



ΑΡΙΣΤΟΤΕΛΕΙΟ
ΠΑΝΕΠΙΣΤΗΜΙΟ
ΘΕΣΣΑΛΟΝΙΚΗΣ

ΤΜΗΜΑ ΗΜΜΥ. ΤΟΜΕΑΣ ΗΛΕΚΤΡΟΝΙΚΗΣ

ΔΙΚΤΥΑ ΥΠΟΛΟΓΙΣΤΩΝ ΙΙ

Αυτοματοποιημένη Ανάπτυξη Τοπολογίας και Ανάλυση Πρωτοκόλλων στο Mininet

Συντάκτης:

Χρήστος Χουτουρίδης [8997]

cchoutou@ece.auth.gr

Διδάσκοντες:

Αναστάσιος Ντελόπουλος

antelopo@ece.auth.gr

Εμανουήλ Τσαρδουλιάς

etsardou@ece.auth.gr

27 Σεπτεμβρίου 2026

1. ΕΙΣΑΓΩΓΗ

Η παρούσα εργασία έχει ως αντικείμενο την **αυτοματοποιημένη ανάπτυξη δικτυακών τοπολογιών στο περιβάλλον Mininet**, με βάση μια δηλωτική περιγραφή σε αρχείο JSON. Στο πλαίσιο αυτό υλοποιείται εφαρμογή σε Python, η οποία διαβάζει περιγραφές της τοπολογίας σε μια πρότυπη **Domain-Specific Language – DSL**, και δημιουργεί δυναμικά τους αντίστοιχους hosts, μεταγωγείς, συνδέσμους και δρομολογητές και εφαρμόζει τις απαιτούμενες ρυθμίσεις δικτύου. Το Mininet χρησιμοποιείται ως περιβάλλον εξομίωσης, ώστε η τοπολογία να μπορεί να κατασκευάζεται και να εξετάζεται με χρήση των ίδιων εργαλείων και πρωτοκόλλων που χρησιμοποιούνται σε ένα πραγματικό σύστημα Linux.

Μετά την ανάπτυξη της τοπολογίας πραγματοποιούνται τρία πειράματα χαμηλού επιπέδου για τη μελέτη των πρωτοκόλλων **DHCP** και **ARP**, καθώς και της **δρομολόγησης μεταξύ διαφορετικών υποδικτύων**. Μέσω καταγραφών με το `tcpdump` εξετάζεται η διαδικασία D.O.R.A. του DHCP, η αντιστοίχιση διευθύνσεων IP σε διευθύνσεις MAC μέσω του ARP και η μεταβολή των Ethernet διευθύνσεων και του TTL κατά τη διέλευση ενός IP πακέτου από τον δρομολογητή. Στόχος των πειραμάτων είναι η σύνδεση της θεωρητικής λειτουργίας των πρωτοκόλλων με την πραγματική μορφή των πακέτων που ανταλλάσσονται στα διαφορετικά επίπεδα του δικτύου.

1.1. Παραδοτέα

Τα παραδοτέα της εργασίας αποτελούνται από:

- Την παρούσα τεχνική αναφορά.
- Το αρχείο [exercise/topology.json](#), το οποίο περιγράφει δηλωτικά την τοπολογία δικτύου που χρησιμοποιείται στην εργασία.
- Τον κατάλογο `source/`, ο οποίος περιέχει το αρχείο [build_network.py](#), καθώς και τα υπόλοιπα αρχεία Python που υλοποιούν την εφαρμογή.
- Τον κατάλογο [exercise/results/](#), ο οποίος περιέχει τις μετρήσεις και τα παραγόμενα αρχεία των πειραμάτων της εργασίας.
- Τους συνδέσμους προς τα αποθετήρια στο [GitHub](#) και στον [προσωπικό μου server](#), τα οποία περιέχουν τον πηγαίο κώδικα, τον κώδικα παραγωγής της αναφοράς και τα αποτελέσματα των πειραμάτων.

2. ΥΛΟΠΟΙΗΣΗ

2.1. Δηλωτική γλώσσα περιγραφής τοπολογίας

Για την περιγραφή των δικτυακών τοπολογιών σχεδιάστηκε μία μικρή **δηλωτική γλώσσα ειδικού σκοπού** (Domain-Specific Language – DSL), η οποία χρησιμοποιεί το JSON ως μορφή αναπαράστασης. Στόχος της γλώσσας είναι να περιγράφει την *επιθυμητή κατάσταση του δικτύου* και όχι τη σειρά των εντολών που απαιτούνται για την κατασκευή του στο Mininet. Με αυτόν τον τρόπο η περιγραφή της τοπολογίας παραμένει ανεξάρτητη από λεπτομέρειες της υλοποίησης, όπως τα ονόματα των εικονικών διεπαφών ή η ακριβής ακολουθία κλήσεων προς το Mininet API. Η αναλυτική σχεδίαση και η γραμματική της γλώσσας τεκμηριώνονται επίσης στο αρχείο [Mininet_DSL_Design.md](#) του αποθετηρίου.

2.1.1. Σχεδιαστική προσέγγιση

Ο σχεδιασμός της γλώσσας πραγματοποιήθηκε **από πάνω προς τα κάτω**, ξεκινώντας από τις έννοιες που είναι φυσικό να χρησιμοποιούνται για την περιγραφή ενός δικτύου, όπως οι *κόμβοι*, οι *διεπαφές*, οι *σύνδεσμοι* και οι *υπηρεσίες*. Παράλληλα, κάθε αφηρημένη έννοια επιλέχθηκε έχοντας υπόψη τις διαθέσιμες πρωτογενείς λειτουργίες του Mininet και του δικτυακού υποσυστήματος του Linux. Η προσέγγιση αυτή επιτρέπει στη γλώσσα να παραμένει υψηλού επιπέδου, χωρίς να εισάγει κατασκευές που δεν μπορούν να μεταφραστούν με σαφή και συστηματικό τρόπο στο υποκείμενο περιβάλλον εκτέλεσης.

Οι βασικές λειτουργίες που απαιτούνται από την εργασία μπορούν να αναχθούν σε ένα σχετικά μικρό σύνολο *πρωτογενών κατασκευών* του Mininet και του Linux. Η δημιουργία hosts και switches, η σύνδεσή τους μέσω links και η εκτέλεση εντολών μέσα στους αντίστοιχους network namespaces επαρκούν για την κατασκευή της απαιτούμενης τοπολογίας. Ένας δρομολογητής μπορεί αντίστοιχα να μοντελοποιηθεί ως host με περισσότερες από μία διεπαφές, κατάλληλη Layer-3 διευθυνσιοδότηση και ενεργοποιημένη προώθηση IP. Ο DHCP server μπορεί να θεωρηθεί ως υπηρεσία που εκτελείται σε έναν κόμβο και συνδέεται με συγκεκριμένη διεπαφή.

Πριν οριστικοποιηθούν οι οντότητες της DSL, εξετάστηκαν οι βασικές λειτουργίες που παρέχει το Mininet και το δικτυακό υποσύστημα του Linux για την κατασκευή και παραμετροποίηση μιας τοπολογίας. Οι κύριες λειτουργίες που χρησιμοποιήθηκαν ως βάση είναι οι `addHost()`, `addSwitch()` και `addLink()`, καθώς και η δυνατότητα εκτέλεσης εντολών μέσα σε κάθε network namespace μέσω του `node.cmd()`. Στις λειτουργίες αυτές προστίθενται η παραμετροποίηση διεπαφών και δρομολόγησης μέσω εργαλείων του Linux και η εκκίνηση διεργασιών, όπως ο `dnsmasq`, μέσα στον αντίστοιχο κόμβο. Η ανάλυση αυτή χρησιμοποιήθηκε για να **σταθεροποιηθεί το σύνολο των αφαιρέσεων της DSL**, έτσι ώστε κάθε υψηλού επιπέδου κατασκευή να έχει σαφή δρόμο μετάφρασης προς τις διαθέσιμες λειτουργίες του backend.

Με βάση τις παρατηρήσεις αυτές επιλέχθηκε ένα σύνολο οντοτήτων ώστε η DSL να **καλύπτει τον χώρο των κατασκευών που απαιτούνται από την εργασία**. Αντίστοιχα, το σύνολο των primitives του Mininet και του Linux που χρησιμοποιείται είναι επαρκές ώστε κάθε κατασκευή της DSL να μπορεί να μεταφραστεί σε συγκεκριμένες λειτουργίες του backend. Με αυτή την έννοια, τόσο η γλώσσα ως προς τον χώρο του προβλήματος όσο και τα χρησιμοποιούμενα primitives ως προς τον χώρο της υλοποίησης λειτουργούν «επί». Δηλαδή, δεν παραμένει απαίτηση της εργασίας χωρίς αναπαράσταση στη DSL ούτε κατασκευή της DSL χωρίς αντίστοιχη υλοποίηση στο Mininet και στο Linux. Οι βασικές οντότητες της γλώσσας είναι οι *κόμβοι*, οι *διεπαφές*, οι *σύνδεσμοι*, οι *ρυθμίσεις κόμβων* και οι *υπηρεσίες*.

2.1.2. Βασική γραμματική

Η τοπολογία περιγράφεται από τρεις κύριες κατηγορίες αντικειμένων: **nodes**, **links** και **services**. Η βασική δομή της γλώσσας μπορεί να συνοψιστεί ως εξής:

```
Network
|-- Nodes
|   |-- name
|   |-- type
|   |-- interfaces
|   `-- settings
|
|-- Links
|   |-- name
|   |-- endpoints
|   |   |-- node
|   |   `-- interface
|   `-- settings
|
`-- Services
    |-- type
    |-- node
    |-- interface
    `-- settings
```

Κάθε κατηγορία έχει **διαφορετική σημασιολογική ευθύνη**. Ένας κόμβος(node) αναπαριστά έναν συμμετέχοντα του δικτύου, μία διεπαφή(interface) ένα σημείο σύνδεσης που ανήκει σε έναν κόμβο και ένας σύνδεσμος(link) συνδέει δύο τέτοια σημεία. Οι ρυθμίσεις(settings) κόμβου περιγράφουν ιδιότητες του ίδιου του

δικτυακού namespace, ενώ οι υπηρεσίες (services) αναπαριστούν ανεξάρτητες διεργασίες που εκτελούνται μέσα σε αυτό.

2.1.3. Κόμβοι και διεπαφές

Οι κόμβοι δηλώνονται με ένα μοναδικό λογικό όνομα και έναν τύπο. Οι τύποι που υποστηρίζονται στην παρούσα υλοποίηση είναι `host`, `switch` και `router`. Ο τύπος εκφράζει τη *σημασιολογία του κόμβου* και όχι απαραίτητα το συγκεκριμένο αντικείμενο του Mininet που θα χρησιμοποιηθεί για την υλοποίησή του.

Οι διεπαφές **ανήκουν στους κόμβους** και προσδιορίζονται από λογικά ονόματα, τα οποία δεν απαιτείται να ταυτίζονται με τα ονόματα που θα δημιουργηθούν τελικά στο Linux. Για παράδειγμα, μία διεπαφή με λογικό όνομα `lan` μπορεί κατά την υλοποίηση να αντιστοιχιστεί στη φυσική διεπαφή `rx0-eth0`. Η επιλογή αυτή επιτρέπει στο αρχείο JSON να χρησιμοποιεί ονόματα που εκφράζουν τον ρόλο μιας διεπαφής και όχι λεπτομέρειες της διαδικασίας δημιουργίας της.

Μία διεπαφή μπορεί να χρησιμοποιεί στατική διευθυνσιοδότηση, DHCP ή να μην διαθέτει καθόλου Layer-3 παραμετροποίηση. Στην περίπτωση στατικής διευθυνσιοδότησης δηλώνεται η διεύθυνση μαζί με το prefix και, προαιρετικά, μία προεπιλεγμένη πύλη. Επιπλέον μπορούν να δηλωθούν ιδιότητες όπως η διεύθυνση MAC και το MTU.

Ιδιαίτερη σημασία έχει η διάκριση μεταξύ της **απουσίας** του πεδίου `interfaces` και της δήλωσης ενός κενού αντικειμένου `interfaces`. Όταν το πεδίο παραλείπεται, ο κόμβος θεωρείται ότι διαθέτει *δυναμικό σύνολο διεπαφών* και νέες διεπαφές μπορούν να δημιουργηθούν κατά την επίλυση των συνδέσμων. Αντίθετα, όταν το πεδίο υπάρχει, ακόμη και αν είναι κενό, το σύνολο των διαθέσιμων διεπαφών θεωρείται *σταθερό*.

2.1.4. Σύνδεσμοι και αυτόματη επιλογή διεπαφών

Κάθε σύνδεσμος αποτελεί αυτοτελή οντότητα και συνδέει ακριβώς δύο endpoints. Κάθε endpoint προσδιορίζει τον κόμβο στον οποίο ανήκει και τη λογική διεπαφή μέσω της οποίας πραγματοποιείται η σύνδεση. Η διεπαφή μπορεί να δηλωθεί ρητά ή να χρησιμοποιηθεί ο ειδικός επιλογέας `auto`.

Όταν χρησιμοποιείται το `auto` σε κόμβο με σταθερό σύνολο διεπαφών, επιλέγεται μία από τις δηλωμένες διεπαφές που δεν έχει ήδη δεσμευτεί από άλλη σύνδεση. Αν δεν υπάρχει ελεύθερη διεπαφή, η τοπολογία θεωρείται μη έγκυρη και **δεν δημιουργείται επιπλέον θύρα σιωπηρά**. Όταν αντίθετα το σύνολο διεπαφών έχει παραλειφθεί, το `auto` επιτρέπει τη δυναμική δημιουργία νέας λογικής διεπαφής για τις ανάγκες του συνδέσμου.

Οι σύνδεσμοι μπορούν επίσης να περιλαμβάνουν παραμέτρους προσομοίωσης χαρακτηριστικών του φυσικού μέσου. Η τρέχουσα γλώσσα υποστηρίζει περιορισμό εύρους ζώνης, καθυστέρηση, απώλεια πακέτων και jitter μέσω των πεδίων `bandwidth_mbps`, `delay_ms`, `loss_percent` και `jitter_ms`.

2.1.5. Ρυθμίσεις κόμβων και υπηρεσίες

Οι ρυθμίσεις που αφορούν τη λειτουργία του ίδιου του κόμβου τοποθετούνται στο αντικείμενο `settings` του αντίστοιχου node. Για τον τύπο `router`, για παράδειγμα, υποστηρίζονται η ενεργοποίηση του IP forwarding και η απενεργοποίηση του reverse path filtering. Οι ιδιότητες αυτές αντιμετωπίζονται ως *χαρακτηριστικά του network namespace* και όχι ως ανεξάρτητες υπηρεσίες.

Οι υπηρεσίες μοντελοποιούνται ξεχωριστά επειδή αντιστοιχούν σε **διεργασίες που εκτελούνται πάνω σε έναν κόμβο**. Στην παρούσα εργασία υποστηρίζεται η υπηρεσία `dhcp-server`, η οποία συνδέεται με έναν κόμβο και μία από τις διεπαφές του. Οι παράμετροι της υπηρεσίας περιλαμβάνουν το εύρος διευθύνσεων που μπορούν να εκχωρηθούν και την προεπιλεγμένη πύλη που διαφημίζεται στους DHCP clients.

Η θέση εκτέλεσης μιας υπηρεσίας και η διεπαφή στην οποία συνδέεται αποτελούν μέρος της ίδιας της ταυτότητάς της και για τον λόγο αυτό δηλώνονται έξω από το αντικείμενο `settings`. Με αυτόν τον διαχωρισμό, το `settings` περιγράφει αποκλειστικά τη συμπεριφορά της υπηρεσίας και όχι το σημείο στο οποίο αυτή τοποθετείται μέσα στην τοπολογία.

2.1.6. Αυστηρότητα της γραμματικής

Η γραμματική σχεδιάστηκε ώστε να είναι **αυστηρή** και να μην αγνοεί άγνωστα πεδία ή μη υποστηριζόμενες τιμές. Ένα τυπογραφικό λάθος σε ένα όνομα ιδιότητας δεν αντιμετωπίζεται επομένως ως απουσία της ιδιότητας, αλλά ως σφάλμα της περιγραφής. Η επιλογή αυτή περιορίζει την πιθανότητα μία συντακτικά αποδεκτή αλλά διαφορετική από την επιθυμητή τοπολογία να δημιουργηθεί χωρίς εμφανή προειδοποίηση.

Η γλώσσα διαχωρίζει επίσης τη **δομική εγκυρότητα** μιας περιγραφής από τη **σημασιολογική συνέπειά** της. Ένα αντικείμενο μπορεί να ακολουθεί σωστά τη γραμματική αλλά να περιγράφει, για παράδειγμα, έναν σύνδεσμο προς ανύπαρκτο κόμβο ή μία στατική πύλη εκτός του αντίστοιχου υποδικτύου. Η διάκριση αυτή αξιοποιείται στη συνέχεια από την αρχιτεκτονική του builder, όπου η ανάλυση της γραμματικής και ο σημασιολογικός έλεγχος πραγματοποιούνται σε **διαφορετικά στάδια**.

2.2. Υλοποίηση του Builder

Η υλοποίηση του builder ακολουθεί τον διαχωρισμό που εισάγει η DSL και οργανώνεται ως μία ακολουθία ανεξάρτητων σταδίων, καθένα από τα οποία μετασχηματίζει την πληροφορία σε μία περισσότερο συγκεκριμένη αναπαράσταση. Η δομή αυτή επιτρέπει τον σαφή διαχωρισμό ανάμεσα στην περιγραφή της τοπολογίας, στον έλεγχο της εγκυρότητάς της και στην τελική δημιουργία και εκτέλεση του δικτύου στο Mininet.

2.2.1. Δομή των αρχείων

Ο πηγαίος κώδικας του builder βρίσκεται στον κατάλογο `source/`, ενώ η περιγραφή και τα αποτελέσματα της συγκεκριμένης εργαστηριακής άσκησης διατηρούνται χωριστά στον κατάλογο `exercise/`. Η βασική δομή του αποθετηρίου είναι η ακόλουθη:

```
.
|-- source/
|   |-- build_network.py
|   |-- topology_loader.py
|   |-- topology_parser.py
|   |-- topology_validator.py
|   |-- topology_plan.py
|   `-- topology_renderer.py
|
|-- exercise/
|   |-- topology.json
|   |-- topology.generated.py
|   `-- results/
|
|-- Mininet_DSL_Design.md
`-- README.md
```

- Το `topology_loader.py` είναι υπεύθυνο αποκλειστικά για τη φόρτωση του αρχείου JSON από το σύστημα αρχείων και την παραγωγή της αντίστοιχης ακατέργαστης Python δομής.
- Το `topology_parser.py` περιέχει τη γραμματική, τα Enum και τα `dataclass` αντικείμενα που αποτελούν το τυποποιημένο σημασιολογικό μοντέλο της DSL.
- Το `topology_validator.py` ελέγχει τη σημασιολογική συνέπεια της τοπολογίας, χωρίς να πραγματοποιεί οποιαδήποτε μεταβολή στο δίκτυο.
- Το `topology_plan.py` επιλύει τις αφηρημένες επιλογές της DSL και παράγει ένα πλήρως συγκεκριμένο `ExecutionPlan`.
- Το `topology_renderer.py` μετατρέπει το σχέδιο αυτό σε εκτελέσιμο Python κώδικα για το Mininet, οποίος εξάγεται σε ξεχωριστό αρχείο και αποτελεί ένα ιδιότυπο *python-listing* του renderer.

- Τέλος, το `build_network.py` αποτελεί τον **coordinator** και το μοναδικό user-facing entry point, συνδέοντας τα επιμέρους στάδια και αναλαμβάνοντας την τελική εκτέλεση του παραγόμενου προγράμματος.

2.2.2. Χρήση

Η κανονική χρήση του builder απαιτεί μόνο το αρχείο JSON που περιγράφει την επιθυμητή τοπολογία. Για την τοπολογία της παρούσας εργασίας η πλήρης κατασκευή και εκτέλεση πραγματοποιείται με την εντολή:

```
sudo python3 source/build_network.py exercise/topology.json
```

Η εκτέλεση αυτή μεταγλωττίζει πρώτα την τοπολογία, δημιουργεί το αρχείο `exercise/topology.generated.py` και στη συνέχεια εκτελεί **το ίδιο ακριβώς παραγόμενο αρχείο**. Μετά την κατασκευή και παραμετροποίηση του δικτύου ανοίγει το Mininet CLI, από το οποίο εκτελούνται τα πειράματα.

Κατά την ανάπτυξη είναι επίσης δυνατό να παραχθεί μόνο το ενδιαμέσο Python πρόγραμμα, χωρίς να δημιουργηθεί κανένα network namespace ή άλλη κατάσταση στο σύστημα. Για παράδειγμα:

```
python3 source/build_network.py exercise/topology.json --emit [Optional file name]
```

Η μορφή αυτή δεν απαιτεί δικαιώματα root και είναι χρήσιμη για την επιθεώρηση του αποτελέσματος της μεταγλώττισης ή για τον εντοπισμό σφαλμάτων πριν από την πραγματική εκτέλεση. Το `--emit` μπορεί επιπλέον να δεχθεί συγκεκριμένο path για το παραγόμενο αρχείο.

Πριν από την εκτέλεση μπορεί επίσης να πραγματοποιηθεί έλεγχος του περιβάλλοντος με:

```
python3 source/build_network.py --env-check
```

Ο έλεγχος επιβεβαιώνει την παρουσία του Mininet, του Open vSwitch και των εργαλείων Linux που απαιτούνται από τον builder και τα πειράματα, όπως τα `ip`, `tc`, `sysctl`, `dnsmasq`, `dhclient`, `tcpdump` και `ping`. Ο ίδιος έλεγχος πραγματοποιείται αυτόματα πριν από την πραγματική εκτέλεση μιας παραγόμενης τοπολογίας.

2.2.3. Διαχωρισμός περιγραφής τοπολογίας και runtime κατάστασης

Βασική σχεδιαστική επιλογή είναι ο **διαχωρισμός της δηλωμένης τοπολογίας από την κατάσταση που δημιουργείται κατά την εκτέλεσή της**. Το αρχείο JSON περιγράφει την επιθυμητή διαμόρφωση και τις πηγές από τις οποίες θα προκύψει η παραμετροποίηση, χωρίς να επιχειρεί να αποθηκεύσει την παροδική κατάσταση ενός εκτελούμενου δικτύου.

Για παράδειγμα, η δήλωση:

```
"net0": {
  "addressing": "dhcp"
}
```

δεν καθορίζει ποια διεύθυνση IP θα έχει τελικά ο host. Δηλώνει ότι η διεύθυνση πρέπει να αποκτηθεί δυναμικά από μία υπηρεσία DHCP κατά την εκτέλεση της τοπολογίας. Αντίστοιχα, ένας στατικός σύνδεσμος στο JSON οδηγεί στη δημιουργία πραγματικών virtual Ethernet interfaces, χωρίς τα τελικά ονόματα αυτών των interfaces να χρειάζεται να εμφανίζονται στην αρχική περιγραφή.

Η ίδια διάκριση ισχύει για τις DHCP leases, τον ARP neighbor table, τους μετρητές πακέτων και τα αρχεία `pcap`, τα οποία αποτελούν *runtime state* και δεν ανήκουν στη δηλωτική γλώσσα. Για τον ίδιο λόγο ο builder δεν εκκινεί αυτόματα το `dhclient` στους hosts που έχουν δηλωθεί ως DHCP clients, καθώς η απόκτηση της διεύθυνσης αποτελεί μέρος της πειραματικής διαδικασίας της εργασίας.

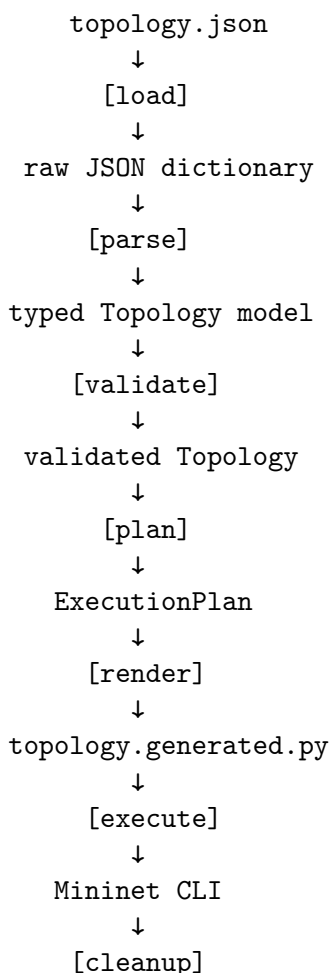
Η αρχή αυτή αποτυπώνεται συνοπτικά ως:

Declarative topology		Runtime state
static address	----->	configured IP address
static gateway	----->	routing-table entry
DHCP addressing	----->	dynamically acquired lease
link declaration	----->	veth / Mininet interfaces
service declaration	----->	running process
ARP entries		
packet captures		
counters		

2.2.4. Compiler-like αρχιτεκτονική

Η εσωτερική αρχιτεκτονική του builder σχεδιάστηκε με λογική παρόμοια με αυτή ενός **μεταγλωττιστή**. Αντί η εφαρμογή να διαβάζει ένα JSON και να καλεί άμεσα συναρτήσεις του Mininet, η περιγραφή περνά από διαδοχικά στάδια, καθένα από τα οποία έχει σαφώς περιορισμένη ευθύνη.

Η συνολική ροή είναι:



Ο διαχωρισμός αυτός επιτρέπει σε κάθε στάδιο να εξετάζεται ανεξάρτητα και μεταφέρει σταδιακά την περιγραφή από μία υψηλού επιπέδου δήλωση πρόθεσης σε μία πλήρως συγκεκριμένη ακολουθία ενεργειών Mininet και Linux. Παράλληλα, κάθε κατηγορία σφάλματος ανήκει στο στάδιο που διαθέτει την απαραίτητη πληροφορία για να την εντοπίσει.

Φόρτωση – topology_loader.py. Το πρώτο στάδιο αφορά αποκλειστικά την πρόσβαση στο αρχείο και την ανάλυση της σύνταξης JSON. Η δημόσια συνάρτηση `load_topology()` διαβάζει το αρχείο και επιστρέφει ένα `Dict[str, Any]`, χωρίς να γνωρίζει οτιδήποτε για τη σημασιολογία της DSL.

```
raw = load_topology("exercise/topology.json")
```

JSON file

```
|
`--> Dict[str, Any]
```

Σφάλματα όπως ανύπαρκτο αρχείο, μη έγκυρο JSON ή root αντικείμενο διαφορετικό από JSON object αναφέρονται ως `TopologyLoadError`. Στο σημείο αυτό δεν εξετάζεται ακόμη αν ένα `type` είναι έγκυρο ή αν ένας σύνδεσμος αναφέρεται σε υπαρκτό κόμβο.

Parsing – topology_parser.py. Το δεύτερο στάδιο μετατρέπει την ακατέργαστη δομή JSON στο τυποποιημένο σημασιολογικό μοντέλο της DSL. Η δημόσια συνάρτηση `parse_topology()` κατασκευάζει αντικείμενα όπως `Topology`, `Node`, `Interface`, `Link` και `Service`, χρησιμοποιώντας `Enum` για τα κλειστά σύνολα τιμών.

Για παράδειγμα, ένα τμήμα JSON όπως:

```
{
    "node": "s1",
    "interface": "auto"
}
```

δεν παραμένει ένα γενικό dictionary, αλλά μετατρέπεται εννοιολογικά σε:

```
LinkEndpoint(
    node="s1",
    interface=LinkInterfaceSelector.AUTO
)
```

Το στάδιο αυτό εφαρμόζει επίσης την *ανστηρή γραμματική* της γλώσσας. Άγνωστα πεδία ή μη υποστηριζόμενες τιμές προκαλούν `TopologyParseError`, αντί να αγνοούνται σιωπηρά.

Validation – topology_validator.py. Μετά το parsing η δομή είναι γραμματικά έγκυρη, αυτό όμως δεν συνεπάγεται ότι περιγράφει και συνεπές δίκτυο. Η δημόσια συνάρτηση `validate()` πραγματοποιεί τους ελέγχους που απαιτούν γνώση περισσότερων από ενός αντικειμένων της τοπολογίας.

Για παράδειγμα, το ακόλουθο JSON μπορεί να αναλυθεί συντακτικά, αλλά δεν περιγράφει έγκυρη στατική διεπαφή:

```
"net0": {
    "addressing": "static"
}
```

Το validation θα απορρίψει τη συγκεκριμένη περιγραφή επειδή απουσιάζει η στατική διεύθυνση. Αντίστοιχα ελέγχονται αναφορές σε ανύπαρκτους κόμβους, επαναχρησιμοποίηση interfaces, διευθύνσεις και gateways, διαθέσιμότητα ports για auto, καθώς και η συνέπεια των ρυθμίσεων της υπηρεσίας DHCP.

Σε αντίθεση με ένα απλό fail-fast μοντέλο, ο validator συλλέγει τα προβλήματα σε αντικείμενα `ValidationIssue` και τα επιστρέφει συγκεντρωτικά μέσω ενός `TopologyValidationError`. Με αυτόν τον τρόπο ο χρήστης μπορεί να διορθώσει περισσότερα από ένα προβλήματα σε έναν κύκλο εκτέλεσης.

Planning – topology_plan.py. Το planning είναι το σημείο στο οποίο η έγκυρη πλέον περιγραφή μετατρέπεται σε μία **πλήρως επιλυμένη ενδιάμεση αναπαράσταση**. Η δημόσια συνάρτηση `plan()` παράγει ένα `ExecutionPlan`, στο οποίο δεν παραμένουν αρηρημένοι selectors όπως `auto` ή άλλες αποφάσεις που πρέπει να ληφθούν από τον backend.

Για παράδειγμα, έστω ένας `switch(node)` με δύο ρητά δηλωμένες λογικές διεπαφές:

```
{
    "name": "s1",
    "type": "switch",
    "interfaces": {
        "client": {},
        "uplink": {}
    }
}
```

και δύο σύνδεσμοι, από τους οποίους ο πρώτος χρησιμοποιεί `auto`, ενώ ο δεύτερος ζητά ρητά τη διεπαφή `uplink`:

```
{
    "endpoints": [
        {"node": "h1", "interface": "net0"},
        {"node": "s1", "interface": "auto"}
    ]
},
{
    "endpoints": [
        {"node": "r0", "interface": "lan"},
        {"node": "s1", "interface": "uplink"}
    ]
}
```

Πριν από την επίλυση των `auto`, ο planner εντοπίζει όλες τις ρητές αναφορές σε interfaces και δεσμεύει τη `uplink`. Το διαθέσιμο σύνολο διεπαφών του `s1` για αυτόματη επιλογή περιέχει επομένως μόνο τη `client`.

```
reserved["s1"]          = {"uplink"}
free_interfaces["s1"] = ["client"]
```

Κατά το planning, το `auto` του πρώτου συνδέσμου επιλύεται συνεπώς στη λογική διεπαφή `client`. Στο συγκεκριμένο παράδειγμα οι λογικές διεπαφές αντιστοιχίζονται στη συνέχεια και στα συγκεκριμένα ονόματα διεπαφών που θα χρησιμοποιήσει το Mininet:

```
h1:net0    -> h1-eth0
s1:client  -> s1-eth0

r0:lan     -> r0-eth0
s1:uplink  -> s1-eth1
```

Με αυτόν τον τρόπο μία προηγούμενη αυτόματη επιλογή δεν μπορεί να καταλάβει interface που έχει ζητηθεί ρητά από μεταγενέστερο link. Το `auto` λειτουργεί επομένως ως *επιλογέας διαθέσιμης λογικής διεπαφής* και όχι ως όνομα διεπαφής.

Η συμπεριφορά είναι διαφορετική όταν το πεδίο `interfaces` έχει παραλειφθεί πλήρως από έναν κόμβο. Σε αυτή την περίπτωση ο planner δημιουργεί συνθετικά λογικά ονόματα, όπως `auto0` και `auto1`, τα οποία αντιστοιχίζονται στη συνέχεια σε φυσικά ονόματα όπως `s1-eth0` και `s1-eth1`.

Στο ίδιο στάδιο αποφασίζονται και backend-specific λεπτομέρειες που δεν πρέπει να εμφανίζονται στη DSL. Έτσι, η σημασιολογία ενός απλού Layer-2 switch μετατρέπεται από τον planner στην επιλογή `switch_fail_mode = "standalone"` για το Open vSwitch.

Ένα απλοποιημένο τμήμα του `ExecutionPlan` για το προηγούμενο παράδειγμα μπορεί επομένως να θεωρηθεί ότι έχει τη μορφή:

```
PlannedLinkEndpoint(
    node="s1",
    logical_interface="client",
    interface_name="s1-eth0"
)

PlannedLinkEndpoint(
    node="s1",
    logical_interface="uplink",
    interface_name="s1-eth1"
)

PlannedNode(
    name="s1",
    type=NodeType.SWITCH,
    switch_fail_mode="standalone"
)
```

Ο planner δεν δημιουργεί interfaces και δεν τα εισάγει στο Mininet. Η ευθύνη του τελειώνει όταν όλες οι αποφάσεις που απαιτούνται για την εκτέλεση έχουν αποτυπωθεί στο `ExecutionPlan`.

Rendering – topology_renderer.py. Το renderer δέχεται το πλήρως επιλυμένο `ExecutionPlan` και το μετατρέπει σε ένα ολοκληρωμένο Python πρόγραμμα μέσω της δημόσιας συνάρτησης `render_python()`. Σε αυτό το στάδιο **δεν πραγματοποιείται νέα σημασιολογική απόφαση**: ο renderer αποτυπώνει σε κώδικα τις επιλογές που έχουν ήδη γίνει από τον planner.

Για το προηγούμενο παράδειγμα μπορούν να παραχθούν εντολές της μορφής:

```
nodes['h1'] = net.addHost('h1', ip=None)
nodes['r0'] = net.addHost('r0', ip=None)
nodes['s1'] = net.addSwitch('s1', failMode='standalone')

net.addLink(
    nodes['h1'],
    nodes['s1'],
    intfName1='h1-eth0',
    intfName2='s1-eth0',
)

net.addLink(
    nodes['r0'],
    nodes['s1'],
    intfName1='r0-eth0',
    intfName2='s1-eth1',
)
```

Παρατηρείται ότι στο παραγόμενο πρόγραμμα δεν υπάρχει πλέον ο selector `auto`. Η επιλογή `s1:client` έχει

ήδη πραγματοποιηθεί από τον planner και ο renderer γνωρίζει μόνο το πλήρως επιλυμένο ζεύγος `s1:client` και `s1-eth0`.

Η χρήση του `ip=None` εμποδίζει το Mininet να εκχωρήσει τις δικές του προεπιλεγμένες διευθύνσεις στους hosts, ώστε η διευθυνσιοδότηση να παραμένει αποκλειστικά υπό τον έλεγχο της DSL. Ανάλογα με το σχέδιο, ο renderer παράγει επίσης εντολές για IP addresses, routes, sysctls, TCLink και εκκίνηση του dnsmasq.

Το παραγόμενο πρόγραμμα περιλαμβάνει ακόμη το πλήρες lifecycle του δικτύου:

```
net.build()
net.start()

# interface configuration
# routes
# sysctls
# services

CLI(net)

# finally:
# stop started services
# net.stop()
```

Το παραγόμενο Python ως ενδιαμέσση “assembly”. Το αρχείο `topology.generated.py` έχει σκόπιμα κεντρικό ρόλο στην αρχιτεκτονική και δεν αποτελεί απλώς ένα προσωρινό implementation detail. Μπορεί να θεωρηθεί ως η “assembly” της DSL: είναι χαμηλότερου επιπέδου από το JSON, πλήρως συγκεκριμένο, άμεσα εκτελέσιμο και ταυτόχρονα αρκετά αναγνώσιμο ώστε να μπορεί να επιθεωρηθεί από τον χρήστη.

Η αντιστοίχιση μπορεί να συνοψιστεί ως:

<code>topology.json:</code>	declarative source
<code>Topology:</code>	typed semantic representation
<code>ExecutionPlan:</code>	resolved intermediate representation
<code>topology.generated.py:</code>	executable "assembly"
<code>Mininet / Linux:</code>	runtime

Ο coordinator αποθηκεύει πάντοτε το παραγόμενο Python πριν από την εκτέλεση και στη συνέχεια εκτελεί ακριβώς αυτό το αρχείο. Δεν υπάρχει δεύτερος, ανεξάρτητος executor που να μεταφράζει απευθείας το `ExecutionPlan` σε Mininet operations, αποφεύγοντας έτσι τον κίνδυνο διαφορετικής συμπεριφοράς μεταξύ του inspectable artifact και του πραγματικού runtime.

Εκτέλεση και συντονισμός – `build_network.py`. Το `build_network.py` συνθέτει τα προηγούμενα στάδια μέσω της συνάρτησης `compile_topology()`:

```
topology      = parse_topology(load_topology(path))
validate(topology)
execution_plan = plan(topology)
source        = render_python(execution_plan)
```

Το παραγόμενο source γράφεται στο `topology.generated.py` και, στην κανονική λειτουργία, εκτελείται ως ξεχωριστό Python process με τα ίδια δικαιώματα με τον coordinator. Η πραγματική εκτέλεση απαιτεί root privileges, ενώ η φόρτωση, το parsing, το validation, το planning και το rendering μπορούν να πραγματοποιηθούν χωρίς αυτά.

Το generated πρόγραμμα δημιουργεί τους κόμβους και τα links, εφαρμόζει τη δικτυακή παραμετροποίηση, εκκινεί τις προβλεπόμενες υπηρεσίες και παραδίδει τον έλεγχο στον χρήστη μέσω του Mininet CLI. Η έξοδος από το CLI οδηγεί σε δομημένο cleanup μέσω `finally`, όπου τερματίζονται οι service processes που ξεκίνησε το ίδιο το πρόγραμμα και στη συνέχεια καλείται το `net.stop()`.

2.2.5. Έλεγχος των επιμέρους σταδίων

Η ανεξαρτησία των σταδίων αξιοποιείται και κατά την ανάπτυξη και τον εντοπισμό σφαλμάτων. Κάθε βασικό module του pipeline διαθέτει ένα μικρό **self-test του δημόσιου interface του**, το οποίο μπορεί να εκτελεστεί χωρίς να δημιουργηθεί πραγματικό Mininet δίκτυο.

Για παράδειγμα:

```
python3 source/topology_loader.py
python3 source/topology_parser.py
python3 source/topology_validator.py
python3 source/topology_plan.py
python3 source/topology_renderer.py
```

Μία επιτυχής εκτέλεση καταλήγει σε μήνυμα της μορφής:

```
topology_parser: self-test passed
```

Τα tests δεν περιορίζονται στην απλή εισαγωγή των modules, αλλά ελέγχουν την κύρια συμπεριφορά του αντίστοιχου δημόσιου API. Ο parser, για παράδειγμα, ελέγχει τόσο σωστή μετατροπή σε typed objects όσο και απόρριψη άγνωστων πεδίων, ενώ ο validator εξετάζει τόσο έγκυρες όσο και πολλαπλά λανθασμένες τοπολογίες. Αντίστοιχα, ο planner ελέγχει την επίλυση του auto, την παραγωγή των physical interface names, την επιλογή **standalone** για τους switches και την παραγωγή των απαιτούμενων sysctls. Ο renderer ελέγχει ότι το παραγόμενο source είναι συντακτικά έγκυρο Python και ότι περιλαμβάνει τις αναμενόμενες κλήσεις Mininet, τις υπηρεσίες και το cleanup.

Ο διαχωρισμός αυτός αποδείχθηκε ιδιαίτερα χρήσιμος κατά την ανάπτυξη, καθώς ένα σφάλμα μπορεί να απομονωθεί στο στάδιο στο οποίο εμφανίζεται, χωρίς να απαιτείται κάθε φορά εκκίνηση του Mininet με δικαιώματα root. Το `--emit` συμπληρώνει αυτή τη διαδικασία, επιτρέποντας την επιθεώρηση της τελικής “assembly” πριν από οποιαδήποτε μεταβολή της πραγματικής δικτυακής κατάστασης του συστήματος.

3. ΠΕΙΡΑΜΑ 1 – DHCP ΚΑΙ ΔΙΑΔΙΚΑΣΙΑ D.O.R.A.

3.1. Σκοπός και πειραματική διαδικασία

Σκοπός του πρώτου πειράματος είναι η παρατήρηση της διαδικασίας μέσω της οποίας ένας host χωρίς στατική διευθυνσιοδότηση αποκτά τις απαραίτητες ρυθμίσεις δικτύου από έναν DHCP server. Ειδικότερα, εξετάζεται η ακολουθία **Discover–Offer–Request–ACK (D.O.R.A.)** και τα αντίστοιχα πεδία των Ethernet, IP και UDP headers.

Η τοπολογία εκκινήθηκε από τον κατάλογο `exercise/` με την εντολή:

```
sudo python3 ../source/build_network.py topology.json
```

Πριν από την καταγραφή αφαιρέθηκε το αποθηκευμένο lease του `dhclient`, ώστε ο `h1` να ξεκινήσει τη διαδικασία χωρίς να επιχειρήσει να επαναχρησιμοποιήσει διεύθυνση από προηγούμενη εκτέλεση. Αφαιρέθηκε επίσης τυχόν προηγούμενο αρχείο καταγραφής και στη συνέχεια ξεκίνησε το `tcpdump` στη διεπαφή `h1-eth0`.

Οι εντολές που εκτελέστηκαν στο Mininet CLI ήταν:

```

minted> h1 rm -f /var/lib/dhcp/dhclient.leases
minted> h1 rm -f /tmp/dhcp.pcap
minted> h1 tcpdump -i h1-eth0 -n -w /tmp/dhcp.pcap &
minted> h1 ps aux | grep tcpdump
minted> h1 dhclient h1-eth0
minted> h1 pkill -INT tcpdump
minted> h1 tcpdump -n -e -vv -r /tmp/dhcp.pcap

```

Η επιλογή `-e` κατά την ανάγνωση της καταγραφής επιτρέπει την εμφάνιση των Ethernet headers και επομένως των MAC διευθύνσεων που απαιτούνται για την ανάλυση. Η καταγραφή ολοκληρώθηκε χωρίς απώλειες από το `tcpdump`, με 12 πακέτα να έχουν καταγραφεί συνολικά.

Η πλήρης συνεδρία της εκτέλεσης βρίσκεται στο [execution.txt](#). Το αρχικό αρχείο καταγραφής διατίθεται στο [dhcp.pcap](#), ενώ η αναλυτικά αποκωδικοποιημένη μορφή του βρίσκεται στο [dhcp-decoded.txt](#).

3.2. Αποτελέσματα καταγραφής – Ζητούμενο 1.1

Η καταγραφή περιλαμβάνει την αναμενόμενη ακολουθία **Discover** → **Offer** → **Request** → **ACK**, η οποία ολοκληρώθηκε επιτυχώς. Στην πραγματική εκτέλεση καταγράφηκαν δύο DHCP Discover με το ίδιο transaction ID πριν από το πρώτο Offer.

Η χρονική ακολουθία των σχετικών DHCP μηνυμάτων ήταν:

```

10:58:29.655251 DHCP Discover
10:58:32.214530 DHCP Discover
10:58:32.670897 DHCP Offer
10:58:32.671193 DHCP Request
10:58:32.673025 DHCP ACK
10:58:32.678132 DHCP Offer

```

Μεταξύ του πρώτου Discover και του πρώτου Offer παρατηρήθηκαν τρία ARP probes από τον DHCP server για τη διεύθυνση 192.168.1.131 και στη συνέχεια ένα ICMP Echo Request προς την ίδια διεύθυνση. Ο έλεγχος αυτός εισήγαγε καθυστέρηση στην πρώτη απάντηση του server, με αποτέλεσμα ο `dhclient` να επαναμεταδώσει το Discover περίπου τρία δευτερόλεπτα μετά το πρώτο.

Και τα δύο Discover χρησιμοποιούν το ίδιο transaction ID, 0x0ecb5a2e, επομένως αποτελούν μέρος της ίδιας προσπάθειας απόκτησης διεύθυνσης και όχι δύο ανεξάρτητες διαδικασίες DHCP. Παρά την επαναμετάδοση, η διαδικασία ολοκληρώθηκε κανονικά με Offer, Request και ACK και ο `h1` έλαβε τη διεύθυνση 192.168.1.131. Μετά το ACK καταγράφηκε ακόμη ένα Offer, χωρίς αυτό να επηρεάζει την ήδη ολοκληρωμένη παραχώρηση της διεύθυνσης.

Η βασική τετράδα D.O.R.A. δίνει τα ακόλουθα στοιχεία:

Μήνυμα	IP	UDP port	MAC
Discover	0.0.0.0 → 255.255.255.255	68 → 67	12:07:f8:bf:6e:ff → ff:ff:ff:ff:ff:ff
Offer	192.168.1.1 → 192.168.1.131	67 → 68	82:3b:16:e9:bb:17 → 12:07:f8:bf:6e:ff
Request	0.0.0.0 → 255.255.255.255	68 → 67	12:07:f8:bf:6e:ff → ff:ff:ff:ff:ff:ff
ACK	192.168.1.1 → 192.168.1.131	67 → 68	82:3b:16:e9:bb:17 → 12:07:f8:bf:6e:ff

Πίνακας 1: Διευθύνσεις και UDP θύρες των μηνυμάτων της διαδικασίας D.O.R.A.

3.3. Ανάλυση της διαδικασίας D.O.R.A.

Στο **Discover** ο h1 δεν διαθέτει ακόμη διεύθυνση IPv4 και αναζητά διαθέσιμο DHCP server στο τοπικό δίκτυο. Για τον λόγο αυτό χρησιμοποιεί ως source IP την 0.0.0.0 και μεταδίδει το μήνυμα προς την περιορισμένη broadcast διεύθυνση 255.255.255.255, με Ethernet destination ff:ff:ff:ff:ff:ff.

Στο **Offer** ο DHCP server του r0, με διεύθυνση 192.168.1.1, προσφέρει στον client τη διεύθυνση 192.168.1.131. Στο ίδιο μήνυμα περιλαμβάνονται επίσης η μάσκα 255.255.255.0, η broadcast διεύθυνση 192.168.1.255 και η προεπιλεγμένη πύλη 192.168.1.1.

Ο h1 απαντά με **Request**, στο οποίο δηλώνει τόσο τον server που επέλεξε όσο και τη διεύθυνση που ζητά να του εκχωρηθεί. Τα αντίστοιχα options που καταγράφηκαν είναι:

```
Server-ID Option:    192.168.1.1
Requested-IP Option: 192.168.1.131
```

Τέλος, με το **ACK** ο server επιβεβαιώνει την παραχώρηση της 192.168.1.131 και αποστέλλει τις τελικές παραμέτρους του lease. Στην καταγραφή το lease time είναι 43200 δευτερόλεπτα και η προεπιλεγμένη πύλη που παρέχεται στον client είναι η 192.168.1.1.

3.4. Broadcast μετάδοση του DHCP Request – Ζητούμενο 1.2

Το DHCP Request της αρχικής διαδικασίας απόκτησης διεύθυνσης μεταδόθηκε ως **broadcast** τόσο στο επίπεδο Ethernet όσο και στο επίπεδο IP. Στην καταγραφή παρατηρούνται συγκεκριμένα:

```
Ethernet destination: ff:ff:ff:ff:ff:ff
IP destination:       255.255.255.255
UDP ports:            68 -> 67
```

Κατά τη στιγμή αποστολής του Request η παραχώρηση της διεύθυνσης δεν έχει ακόμη ολοκληρωθεί και ο client συνεχίζει να χρησιμοποιεί την 0.0.0.0 ως source IP. Η broadcast μετάδοση επιτρέπει επιπλέον σε όλους τους DHCP servers του τοπικού δικτύου να παρατηρήσουν ποια προσφορά επέλεξε ο client.

Ο επιλεγμένος server αναγνωρίζεται από το Server-ID του Request, ενώ τυχόν άλλοι servers μπορούν να διαπιστώσουν ότι οι δικές τους προσφορές δεν επιλέχθηκαν. Το ACK του 192.168.1.1 ολοκληρώνει τελικά τη διαδικασία και επιβεβαιώνει ότι η 192.168.1.131 μπορεί πλέον να χρησιμοποιηθεί από τον h1.

4. ΠΕΙΡΑΜΑ 2 – ΠΡΩΤΟΚΟΛΛΟ ARP

4.1. Σκοπός και πειραματική διαδικασία

Σκοπός του δεύτερου πειράματος είναι η παρατήρηση της διαδικασίας αντιστοίχισης μιας γνωστής διεύθυνσης IPv4 στην αντίστοιχη διεύθυνση MAC μέσω του πρωτοκόλλου **ARP**. Ειδικότερα, εξετάζεται η αρχική επίλυση της διεύθυνσης του h2 από τον h1 και η διαφορετική μορφή μετάδοσης του ARP Request και του ARP Reply.

Η τοπολογία εκκινήθηκε εκ νέου και οι δύο hosts του Subnet A απέκτησαν διευθύνσεις μέσω DHCP. Οι διευθύνσεις που εκχωρήθηκαν κατά τη συγκεκριμένη εκτέλεση ήταν:

- 192.168.1.131/24 για τον h1 και
- 192.168.1.164/24 για τον h2.

Οι αρχικές εντολές της εκτέλεσης ήταν:

```
sudo python3 ../source/build_network.py topology.json
```

```
minted> h1 ip -4 -br addr
```

```

minted> h2 ip -4 -br addr
minted> h1 dhclient h1-eth0
minted> h2 dhclient h2-eth0
minted> h1 ip -4 -br addr
minted> h2 ip -4 -br addr

```

Πριν από το ping διαγράφηκε ο neighbor table του h1, ώστε να μην υπάρχει ήδη γνωστή αντιστοίχιση μεταξύ της IP και της MAC διεύθυνσης του h2. Η κενή κατάσταση του πίνακα επιβεβαιώθηκε με την εντολή `ip neigh show`.

```

minted> h1 ip neigh flush all
minted> h1 ip neigh show
minted> h1 rm -f /tmp/arp.pcap
minted> h1 tcpdump -i h1-eth0 -n arp -w /tmp/arp.pcap &
minted> h1 ping -c 1 192.168.1.164
minted> h1 ip neigh show
minted> h1 pkill -INT tcpdump
minted> h1 tcpdump -n -e -vv -r /tmp/arp.pcap

```

Το φίλτρο `arp` του `tcpdump` περιορίζει την καταγραφή στα ARP frames, ενώ η επιλογή `-e` κατά την ανάγνωση εμφανίζει τα Ethernet headers και επομένως τις MAC διευθύνσεις πηγής και προορισμού. Η καταγραφή ολοκληρώθηκε χωρίς απώλειες, με τέσσερα ARP frames να έχουν καταγραφεί συνολικά.

Η πλήρης συνεδρία της εκτέλεσης βρίσκεται στο [execution.txt](#). Το αρχικό αρχείο καταγραφής διατίθεται στο [arp.pcap](#), ενώ η αποκωδικοποιημένη μορφή του βρίσκεται στο [arp-decoded.txt](#).

4.2. Αποτελέσματα καταγραφής

Μετά την εκκαθάριση του neighbor table, το πρώτο ping από τον h1 προς τον h2 απαίτησε την επίλυση της MAC διεύθυνσης που αντιστοιχεί στην 192.168.1.164. Το πρώτο ζεύγος πακέτων της καταγραφής ήταν:

```

4a:6f:2a:18:f4:80 > ff:ff:ff:ff:ff:ff
Request who-has 192.168.1.164 tell 192.168.1.131

82:c0:6d:ae:3f:56 > 4a:6f:2a:18:f4:80
Reply 192.168.1.164 is-at 82:c0:6d:ae:3f:56

```

Από την καταγραφή προκύπτει επομένως ότι στη συγκεκριμένη εκτέλεση οι δύο hosts είχαν τα ακόλουθα στοιχεία:

Host	IPv4	MAC
h1	192.168.1.131	4a:6f:2a:18:f4:80
h2	192.168.1.164	82:c0:6d:ae:3f:56

Πίνακας 2: Διευθύνσεις των hosts που συμμετέχουν στο πείραμα ARP.

Το ping ολοκληρώθηκε επιτυχώς και, μετά την ανταλλαγή ARP, ο neighbor table του h1 περιείχε (λογικά) την αντιστοίχιση:

```
192.168.1.164 dev h1-eth0 lladdr 82:c0:6d:ae:3f:56 REACHABLE
```

Το αποτέλεσμα αυτό επιβεβαιώνει ότι ο h1 έμαθε τη MAC διεύθυνση του h2 μέσω της προηγούμενης ανταλλαγής ARP και μπορεί πλέον να δημιουργεί απευθείας Ethernet frames προς αυτόν.

4.3. Ανάλυση του ARP Request και Reply – Ζητούμενο 2.1

Κατά την αποστολή του αρχικού ARP Request ο h1 γνωρίζει τη διεύθυνση IP του προορισμού, αλλά **δεν γνωρίζει ακόμη τη MAC διεύθυνσή του**. Δεν μπορεί επομένως να δημιουργήσει ένα unicast Ethernet frame προς τον h2.

Για τον λόγο αυτό το ARP Request μεταδίδεται προς την Ethernet broadcast διεύθυνση:

```
Source MAC:      4a:6f:2a:18:f4:80
Destination MAC: ff:ff:ff:ff:ff:ff
```

```
Who has 192.168.1.164?
Tell 192.168.1.131
```

Όλοι οι κόμβοι του ίδιου Layer-2 broadcast domain μπορούν να λάβουν το Request, αλλά μόνο ο host στον οποίο ανήκει η 192.168.1.164 πρέπει να απαντήσει. Στην συγκεκριμένη περίπτωση ο host αυτός είναι ο h2.

Το ARP Reply, αντίθετα, μεταδίδεται ως **unicast**:

```
Source MAC:      82:c0:6d:ae:3f:56
Destination MAC: 4a:6f:2a:18:f4:80
```

```
192.168.1.164 is-at 82:c0:6d:ae:3f:56
```

Ο h2 μπορεί να στείλει την απάντηση απευθείας στον h1, επειδή το ARP Request περιλαμβάνει τα στοιχεία του αποστολέα και επομένως παρέχει στον h2 τη MAC διεύθυνση στην οποία πρέπει να απαντήσει. Δεν υπάρχει συνεπώς ανάγκη το Reply να μεταδοθεί σε όλους τους κόμβους του broadcast domain.

Η MAC προορισμού του αρχικού ARP Request είναι επομένως ff:ff:ff:ff:ff:ff, ενώ το ARP Reply επιστρέφει ως unicast στη MAC 4a:6f:2a:18:f4:80 του h1.

4.4. Πρόσθετη παρατήρηση

Στην καταγραφή εμφανίστηκε περίπου πέντε δευτερόλεπτα αργότερα και ένα δεύτερο ζεύγος ARP Request/Reply προς την αντίθετη κατεύθυνση:

```
82:c0:6d:ae:3f:56 > 4a:6f:2a:18:f4:80
Request who-has 192.168.1.131 tell 192.168.1.164
```

```
4a:6f:2a:18:f4:80 > 82:c0:6d:ae:3f:56
Reply 192.168.1.131 is-at 4a:6f:2a:18:f4:80
```

Σε αντίθεση με το αρχικό Request, το δεύτερο Request έχει ήδη ως Ethernet destination τη MAC του h1 και επομένως είναι unicast. Η ανταλλαγή αυτή σχετίζεται με τη διαχείριση και επιβεβαίωση της κατάστασης του neighbor cache από το Linux και δεν αποτελεί την αρχική επίλυση που προκαλεί το ping.

Για το ζητούμενο του πειράματος εξετάζεται συνεπώς το **πρώτο ζεύγος Request/Reply**, στο οποίο φαίνεται καθαρά η μετάβαση από broadcast αναζήτηση σε unicast απάντηση. Παρόλα αυτά το γεγονός ότι υπήρχε δεύτερο ζεύγος μας προβλημάτισε ιδιαίτερα μέχρι να ανακαλύψουμε τον λόγο ύπαρξής του!

4.5. Συμπεράσματα

...

5. ΣΥΝΟΨΗ

Τέλος, ...