Network Optimizer ODL-App

Μπέλμπας, Θεόφιλος
1115201400119

Μανωλάς, Σταμάτιος
1115201400094

Χρονόπουλος, Ευστράτιος
1115201600203

September 16, 2019

Contents

1 Εισαγωγή

Καταλήξαμε στην επιλογή του $1^{ου}$ Project καθώς μας ενδιέφερε ιδιαίτερα η ιδέα της αφαίρεσης του *Control Plane* από τα *low-level* μηχανήματα και η διαχείριση τους από ένα κεντρικό *Controller*. Έτσι δίνεται η δυνατότητα για καλύτερο προγραμματισμό του δικτύου με δυναμικό τρόπο. Αυτός είναι και ο στόχος τις εφαρμογής μας.

1.1 Ανάθεση Εργασιών

- Υλοποίηση Network Optimizer: Μπέλμπας Θεόφιλος
- Υλοποίηση Πειραμάτων: Μανωλάς Σταμάτιος, Χρονόπουλος Ευστράτιος
- Συγγραφή Report: Μανωλάς Σταμάτιος, Χρονόπουλος Ευστράτιος

2 Εγκατάσταση

2.1 Python

Η εφαρμογή υλοποιήθηκε σε *Python2.7* και χρησιμοποιήθηκαν ορισμένα επιπλέον πακέτα. Για αυτό θα προτείναμε την δημιουργία ενός conda enviroment με τις εξής εντολές:

```
conda create -n noapp python=2.7
conda activate noapp
conda install networkx=2.2
conda install requests
```

2.2 Java

Είναι απαραίτητη η εγκατάσταση Java-8 για την ορθή λειτουργία του OpenDayLight. Επίσης χρειάζεται να δοθεί τιμή στην $JAVA\_HOME$ environmental variable, που να δείχνει το σημείο εγκατάστασής της JAVA.

2.3 OpenDayLight

Χρησιμοποιήσαμε τις οδηγίες του PDF που μας δόθηκε για να εγκαταστήσουμε την έκδοση BORON (5.4) του OpenDayLight που έχει ενσωματωμένο τον ApacheKaraf. Βέβαια τρέχοντας το σε UBUNTU 18.04 αντιμετωπίσαμε ένα error το οποίο μέτα από αρκετό GoogleSearch ανακαλύψαμε οτι χρειαζόταν αυτή την εντολή πριν την εκτέλεση του Karaf:

```
export TERM=xterm-color
```

2.4 Mininet

Για το mininet ακολουθήσαμε τον $1^ο$ τρόπο εγκατάστασης που προτείνετε στο PDF, δηλαδή download από το GitHub και run το installation script. Αλλά είχαμε πρόβλημα με την αποστολή των flows από τον ODL-server στα switches του mininet. Για να λυθεί το πρόβλημα έπρεπε να το εγκαταστήσουμε σε VM.

3 Περιγραφή του Network Optimizer

Η ιδέα για την εφαρμογή ξεκίνησε με την παρατήρηση των flows στα switches μετά την αρχικοποίηση του mininet. Τα πακέτα που φτάνουν σε μια πόρτα κάθε switch προωθούνται σε όλες τις υπόλοιπες με την βοήθεια των default flows. Αυτή η πρακτική κάνει flood το δίκτυο με αποτέλεσμα να μειώνει το bandwidth (δείτε Seciton4.1).

3.1 Συμβολισμοί

Πριν προχωρήσουμε θα θέλαμε να περιγράψουμε κάποιους συμβολισμούς που θα χρησιμοποιηθούν αργότερα. Ορίζουμε σαν $G(V, E, W)$ τον γράφο που αντιπροσωπεύει ένα topology, όπου οι κόμβοι $v \in V$ είναι τα Switches και οι Hosts. Για κάθε σύνδεση μεταξύ δύο κόμβων $u, v \in V$ στο topology υπάρχει μια ακμή $e_{u,v} \in E$. Η συνάρτηση W μας δίνει το βάρος μεταξύ δύο οποιονδήποτε κόμβων $u, v \in V$ μόνο αν υπάρχει ακμή $e_{u,v} \in E$ που τους συνδέει.

Σαν μονοπάτι $path(u, v)$ ορίζουμε μια ακολουθία κόμβων 2^{ou} βαθμού με αρχή τον u και τέλος τον v όπου κάθε κόμβος συνδέεται με τον επόμενο του με μια ακμή. Τέλος σαν κόστος του μονοπατιού $path(u, v)$ ορίζουμε το άθροισμα των βαρών των ακμών που χρησιμοποιούνται στο μονοπάτι.

3.2 Υλοποίηση

Ο *NetworkOptimizer* αλλάζει τα flows στα Switches ώστε τα πακέτα, ανάλογα με τον παραλήπτη, να χρησιμοποιούν το μονοπάτι με το μικρότερο κόστος.

3.2.1 Topology Recognizer

Η εφαρμογή μας αναγνωρίζει αυτόματα την τοπολογία που έχει αρχικοποιηθεί στο mininet κάνοντας parse το json που επιστέφει το HttpRequest στο endpoit:'/operational/network-topology:network-topology' του ODL-server.

3.2.2 Simple Optimization

Μια λειτουργία της εφαρμογής είναι η δημιουργία flows τα οποία εξασφαλίζουν ότι τα πακέτα που μεταφέρονται στο δίκτυο χρησιμοποιούν τα μονοπάτια με το μικρότερο κόστος. Αυτό επιτυγχάνεται φτιάχνοντάς όλα τα πιθανά ζευγάρια μεταξύ host και βρίσκοντας τα Dijkstra μονοπάτια που τα συνδέουν. Για κάθε μονοπάτι προσθέτουμε ένα flow σε κάθε Switch που περιέχεται σε αυτό, ώστε να

προωθεί σωστά τα πακέτα για τους δυο Hosts που βρίσκονται στο συγκεκριμένο μονοπάτι. Εδώ σαν συνάρτησή βάρους χρησιμοποιούμε την

$$W(u,v) = 1 \quad if \quad \exists \quad e_{u,v} \in E$$

3.2.3 Daemon Optimization

Η ιδέα εδώ είναι η ίδια με την παραπάνω, φτιάχνουμε flows που εκμεταλλεύονται τα Dijkstra μονοπάτια. Το μόνο που αλλάζει είναι η συνάρτηση κόστους για τον G. Έχοντάς πρώτα εκτελέσει ένα *SimpleOptimization* ξεκινάμε έναν Daemon που μετράει πόσα πακέτα προωθεί κάθε ένα από τα flows που προσθέσαμε, σε διάστημα τ. Έτσι η νέα συνάρτηση κόστους για τον γράφο είναι

$$W(u,v) = F(f_{packet\_count}) \quad where$$
$$F(f_{packet\_count}) = f^{t+\tau}_{packet\_count} - f^{t-\tau}_{packet\_count}$$

όπου f είναι το flow που αντιστοιχεί στο e που συνδέει τα u, v. Η αντιστοιχία f και e είναι 1 προς 1 καθώς το flow που προωθεί πακέτα ορίζεται με την βοήθεια του κόμβου που έχει το πακέτο (u), και του κόμβου στο οποίο πρέπει να το προωθήσει (v), όπως ακριβώς και η ακμή.

Με βάση τα νέα κόστη υπολογίζουμε ξανά τα Dijkstra μονοπάτια και κάνουμε update τα flows ώστε να τα απεικονίζουν. Έτσι προσπαθούμε να κρατάμε συνέχεια το δίκτυο Optimized ακόμα και σε περιόδους που αυξάνεται ο αριθμός πακέτων μεταξύ μονοπατιών.

4 Πειράματα

Πριν την εκτέλεση οποιουδήποτε πειράματος πρέπει να συμπληρώσετε τις παραμέτρους που θα χρησιμοποιεί η εφαρμογή μας. Κατευθυνθείτε στο αρχείο *src/params.py* και συμπληρώστε τα εξής:

```
server_ip = '<your_odl_server_ip>'
server_port = '<your_odl_server_port>'
monitor_interval = <daemon_monitor_interval>  # T in milliseconds
```

4.1 Simple Optimization

4.1.1 Περιγραφή

Στο παρακάτω πείραμα αποδεικνύουμε ότι τα αρχικά flows των Switches επηρεάζουν το bandwidth κάνοντας flood το δίκτυο. Ενώ με την χρήση του Optimizer τα πακέτα ακολουθούν μονο το συντομότερο μονοπάτι και το bandwidth αυξάνεται. Για την εκτέλεση αυτού του πειράματος ορίζουμε ένα topology (Figure 1) και χρησιμοποιούμε τον host1 σαν server και τον host3 σαν client, καλώντας τις αντίστοιχες εντολές 'iperf', και μετράμε το bandwidth και τα bytes που μεταφέρθηκαν.

Figure 1: Simple Optimization Topology

4.1.2 Εκτέλεση

1. Έχοντας ανοιχτό τον Karaf πηγαίνετε στο φάκελο *tests/* και επιλέγετε το αρχείο *optimizer_test.py*. Αν δουλεύετε με VM θα πρέπει να το μεταφέρετε εκεί. Τρέχετε το αρχείο με την εξής εντολή:

   ```
   sudo python optimizer_test.py <server_ip> <server_port>
   ```

2. Σας ζητάει να τρέξετε τον *SimpleOptimizer*, οπότε από τον αρχικό φάκελο του Project μας εκτελέστε :

   ```
   sudo python –m src.simple_optimizer simple_optimizer.py
   ```

3. Αφού τρέξει η εφαρμογή πατήστε *enter* πίσω στο *optimizer_test.py* και βλέπετε το εξής output:

```
STARTING TEST: 'Network Optimizer'
+ Run iperf without optimizing the network...
|-Run iperf on h1 (as server)
|-Run iperf on h3 (client) to h1 (server)
|-OUTPUT:
------------------------------------------------------------
[  3] local 10.0.0.3 port 36502 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.0 sec  90.2 MBytes   379 Mbits/sec
[  3]  2.0- 4.0 sec  86.5 MBytes   363 Mbits/sec
[  3]  4.0- 6.0 sec  86.5 MBytes   363 Mbits/sec
[  3]  0.0- 6.0 sec   263 MBytes   367 Mbits/sec

+ Run the optimizer_test.py and press enter...
|-Run iperf again on h3 (client) to h1 (server)
|-OUTPUT:
------------------------------------------------------------
[  3] local 10.0.0.3 port 36504 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0- 2.0 sec  5.10 GBytes  21.9 Gbits/sec
[  3]  2.0- 4.0 sec  5.05 GBytes  21.7 Gbits/sec
[  3]  0.0- 6.0 sec  15.3 GBytes  21.9 Gbits/sec

READING SERVER'S OUTPUT
[  4] local 10.0.0.1 port 5001 connected with 10.0.0.3 port 36502
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0- 6.1 sec   263 MBytes   362 Mbits/sec
[  5] local 10.0.0.1 port 5001 connected with 10.0.0.3 port 36504
[  5]  0.0- 6.0 sec  15.3 GBytes  21.8 Gbits/sec
```

4. Για τον καθαρισμό των νέων flows που προστέθηκαν από τον Optimizer πατήστε *enter* στο *src/simple_optimizer.py*.

4.1.3 Συμπέρασμα

Είναι πολύ εύκολο να δούμε (από το output του server στο h1) ότι **χωρίς τον Optimizer** μεταφέρθηκαν μόνο 264 mb ενώ **με τον Optimizer** μεταφέρθηκαν 15.2 gb στην ίδια χρονική περίοδο (6 sec). Ο λόγος είναι ότι τα πακέτα παραμένουν στο δίκτυο μέχρι να λήξει το ttl (time to live) τους, επιβαρύνοντάς το με συνεχείς εκπομπές από τα Switches.

4.2 Daemon Optimization

4.2.1 Περιγραφή

Όμως ένα απλό Optimization δεν αρκεί. Καθώς αυξάνεται η κίνηση στο δίκτυο τα συντομότερα μονοπάτια που επιλέχθηκαν στην αρχή μπορεί πλέον να μην είναι τόσο σύντομα. Σε αυτό το πείραμα συγκρίνουμε τον *SimpleOptimizer* με τον *DaemonOptimizer* για να αποδείξουμε τον παραπάνω ισχυρισμό.

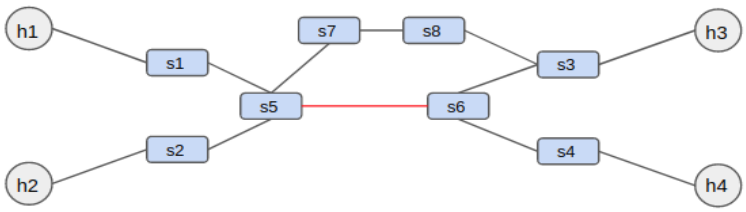

Figure 2: Daemon Optimization Topology

Από το Figure2 βλέπουμε ότι το συντομότερο μονοπάτι που συνδέει κάθε ζευγάρι host περνάει από την ακμή μεταξύ των switch5 και switch6. Ξεκινώντας όμως να μεταφέρουμε πακέτα ταυτόχρονα από τον host3 στον host1 και από τον host4 στον host2 σίγουρα η παραπάνω ακμή θα έχει πολύ κίνηση με αποτέλεσμα να χάνουμε bandwidth. Ο *Daemon Optimizer* θα το αναγνωρίσει και θα κάνει re-route κάποια πακέτα από μονοπάτια με λιγότερο κόστος, όπως το switch5-switch7-switch8-switch3. Όπως και πριν έτσι και εδώ η δημιουργία κίνησης στο δίκτυο γίνεται με τη εντολή 'iperf'.

4.2.2 Εκτέλεση

1. Έχοντας ανοιχτό τον Karaf πηγαίνετε στο φάκελο *tests/* και επιλέγετε το αρχείο *daemon_test.py*. Αν δουλεύετε με VM θα πρέπει να το μεταφέρετε εκεί. Τρέχετε το αρχείο με την εξής εντολή:

   ```
   sudo python daemon_test.py <server_ip> <server_port>
   ```

2. Αρχικά σας ζητάει να τρέξετε την εφαρμογή *Simple Optimizer*. Από τον αρχικό φάκελο του Project μας εκτελέστε:

   ```
   python -m src.simple_optimizer simple_optimizer.py
   ```

3. Αφού τρέξει η εφαρμογή πατήστε *enter* πίσω στο *daemon_test.py* και περιμένετε (30sec) για την ολοκλήρωση των μετρήσεων.

4. Μετά σας ζητάει να τρέξετε την εφαρμογή *Daemon Optimizer*. Πριν γίνει αυτό για τον καθαρισμό των νέων flows που προστέθηκαν από τον *Simple Optimizer* πατήστε πρώτα *enter* στο *src/simple_optimizer.py* και μετά εκτελέστε την εξής εντολή:

   ```
   python -m src.daemon_optimizer daemon_optimizer.py
   ```

5. Αφού τρέξει η εφαρμογή πατήστε *enter* πίσω στο *daemon_test.py* και μετά από 30sec θα εμφανιστεί αυτό το output:

```
STARTING TEST: 'Simple Optimizer vs Daemon Optimizer'
|-Run iperf on h1 and h2 (as servers)
+ Run the simple_optimizer.py and press enter...

|-Run iperf on h3 (client) to h1 (server)
|-Run iperf on h4 (client) to h2 (server)
+ wait iperf to complete...(~30sec)
+ Run the daemon_optimizer.py and press enter...

|-Run iperf on h3 (client) to h1 (server)
|-Run iperf on h4 (client) to h2 (server)
+ wait iperf to complete...(~30sec)
|-Kill server...
|-Kill server...
*

READING SERVER'S OUTPUT
FILE:  /tmp/h1.out
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.0.0.1 port 5001 connected with 10.0.0.3 port 38630
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-30.0 sec  34.8 GBytes  9.95 Gbits/sec
[  5] local 10.0.0.1 port 5001 connected with 10.0.0.3 port 38634
[  5]  0.0-30.0 sec  43.4 GBytes  12.4 Gbits/sec


FILE:  /tmp/h2.out
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  4] local 10.0.0.2 port 5001 connected with 10.0.0.4 port 59776
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-30.0 sec  34.8 GBytes  9.95 Gbits/sec
[  5] local 10.0.0.2 port 5001 connected with 10.0.0.4 port 59780
[  5]  0.0-30.0 sec  46.7 GBytes  13.4 Gbits/sec
```

(Κάποια στατιστικά διαγράφηκαν για εξοικονόμηση χώρου).

6. Όταν τελειώσουν και οι δεύτερες μετρήσεις πατήστε CTR+Z στο terminal που τρέχει ο *src/daemon_optimizer.py* για να καθαρίσει κι αυτός τα flows που τοποθέτησε στον ODL-server.

4.2.3 Συμπέρασμα

Παρατηρώντας το output των εντολών 'iperf' για τους server h1 και h2, βλέπουμε ότι στις δεύτερες μετρήσεις (με την χρήση του ***Daemon Optimizer***) έχουν μεταφερθεί περίπου 10 gb παραπάνω και έχουμε 25% μεγαλύτερο bandwidth και στους δύο server.