκευές είναι αρκετά φορτωμένες, ελάχιστα φορτωμένες ή αδρανείς. Για να διευκολυνθεί η ερμηνεία και εκμετάλλευση των αποτελεσμάτων, δημιουργούνται τόσα αρχεία εξόδου όσες είναι και οι δικτυακές συσκευές με το κάθε ένα να περιλαμβάνει τα στατιστικά δεδομένα ενός μεταγωγέα. Τα αρχεία φέρουν ως όνομα το *datapath_id* του κάθε μεταγωγέα. Για παράδειγμα το αρχείο εξόδου με όνομα 1.txt αντιστοιχεί στον μεταγωγέα με datapath_id=0000000000000001. Υπενθυμίζεται, πως datapath_id είναι το μοναδικό αναγνωριστικό που αντιστοιχεί σε κάθε OpenFlow μεταγωγέα.
Για την επίτευξη της δυναμικής δρομολόγησης υιοθετείται η τεχνική *polling*. Σύμφωνα με αυτή:

- Ο ελεγκτής στέλνει περιοδικά μηνύματα *OFPPortStatsRequest* σε κάθε μεταγωγέα του δικτύου, ζητώντας έτσι τις πληροφορίες για τα στατιστικά στοιχεία που έχει συλλέξει το κάθε ένα για κάθε θύρα του.
- Οι μεταγωγείς ακολούθως απαντούν στον ελεγκτή με μηνύματα *OFPPortStatsReply* που περιλαμβάνουν όσα έχουν ζητηθεί.

Τα στοιχεία που έχουν συλλεχθεί και αποθηκευτεί στα αρχεία [DPID].txt τροφοδοτούνται σε κάποια σουίτα βελτιστοποίησης (π.χ. *CPLEX*), όπου εκτελείται ο αλγόριθμος βελτιστοποίησης ενεργειακής απόδοσης. Ακολούθως, μπορεί να ληφθεί απόφαση σχετικά με το ποιες θύρες θα παραμείνουν σε κατάσταση λειτουργίας και ποιες όχι. Οι υποδείξεις του προγράμματος αυτού αποθηκεύονται στο αρχείο ports_mod.txt με σκοπό την εκμετάλλευσή του.
Το πρόγραμμα αυτό δημιουργεί την κλάση SimpleMonitor που κληρονομεί από την κλάση simple_switch_13.SimpleSwitch13 της εφαρμογής Simple Switch που περιεγράφηκε εκτενώς στο Κεφάλαιο 3. Συνεπώς, η εκτέλεση αυτού του

προγράμματος επιτρέπει την επικοινωνία μεταξύ τερματικών στο υφιστάμενο δίκτυο. Ένας μεταγωγέας γίνεται «στόχος» όταν ανιχνευθεί αλλαγή στο μονοπάτι δεδομένων (*datapath*) και αυτό διαρκεί από τη στιγμή που γίνεται MAIN_DISPATCHER έως ότου γίνει DEAD DISPATCHER.

Σημειώνεται, όμως, πως αυτό δεν είναι αρκετό για τοπολογίες που περιλαμβάνουν βρόχους. Σε τέτοιες περιπτώσεις απαιτείται η παράλληλη χρήση και εφαρμογή του δικτυακού πρωτοκόλλου *Spanning Tree*.

Υπενθυμίζεται, πως πρόκειται για πρωτόκολλο με στόχο την κατάργηση των βρόχων επιτρέποντας παράλληλα την ύπαρξη πλεοναζόντων συνδέσεων, όπως είναι αναγκαίο για την εξασφάλιση της επικοινωνίας σε περίπτωση που αντιμετωπίζει πρόβλημα κάποια ζεύξη και όπως είναι, φυσικά, απαραίτητο για τα πλαίσια που εξετάζονται στην παρούσα εργασία.

Για το σκοπό αυτό επεκτείνεται η λειτουργικότητα του προγράμματος έτσι ώστε η κλάση SimpleMonitor να κληρονομεί από μία εφαρμογή Simple Switch που εφαρμόζει παράλληλα το πρωτόκολλο STP.

Αξίζει να σημειωθεί, πως η χρήση του πρωτοκόλλου STP μπορεί και πρέπει να αποφευχθεί στην περίπτωση της ενεργειακής αναδιάρθρωσης του δικτύου, καθώς το ρόλο αυτού πρέπει να αναλάβει ο αλγόριθμος βελτιστοποίησης επιτελώντας τον με αποδοτικότερο τρόπο και πιο εξειδικευμένα κριτήρια.

Το τελευταίο βήμα προς την πραγματοποίηση του στόχου της παρούσας εργασίας είναι η τροποποίηση της κατάστασης των θυρών που έχει υποδείξει το πρόγραμμα βελτιστοποίησης. Αναπτύχθηκε, λοιπόν, το πρόγραμμα ***modify_ports.py*** με στόχο να ορίζει ποιες θύρες θα επιτρέπουν ροή δεδομένων και ποιες όχι. Για κάθε μία από τις θύρες που συναντώνται στο αρχείο-πόρισμα του προγράμματος βελτιστοποίησης:

- Ο ελεγκτής στέλνει μήνυμα OFPPortMod στον μεταγωγέα που φέρει τη συγκεκριμένη θύρα. Στο μήνυμα διευκρινίζονται το *datapath_id*, η θύρα και η κατάσταση στην οποία αυτή πρέπει να εισέλθει.
- Με τη λήψη του παραπάνω μηνύματος, ο μεταγωγέας πραγματοποιεί την μεταβολή και ακολούθως απαντά στον ελεγκτή με μήνυμα OFPPortStatus, για να τον ενημερώσει για την μεταβολή που συνέβη.
- Τέλος, όταν ο ελεγκτής ανιχνεύσει EventOFPPortStatus επιβεβαιώνει την τροποποίηση στην κατάσταση της συγκεκριμένης θύρας.

Τα τμήματα κώδικα που αναπτύχθηκαν και περιεγράφηκαν σε αυτή την παράγραφο συνοψίζονται στον πίνακα που ακολουθεί:

Πίνακας 6-1 Components που δημιουργήθηκαν στα πλαίσια της εργασίας

Χρησιμότητα	Όνομα Component	Αρχεία Εξόδου
Ανίχνευση της τοπολογίας του δικτύου	**get_topology_mini.py**	**topology.txt**
Δημιουργία στατιστικών στοιχείων για κάθε μεταγωγέα του δικτύου	**traffic_mon.py**	**τόσα αρχεία όσοι και οι μεταγωγείς του δικτύου, το καθένα από αυτά έχει όνομα τύπου: [DPID].txt**
Δημιουργία στατιστικών στοιχείων για κάθε μεταγωγέα του δικτύου και παράλληλη εφαρμογή του STP	**Traffic_mon_stp.py**	**τόσα αρχεία όσοι και οι μεταγωγείς του δικτύου, το καθένα από αυτά έχει όνομα τύπου: [DPID].txt**
Λαμβάνει αρχείο εισόδου που υποδεικνύει ποιες θύρες πρέπει να πάψουν να λειτουργούν ή να αρχίσουν να λειτουργούν και εφαρμόζει αυτούς τους κανόνες	**modify_ports.py**	**Κανένα**

7

Τεχνικές λεπτομέρειες

Για την υλοποίηση όσων αναφέρθηκαν στο Κεφάλαιο 6, απαιτείται η δημιουργία ενός εικονικού μηχανήματος και η εγκατάσταση κάποιων λογισμικών σε αυτό. Ακόμη, αναπτύχθηκαν ορισμένα τμήματα κώδικα και εκτελέστηκαν στο εικονικό μηχάνημα.

7.1 Εγκατάσταση Λογισμικών και Εργαλείων

Αρχικά, εγκαταστάθηκε το λογισμικό προσομοίωσης Mininet, που περιεγράφηκε αναλυτικά στο κεφάλαιο 4, ακολουθώντας τα παρακάτω βήματα:

- apt-get update: κάνει λήψη της λίστας των προγραμμάτων λογισμικού από τα αποθετήρια και τα ενημερώνει να λαμβάνουν πληροφορίες για τις νεότερες εκδόσεις των προγραμμάτων λογισμικού και των εξαρτήσεών τους
- apt-get install mininet
- apt-get install xterm: για την εγκατάσταση του προσομοιωτή τερματικού που ίσως χρειαστεί προκειμένου να υπάρχει η δυνατότητα χρήσης επιπρόσθετων παραθύρων κατά τη χρήση του Mininet

Για την εγκατάσταση του ελεγκτή Opendaylight, που όμως δεν χρησιμοποιήθηκε στην τελική υλοποίηση τα βήματα ήταν τα ακόλουθα:

- apt-get install maven git openjdk-7-jre openjdk-7-jdk: κάνει λήψη των προ απαιτούμενων λογισμικών maven, git, openjdk-7-jre και openjdk-7-jdk
- git clone https://git.opendaylight.org/gerrit/p/integration.git
- mvn clean install

Για να τρέξει ο ελεγκτής: χρήση της εντολής run ./bin/karaf από τη διεύθυνση όπου έχει αποθηκευτεί.

Αξίζει να σημειωθεί, ότι για να δοθεί οποιαδήποτε λειτουργικότητα σε αυτόν τον ελεγκτή πρέπει να εγκατασταθούν συγκεκριμένα τμήματα λογισμικού (*components*) από μία μακρά λίστα επιλογών με τον εξής τρόπο:

- feature: install <feature name>

Στη συνέχεια, δοκιμάστηκε ο ελεγκτής ONOS που εγκαταστάθηκε ως εξής:

- εγκατάσταση του προαπαιτούμενου OracleJDK8:
  - apt-get install software-properties-common -y
  - add-apt-repository ppa:webupd8team/java –y
  - apt-get update
  - apt-get install oracle-java8-installer oracle-java8-set-default –y
- apt-get install git
- git clone https://gerrit.onosproject.org/onos -b 1.0.1: για την εγκατάσταση της τελευταίας σταθερής έκδοσης (*Advocet*)
- wget http://www.apache.org/dist/maven/binaries/apache-maven-3.2.2-bin.tar.gz: εγκατάσταση maven
- wget http://download.nextag.com/apache/karaf/3.0.2/apache-karaf-3.0.2.tar.gz: εγκατάσταση karaf
- tar –zxvf <maven filename> -C *directory*
- tar –zxvf <karaf filename> -C *directory*
- mvn clean install

Τελικά, εγκαταστάθηκε και επιλέχθηκε ο ελεγκτής Ryu. Η εγκατάσταση του ήταν απλή και έγινε με χρήση της εντολής *pip install*. Υπενθυμίζεται, πως το *pip* είναι ένα σύστημα διαχείρισης τμημάτων λογισμικού που χρησιμοποιείται για την εγκατάσταση και τη διαχείριση οποιουδήποτε λογισμικού γραμμένου σε Python. Συγκεκριμένα, η εντολή που χρησιμοποιήθηκε είναι:

- pip install ryu

7.2 Ανάπτυξη Λογισμικών

Δημιουργία τοπολογίας Fat Tree – κλασική δομή

- Με τη χρήση του παρακάτω κώδικα (fattree_3.py) δημιουργείται η τοπολογία που παρουσιάστηκε στο Κεφάλαιο 4 και περιέχει τέσσερα *core switches, 8 aggregate switches* και *8 edge switches* (Σχήμα 4.1).

```
from mininet.topo import Topo

class FatTree( Topo ):

    def __init__( self ):
        iNumber = 4
        coreSwitchnumber = iNumber
        aggregateSwitchnumber = iNumber*2
        racknumber = iNumber*2
        racksize = iNumber/2

        # Initialize topology
        Topo.__init__( self )

        coreSwitches = []
        aggregateSwitches = []
        torSwitches = []

        '''
       Create the three layers of switches
        '''

        # Core Switches
        for x in range(0, coreSwitchnumber):
            coreSwitches.append(self.addSwitch("100" + str(x)))
        # Aggregate Switches
        for x in range(0, aggregateSwitchnumber):
            aggregateSwitches.append(self.addSwitch("200" + str(x)))
        # TOR Switches (Top-Of-Rack)
        for x in range(0, racknumber):
            torSwitches.append(self.addSwitch("300" + str(x)))

        '''
        Create links between layers of switches
        '''

        # Connections
        # Aggregate -> Core Connection
        for x in range(0, len(coreSwitches)/2):
            for y in range(0, len(aggregateSwitches),2):
                self.addLink(coreSwitches[x], aggregateSwitches[y])
        for x in range((len(coreSwitches)/2) ,len(coreSwitches)):
            for y in range(1, len(aggregateSwitches),2):
                self.addLink(coreSwitches[x], aggregateSwitches[y ])

        # TOR -> Aggregate Connection
        for x in range(0, len(aggregateSwitches),2):
                self.addLink(torSwitches[x], aggregateSwitches[x])
                self.addLink(torSwitches[x+1], aggregateSwitches[x])
                self.addLink(torSwitches[x], aggregateSwitches[x+1])
                self.addLink(torSwitches[x+1],
aggregateSwitches[x+1])
        '''
        Addition of hosts in the network
        '''

        # Hosts -> TOR switches
        for x in range(0, len(torSwitches)):
            for y in range(0, racksize):
```

```
                self.addLink(torSwitches[x],    self.addHost("400"    +
str(x) + str(y)))

topos = { 'fattree': ( lambda: FatTree() ) }
```

• Με τη χρήση του παρακάτω κώδικα (fattree_3_2_6_12.py) δημιουργείται τοπολογία που περιλαμβάνει 2 *core switches, 6 aggregate switches* και *12 edge switches* (Σχήμα 4.2).

```
from mininet.topo import Topo

class FatTree( Topo ):

    def __init__( self ):
        coreSwitchnumber = 2
        aggregateSwitchnumber = 6
        torSwitchnumber = 12
        racksize = 2

        # Initialize topology
        Topo.__init__( self )

        coreSwitches = []
        aggregateSwitches = []
        torSwitches = []

        # Core Switches
        for x in range(0, coreSwitchnumber):
            coreSwitches.append(self.addSwitch("100" + str(x)))
        # Aggregate Switches
        for x in range(0, aggregateSwitchnumber):
            aggregateSwitches.append(self.addSwitch("200" + str(x)))
        # TOR Switches (Top-Of-Rack)
        for x in range(0, torSwitchnumber):
            torSwitches.append(self.addSwitch("300" + str(x)))

        # Aggregate -> Core Connection
        for y in range(0, len(aggregateSwitches),2):
            self.addLink(coreSwitches[0], aggregateSwitches[y])
            print 'Core:' + str(0) + ' Ag:' + str(y)

        for y in range(1, len(aggregateSwitches),2):
            self.addLink(coreSwitches[1], aggregateSwitches[y])
            print 'Core:' + str(1) + ' Ag:' + str(y)

        # TOR -> Aggregate Connection
        for x in range(0, len(aggregateSwitches), 2):
            for y in range(x*len(torSwitches)/len(aggregateSwitches),
(x+2)*len(torSwitches)/len(aggregateSwitches),1):
                self.addLink(aggregateSwitches[x],torSwitches[y])
                self.addLink(aggregateSwitches[x+1],torSwitches[y])

        # Hosts -> TOR switches
        for x in range(0, len(torSwitches)):
            for y in range(0, racksize):
                self.addLink(torSwitches[x],    self.addHost("400"    +
str(x) + str(y)))

topos = { 'fattree': ( lambda: FatTree() ) }
```

Εφαρμογές του Ryu που χρησιμοποιήθηκαν

- Η εφαρμογή Simple Switch με την έκδοση 1.3 του πρωτοκόλλου OpenFlow

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet


class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst                                                    =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]

        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]
```

```
        dst = eth.dst
        src = eth.src

        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})

        self.logger.info("packet in %s %s %s %s", dpid, src, dst,
in_port)

        # learn a mac address to avoid FLOOD next time.
        self.mac_to_port[dpid][src] = in_port

        if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofproto.OFPP_FLOOD

        actions = [parser.OFPActionOutput(out_port)]

        # install a flow to avoid packet_in next time
        if out_port != ofproto.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
            self.add_flow(datapath, 1, match, actions)

        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out          =          parser.OFPPacketOut(datapath=datapath,
buffer_id=msg.buffer_id,
                                   in_port=in_port,   actions=actions,
data=data)
        datapath.send_msg(out)
```

Simple Switch με προσθήκη της λειτουργίας STP:

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import dpid as dpid_lib
from ryu.lib import stplib
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet


class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'stplib': stplib.Stp}

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.stp = kwargs['stplib']

        # Sample of stplib config.
        #  please refer to stplib.Stp.set_config() for details.
        config = {dpid_lib.str_to_dpid('0000000000000001'):
                     {'bridge': {'priority': 0x8000}},
                  dpid_lib.str_to_dpid('0000000000000002'):
```

```
                    {'bridge': {'priority': 0x9000}},
                dpid_lib.str_to_dpid('0000000000000003'):
                    {'bridge': {'priority': 0xa000}}}
        self.stp.set_config(config)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        # install table-miss flow entry
        #
        # We specify NO BUFFER to max_len of the output action due to
        # OVS bug. At this moment, if we specify a lesser number, e.g.,
        # 128, OVS will send Packet-In with invalid buffer_id and
        # truncated packet data. In that case, we cannot output packets
        # correctly.
        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst                                                              =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                             actions)]

        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)
        datapath.send_msg(mod)

    def delete_flow(self, datapath):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        for dst in self.mac_to_port[datapath.id].keys():
            match = parser.OFPMatch(eth_dst=dst)
            mod = parser.OFPFlowMod(
                datapath, command=ofproto.OFPFC_DELETE,
                out_port=ofproto.OFPP_ANY,
out_group=ofproto.OFPG_ANY,
                priority=1, match=match)
            datapath.send_msg(mod)

    @set_ev_cls(stplib.EventPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]

        dst = eth.dst
        src = eth.src
```

```
        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})

        self.logger.info("packet in %s %s %s %s", dpid, src, dst,
in_port)

        # learn a mac address to avoid FLOOD next time.
        self.mac_to_port[dpid][src] = in_port

        if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofproto.OFPP_FLOOD

        actions = [parser.OFPActionOutput(out_port)]

        # install a flow to avoid packet_in next time
        if out_port != ofproto.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
            self.add_flow(datapath, 1, match, actions)

        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out          =          parser.OFPPacketOut(datapath=datapath,
buffer_id=msg.buffer_id,
                                  in_port=in_port,   actions=actions,
data=data)
        datapath.send_msg(out)

    @set_ev_cls(stplib.EventTopologyChange, MAIN_DISPATCHER)
    def _topology_change_handler(self, ev):
        dp = ev.dp
        dpid_str = dpid_lib.dpid_to_str(dp.id)
        msg = 'Receive topology change event. Flush MAC table.'
        self.logger.debug("[dpid=%s] %s", dpid_str, msg)

        if dp.id in self.mac_to_port:
            self.delete_flow(dp)
            del self.mac_to_port[dp.id]

    @set_ev_cls(stplib.EventPortStateChange, MAIN_DISPATCHER)
    def _port_state_change_handler(self, ev):
        dpid_str = dpid_lib.dpid_to_str(ev.dp.id)
        of_state = {stplib.PORT_STATE_DISABLE: 'DISABLE',
                    stplib.PORT_STATE_BLOCK: 'BLOCK',
                    stplib.PORT_STATE_LISTEN: 'LISTEN',
                    stplib.PORT_STATE_LEARN: 'LEARN',
                    stplib.PORT_STATE_FORWARD: 'FORWARD'}
        self.logger.debug("[dpid=%s][port=%d] state=%s",
                          dpid_str,                    ev.port_no,
of_state[ev.port_state])
```

Δημιουργία τμημάτων λογισμικού (components) για τον ελεγκτή

- Τμήμα λογισμικού για την ανίχνευση της τοπολογίας του δικτύου (*component get_topology_mini.py*)

```
'''
 Module to Discover Network Topology
 Run    with    :    PYTHONPATH=.    ./bin/ryu-manager    --observe-links
ryu/app/get_topology_mini.py
 Results: extracts the topology in file topology.txt
 Format:
----------------    ----------------    --------    -----------------
----------
    Src DPID               Dst DPID             Port             Time:
1455823926.56
----------------    ----------------    --------    -----------------
----------
0000000000000003 -> 0000000000000002        3
0000000000000005 -> 0000000000000006        1
0000000000000004 -> 0000000000000002        3
0000000000000007 -> 0000000000000005        3
0000000000000002 -> 0000000000000003        1
0000000000000002 -> 0000000000000004        2

Explanation:
    If we want from Src DPID: 0000000000000003 to reach Dst DPID:
0000000000000002
    then the egress Port of Src DPID is 3.
    The Time represents the epoch time of your sample.
    See topology.txt file for the exact format.

'''
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from ryu.topology import event
from ryu.topology.api import get_switch, get_link, get_host
from ryu.topology import switches
import time
from ryu.lib import hub


'''
  Uncomment to Run Simple Learning Switch in parallel so that hosts
can reach each other
'''
# from ryu.app import simple_switch_13
# class SimpleMonitor(simple_switch_13.SimpleSwitch13):
#
#     def __init__(self, *args, **kwargs):
#         super(SimpleMonitor, self).__init__(*args, **kwargs)


class GetTopologyApp(app_manager.RyuApp):

    def __init__(self, *args, **kwargs):
        super(GetTopologyApp, self).__init__(*args, **kwargs)
        self.monitor_thread = hub.spawn(self._monitor)
        self.time_now = time.time()
        self.prv_time = self.time_now
```

```
        self.delta_time = 0
        self.max_age    =    20    #    Timer    in    seconds    to    call
_topo_change_handler
        self.min_age = 5     # Threshold between two samples. Avoid
logging to file sooner than 5 secs

    def _monitor(self):
        ev = 'Max Age timer caught'
        while True:
            self._topo_change_handler(ev)
            hub.sleep(self.max_age)

    @set_ev_cls([event.EventSwitchEnter,      event.EventSwitchLeave,
event.EventSwitchReconnected,                  event.EventPortModify,
ofp_event.EventOFPStateChange])
    def _topo_change_handler(self,ev):
        self.time_now = time.time()
        self.delta_time = self.time_now - self.prv_time
        if (self.delta_time > self.min_age):
            print('Entering _topo_change_handler : Reason %s'  % ev)
            switch_list = get_switch(self)
            switches=[switch.dp.id for switch in switch_list]
            links_list = get_link(self)

links=[(link.src.dpid,link.dst.dpid,{'port':link.src.port_no})    for
link in links_list]
            self.logger.info("Active Topology. Switches:%s Links:%s "
%(switches,links))
            self.prv_time = self.time_now
            try:
                with open('topology.txt', 'a') as f:
                    f.write( '----------------   ----------------   -
-------    ---------------------------\n'
                             '     Src  DPID                Dst  DPID
Port          Time: %s\n'
                             '----------------   ----------------   -
-------    ---------------------------\n' % self.time_now)
                    for link in links_list:
                        f.write('{:016x} -> {:016x} {:8d} \n'.format(
                                link.src.dpid,        link.dst.dpid,
link.src.port_no))
            except Exception,e:
                print e
```

- Τμήμα λογισμικού για τη συλλογή στατιστικών στοιχείων (*component traffic_mon.py*)

```
'''
 Module for Traffic Monitoring
 Run with : PYTHONPATH=. ./bin/ryu-manager ryu/app/traffic_mon.py
 Results: Monitors each port of each switch in the network.
          Each switch has its own log file (e.g. 1.txt represents the
log of switch 1 )

 Example Format : The file with name 1.txt has the following format

---------------- -------- -------- -------- -------- -------- -------
- -------- -------------------------
datapath         port     rx-pkts  rx-bytes rx-error tx-pkts  tx-bytes
tx-error Time: 1455825611.3
```

```
---------------- -------- -------- -------- -------- -------- -------
- ----------------------------------
0000000000000001        1       58     4656        0       59     4758
0
0000000000000001        2       55     4458        0       63     5046
0
0000000000000001 fffffffe        0        0        0       88     6936
0
---------------- -------- -------- -------- -------- -------- -------
- -------- -------------------------
datapath         port     rx-pkts  rx-bytes rx-error tx-pkts  tx-bytes
tx-error Time: 1455825616.29
---------------- -------- -------- -------- -------- -------- -------
- ----------------------------------
0000000000000001        1       58     4656        0       59     4758
0
0000000000000001        2       55     4458        0       63     5046
0
0000000000000001 fffffffe        0        0        0       88     6936
0


'''
from operator import attrgetter
from ryu.app import simple_switch_13
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub
from _ast import With
import time


class SimpleMonitor(simple_switch_13.SimpleSwitch13):

    def __init__(self, *args, **kwargs):
        super(SimpleMonitor, self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.monitor_thread = hub.spawn(self._monitor)


    @set_ev_cls(ofp_event.EventOFPStateChange,[MAIN_DISPATCHER,
DEAD_DISPATCHER])
    def _state_change_handler(self, ev):
        datapath = ev.datapath
        # Register a new Datapath
        if ev.state == MAIN_DISPATCHER:
            if not datapath.id in self.datapaths:
                self.logger.debug('register     datapath:     %016x',
datapath.id)
                self.datapaths[datapath.id] = datapath
        # Unregister a Datapath
        elif ev.state == DEAD_DISPATCHER:
            if datapath.id in self.datapaths:
                self.logger.debug('unregister    datapath:    %016x',
datapath.id)
                del self.datapaths[datapath.id]

    def _monitor(self):
        while True:
            for dp in self.datapaths.values():
```

```
                self._request_stats(dp)
            hub.sleep(5)

    def _request_stats(self, datapath):
        self.logger.debug('send stats request: %016x', datapath.id)
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        req      =      parser.OFPPortStatsRequest(datapath,      0,
ofproto.OFPP_ANY)
        datapath.send_msg(req)


    @set_ev_cls(ofp_event.EventOFPPortStatsReply, MAIN_DISPATCHER)
    def _port_stats_reply_handler(self, ev):
        body = ev.msg.body

        self.logger.info('datapath         port     '
                         'rx-pkts  rx-bytes rx-error '
                         'tx-pkts  tx-bytes tx-error')
        self.logger.info('---------------- -------- '
                         '-------- -------- -------- '
                         '-------- -------- --------')

        time_now = time.time()
        with open(str(ev.msg.datapath.id) + '.txt', 'a') as f:
            f.write( '---------------- -------- '
                     '-------- -------- -------- '
                     '-------- -------- -------- '
                     '-------------------------\n'
                     'datapath         port     '
                     'rx-pkts  rx-bytes rx-error '
                     'tx-pkts  tx-bytes tx-error | Time: %s\n'
                     '---------------- -------- '
                     '-------- -------- -------- '
                     '-------- -------- -------- '
                     '-------------------------\n' % time_now)
            self.logger.info("File %x created " % ev.msg.datapath.id)

        for stat in sorted(body, key=attrgetter('port_no')):
            self.logger.info('%016x %8x %8d %8d %8d %8d %8d %8d',
                             ev.msg.datapath.id, stat.port_no,
                             stat.rx_packets,          stat.rx_bytes,
stat.rx_errors,
                             stat.tx_packets,          stat.tx_bytes,
stat.tx_errors)

            try:
                with open(str(ev.msg.datapath.id) + '.txt', 'a') as f:
                    f.write('{:016x} {:8x} {:8d} {:8d} {:8d} {:8d}
{:8d} {:8d}\n'.format(ev.msg.datapath.id, stat.port_no,
                                stat.rx_packets,      stat.rx_bytes,
stat.rx_errors,
                                stat.tx_packets,      stat.tx_bytes,
stat.tx_errors))
            except Exception,e:
                self.logger.error("File Error!", e)
```

- Τμήμα λογισμικού για την τροποποίηση της κατάστασης των θυρών (*component modify_ports.py*)

```
"""
    Modify Ports , based on rules inside ports_mod.txt
    Uses OFPPortMod message to change port status/configuration
    Rules can have the Letters:
    -F : Forward all traffic
    -B : Block all traffic
    The Format Should be:  e.g.     For DPID 1000 make a Block rule on
PORT 1 and Forward rule on PORT 1 of DPID 2000.
    DPID      PORT     STATUS
    1000         1         B
    2000         1         F

    Output Sample:
    Parsing file:/home/sdn/git/ryu/ryu/app/ports_mod.txt
    Found        new        ports        to        modify        on
file:/home/sdn/git/ryu/ryu/app/ports_mod.txt  with  Rules:  {1000:  {1:
'B'}, 2000: {1: 'F'}}
    Modify port of DPID:1000 PortNo:1 Reason:B
    Modify port of DPID:2000 PortNo:1 Reason:F
    OFPPortStatus               received:               reason=MODIFY
desc=OFPPort(port_no=1,hw_addr='c2:aa:4f:79:e1:6f',name='1000-
eth1',config=101,state=1,curr=2112,advertised=0,supported=0,peer=0,cu
rr_speed=10000000,max_speed=0)
    OFPPortStatus               received:               reason=MODIFY
desc=OFPPort(port_no=1,hw_addr='c6:1e:08:ff:3a:51',name='2000-
eth1',config=0,state=1,curr=2112,advertised=0,supported=0,peer=0,curr
_speed=10000000,max_speed=0)

"""
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.mac import haddr_to_bin
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet
from ryu.lib.packet import ether_types
from operator import attrgetter
from ryu.app import simple_switch_13
from ryu.controller import ofp_event
from ryu.controller.handler import MAIN_DISPATCHER, DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.lib import hub
from _ast import With
import time ,os, re



class ModifyPorts(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]

    def __init__(self, *args, **kwargs):
        super(ModifyPorts, self).__init__(*args, **kwargs)
        self.datapaths = {}
        self.monitor_thread = hub.spawn(self._monitor)
        self.working_file                                              =
os.path.join(os.getcwd(),'ryu/app/ports_mod.txt')
        self.port_mod = {}
```

```
    @set_ev_cls(ofp_event.EventOFPStateChange,[MAIN_DISPATCHER,
DEAD_DISPATCHER])
    def _state_change_handler(self, ev):
        datapath = ev.datapath
        if ev.state == MAIN_DISPATCHER:
            if not datapath.id in self.datapaths:
                self.logger.debug('register     datapath:     %016x',
datapath.id)
                self.datapaths[datapath.id] = datapath

        elif ev.state == DEAD_DISPATCHER:
            if datapath.id in self.datapaths:
                self.logger.debug('unregister    datapath:    %016x',
datapath.id)
                del self.datapaths[datapath.id]

    def _monitor(self):
        while True:
            self._parse_file()
            for dp in self.datapaths.values():
                self._print_port_status(dp)
                self._modify_port_status(dp)
            hub.sleep(10)

    def _parse_file(self):
        '''
        Function to parse "ports_mod.txt" file, and to decide the
modified ports.
        Updates the self.port_mod dictionary with the ports of each
dpid that needs to be modified.
        e.g.

        '''
        self.port_mod = {}
        self.logger.info("Parsing file:%s"%self.working_file)
        try:
            with open(self.working_file,'r') as f:
                lines = f.readlines()
                 # Avoid lines with # or blank lines
                filtered      =      filter(lambda      x:      not
re.match(r'\s*#|^\s*$', x), lines)
                for line in filtered:
                    dpid = int(line.strip().split()[0])
                    port_no = int(line.strip().split()[1])
                    config = line.strip().split()[2]
                    self.port_mod.setdefault(dpid, {})
                    self.port_mod[dpid][port_no] = config
                self.logger.info("Found new ports to modify on file:%s
with Rules: %s"%(self.working_file, self.port_mod))
        except Exception as e:
            self.logger.error(e)


    def _print_port_status(self,datapath):
        self.logger.debug('Printing   Port   Status   on   DPID:   %s',
datapath.id)
        ofp = datapath.ofproto
        ofp_parser = datapath.ofproto_parser
        for port_no, port in datapath.ports.items():
            self.logger.debug("PortNo:%s , Port:%s" %(port_no,port))
```

```
    def _modify_port_status(self,datapath):
        """
        Function to modify port status. The rules are on the
self.port_mod dictionary.
        Uses the OFPPortMod message, to modify port status.
        """
        ofp = datapath.ofproto
        ofp_parser = datapath.ofproto_parser
        if datapath.id in self.port_mod:
            dpid_to_modify = datapath.id
            for port_no, port in datapath.ports.items():
                if port_no in self.port_mod[dpid_to_modify]:
                    reason = self.port_mod[dpid_to_modify][port_no]
                    self.logger.info("Modify port of DPID:%s PortNo:%s
Reason:%s" %(dpid_to_modify,port_no,reason))
                    if reason == 'B':
                        config      =      (ofp.OFPPC_PORT_DOWN      |
ofp.OFPPC_NO_RECV |
                                            ofp.OFPPC_NO_FWD                    |
ofp.OFPPC_NO_PACKET_IN)
                    elif reason == 'F':
                        config = 0
                    else:
                        self.logger.info("Unknown          Reason:%s"
%(reason))
                    mask = 0b1111111
                    req  =  ofp_parser.OFPPortMod(datapath,  port_no,
port.hw_addr, config,
                                                  mask)
                    datapath.send_msg(req)


    @set_ev_cls(ofp_event.EventOFPPortStatus, MAIN_DISPATCHER)
    def port_status_handler(self, ev):
        msg = ev.msg
        dp = msg.datapath
        ofp = dp.ofproto
        if msg.reason == ofp.OFPPR_ADD:
            reason = 'ADD'
        elif msg.reason == ofp.OFPPR_DELETE:
            reason = 'DELETE'
        elif msg.reason == ofp.OFPPR_MODIFY:
            reason = 'MODIFY'
        else:
            reason = 'unknown'
        self.logger.info('OFPPortStatus received: reason=%s desc=%s',
reason, msg.desc)
```

- Η εφαρμογή Simple Switch με παράλληλη εφαρμογή του πρωτοκόλλου STP (simple_switch_stp_13.py):

```
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import CONFIG_DISPATCHER, MAIN_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib import dpid as dpid_lib
from ryu.lib import stplib
```

```
from ryu.lib.packet import packet
from ryu.lib.packet import ethernet


class SimpleSwitch13(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    _CONTEXTS = {'stplib': stplib.Stp}

    def __init__(self, *args, **kwargs):
        super(SimpleSwitch13, self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.stp = kwargs['stplib']

        # Sample of stplib config.
        #  please refer to stplib.Stp.set_config() for details.
        config = {dpid_lib.str_to_dpid('0000000000000001'):
                     {'bridge': {'priority': 0x8000}},
                  dpid_lib.str_to_dpid('0000000000000002'):
                     {'bridge': {'priority': 0x9000}},
                  dpid_lib.str_to_dpid('0000000000000003'):
                     {'bridge': {'priority': 0xa000}}}
        self.stp.set_config(config)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures, CONFIG_DISPATCHER)
    def switch_features_handler(self, ev):
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match = parser.OFPMatch()
        actions = [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
                                          ofproto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)

    def add_flow(self, datapath, priority, match, actions):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
                                                actions)]

        mod = parser.OFPFlowMod(datapath=datapath, priority=priority,
                                match=match, instructions=inst)
        datapath.send_msg(mod)

    def delete_flow(self, datapath):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        for dst in self.mac_to_port[datapath.id].keys():
            match = parser.OFPMatch(eth_dst=dst)
            mod = parser.OFPFlowMod(
                datapath, command=ofproto.OFPFC_DELETE,
                out_port=ofproto.OFPP_ANY,
out_group=ofproto.OFPG_ANY,
                priority=1, match=match)
            datapath.send_msg(mod)

    @set_ev_cls(stplib.EventPacketIn, MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
```

```
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocols(ethernet.ethernet)[0]

        dst = eth.dst
        src = eth.src

        dpid = datapath.id
        self.mac_to_port.setdefault(dpid, {})

        self.logger.info("packet in %s %s %s %s", dpid, src, dst,
in_port)

        # learn a mac address to avoid FLOOD next time.
        self.mac_to_port[dpid][src] = in_port

        if dst in self.mac_to_port[dpid]:
            out_port = self.mac_to_port[dpid][dst]
        else:
            out_port = ofproto.OFPP_FLOOD

        actions = [parser.OFPActionOutput(out_port)]

        # install a flow to avoid packet_in next time
        if out_port != ofproto.OFPP_FLOOD:
            match = parser.OFPMatch(in_port=in_port, eth_dst=dst)
            self.add_flow(datapath, 1, match, actions)

        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(datapath=datapath,
buffer_id=msg.buffer_id,
                                  in_port=in_port, actions=actions,
data=data)
        datapath.send_msg(out)

    @set_ev_cls(stplib.EventTopologyChange, MAIN_DISPATCHER)
    def _topology_change_handler(self, ev):
        dp = ev.dp
        dpid_str = dpid_lib.dpid_to_str(dp.id)
        msg = 'Receive topology change event. Flush MAC table.'
        self.logger.debug("[dpid=%s] %s", dpid_str, msg)

        if dp.id in self.mac_to_port:
            self.delete_flow(dp)
            del self.mac_to_port[dp.id]

    @set_ev_cls(stplib.EventPortStateChange, MAIN_DISPATCHER)
    def _port_state_change_handler(self, ev):
        dpid_str = dpid_lib.dpid_to_str(ev.dp.id)
        of_state = {stplib.PORT_STATE_DISABLE: 'DISABLE',
                    stplib.PORT_STATE_BLOCK: 'BLOCK',
                    stplib.PORT_STATE_LISTEN: 'LISTEN',
                    stplib.PORT_STATE_LEARN: 'LEARN',
```

```
                stplib.PORT_STATE_FORWARD: 'FORWARD'}
    self.logger.debug("[dpid=%s][port=%d] state=%s",
                              dpid_str,                  ev.port_no,
of_state[ev.port_state])
```

8

Εκτέλεση πειράματος με χρήση των λογισμικών που αναπτύχθηκαν και τελικά συμπεράσματα

Στο Κεφάλαιο αυτό παρουσιάζεται η εκτέλεση όσων αναπτύχθηκαν και παρουσιάστηκαν στα Κεφάλαια 6 και 7 συνοδευόμενη από στιγμιότυπα οθόνης καθώς και τα τελικά συμπεράσματα της εργασίας.

8.1 Εκτέλεση στο εικονικό μηχάνημα

Προκειμένου να διεξαχθεί το πείραμα γίνεται σύνδεση *ssh* στο εικονικό μηχάνημα που δημιουργήθηκε για αυτό το σκοπό και λειτουργεί σε περιβάλλον Linux. Προκειμένου να πραγματοποιηθεί η σύνδεση έγινε χρήση της εφαρμογής εξομοίωσης τερματικών PuTTY σε λειτουργικό σύστημα Windows.
Στον πίνακα που ακολουθεί φαίνονται οι εκδόσεις των λογισμικών που χρησιμοποιήθηκαν για την προσομοίωση:

Πίνακας 8-1 Οι εκδόσεις των Mininet, OVS, Ryu

Mininet	**2.1.0**
OVS	**2.5.90**
Ryu	**3.3**

Το δίκτυο που εξομοιώνεται έχει τη μορφή *Fat Tree*, όπως αυτή παρουσιάζεται στο σχήμα 4.2 και αναλύεται στην παράγραφο 5.2.2.
Η εξομοίωση γίνεται εκτελώντας την ακόλουθη εντολή για ενεργοποίηση του Mininet:

sudo mn --controller=remote,IP=127.0.0.1 --mac --switch ovsk --custom ~/temp/fattree_3_2_6_12.py --topo fattree

Καταρχήν, για την εκτέλεση του Mininet επιβάλλεται ο χρήστης να έχει δικαιώματα διαχειριστή. Για το λόγο αυτό η εντολή *mn* που το εκκινεί ακολουθεί την εντολή sudo. Επιπλέον, το Mininet παρέχει αρκετές επιλογές εκκίνησης. Από αυτές έγινε χρήση των παρακάτω:

- Controller: η επιλογή αυτή υποδεικνύει πως ο ελεγκτής βρίσκεται απομακρυσμένα και ορίζει τη διεύθυνση *IP* του. Στην συγκεκριμένη περίπτωση ο *OpenFlow* ελεγκτής είναι εγκατεστημένος στο ίδιο εικονικό μηχάνημα. Για αυτό η διεύθυνση *IP* που του αντιστοιχεί είναι η διεύθυνση της διεπαφής *loopback*.
- Mac: με την επιλογή αυτή ορίζεται οι διευθύνσεις *MAC* να είναι μικρά, μοναδικά και απλά *ID* για λόγους απλούστευσης και διευκόλυνσης του χρήση κυρίως κατά τη διαδικασία της αποσφαλμάτωσης.
- Switch: με την επιλογή αυτή καθορίζεται να είναι τύπου ovsk (Open vSwitch) οι μεταγωγείς
- Custom: ορίζει πως η τοπολογία που θα προσομοιωθεί είναι ένα *Python script* καθώς και τη διαδρομή στην οποία θα βρεθεί αυτό το αρχείο.
- Topo: εδώ ορίζεται το όνομα της τοπολογίας

Κατά την εκκίνηση, λοιπόν, του Mininet, παρατηρούνται τα ακόλουθα:

```
*** Creating network
*** Adding controller
*** Adding hosts:
40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40
061 40070 40071 40080 40081 40090 40091 400100 400101 400110 400111
*** Adding switches:
1000 1001 2000 2001 2002 2003 2004 2005 3000 3001 3002 3003 3004 3005 3006 3007
3008 3009 30010 30011
*** Adding links:
(1000, 2000) (1000, 2002) (1000, 2004) (1001, 2001) (1001, 2003) (1001, 2005) (2
000, 3000) (2000, 3001) (2000, 3002) (2000, 3003) (2001, 3000) (2001, 3001) (200
1, 3002) (2001, 3003) (2002, 3004) (2002, 3005) (2002, 3006) (2002, 3007) (2003,
 3004) (2003, 3005) (2003, 3006) (2003, 3007) (2004, 3008) (2004, 3009) (2004, 3
0010) (2004, 30011) (2005, 3008) (2005, 3009) (2005, 30010) (2005, 30011) (3000,
 40000) (3000, 40001) (3001, 40010) (3001, 40011) (3002, 40020) (3002, 40021) (3
003, 40030) (3003, 40031) (3004, 40040) (3004, 40041) (3005, 40050) (3005, 40051
) (3006, 40060) (3006, 40061) (3007, 40070) (3007, 40071) (3008, 40080) (3008, 4
0081) (3009, 40090) (3009, 40091) (30010, 400100) (30010, 400101) (30011, 400110
) (30011, 400111)
*** Configuring hosts
40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40
061 40070 40071 40080 40081 40090 40091 400100 400101 400110 400111
*** Starting controller
*** Starting 20 switches
1000 1001 2000 2001 2002 2003 2004 2005 3000 3001 3002 3003 3004 3005 3006 3007
3008 3009 30010 30011
*** Starting CLI:
```

Στην παραπάνω εικόνα φαίνεται η δημιουργία του δικτύου, που περιλαμβάνει 20 hosts και 20 μεταγωγείς (Κεφ. 4), η σύνδεση με τον ελεγκτή και η δημιουργία των ζεύξεων.

Με την εκτέλεση της εντολής links εμφανίζονται οι συνδέσεις του δικτύου, ενδεικτικά:

```
40070 40070-eth0:3007-eth3
40071 40071-eth0:3007-eth4
40080 40080-eth0:3008-eth3
40081 40081-eth0:3008-eth4
40090 40090-eth0:3009-eth3
40091 40091-eth0:3009-eth4
400100 400100-eth0:30010-eth3
400101 400101-eth0:30010-eth4
400110 400110-eth0:30011-eth3
400111 400111-eth0:30011-eth4
1000 lo:  1000-eth1:2000-eth1 1000-eth2:2002-eth1 1000-eth3:2004-eth1
1001 lo:  1001-eth1:2001-eth1 1001-eth2:2003-eth1 1001-eth3:2005-eth1
2000 lo:  2000-eth1:1000-eth1 2000-eth2:3000-eth1 2000-eth3:3001-eth1 2000-eth4:3002-eth1 2000-eth5:3003-eth1
2001 lo:  2001-eth1:1001-eth1 2001-eth2:3000-eth2 2001-eth3:3001-eth2 2001-eth4:3002-eth2 2001-eth5:3003-eth2
2002 lo:  2002-eth1:1000-eth2 2002-eth2:3004-eth1 2002-eth3:3005-eth1 2002-eth4:3006-eth1 2002-eth5:3007-eth1
2003 lo:  2003-eth1:1001-eth2 2003-eth2:3004-eth2 2003-eth3:3005-eth2 2003-eth4:3006-eth2 2003-eth5:3007-eth2
2004 lo:  2004-eth1:1000-eth3 2004-eth2:3008-eth1 2004-eth3:3009-eth1 2004-eth4:30010-eth1 2004-eth5:30011-eth1
2005 lo:  2005-eth1:1001-eth3 2005-eth2:3008-eth2 2005-eth3:3009-eth2 2005-eth4:30010-eth2 2005-eth5:30011-eth2
3000 lo:  3000-eth1:2000-eth2 3000-eth2:2001-eth2 3000-eth3:40000-eth0 3000-eth4:40001-eth0
3001 lo:  3001-eth1:2000-eth3 3001-eth2:2001-eth3 3001-eth3:40010-eth0 3001-eth4:40011-eth0
3002 lo:  3002-eth1:2000-eth4 3002-eth2:2001-eth4 3002-eth3:40020-eth0 3002-eth4:40021-eth0
3003 lo:  3003-eth1:2000-eth5 3003-eth2:2001-eth5 3003-eth3:40030-eth0 3003-eth4:40031-eth0
3004 lo:  3004-eth1:2002-eth2 3004-eth2:2003-eth2 3004-eth3:40040-eth0 3004-eth4:40041-eth0
3005 lo:  3005-eth1:2002-eth3 3005-eth2:2003-eth3 3005-eth3:40050-eth0 3005-eth4:40051-eth0
3006 lo:  3006-eth1:2002-eth4 3006-eth2:2003-eth4 3006-eth3:40060-eth0 3006-eth4:40061-eth0
3007 lo:  3007-eth1:2002-eth5 3007-eth2:2003-eth5 3007-eth3:40070-eth0 3007-eth4:40071-eth0
3008 lo:  3008-eth1:2004-eth2 3008-eth2:2005-eth2 3008-eth3:40080-eth0 3008-eth4:40081-eth0
3009 lo:  3009-eth1:2004-eth3 3009-eth2:2005-eth3 3009-eth3:40090-eth0 3009-eth4:40091-eth0
30010 lo:  30010-eth1:2004-eth4 30010-eth2:2005-eth4 30010-eth3:400100-eth0 30010-eth4:400101-eth0
30011 lo:  30011-eth1:2004-eth5 30011-eth2:2005-eth5 30011-eth3:400110-eth0 30011-eth4:400111-eth0
```

Με την εκτέλεση της εντολής nodes φαίνονται οι κόμβοι του δικτύου, ενδεικτικά:

```
available nodes are:
1000 1001 2000 2001 2002 2003 2004 2005 3000 3001 30010 30011 3002 3003 3004 3005 3006 3007 3008 3009 40000 40001 40010 400100 40
030 40031 40040 40041 40050 40051 40060 40061 40070 40071 40080 40081 40090 40091 c0
```

Με την εκτέλεση της εντολής dump εμφανίζονται πληροφορίες για όλους τους κόμβους, ενδεικτικά:

```
<Host 40000: 40000-eth0:10.0.0.1 pid=25691>
<Host 40001: 40001-eth0:10.0.0.2 pid=25692>
<Host 40010: 40010-eth0:10.0.0.3 pid=25693>
<Host 40011: 40011-eth0:10.0.0.4 pid=25696>
<Host 40020: 40020-eth0:10.0.0.5 pid=25697>
<Host 40021: 40021-eth0:10.0.0.6 pid=25698>
<Host 40030: 40030-eth0:10.0.0.7 pid=25699>
<Host 40031: 40031-eth0:10.0.0.8 pid=25700>
<Host 40040: 40040-eth0:10.0.0.9 pid=25701>
<Host 40041: 40041-eth0:10.0.0.10 pid=25702>
<OVSSwitch 3002: lo:127.0.0.1,3002-eth1:None,3002-eth2:None,3002-eth3:None,3002-eth4:None pid=25769>
<OVSSwitch 3003: lo:127.0.0.1,3003-eth1:None,3003-eth2:None,3003-eth3:None,3003-eth4:None pid=25774>
<OVSSwitch 3004: lo:127.0.0.1,3004-eth1:None,3004-eth2:None,3004-eth3:None,3004-eth4:None pid=25779>
<OVSSwitch 3005: lo:127.0.0.1,3005-eth1:None,3005-eth2:None,3005-eth3:None,3005-eth4:None pid=25784>
<OVSSwitch 3006: lo:127.0.0.1,3006-eth1:None,3006-eth2:None,3006-eth3:None,3006-eth4:None pid=25789>
<OVSSwitch 3007: lo:127.0.0.1,3007-eth1:None,3007-eth2:None,3007-eth3:None,3007-eth4:None pid=25794>
<OVSSwitch 3008: lo:127.0.0.1,3008-eth1:None,3008-eth2:None,3008-eth3:None,3008-eth4:None pid=25799>
<OVSSwitch 3009: lo:127.0.0.1,3009-eth1:None,3009-eth2:None,3009-eth3:None,3009-eth4:None pid=25804>
<OVSSwitch 30010: lo:127.0.0.1,30010-eth1:None,30010-eth2:None,30010-eth3:None,30010-eth4:None pid=25809>
<OVSSwitch 30011: lo:127.0.0.1,30011-eth1:None,30011-eth2:None,30011-eth3:None,30011-eth4:None pid=25814>
<RemoteController c0: 127.0.0.1:6633 pid=25684>
```

Σε νέο τερματικό, παράλληλα με το Mininet μπορεί να γίνει εκτέλεση της εφαρμογής που δημιουργήθηκε. Μπορεί να δοκιμαστεί η λειτουργία κάθε τμήματος ξεχωριστά, εκτελώντας την παρακάτω εντολή, η οποία χρησιμοποιεί για παράδειγμα το *Traffic Monitor* που κληρονομεί από το Simple Switch που εφαρμόζει το *STP*:

sudo PYTHONPATH=. ./bin/ryu run ryu/app/traffic_mon_stp.py

Ωστόσο, μπορεί να εκτελεστεί ταυτόχρονα οποιοσδήποτε συνδυασμός των *components* που έχουν δημιουργηθεί ανάλογα με τις ανάγκες που υπάρχουν.

Σημειώνεται, πως η παράμετρος – verbose στην παραπάνω εντολή αφορά την εμφάνιση μηνυμάτων για σκοπούς αποσφαλμάτωσης (*debugging*) που συμπεριλαμβάνονται στον κώδικα που εκτελείται. Παράδειγμα τέτοιας εντολής αποτελεί η

```
self.logger.debug('register datapath: %016x', datapath.id)
```

που συναντάται στο *traffic_mon.py*.

Για τις ανάγκες της εργασίας πραγματοποιούνται ξεχωριστά τα ακόλουθα πειράματα/σενάρια:

Ανίχνευση Τοπολογίας:

Εκτέλεση της εντολής:

```
sudo PYTHONPATH=. ./bin/ryu run –observe-links ryu/app/get_topology_mini.py
```

Σε αυτή την περίπτωση είναι απαραίτητη η παράμετρος –observe-links που αφορά την ανίχνευση συνδέσεων προκειμένου να είναι επιτυχής η εκτέλεση.

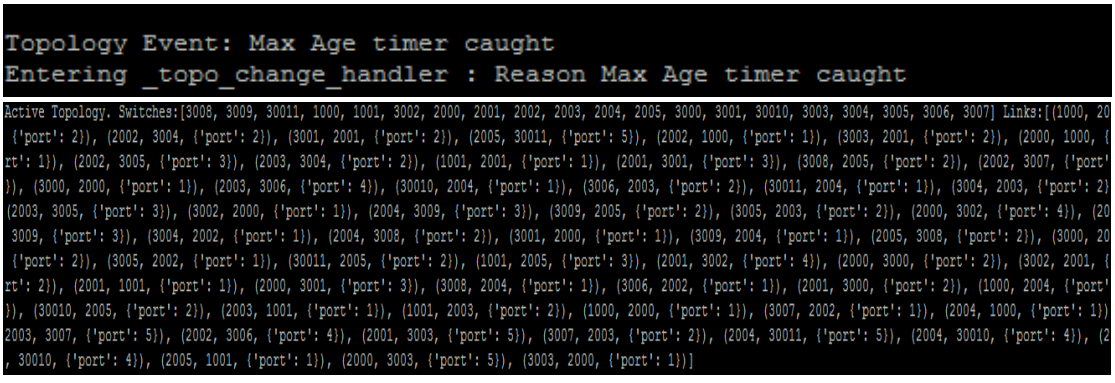
Σκοπός είναι η παρατήρηση της δημιουργίας του αρχείου που περιέχει την τοπολογία που προσομοιώνεται (topology.txt). Τα αποτελέσματα μπορούν να αξιοποιηθούν από οποιαδήποτε εφαρμογή χρειάζεται να διατηρεί εποπτεία της υπάρχουσας τοπολογίας.

Καταρχήν, εμφανίζεται ο λόγος που γίνεται εκκίνηση της διαδικασίας ανίχνευσης τοπολογίας, όπως υποδεικνύει η εντολή:

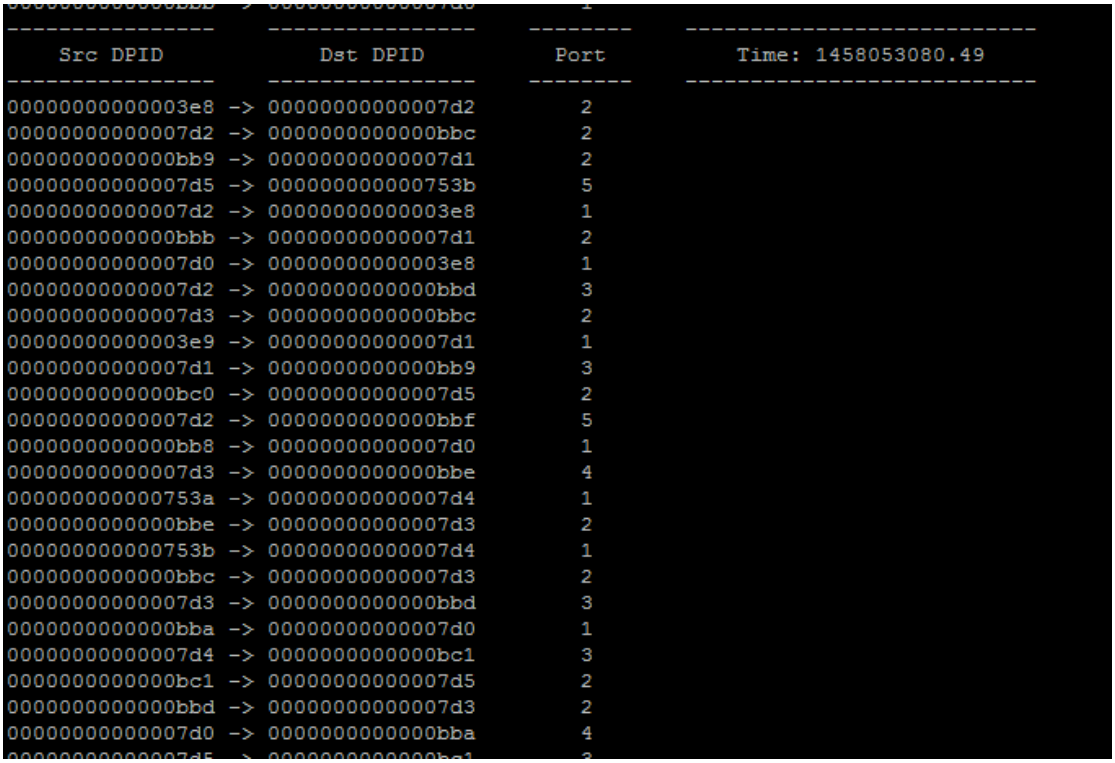
```
print('Entering _topo_change_handler : Reason %s' % ev)
```

Ακόμη, κάθε φορά που γίνεται η ανίχνευση εμφανίζονται μηνύματα που περιλαμβάνουν τους μεταγωγείς και τις συνδέσεις μεταξύ του και ακολούθως αυτά τα στοιχεία τοποθετούνται στο αρχείο της τοπολογίας:

```
self.logger.info("Active Topology. Switches:%s Links:%s " %(switches,links))
```



Η εμφάνιση του αρχείου που έχει πλέον δημιουργηθεί γίνεται με την εντολή cat topology.txt στη διεύθυνση που αυτό είναι αποθηκευμένο. Ενδεικτικά κάποιες από τις εγγραφές που περιλαμβάνει:



Src DPID		Dst DPID	Port	Time: 1458053080.49
00000000000003e8	->	00000000000007d2	2	
00000000000007d2	->	0000000000000bbc	2	
0000000000000bb9	->	00000000000007d1	2	
00000000000007d5	->	000000000000753b	5	
00000000000007d2	->	00000000000003e8	1	
0000000000000bbb	->	00000000000007d1	2	
00000000000007d0	->	00000000000003e8	1	
00000000000007d2	->	0000000000000bbd	3	
00000000000007d3	->	0000000000000bbc	2	
00000000000003e9	->	00000000000007d1	1	
00000000000007d1	->	0000000000000bb9	3	
0000000000000bc0	->	00000000000007d5	2	
00000000000007d2	->	0000000000000bbf	5	
0000000000000bb8	->	00000000000007d0	1	
00000000000007d3	->	0000000000000bbe	4	
000000000000753a	->	00000000000007d4	1	
0000000000000bbe	->	00000000000007d3	2	
000000000000753b	->	00000000000007d4	1	
0000000000000bbc	->	00000000000007d3	2	
00000000000007d3	->	0000000000000bbd	3	
0000000000000bba	->	00000000000007d0	1	
00000000000007d4	->	0000000000000bc1	3	
0000000000000bc1	->	00000000000007d5	2	
0000000000000bbd	->	00000000000007d3	2	
00000000000007d0	->	0000000000000bba	4	

Σημειώνεται, πως καθώς σε αυτό το στάδιο της διαδικασίας δεν έχει γίνει κάποια αλλαγή στην τοπολογία, η εκκίνηση της ανίχνευσης γίνεται κάθε 20 δευτερόλεπτα, όπως έχει οριστεί. Η τιμή αυτή μπορεί να αλλάξει ανάλογα με τις ανάγκες του εκάστοτε πειράματος. Συνεπώς, στο αρχείο topology.txt παρατηρούνται οι ίδιες εγγραφές επανειλημμένα.

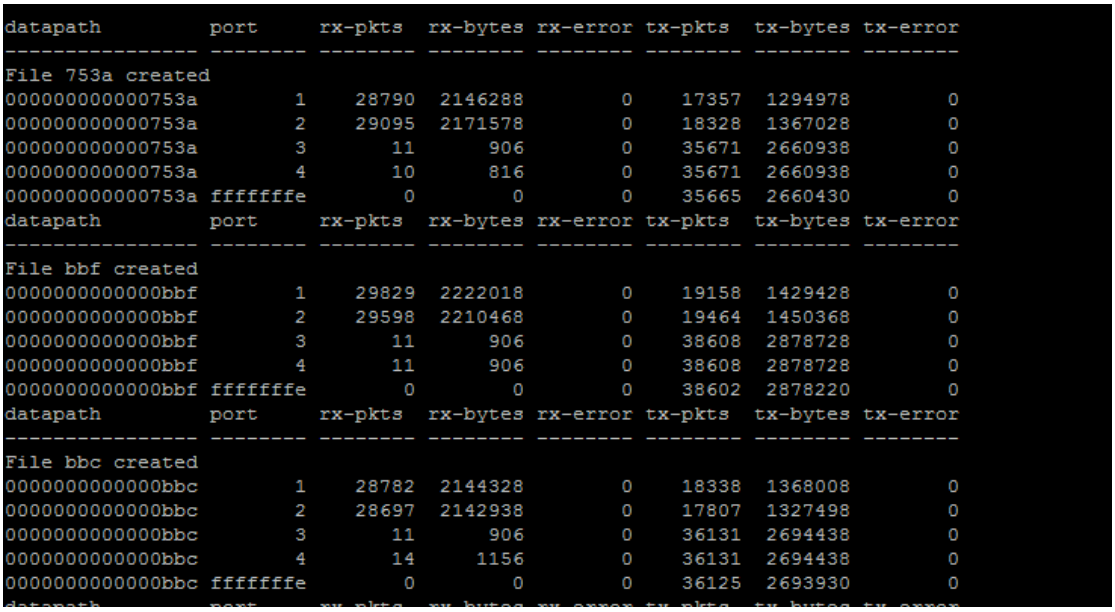
Δημιουργία αρχείων με στατιστικά στοιχεία για τις δικτυακές συσκευές

Εκτέλεση της εντολής:

```
sudo PYTHONPATH=. ./bin/ryu run –observe-links  ryu/app/traffic_mon.py
```

Σκοπός είναι η παρατήρηση της δημιουργίας των 20 αρχείων (ένα για κάθε μεταγωγέα) που περιλαμβάνουν τα στατιστικά στοιχεία. Σημειώνεται, πως σε αυτό το σημείο δεν μπορεί να πραγματοποιηθεί επικοινωνία μεταξύ των τερματικών στη συγκεκριμένη τοπολογία, λόγω της ύπαρξης των βρόχων. Σε μία πιο απλή τοπολογία (π.χ. ένα απλό δένδρο) θα μπορούσε να πραγματοποιηθεί λόγω της εφαρμογής Simple Switch, από την οποία κληρονομεί το Traffic Monitor. Τα αποτελέσματα μπορούν να αξιοποιηθούν από οποιαδήποτε σουίτα βελτιστοποίησης.

Τα μηνύματα που εμφανίζονται κατά τη διάρκεια της εκτέλεσης της παραπάνω εντολής παρουσιάζουν τις εγγραφές που γίνονται στα αρχεία:



```
datapath         port     rx-pkts  rx-bytes rx-error tx-pkts  tx-bytes tx-error
---------------- -------- -------- -------- -------- -------- -------- --------
File 753a created
000000000000753a        1    28790  2146288        0    17357  1294978        0
000000000000753a        2    29095  2171578        0    18328  1367028        0
000000000000753a        3       11      906        0    35671  2660938        0
000000000000753a        4       10      816        0    35671  2660938        0
000000000000753a fffffffe        0        0        0    35665  2660430        0
datapath         port     rx-pkts  rx-bytes rx-error tx-pkts  tx-bytes tx-error
---------------- -------- -------- -------- -------- -------- -------- --------
File bbf created
0000000000000bbf        1    29829  2222018        0    19158  1429428        0
0000000000000bbf        2    29598  2210468        0    19464  1450368        0
0000000000000bbf        3       11      906        0    38608  2878728        0
0000000000000bbf        4       11      906        0    38608  2878728        0
0000000000000bbf fffffffe        0        0        0    38602  2878220        0
datapath         port     rx-pkts  rx-bytes rx-error tx-pkts  tx-bytes tx-error
---------------- -------- -------- -------- -------- -------- -------- --------
File bbc created
0000000000000bbc        1    28782  2144328        0    18338  1368008        0
0000000000000bbc        2    28697  2142938        0    17807  1327498        0
0000000000000bbc        3       11      906        0    36131  2694438        0
0000000000000bbc        4       14     1156        0    36131  2694438        0
0000000000000bbc fffffffe        0        0        0    36125  2693930        0
```

Σημειώνεται, πως τα ονόματα των αρχείων είναι στο δεκαδικό σύστημα, ενώ στα μηνύματα log στο δεκαεξαδικό. Για παράδειγμα στην παραπάνω εικόνα το αρχείο 753a αντιστοιχεί στο αρχείο 30010.txt, στο αρχείο δηλαδή του προ τελευταίου μεταγωγέα στο επίπεδο πρόσβασης (Κεφ. 4 – εικόνα****)

Αλλαγή στην τοπολογία με την εφαρμογή κανόνων σε ορισμένες θύρες

Εκτέλεση της εντολής:

```
sudo PYTHONPATH=. ./bin/ryu run –observe-links ryu/app/get_topology_mini.py ryu/app/modify_ports.py
```

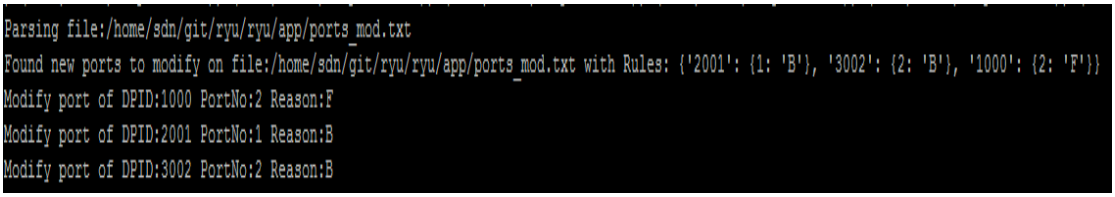
Σκοπός είναι η εφαρμογή κανόνων σε κάποιες δικτυακές θύρες και ακολούθως η παρατήρηση της αλλαγής της τοπολογίας στο αντίστοιχο αρχείο.

Το αρχείο txt θεωρείται ότι έχει την ακόλουθη μορφή:

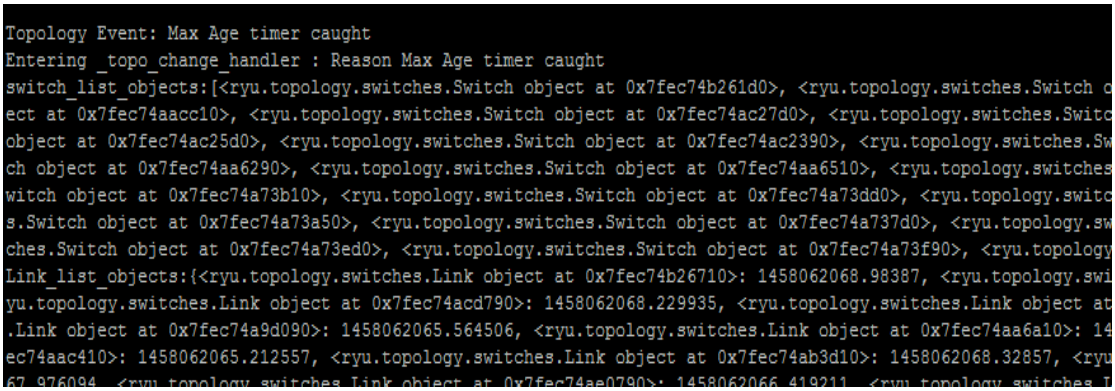
# DPID	**# PORT**	**# RULE**
1000	**2**	**F**
3002	**2**	**B**
2001	**1**	**B**

Υποδεικνύει δηλαδή την απαγόρευση διέλευσης κίνησης από τη θύρα 1 του μεταγωγέα 2001 και από την θύρα 2 του μεταγωγέα 3002.

Κατά την εκτέλεση της παραπάνω εντολής εμφανίζονται μηνύματα που αναφέρουν τις αιτίες για τις οποίες καλείται η συνάρτηση που εκτελεί την ανίχνευση της τοπολογίας και οι αλλαγές που πρέπει να εφαρμοστούν. Ενδεικτικά παρουσιάζονται κάποια από τα μηνύματα:



```
Parsing file:/home/sdn/git/ryu/ryu/app/ports_mod.txt
Found new ports to modify on file:/home/sdn/git/ryu/ryu/app/ports_mod.txt with Rules: {'2001': {1: 'B'}, '3002': {2: 'B'}, '1000': {2: 'F'}}
Modify port of DPID:1000 PortNo:2 Reason:F
Modify port of DPID:2001 PortNo:1 Reason:B
Modify port of DPID:3002 PortNo:2 Reason:B
```



Επισημαίνεται, πως στην αρχική τοπολογία υπάρχουν οι ζεύξεις 2001-1001 και 3002-2001. Συνεπώς στο αρχείο της τοπολογίας αυτό θα εμφανιζόταν ως εξής:

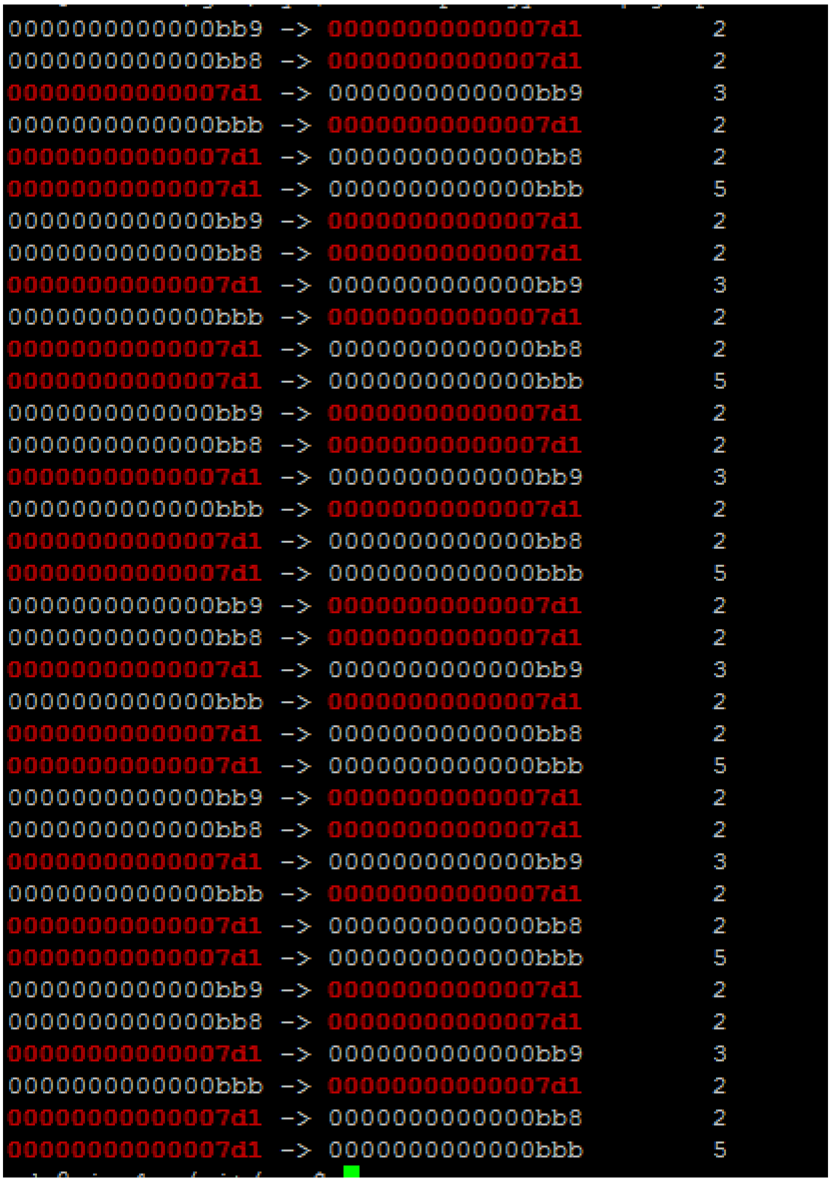
0000000000000bba -> 00000000000007d1 2

00000000000007d1 -> 00000000000003e9 1

Με την εκτέλεση των εντολών cat topology.txt | grep ‘0000000000000bba’ και cat topology.txt | grep ‘00000000000007d’ εμφανίζονται οι εγγραφές που αφορούν συνδέσεις τον μεταγωγέων 3002 και 2001:

Παρατηρείται, λοιπόν, να εμφανίζεται η αμφίδρομη ζεύξη μεταξύ 3002 και 2000 έξι φορές. Η επανάληψη των δύο εγγραφών οφείλεται στο ότι η συνάρτηση που ανιχνεύει την τοπολογία εκτελέστηκε τόσες φορές κατά τη διάρκεια του πειράματος. Επιβεβαιώνεται, έτσι, πως η απαγόρευση λειτουργίας στη θύρα 2 του μεταγωγέα 3002 οδήγησε σε κατάργηση αυτής της εγγραφής από το αρχείο.

Αντίστοιχα παρατηρούνται και για τον μεταγωγέα 2001, με τη διαφορά ότι εκείνος διατηρεί 3 συνδέσεις:



```
00000000000000bb9 -> 00000000000007d1        2
00000000000000bb8 -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb9        3
00000000000000bbb -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb8        2
00000000000007d1 -> 0000000000000bbb        5
00000000000000bb9 -> 00000000000007d1        2
00000000000000bb8 -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb9        3
00000000000000bbb -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb8        2
00000000000007d1 -> 0000000000000bbb        5
00000000000000bb9 -> 00000000000007d1        2
00000000000000bb8 -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb9        3
00000000000000bbb -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb8        2
00000000000007d1 -> 0000000000000bbb        5
00000000000000bb9 -> 00000000000007d1        2
00000000000000bb8 -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb9        3
00000000000000bbb -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb8        2
00000000000007d1 -> 0000000000000bbb        5
00000000000000bb9 -> 00000000000007d1        2
00000000000000bb8 -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb9        3
00000000000000bbb -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb8        2
00000000000007d1 -> 0000000000000bbb        5
00000000000000bb9 -> 00000000000007d1        2
00000000000000bb8 -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb9        3
00000000000000bbb -> 00000000000007d1        2
00000000000007d1 -> 0000000000000bb8        2
00000000000007d1 -> 0000000000000bbb        5
```

Δημιουργία αρχείων με στατιστικά στοιχεία και παράλληλη εφαρμογή του STP

Εκτέλεση της εντολής:

```
sudo PYTHONPATH=. ./bin/ryu run –observe-links ryu/app/traffic_mon_stp.py
```

Σκοπός σε αυτό το σημείο είναι και πάλι η ανάκτηση των στατιστικών δεδομένων επιτρέποντας παράλληλα την επικοινωνία μεταξύ των τερματικών εφαρμόζοντας τις αρχές του πρωτοκόλλου STP.

Σημειώνεται, πως σε αυτό το πείραμα ουσιαστικά υποκαθίσταται η λειτουργία της ενεργειακής βελτιστοποίησης και του *component* για την τροποποίηση της κατάστασης των θυρών από αυτή του πρωτοκόλλου STP, καθώς το τελευταίο αναλαμβάνει την κατάργηση των περιττών ζεύξεων.

Κατά την εκτέλεση, λοιπόν, της παραπάνω εντολής, παρατηρεί ο χρήστης αρχικά τα μηνύματα που αφορούν τα στάδια λειτουργίας του STP, όπως αυτά έχουν περιγραφεί στο κεφάλαιο 3. Ενδεικτικά, παρουσιάζονται τα ακόλουθα στιγμιότυπα:

```
[STP][INFO] dpid=00000000000007d2: [port=1] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000007d2: [port=2] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000007d2: [port=3] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000007d2: [port=4] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000007d5: Join as stp bridge.
[STP][INFO] dpid=00000000000007d2: [port=5] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000007d5: [port=1] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000007d5: [port=2] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000007d5: [port=3] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000007d5: [port=4] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000007d3: Join as stp bridge.
[STP][INFO] dpid=00000000000007d5: [port=5] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000bba: [port=1] ROOT_PORT           / LISTEN
[STP][INFO] dpid=0000000000000bba: [port=2] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000bba: [port=3] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000bba: [port=4] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000003e8: [port=3] Receive superior BPDU.
[STP][INFO] dpid=00000000000003e8: [port=1] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=00000000000003e8: [port=2] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=00000000000003e8: [port=3] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=00000000000003e8: Root bridge.
[STP][INFO] dpid=00000000000003e8: [port=1] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000003e8: [port=2] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=00000000000003e8: [port=3] DESIGNATED_PORT     / LISTEN
[STP][INFO] dpid=0000000000000bb9: [port=2] Receive superior BPDU.
[STP][INFO] dpid=0000000000000bb9: [port=1] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000bb9: [port=2] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000bb9: [port=3] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000bb9: [port=4] DESIGNATED_PORT     / BLOCK
[STP][INFO] dpid=0000000000000bb9: Root bridge.
```

Ακόμη, εκτελείται δοκιμή pingall, δηλαδή αποστολή μηνυμάτων ICMP από όλα τα τερματικά προς τα υπόλοιπα, για να επαληθευτεί η επιτυχής επικοινωνία:

```
*** Results: 0% dropped (548/552 received)
mininet> pingall
*** Ping: testing ping reachability
40000 -> 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 40070 4007
40001 -> 40000 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 40070 4007
40010 -> 40000 40001 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 40070 4007
40011 -> 40000 40001 40010 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 40070 4007
40020 -> 40000 40001 40010 40011 40021 40030 40031 40040 40041 40050 40051 40060 40061 40070 4007
40021 -> 40000 40001 40010 40011 40020 40030 40031 40040 40041 40050 40051 40060 40061 40070 4007
40030 -> 40000 40001 40010 40011 40020 40021 40031 40040 40041 40050 40051 40060 40061 40070 4007
40031 -> 40000 40001 40010 40011 40020 40021 40030 40040 40041 40050 40051 40060 40061 40070 4007
40040 -> 40000 40001 40010 40011 40020 40021 40030 40031 40041 40050 40051 40060 40061 40070 4007
40041 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40050 40051 40060 40061 40070 4007
40050 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40051 40060 40061 40070 4007
40051 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40060 40061 40070 4007
40060 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40061 40070 4007
40061 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40070 4007
40070 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 4007
40071 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 4007
40080 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 4007
40081 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 4007
40090 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 4007
40091 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 4007
400100 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 400
400101 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 400
400110 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 400
400111 -> 40000 40001 40010 40011 40020 40021 40030 40031 40040 40041 40050 40051 40060 40061 400
*** Results: 0% dropped (552/552 received)
```

Σημειώνεται, πως απαιτείται η ολοκλήρωση των εργασιών του STP, για να υπάρξει επιτυχής επικοινωνία.

8.2 Συμπεράσματα

Καθώς η τεχνολογία *SDN* εισβάλει δυναμικά στο χώρο των δικτύων για να αντιμετωπίσει τις προκλήσεις του μέλλοντος και να βελτιώσει προβλήματα του παρελθόντος, δεν γινόταν παρά να χρησιμοποιηθεί σε πειραματικό στάδιο και στο χώρο των *Data Center* με στόχο την υιοθέτησή της κάποια στιγμή στο μέλλον. Για το λόγο αυτό αποτέλεσε έναυσμα για την εκπόνηση αυτής της διπλωματικής εργασίας που στοχεύει να διευκολύνει την επίλυση του ενεργειακού προβλήματος των δικτύων *Data Center* παρέχοντας την υποδομή για την εφαρμογή αλγορίθμων βελτιστοποίησης της ενεργειακής κατανάλωσης.
Με τη χρήση του ελεγκτή *Ryu*, που αποτελεί ιδιαίτερα χρηστικό ελεγκτή για ερευνητικές και πειραματικές μελέτες, αναπτύχθηκαν τα απαραίτητα εργαλεία που καθιστούν εφικτή την εποπτεία ενός τέτοιου δικτύου και τελικά την τροποποίηση της κατάστασης των θυρών των προωθητικών συσκευών.

Συγκεκριμένα τα τμήματα κώδικα (*components*) που αναπτύχθηκαν μπορούν χωρίς περαιτέρω επέκταση και επεξεργασία να πετύχουν τα ακόλουθα:

- Δυναμική παρακολούθηση και έλεγχο του δικτύου με την ταυτόχρονη εκτέλεση των τμημάτων *Traffic Monitor* και *Get Topology*.
- Δυναμική προσαρμογή και αναδιάρθρωση του δικτύου με την εκτέλεση του τμήματος *Port Modify*.
- Δυναμική προσαρμογή και αναδιάρθρωση του δικτύου με στόχο την αποφυγή βρόχων στη τοπολογία με την εκτέλεση του τμήματος *Traffic Monitor STP*.

Ακόμη, λαμβάνοντας υπόψη το ερευνητικό ενδιαφέρον που συναντάται γενικά στο χώρο των SDN και ειδικότερα στα δίκτυα *Data Center* η παρούσα εργασία θα μπορούσε να αποτελέσει εφαλτήριο για τη μελέτη των ακόλουθων:

- Επιλογή ενός ενεργειακού αλγορίθμου και έλεγχος της επίδοσης του σε διαφόρων διαστάσεων Fat Tree δίκτυα.
- Επιλογή κάποιων ενεργειακών αλγορίθμων και σύγκριση των επιδόσεων τους στη δημοφιλέστερη αρχιτεκτονική Fat Tree.
- Υιοθέτηση ενός ενεργειακού αλγορίθμου, πραγματοποίηση ενεργειακής βελτιστοποίησης με αυτόν και χρήση των πορισμάτων της για τροποποίηση της κατάστασης των θυρών του δικτύου Fat Tree που δημιουργήθηκε στα πλαίσια αυτής της εργασίας.

Αξίζει να σημειωθεί, πως τα αρχεία εξόδου των τμημάτων κώδικα που έχουν δημιουργηθεί, δηλαδή αυτό που περιλαμβάνει τη μορφή του δικτύου (topology.txt) και αυτά με τα στατιστικά στοιχεία που αφορούν κάθε κόμβο ξεχωριστά ([DPID].txt), μπορούν να συνεισφέρουν στην εφαρμογή οποιουδήποτε κεντρικοποιημένου αλγορίθμου. Για παράδειγμα, η δυναμική απόκτηση των στοιχείων που

περιλαμβάνουν τα προαναφερθέντα αρχεία θα μπορούσαν να εκμεταλλευτούν από εφαρμογές που αφορούν την εξισορρόπηση του φορτίου (*Load balancing*) ή την εφαρμογή κανόνων QoS στη δρομολόγηση.

Τέλος, επέκταση της εργασίας αυτής εκτός των πλαισίων των Data Center θα μπορούσε να αποτελέσει η αντικατάσταση της τοπολογίας, που έχει προσομοιωθεί από το λογισμικό *Mininet,* από κάποιο δίκτυο αποτελούμενο από φυσικούς μεταγωγείς συμβατούς με το πρωτόκολλο *OpenFlow* (HP 2920 Switch Series, Extreme Networks -–BlackDiamond X8, κοκ). Έτσι θα μπορούσε να μελετηθεί το πρόβλημα με δεδομένα πραγματικής κίνησης και σε πιο ρεαλιστικές συνθήκες, με στόχο την μακροπρόθεσμη υιοθέτηση της τεχνολογίας *SDN* για εμπορική χρήση όπου ενδεχομένως θα αποτελούσαν καταλληλότερη επιλογή οι ελεγκτές που προορίζονται για τέτοια χρήση, δηλαδή οι OpenDaylight και ONOS.

9

Βιβλιογραφία

[1] "OpenFlow Specifications Sheet v1.4.0," Open Networking Foundation, 2013.

[2] N. McKeown, G. Parulkar, T. Anderson, L. Peterson, H. Balakrishnan, J. Rexford, S. Shenker και J. Turner, «OpenFlow: Enabling Innovation in Campus Networks,» 2008.

[3] T. Varun, R. Parekh και V. Patel, «A Survey on Vulnerabilities of OpenFlow Network and its Impact on SDN/Openflow Controller,» *World Academics Journal of Engineering Sciences,* p. 5, 2014.

[4] K. Rowan, V. Kotronis και P. Smith, «Openflow: A Security Analysis».

[5] C. Luca, D. Ciullo και M. Meo, «Modeling Sleep Mode Gains in Energy-Aware Networks».

[6] S. L. Alexandru, S. Halunga, A. Vulpe, G. Suciu, O. Fratu και E. C. Popovici, «A Comparison between Several Software Defined Networking Controllers,» 2015.

[7] L. Bob, B. Heller και N. McKeown, «A Network in a Laptop: Rapid Prototyping for Software-Defined Networks».

[8] «Sdn series part eight: comparison of open-source sdn controllers,» 2015. [Ηλεκτρονικό]. Available: http://thenewstack.io/sdn-series-part-eight-comparison-of-open-source-sdn-controllers/.

[9] S. Dimitri, S. Sharma, D. Colle, M. Pickavet και P. Demeester, «Sotware Defined Networking: Meeting Carrier Grade Requirement».

[10] M. Jan, R. Varga, A. Tkacik και K. Gray, «OpenDaylight: Towards a Model-Driven SDN Controller Architecture».

[11] A. Shalimov, D. Zuikov, D. Zimarina, V. Pashkov και R. Smeliansky, «Advanced Study of SDN/OpenFlow controllers».

[12] Ashton, Metzler & Associates, «Ten Things to Look for in an SDN controller».

[13] R. Mijumbi, J. Serrat, J. R. Loyola, N. Buten, F. de Turck και S. Latre, «Dynamic Resource Management in SDN-based Virtualized Networks».

[14] R. P. Team, «RYU SDN Framework Using OpenFlow 1.3».

[15] V. Yazici, «On the OpenFlow Controllers,» [Ηλεκτρονικό]. Available: http://vlkan.com/blog/post/2013/07/31/openflow-controllers/.

[16] «Software-defined networking,» [Ηλεκτρονικό]. Available: wikipedia.

[17] N. Feamster, J. Rexford και E. Zegura, «The road to SDN: An Intellectual History of Programmable Networks».

[18] «ONOS Wiki,» [Ηλεκτρονικό]. Available: https://wiki.onosproject.org/display/ONOS/Wiki+Home.

[19] Y. Xiaowei, «Datacenter Network: Topology and Routing».

[20] P. Charalampou, A. Zafeiropoulos, C. Vassilakis, C. Tziouvaras, V. Giannikopoulou και N. Laoutaris, «Empirical evaluation of energy saving margins in backbone networks».