



دانشکده مهندسی کامپیوتر

دکتر وصال حکمی

پاییز ۱۴۰۰

پیاده‌سازی یک سامانه دیواره آتش مبتنی بر شبکه نرم افزار محور

امیر رضائی

پروژه کارشناسی

شماره دانشجویی: ۹۶۵۲۱۲۲۷

---

## چکیده

مسئله‌ای که در پیش‌رو مورد بررسی و پیاده‌سازی قرار گرفته است، یک سامانه دیواره آتش<sup>۱</sup> مبتنی بر شبکه نرم افزار محور<sup>۲</sup> و بستر مقلد مینی‌نت<sup>۳</sup> است. در ابتدای مسیر قصد آشنایی با بستر مقلد مینی‌نت را داریم و با پیاده‌سازی یک شبکه، تشکیل شده از چند میزبان<sup>۴</sup> و سوئیچ، روش کار با این بستر را فرامی‌گیریم. سپس به دنبال روشی برای کنترل بسته‌های ترافیکی موجود در شبکه می‌رویم. بدین منظور با کنترل‌کننده در شبکه نرم‌افزار محور آشنا می‌شویم. کنترل‌کننده‌ای که مورد بررسی قرار می‌گیرد، کنترل‌کننده OpenFlow است. این کنترل‌کننده در زبان برنامه‌نویسی پایتون<sup>۵</sup> با عنوان POX عرضه شده است. برای کار با POX در ابتدا با دستورهای آن در زبان برنامه‌نویسی پایتون آشنا شده سپس به سراغ پیاده‌سازی چندین شبکه که از این کنترل‌کننده استفاده می‌کنند می‌رویم. و سامانه دیواره آتش را برای یک سازمان با استفاده از کنترل‌کننده پیاده‌سازی می‌کنیم. در انتها یک طرح و سناریوی مبتنی بر قانون را تعریف کرده و با این کنترل‌کننده آن را پیاده‌سازی می‌کنیم.

## واژه‌های کلیدی

دیواره آتش، شبکه نرم‌افزار محور، مینی‌نت، OpenFlow ، POX

---

<sup>۱</sup>Firewall

<sup>۲</sup>Software-Defined Networking

<sup>۳</sup>Mininet Emulator

<sup>۴</sup>Host

<sup>۵</sup>Python

## فهرست مطالب

۱	مقدمه	۱
۳	معرفی دوره‌های آموزشی	۲
۴	پیاده‌سازی یک شبکه ساده در mininet	۳
۱۳	پیاده‌سازی یک کنترل‌کننده ساده با استفاده از POX	۴
۱۳	۱.۴ OpenFlow	۱۳
۱۴	۲.۴ POX	۱۴
۱۴	۳.۴ آشنایی با دستورات و توابع POX	۱۴
۱۶	۴.۴ پیاده‌سازی کنترل‌کننده و مشاهده نتایج	۱۶
۱۹	۵ پیاده‌سازی دیواره آتش با استفاده از کنترل‌کننده راه دور	۱۹
۲۲	۱.۵ پیاده‌سازی دیواره آتش و مشاهده نتایج	۲۲
۲۴	۶ پیاده‌سازی دیواره آتش واقعی برای یک سازمان	۲۴
۲۴	۱.۶ معرفی ساختار سازمان و قوانین دیواره آتش	۲۴
۲۵	۲.۶ پیاده‌سازی دیواره آتش سازمان و مشاهده نتایج	۲۵
۲۹	۷ پیاده‌سازی سناریوی مبتنی بر قانون در دیواره آتش	۲۹

۱.۷	اتصال از راه دور به سرور لینوکسی و باز نمودن پنجره	
۲۹	گرافیکی . . . . .	
۳۱	ایجاد TCP Connection . . . . .	۲.۷
۳۲	معرفی ساختار شبکه برای پیاده سازی سناریو . . . . .	۳.۷
۳۴	معرفی سناریوی مبتنی بر قانون . . . . .	۴.۷
۳۷	نتایج اعمال سیاست در تصمیم گیری سوئیچ . . . . .	۵.۷
۴۷		۸ مراجع
۵۰		۹ پیوست

## ۱ مقدمه

دستگاه‌های موجود در شبکه ( سوئیچ‌ها <sup>۱</sup> ) پیچیده هستند. به دلیل اینکه در آن‌ها، لایه‌ی کنترلی <sup>۲</sup> و لایه‌ی ارسال داده <sup>۳</sup> به صورت یکجا قرار داده شده‌است. این ساختار پیچیده، عملکرد شبکه را از نظر تحویل با تاخیر بسته‌ها و همچنین در مواردی، ارسال مجدد بسته‌ها تحت تاثیر قرار می‌دهد [۱]. شبکه نرم‌افزار محور، گونه‌ای از معماری شبکه است که قابلیت کنترل به صورت هوشمندانه و مرکزی را به شبکه اضافه می‌کند. OpenFlow راه‌کاری است که این معماری معرفی کرده است که با جدا کردن عملکرد کنترلی از عملکرد ارسال بسته‌ها باعث افزایش عملکرد شبکه می‌شود [۱]. کنترل این شبکه‌ها می‌تواند با استفاده از برنامه‌های نرم‌افزاری <sup>۴</sup> بدون توجه به این که فناوری مورد استفاده در اجزای شبکه چگونه است، صورت گیرد [۲].

در این گزارش، سازوکار دیواره‌های آتش مبتنی بر این معماری را بررسی می‌کنیم. این نوع از دیواره آتش می‌تواند به عنوان افزونه به کنترل‌کننده‌های شبکه اضافه شود. این دیواره آتش توانایی شناسایی برنامه‌های موجود در شبکه که عملکرد شبکه را کاهش می‌دهند دارد. با اعمال کردن محدودیت‌ها به صورت پویا، کنترل بیشتری روی شبکه اعمال می‌کند [۳].

<sup>۱</sup> Switches  
<sup>۲</sup> Control Plane  
<sup>۳</sup> Data Plane  
<sup>۴</sup> Software Applications

این دیواره آتش، ترافیک عبوری شبکه را در لایه‌های شبکه<sup>۱</sup>، انتقال<sup>۲</sup> و برنامه<sup>۳</sup> بررسی می‌کند. و دستورالعمل‌های امنیتی مدیر شبکه را بر شبکه اعمال می‌کند [۳]. دیواره آتش ارائه شده، بر روی کنترل‌کننده POX که مبتنی بر زبان برنامه‌نویسی Python است، پیاده‌سازی شده است. همچنین ساختار شبکه‌ها به وسیله‌ی شبیه‌ساز مینی‌نت ساخته شده است. در فصل دوم این گزارش، ابتدا چند دوره‌ی آموزشی مفید را که به فراگیری مفاهیم SDN و آشنایی با مینی‌نت کمک می‌کند، معرفی می‌کنیم. در فصل سوم با شبیه‌ساز مینی‌نت سروکار داریم و با پیاده‌سازی یک شبکه در آن، با محیط این شبیه‌ساز آشنا می‌شویم. در فصل چهارم به سراغ آشنایی با POX رفته و با تابع‌های آن آشنا می‌شویم و یک کنترل‌کننده ساده با POX می‌نویسیم و آنرا در محیط مینی‌نت آزمایش می‌کنیم. پس از آشنایی‌های اولیه به فصل پنجم رسیده و در آن، دیواره آتش را با استفاده از POX در بستر مینی‌نت به صورت اولیه پیاده‌سازی و آزمایش می‌کنیم. در فصل ششم یک دیواره آتش را مطابق با نیازهای واقعی یک سازمان پیاده‌سازی کرده و شبکه این سازمان را در بستر مینی‌نت ترسیم کرده و دیواره آتش را برای این سازمان آزمایش می‌کنیم. و در فصل هفتم با استفاده از تعریف یک طرح، دیواره آتش متناسب با آن را پیاده‌سازی می‌کنیم.

---

<sup>۱</sup>Network Layer<sup>۲</sup>Transport Layer<sup>۳</sup>Application Layer

## ۲ معرفی دوره‌های آموزشی

در ابتدا با توجه به مفاهیم و عباراتی که در توضیح پروژه بکار رفته است دوره آموزشی Udemy Complete, practical SDN and Open-Flow Fundamentals: over 8hrs [۴] را به شما معرفی می‌کنیم. در ابتدای این دوره مفاهیم و پیش‌زمینه‌های مربوط به SDN, OpenFlow و OpenDayLight معرفی می‌شود که برای شروع، فراگیری این مفاهیم ضروری است.

محتوای آغاز این دوره به صورت زیر است:

- – Fundamentals Welcome
- – Welcome to the SDN and OpenFlow course!
- – SDN Terms and Definitions
- – Fundamentals Course: OpenFlow Theory
- – Mininet and OpenDaylight (ODL)

### ۳ پیاده‌سازی یک شبکه ساده در mininet

پس از فراگیری مفهومی‌های دوره، به سراغ نصب شبیه‌ساز<sup>۱</sup> mininet می‌رویم. این شبیه‌ساز بر روی تمامی سیستم‌عامل‌های موجود در دسترس است اما برای راحتی کار، این شبیه‌ساز را به صورت مجازی بر روی VMware Workstation Pro اجرا می‌کنیم.

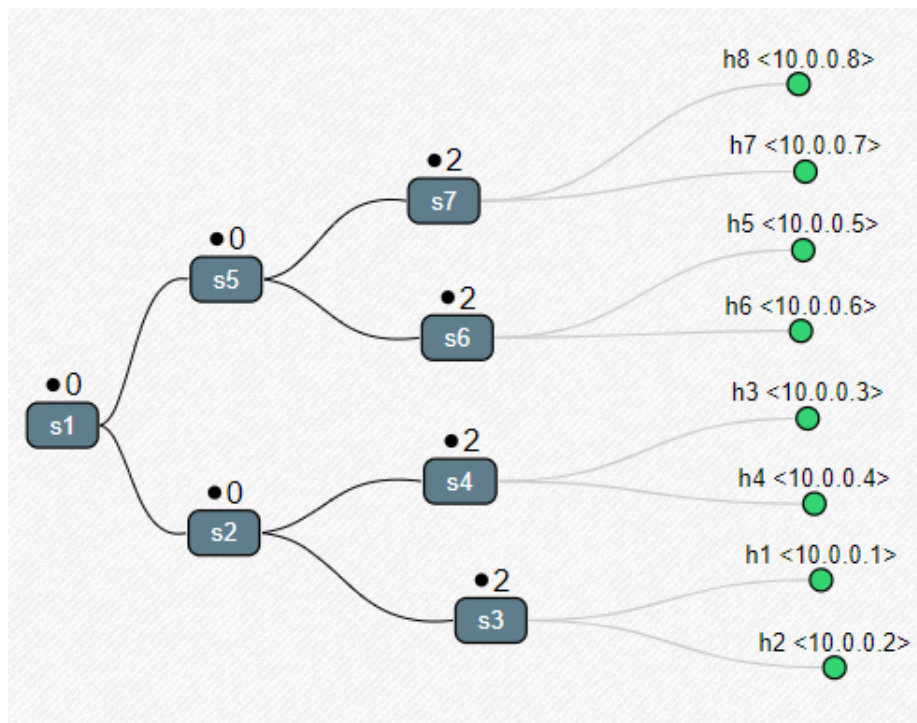
با مراجعه به [An expedient introduction to Mininet](#) [۵] با ساختن شبکه در mininet و ساخت میزبان و سوئیچ و متصل کردن آن‌ها به یکدیگر آشنا می‌شویم. و با پیاده‌سازی یک شبکه ابتدایی با تعداد ۸ میزبان و تعداد ۷ عدد سوئیچ، آن را در محیط mininet اجرا می‌کنیم. ساختار این شبکه به صورت شکل ۱ است. که در مورد نحوه بدست آوردن شکل و توپولوژی<sup>۲</sup> شبکه نیز در قسمت‌های بعدی صحبت خواهیم کرد. این شبکه دارای ساختاری درختی بوده که از ۷ سوئیچ که برخی از آن‌ها به یکدیگر به صورت مستقیم به یکدیگر متصل‌اند، تشکیل شده است. و ۸ میزبان که از بازه آدرسی <10.0.0.1> تا <10.0.0.8> هستند به سوئیچ‌ها متصل‌اند.

---

<sup>۱</sup> Emulator

<sup>۲</sup> Topolog





شکل ۱: طرح گرافیکی شبکه

در هنگام پیاده‌سازی، متوجه این قضیه شدیم که این شبکه، خود به صورت پیش فرض از کنترل کننده<sup>۱</sup> خارجی استفاده نمی‌کند و تنها از کنترل کننده‌ای که توسط mininet در اختیار آن قرار داده شده‌است استفاده می‌کند. قسمت‌های کلیدی برنامه این شبکه در الگوریتم ۱ مشخص شده است.

<sup>۱</sup> Controller

الگوریتم ۱: tree.py

```
1 from mininet.node import Controller
2
3 net = Mininet(controller=Controller)
4 net.addController('c0')
5 h1 = net.addHost('h1', ip='10.0.0.1')
6 s1 = net.addSwitch('s1')
7 net.addLink(h1, s3)
8 net.start()
9 CLI(net)
```

در این برنامه با ساختن یک شبکه با نام *net* که دارای یک کنترل کننده پیش فرض است، با تابع *addHost*، میزبان‌های مورد نظر را به شبکه اضافه می‌کنیم. همچنین با تابع *addSwitch*، سویچ‌های مورد نظر را به شبکه اضافه می‌کنیم. سپس با تابع *addLink* می‌توانیم میزبان‌ها و سویچ‌ها را به یکدیگر متصل کنیم.

پس از کامل کردن و ساخت شبکه‌ی مورد نظر آن را با دستور موجود در الگوریتم ۲ در *mininet* اجرا کرده و با نتیجه (شکل ۲) روبرو می‌شویم.

الگوریتم ۲: دستور اجرای برنامه tree.py

```
1 sudo python3 tree.py
```

```
mininet@mininet-vm:~/FFS$ sudo python3 tree.py
*** Adding controller
*** Adding hosts
*** Adding switches
*** Creating links
*** Starting network
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8
*** Starting controller
c0
*** Starting 7 switches
s1 s2 s3 s4 s5 s6 s7 ...
*** Running CLI
*** Starting CLI:
mininet> █
```

شکل ۲: نتیجه اجرای برنامه tree.py

با اجرای دستور net در محیط CLI با نتیجه شکل ۳ مواجه می‌شویم که شبکه و اتصال میان سویچ‌ها و میزبان‌ها را در CLI نمایش می‌دهد.

```
mininet> net
h1 h1-eth0:s3-eth1
h2 h2-eth0:s3-eth2
h3 h3-eth0:s4-eth1
h4 h4-eth0:s4-eth2
h5 h5-eth0:s6-eth1
h6 h6-eth0:s6-eth2
h7 h7-eth0:s7-eth1
h8 h8-eth0:s7-eth2
s1 lo: s1-eth1:s2-eth1 s1-eth2:s5-eth1
s2 lo: s2-eth1:s1-eth1 s2-eth2:s3-eth3 s2-eth3:s4-eth3
s3 lo: s3-eth1:h1-eth0 s3-eth2:h2-eth0 s3-eth3:s2-eth2
s4 lo: s4-eth1:h3-eth0 s4-eth2:h4-eth0 s4-eth3:s2-eth3
s5 lo: s5-eth1:s1-eth2 s5-eth2:s6-eth3 s5-eth3:s7-eth3
s6 lo: s6-eth1:h5-eth0 s6-eth2:h6-eth0 s6-eth3:s5-eth2
s7 lo: s7-eth1:h7-eth0 s7-eth2:h8-eth0 s7-eth3:s5-eth3
c0
```

شکل ۳: نتیجه اجرای دستور net

همچنین بصورت جداگانه نیز با اجرای دستورهای links در شکل ۴ و dump در شکل ۵ می‌توان به ترتیب به دستگاه‌های بکار برده شده و ویژگی‌های آن‌ها در شبکه و اتصال‌های موجود میان این دستگاه‌ها پی برد.

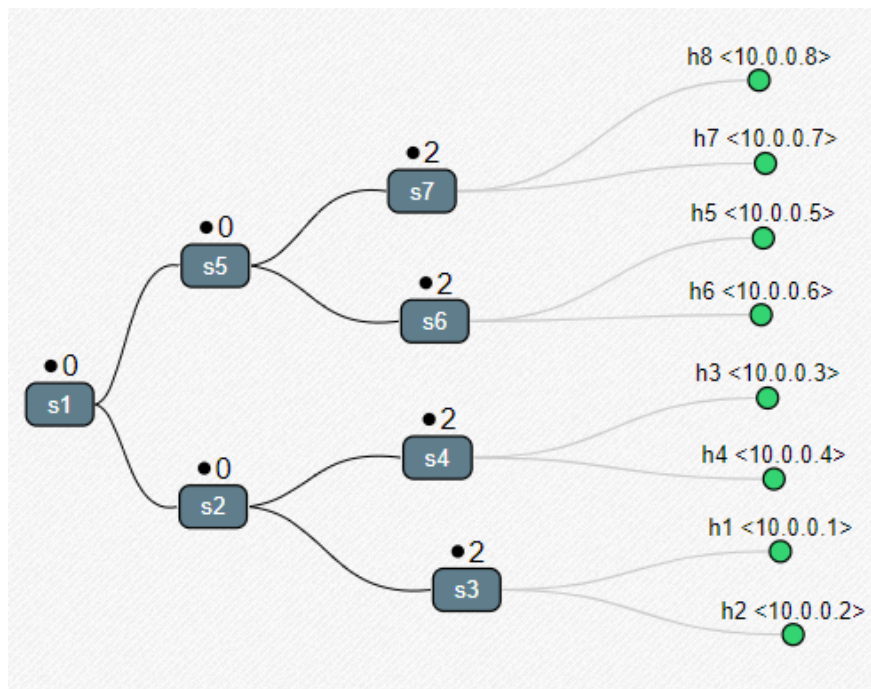
```
mininet> links
h1-eth0<->s3-eth1 (OK OK)
h2-eth0<->s3-eth2 (OK OK)
h3-eth0<->s4-eth1 (OK OK)
h4-eth0<->s4-eth2 (OK OK)
h5-eth0<->s6-eth1 (OK OK)
h6-eth0<->s6-eth2 (OK OK)
h7-eth0<->s7-eth1 (OK OK)
h8-eth0<->s7-eth2 (OK OK)
s1-eth1<->s2-eth1 (OK OK)
s2-eth2<->s3-eth3 (OK OK)
s2-eth3<->s4-eth3 (OK OK)
s1-eth2<->s5-eth1 (OK OK)
s5-eth2<->s6-eth3 (OK OK)
s5-eth3<->s7-eth3 (OK OK)
```

شکل ۴: نتیجه اجرای دستور links

```
mininet> dump
<Host h1: h1-eth0:10.0.0.1 pid=2081>
<Host h2: h2-eth0:10.0.0.2 pid=2083>
<Host h3: h3-eth0:10.0.0.3 pid=2085>
<Host h4: h4-eth0:10.0.0.4 pid=2087>
<Host h5: h5-eth0:10.0.0.5 pid=2089>
<Host h6: h6-eth0:10.0.0.6 pid=2091>
<Host h7: h7-eth0:10.0.0.7 pid=2093>
<Host h8: h8-eth0:10.0.0.8 pid=2095>
<OVSSwitch s1: lo:127.0.0.1,s1-eth1:None,s1-eth2:None pid=2100>
<OVSSwitch s2: lo:127.0.0.1,s2-eth1:None,s2-eth2:None,s2-eth3:None pid=2103>
<OVSSwitch s3: lo:127.0.0.1,s3-eth1:None,s3-eth2:None,s3-eth3:None pid=2106>
<OVSSwitch s4: lo:127.0.0.1,s4-eth1:None,s4-eth2:None,s4-eth3:None pid=2109>
<OVSSwitch s5: lo:127.0.0.1,s5-eth1:None,s5-eth2:None,s5-eth3:None pid=2112>
<OVSSwitch s6: lo:127.0.0.1,s6-eth1:None,s6-eth2:None,s6-eth3:None pid=2115>
<OVSSwitch s7: lo:127.0.0.1,s7-eth1:None,s7-eth2:None,s7-eth3:None pid=2118>
<Controller c0: 127.0.0.1:6653 pid=2074>
```

شکل ۵: نتیجه اجرای دستور dump

با مراجعه به سایت دموئی شبیه‌ساز طرح شبکه mininet از طریق این [لینک](#) با استفاده از دستورهای dump و links می‌توان به صورت گرافیکی به طرح شبکه رسید [۶]. که طرح گرافیکی شبکه‌ی ایجاد شده توسط اجرای برنامه tree.py بصورت شکل ۶ است. که پیش‌تر در شکل ۱ نیز به آن اشاره کرده بودیم.



شکل ۶: طرح گرافیکی شبکه

برای اطمینان از اتصال تمامی میزبان‌ها به یکدیگر با اجرای دستور pingall در شکل ۷ از اتصال، اطمینان حاصل می‌کنیم. که مشاهده می‌شود 0% بسته‌ها رها<sup>۱</sup> شده‌اند. یا به عبارت دیگر، هیچ بسته‌ای رها نشده‌است.

<sup>۱</sup>Drop

```
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 h7 h8
h2 -> h1 h3 h4 h5 h6 h7 h8
h3 -> h1 h2 h4 h5 h6 h7 h8
h4 -> h1 h2 h3 h5 h6 h7 h8
h5 -> h1 h2 h3 h4 h6 h7 h8
h6 -> h1 h2 h3 h4 h5 h7 h8
h7 -> h1 h2 h3 h4 h5 h6 h8
h8 -> h1 h2 h3 h4 h5 h6 h7
*** Results: 0% dropped (56/56 received)
mininet> |
```

شکل ۷: نتیجه اجرای برنامه pingall



## ۴ پیاده‌سازی یک کنترل‌کننده ساده با استفاده از POX

در ادامه پس از پیاده‌سازی موفقیت‌آمیز یک شبکه و اتصال‌های موجود میان میزبان‌های آن. متوجه این قضیه شدیم که بسته‌های ارسالی میان این میزبان‌ها توسط کنترل‌کننده که بصورت پیش‌فرض در mininet تعبیه شده است، کنترل می‌شود. حال با استفاده از OpenFlow می‌خواهیم کنترل این بسته‌های ارسالی در شبکه را در دست خود بگیریم.

### ۱.۴ OpenFlow

OpenFlow قراردادی ارتباطی در معماری نرم‌افزار محور است. که قابلیت ارتباط مستقیم لایه‌ی کنترلی<sup>۱</sup> با لایه‌ی ارسالی<sup>۲</sup> دستگاه‌های موجود در شبکه مانند مسیریاب‌ها و سوئیچ‌ها را به شبکه اضافه می‌کند [۷]. OpenFlow اجازه‌ی کنترل از راه دور جدول‌های سوئیچ<sup>۳</sup> را به ما می‌دهد. به عبارت دیگر عمل ارسال کردن<sup>۴</sup> در سوئیچ‌ها می‌تواند به صورت پویا، با اضافه یا کم کردن قاعده‌ها تغییر کند.

---

<sup>۱</sup>Control Plane

<sup>۲</sup>Forwarding Plane

<sup>۳</sup>Forwarding Table

<sup>۴</sup>Forward

## ۲.۴ POX

تعداد زیادی از کنترل‌کننده‌های OpenFlow با پشتیبانی از زبان‌های مختلف برنامه‌نویسی وجود دارد. بطور مثال در زبان پایتون<sup>۱</sup> این کنترل‌کننده با نام POX عرضه شده است. یا در زبان‌های برنامه‌نویسی دیگر نظیر جاوا این کنترل‌کننده با نام FloodLight [۸] عرضه شده است. تعدادی از این کنترل‌کننده‌ها در جدول ۱ گردآوری شده است. POX که کنترل‌کننده OpenFlow در زبان پایتون است، به سادگی از طریق مخزن گیت **POX** قابل دسترسی است. همچنین مستند چگونگی استفاده از این ابزار از طریق این **لینک** و این **لینک** قابل دسترسی است [۹].

جدول ۱: کنترل‌کننده‌های مختلف OpenFlow

ODL	Floodlight	Trema	Ryu	POX	
Java	Java	C, Ruby	Python	Python	زبان
v۱.۰	v۱.۰	v۱.۰	v۱.۰، v۲.۰، v۳.۰	v۱.۰	OpenFlow
بله	بله	بله	بله	بله	متن‌باز

## ۳.۴ آشنایی با دستورات و توابع POX

`ofp_flow_mod()`

این دستور برای ایجاد یک entry در جدول سوئیچ استفاده می‌شود.

---

<sup>۱</sup>Python

## priority

برای اولویت اختصاص دادن به entry های ایجاد شده در جدول سوئیچ بکار می‌رود. در صورت وجود چند entry که قابلیت اعمال شدن داشته باشند. آن‌گاه وجود priority لازم است.

## match()

برای بررسی انطباق شرایط بسته با entry موجود در جدول سوئیچ‌ها استفاده می‌شود. که می‌توان موارد زیر را برای آن بررسی کرد.

- nw\_src: برای بررسی ip address مبدا
- nw\_dst: برای بررسی ip address مقصد
- nw\_proto: برای بررسی ip protocol که بعنوان مثال عدد 1 برای ICMP و عدد 6 برای TCP است.

همچنین موارد دیگری نیز برای بررسی مثل MAC\_Address وجود دارند، که لیست تمامی آن‌ها در مستندات POX موجود است.

## ofp\_action\_output()

برای تعیین پورت خروجی بسته از سوئیچ مورد استفاده قرار می‌گیرد.

## send()

برای ارسال entry مورد نظر به سوئیچ مورد استفاده قرار می‌گیرد.

## ۴.۴ پیاده‌سازی کنترل کننده و مشاهده نتایج

با مراجعه به این [سایت](#)، در ادامه پیاده‌سازی شبکه‌ای که در فصل قبل داشتیم، با توجه به اینکه شبکه پیاده‌سازی شده، از کنترل کننده پیش فرض mininet استفاده می‌کند، قرار است، کنترل کننده‌ای با استفاده از POX بنویسیم. جهت انجام این کار برنامه شبکه را مطابق الگوریتم ۳ تغییر می‌دهیم.

الگوریتم ۳: remote-controlled-tree.py

```
۱ from mininet.node import RemoteController
۲ net = Mininet( controller=RemoteController )
```

در واقع تغییری که این برنامه نسبت به برنامه قبلی داشته است، این است که در این برنامه دیگر از کنترل کننده پیش فرض در mininet استفاده نمی‌شود و شبکه از کنترل کننده که دستورهای را به صورت از راه دور یا بعبارت دیگر remote به سوییچ‌ها ارسال می‌کند استفاده می‌کند [۱۰]. این کنترل کننده قرار است قاعده‌های زیر را اجرا کند:

- بسته‌های ارسالی میان  $h1$  و  $h2$  بصورت دو طرفه مسدود می‌شود.
- بسته‌های ارسالی میان  $h2$  و  $h4$  بصورت دو طرفه مسدود می‌شود.
- بسته‌های ارسالی میان  $h2$  و  $h7$  بصورت دو طرفه مسدود می‌شود.
- بسته‌های ارسالی میان  $h3$  و  $h8$  بصورت دو طرفه مسدود می‌شود.

در این برنامه یک کنترل‌کننده بصورت ساده پیاده‌سازی می‌شود که بسته‌های ارسالی میان دو میزبان مشخص را مسدود می‌کند. این برنامه دارای تابعی با نام `handle_ConnctionUp()` است که این تابع در هر باری که یک میزبان سعی دارد از طریق سویچ‌ها به میزبان دیگری دسترسی پیدا کند، اجرا می‌شود. در این برنامه قاعده‌ای مبنی بر بررسی مبدا و مقصد بسته‌ها و در صورت تطابق، مسدود کردن بسته، به سویچ‌ها ارسال می‌شود و سویچ‌ها این دستور را اجرا می‌کنند. برای اجرای این کنترل‌کننده مخزن گیت POX را از [این آدرس](#)، دریافت کرده و فایل کنترل‌کننده را در مسیر زیر قرار می‌دهیم.

`pox/pox/misc`

با اجرای این کنترل‌کننده با استفاده از دستور موجود در الگوریتم ۴ و متصل شدن شبکه به آن (همانند آنچه در شکل ۸ نشان داده شده است.) دستور `pingall` را اجرا می‌کنیم و انتظار داریم که قاعده‌های یاد شده اجرا شوند. می‌توان نتیجه اجرای دستور `pingall` را در شکل ۹ مشاهده کرد.

الگوریتم ۴: دستور اجرای کنترل‌کننده

```
./pox.py log.level --DEBUG openflow.of_01 forwarding.  
l2_learning misc.firewall
```

```
mininet@mininet-vm:~/FFS/pox$ ./pox.py log.level --DEBUG openflow.
of_01 forwarding.12_learning misc.firewall
POX 0.7.0 (gar) / Copyright 2011-2020 James McCauley, et al.
DEBUG:core:POX 0.7.0 (gar) going up...
DEBUG:core:Running on CPython (3.8.5/Jul 28 2020 12:59:40)
DEBUG:core:Platform is Linux-5.4.0-42-generic-x86_64-with-glibc2.2
9
WARNING:version:Support for Python 3 is experimental.
INFO:core:POX 0.7.0 (gar) is up.
DEBUG:openflow.of_01:Listening on 0.0.0.0:6633
INFO:openflow.of_01:[00-00-00-00-00-01 2] connected
DEBUG:forwarding.12_learning:Connection [00-00-00-00-00-01 2]
INFO:openflow.of_01:[00-00-00-00-00-02 3] connected
DEBUG:forwarding.12_learning:Connection [00-00-00-00-00-02 3]
INFO:openflow.of_01:[00-00-00-00-00-03 4] connected
DEBUG:forwarding.12_learning:Connection [00-00-00-00-00-03 4]
INFO:openflow.of_01:[00-00-00-00-00-04 5] connected
DEBUG:forwarding.12_learning:Connection [00-00-00-00-00-04 5]
INFO:openflow.of_01:[00-00-00-00-00-05 6] connected
DEBUG:forwarding.12_learning:Connection [00-00-00-00-00-05 6]
INFO:openflow.of_01:[00-00-00-00-00-06 7] connected
DEBUG:forwarding.12_learning:Connection [00-00-00-00-00-06 7]
INFO:openflow.of_01:[00-00-00-00-00-07 8] connected
DEBUG:forwarding.12_learning:Connection [00-00-00-00-00-07 8]
```

شکل ۸: راه اندازی کنترل کننده به صورت از راه دور

با اجرای دستور pingall با نتایج شکل ۹ روبرو می شویم. از آنجایی که ۴ قاعده مسدودسازی به صورت دو طرفه داشتیم. با اجرای دستور pingall، ۸ بسته ارسالی رها شده و به مقصد نمی رسند، که در شکل ۹ قابل مشاهده است.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> x h3 h4 h5 h6 h7 h8
h2 -> x h3 x h5 h6 x h8
h3 -> h1 h2 h4 h5 h6 h7 x
h4 -> h1 x h3 h5 h6 h7 h8
h5 -> h1 h2 h3 h4 h6 h7 h8
h6 -> h1 h2 h3 h4 h5 h7 h8
h7 -> h1 x h3 h4 h5 h6 h8
h8 -> h1 h2 x h4 h5 h6 h7
*** Results: 14% dropped (48/56 received)
mininet>
```

شکل ۹: نتیجه اجرای دستور pingall

## ۵ پیاده سازی دیواره آتش با استفاده از کنترل کننده راه دور

در این بخش نیز همانند فصل گذشته، بجای استفاده از کنترل کننده‌ی پیش فرض، قصد داریم که بوسیله POX کنترل کننده مورد نظر خود را بنویسیم. در واقع در این تمرین قصد داریم یک دیواره آتش یا Firewall بنویسیم.

کلمه Firewall در حالت عمرانی به این معناست که در ساختمان‌ها از دیواری به نام Firewall برای جلوگیری از پخش آتش به باقی مکان‌های ساختمان استفاده می‌شود. در معنای شبکه‌ای، این دیواره با اجازه ندادن عبور ترافیک مشخص از خود باعث ایجاد امنیت در شبکه می‌شود.

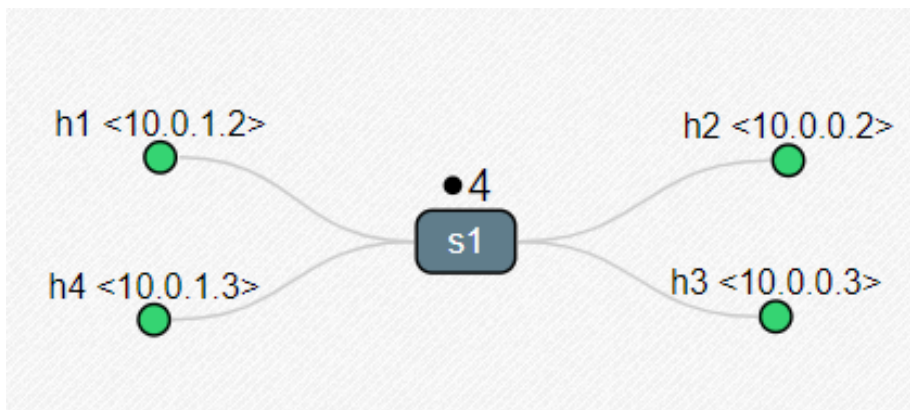
در این قسمت با دو فایل در زبان برنامه‌نویسی پایتون سروکار داریم که در فایل part2.py یک شبکه متشکل از ۴ عدد میزبان داریم که دو به دو در یک زیر شبکه<sup>۱</sup> قرار دارند. همچنین شبکه دارای یک سویچ است که این ۴ میزبان را به یکدیگر متصل کرده است. ساختار این شبکه در شکل ۱۰ نشان داده شده است.

در این شبکه، از کنترل کننده پیش فرض mininet استفاده نمی‌شود و فرض بر این است که کنترل کننده‌ای روی پورت<sup>۲</sup> 6633 در حال اجراست تا شبکه به آن متصل شود.

---

<sup>۱</sup>Subnet

<sup>۲</sup>Port



شکل ۱۰: ساختار شبکه متشکل از ۴ میزبان و یک سوئیچ

در فایل دیگر که `part2Controller.py` نام دارد، قصد داریم Firewall مورد نظر را پیاده سازی کنیم.

برای اجرای این کنترل کننده روی پورت 6633 دستور موجود در الگوریتم ۵ را اجرا می کنیم:

الگوریتم ۵: دستور اجرای کنترل کننده

```
pox/pox.py misc.part2Controller
```

در صورت مواجه شدن با خطای مشغول بودن پورت مورد نظر، دستور موجود در الگوریتم ۶ را اجرا می کنیم تا برنامه ای که پورت مورد نیاز را مشغول کرده شناسایی کرده و آن را متوقف کنیم:

الگوریتم ۶: دستور شناسایی کنترل کننده ها

```
ps -aux | grep controller
```



با اجرای این دستور تمامی کنترل‌کننده‌های در حال اجرا شناسایی شده و با داشتن شناسه<sup>۱</sup> آن برنامه، با اجرای دستور موجود در الگوریتم ۷ آنرا متوقف می‌کنیم:

الگوریتم ۷: دستور خاموش کردن کنترل‌کننده‌ها

```
sudo kill -9 #pid
```

پس از اجرای صحیح کنترل‌کننده، فایل شبکه را اجرا می‌کنیم. باید دقت داشته باشیم که در حالت دسترسی مدیر<sup>۲</sup> این برنامه را اجرا کنیم. این دیواره‌آتش باید ترافیک‌های ICMP<sup>۳</sup> و ARP<sup>۴</sup> را از خود عبور داده و به باقی ترافیک اجازه عبور ندهد و آنرا رها کند.

---

<sup>۱</sup>pid

<sup>۲</sup>admin

<sup>۳</sup>Internet Control Message Protocol

<sup>۴</sup>Address Resolution Protocol

## ۱.۵ پیاده سازی دیواره آتش و مشاهده نتایج

با پیاده سازی دیواره آتش با دستورهای ذکر شده ، آنرا اجرا کرده و بعنوان کنترل کننده شبکه استفاده می کنیم.

با اجرای دستور pingall ، از آنجاییکه *host1* و *host4* در یک زیر شبکه قرار دارند. میتوانند یکدیگر را ping کنند. دو میزبان دیگر نیز به همین ترتیب. و ping باقی میزبان ها به یکدیگر موفقیت آمیز نیست. نتیجه اجرای دستور pingall در شکل ۱۱ مشخص است.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> x x h4
h2 -> x h3 x
h3 -> x h2 x
h4 -> h1 x x
*** Results: 66% dropped (4/12 received)
mininet>
```

شکل ۱۱: نتیجه اجرای دستور pingall

همانطور که پیداست *host1* و *host4* یکدیگر را ping کرده اند. همچنین *host2* و *host3* یکدیگر را ping کرده اند. علاوه بر آن بقیه آن ها رها شده اند. به دلیل اینکه ترافیک ip توسط دیواره آتش مسدود می شود، دستور iperf نیز به درستی اجرا نمی شود و در حالت معلق<sup>۱</sup> ماند و خروجی نشان نمی دهد. نتیجه آن در شکل ۱۲ مشخص است.

---

<sup>۱</sup>hang

```
mininet> iperf
*** Iperf: testing TCP bandwidth between h1 and h4
```

شکل ۱۲: نتیجه اجرای دستور iperf

با اجرای دستور dpctl dump-flows تمامی قاعده‌هایی که توسط دیواره‌آتش تنظیم شده و روی سویچ‌ها اعمال شده است، نمایش داده می‌شود که در شکل ۱۳ پیداست.

```
mininet> dpctl dump-flows
*** s1 -----
cookie=0x0, duration=105.940s, table=0, n_packets=8, n_bytes=784, priority=10,icmp actions=FLOOD
cookie=0x0, duration=105.940s, table=0, n_packets=10, n_bytes=420, priority=9,arp actions=FLOOD
cookie=0x0, duration=105.940s, table=0, n_packets=5, n_bytes=370, priority=0 actions=drop
mininet> |
```

شکل ۱۳: نتیجه اجرای دستور dpctl dump-flows

مشاهده می‌شود که ترافیک ICMP با اولویت 10 از تمامی پورت‌ها ارسال می‌شود یا به عبارت دیگر Flood می‌شود. این اتفاق برای ترافیک ARP با اولویت 9 نیز می‌افتد. همچنین در سطر سوم مشخص است که باقی ترافیک‌ها رها می‌شوند.

## ۶ پیاده سازی دیواره آتش واقعی برای یک سازمان

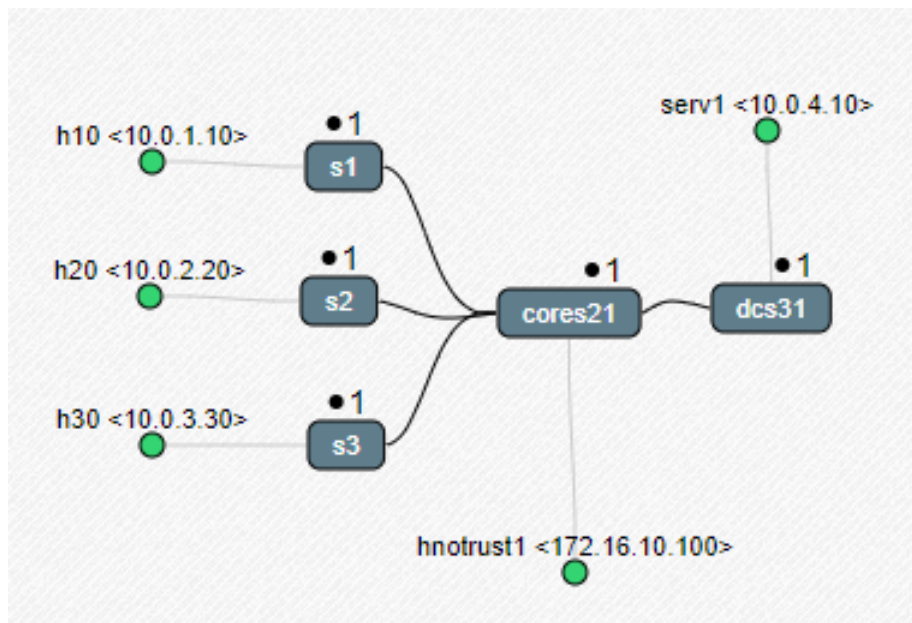
### ۱.۶ معرفی ساختار سازمان و قوانین دیواره آتش

در این بخش قصد داریم که یک شبکه واقعی برای یک شرکت پیاده سازی کنیم. این شرکت دارای سه طبقه و یک مرکز داده<sup>۱</sup> است. هریک از طبقات دارای تعدادی میزبان و یک سویچ است که میزبان ها به آن متصل می شوند. هر سویچ به یک سویچ مرکزی متصل است که طبقات مختلف را به یکدیگر متصل می کند. علاوه بر اتصال طبقات مختلف، داده ی بیرونی نیز از طریق این سویچ به شبکه وارد می شود. همچنین یک سویچ دیگر وجود دارد که مرکز داده به آن متصل شده و به باقی شبکه ملحق می شود. طرح کلی شبکه در شکل ۱۴ مشخص شده است.

در این قسمت قصد داریم که تمامی میزبان ها بتوانند به یکدیگر دسترسی داشته باشند. همچنین برای محافظت از مرکز داده از خطرات احتمالی، اجازه دسترسی از خارج شبکه به مرکز داده، اعطا نمی شود. علاوه بر آن برای جلوگیری از پخش شدن ip های داخلی اجازه عبور ترافیک ip از خارج داده نمی شود.

---

<sup>۱</sup>Data Center



شکل ۱۴: ساختار شبکه موجود در شرکت

## ۲.۶ پیاده‌سازی دیواره آتش سازمان و مشاهده نتایج

با اجرای دستور pingall همه میزبان‌ها می‌توانند یکدیگر را ping کنند. درحالی‌که میزبانی که در خارج از زیرشبکه قرار دارد نمی‌تواند بقیه میزبان‌ها را ping کند. نتیجه اجرای این دستور در شکل ۱۵ مشخص است.

```

mininet> pingall
*** Ping: testing ping reachability
h10 -> h20 h30 x serv1
h20 -> h10 h30 x serv1
h30 -> h10 h20 x serv1
hnotrust1 -> x x x x
serv1 -> h10 h20 h30 x
*** Results: 40% dropped (12/20 received)
  
```

شکل ۱۵: نتیجه اجرای دستور pingall در ساختار شبکه شرکت

با اجرای دستور iperf برای میزبان‌های مختلف مشاهده می‌شود که *hostNoTrust* نمی‌تواند به مرکز داده دسترسی داشته باشد و این دستور در حالت معلق قرار می‌گیرد. اما برای بقیه میزبان‌ها این اتفاق نمی‌افتد. و پاسخ دریافت می‌شود.

```
mininet> iperf hnotrust1 h10
*** Iperf: testing TCP bandwidth between hnotrust1 and h10
*** Results: ['5.54 Gbits/sec', '5.54 Gbits/sec']
mininet> iperf h10 serv1
*** Iperf: testing TCP bandwidth between h10 and serv1
*** Results: ['5.90 Gbits/sec', '5.91 Gbits/sec']
mininet> iperf hnotrust1 serv1
*** Iperf: testing TCP bandwidth between hnotrust1 and serv1
```

شکل ۱۶: نتیجه اجرای دستور iperf برای *hnotrust1*

با اجرای دستور `dpctl dump-flows` تمامی قاعده‌هایی که توسط دیواره‌آتش تنظیم شده و روی سوئیچ‌ها اعمال شده است، نمایش داده می‌شود که در شکل ۱۷ پیداست.

```
mininet> dpctl dump-flows
*** cores21 -----
--
cookie=0x0, duration=1312.951s, table=0, n_packets=8, n_bytes=784, priority=20,icmp,nw_src=172.16.10.100 actions=drop
cookie=0x0, duration=1312.951s, table=0, n_packets=13, n_bytes=962, priority=10,icmp,nw_src=172.16.10.100,nw_dst=10.0.4.10 actions=drop
cookie=0x0, duration=1312.951s, table=0, n_packets=306354, n_bytes=7423036088, priority=5,ip,nw_dst=10.0.1.10 actions=output:"cores21-eth1"
cookie=0x0, duration=1312.950s, table=0, n_packets=6, n_bytes=588, priority=5,ip,nw_dst=10.0.2.20 actions=output:"cores21-eth2"
cookie=0x0, duration=1312.947s, table=0, n_packets=6, n_bytes=588, priority=5,ip,nw_dst=10.0.3.30 actions=output:"cores21-eth3"
cookie=0x0, duration=1312.947s, table=0, n_packets=172342, n_bytes=7583338324, priority=5,ip,nw_dst=10.0.4.10 actions=output:"cores21-eth4"
cookie=0x0, duration=1312.947s, table=0, n_packets=131053, n_bytes=8649970, priority=5,ip actions=output:"cores21-eth5"
cookie=0x0, duration=1312.947s, table=0, n_packets=66, n_bytes=2772, priority=2 actions=FLOOD
cookie=0x0, duration=1312.945s, table=0, n_packets=0, n_bytes=0, priority=1 actions=drop
*** dcs31 -----
cookie=0x0, duration=1312.942s, table=0, n_packets=311875, n_bytes=7592546402, priority=2 actions=FLOOD
cookie=0x0, duration=1312.941s, table=0, n_packets=0, n_bytes=0, priority=1 actions=drop
*** s1 -----
cookie=0x0, duration=1312.966s, table=0, n_packets=609812, n_bytes=15015026860, priority=2 actions=FLOOD
cookie=0x0, duration=1312.965s, table=0, n_packets=0, n_bytes=0, priority=1 actions=drop
*** s2 -----
cookie=0x0, duration=1312.971s, table=0, n_packets=79, n_bytes=4046, priority=2 actions=FLOOD
cookie=0x0, duration=1312.970s, table=0, n_packets=0, n_bytes=0, priority=1 actions=drop
*** s3 -----
cookie=0x0, duration=1312.991s, table=0, n_packets=79, n_bytes=4046, priority=2 actions=FLOOD
cookie=0x0, duration=1312.989s, table=0, n_packets=0, n_bytes=0, priority=1 actions=drop
mininet> |
```

شکل ۱۷: نتیجه اجرای دستور `dpctl dump-flows` در سوئیچ‌ها

مشاهده می‌شود که سوئیچ‌های *dcs31*, *s1*, *s2*, *s3* تمامی بسته‌های ورودی را از تمامی خروجی‌های خود ارسال می‌کنند. اما سوئیچ *cores21* که نقش کلیدی را در این شبکه ایفا می‌کند، با اولویت بسیار بالایی بسته‌هایی

که فرستنده آن‌ها ip آدرس  $\langle 172.16.10.100 \rangle$  داشته باشد، آن‌ها را رها می‌کند. در واقع ترافیک ICMP را از مقصد خارج از زیرشبکه، رها می‌کند. همچنین در سطر دوم تمامی ترافیک ارسالی از خارج از زیرشبکه ( $\langle 172.16.10.100 \rangle$ ) به مقصد مرکز داده ( $\langle 10.0.4.10 \rangle$ ) را نیز رها می‌کند.

در سطرهای بعدی ترافیک ارسالی به سمت  $\langle 10.0.1.10 \rangle$  و  $\langle 10.0.1.20 \rangle$  و  $\langle 10.0.1.30 \rangle$  و  $\langle 10.0.4.10 \rangle$  را به ترتیب از خروجی‌های 1 و 2 و 3 و 4 ارسال می‌کند.

بسته‌های ارسالی به خارج از زیرشبکه از تمامی میزبان‌ها، از خروجی 5 سوییچ ارسال می‌شود. باقی ترافیک نیز از تمامی پورت‌ها ارسال می‌شود.



## ۷ پیاده سازی سناریوی مبتنی بر قانون در دیواره آتش

### ۱.۷ اتصال از راه دور به سرور لینوکسی و باز نمودن پنجره گرافیکی

۱. ابتدا برنامه Xming را نصب می کنیم. برای نصب می توان از این

[لینک](#) استفاده کرد.

۲. پس از نصب برنامه آن را اجرا می نماییم. برای اطمینان از اجرای

برنامه، آیکون این برنامه در TaskBar نمایان می شود.

۳. برای اتصال به سرور لینوکسی از برنامه PuTTY استفاده می کنیم.

- برای اتصال، مشخصات سرور را در HostName وارد می کنیم.

- باید مطمئن شد که اتصال از نوع SSH است.

- به مسیر (Connection > SSH > X11) رفته و X11

Forwarding را فعال می نماییم.

- Open را کلیک نموده و Username و Password سرور را

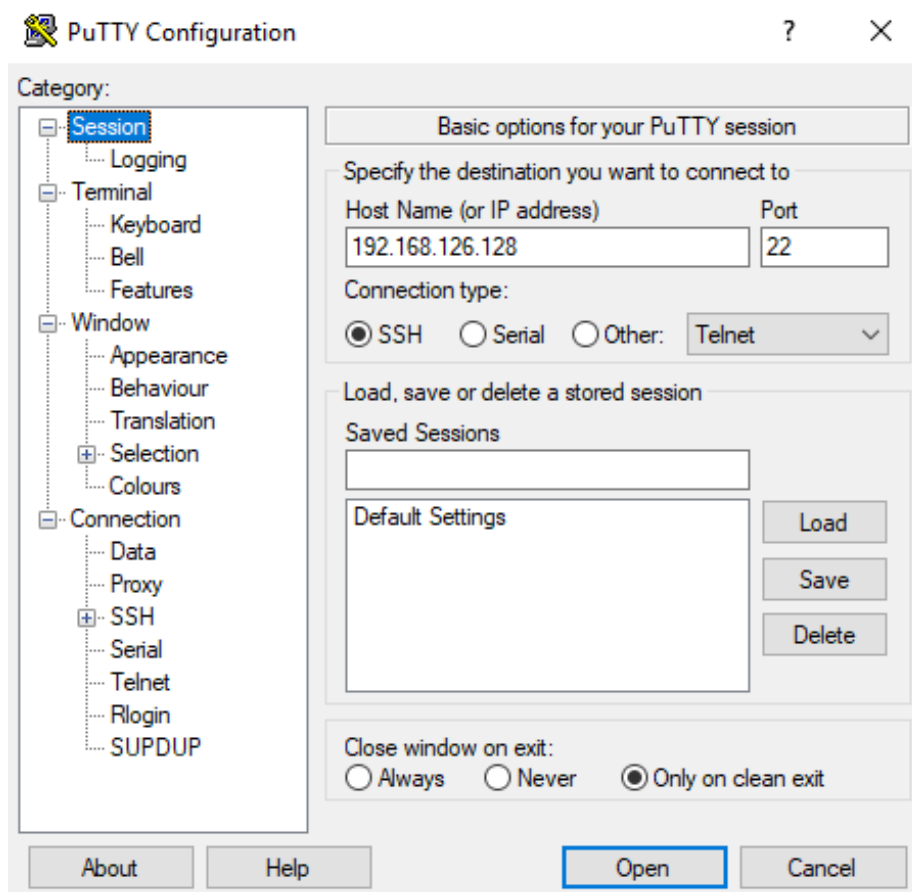
وارد می کنیم. ( مراحل در شکل ۱۸ نشان داده شده است ).

۴. پس از ورود موفق به سرور، با استفاده از دستور [xterm](#) ، یک پنجره

جدید ایجاد می کنیم.

برای این کار در سیستم عامل هایی نظیر Linux, MacOS می توان از این

[لینک](#) استفاده نمود. [۱۱]



شکل ۱۸: استفاده از برنامه PuTTY برای اتصال به سرور

## ۲.۷ ایجاد TCP Connection

برای ایجاد یک TCP Server در یک پورت خاص از دستگاه، از دستور موجود در الگوریتم ۸ استفاده می‌کنیم.

الگوریتم ۸: دستور ایجاد یک سرور TCP

```
iperf -s -p #port-number -i #num
```

که در آن `#port-number` پورتهایی است که قصد داریم سرور TCP را در آن اجرا کنیم. و قصد داریم هر `#num` ثانیه نتیجه را رهیابی کنیم [۱۲]. به طور مثال دستور موجود در الگوریتم ۹ یک TCP Server در پورت ۵۵۶۶ دستگاه اجرا می‌کند.

الگوریتم ۹: دستور ایجاد یک سرور TCP بر روی پورت ۵۵۶۶

```
iperf -s -p 5566 -i 1
```

برای ایجاد یک TCP Client در یک پورت خاص از دستگاه، از دستور موجود در الگوریتم ۱۰ استفاده می‌نماییم. [۱۳]

الگوریتم ۱۰: دستور ایجاد یک TCP Client

```
iperf -B #ip-number:#port-number -c #ip-number -p #port-number -t #times
```

به طور مثال دستور موجود در الگوریتم ۱۱ یک کلاینت TCP به روی پورت 78 دستگاهی با آدرس ، 192.168.10.100 ایجاد کرده که به سرور TCP که به روی پورت 5566 دستگاهی با آدرس ، 192.168.10.200 در حال اجراست درخواست اتصال TCP ارسال می کند. و به مدت ۵ ثانیه این کار را انجام می دهد.

الگوریتم ۱۱: دستور ایجاد یک TCP Client بر روی پورت 78

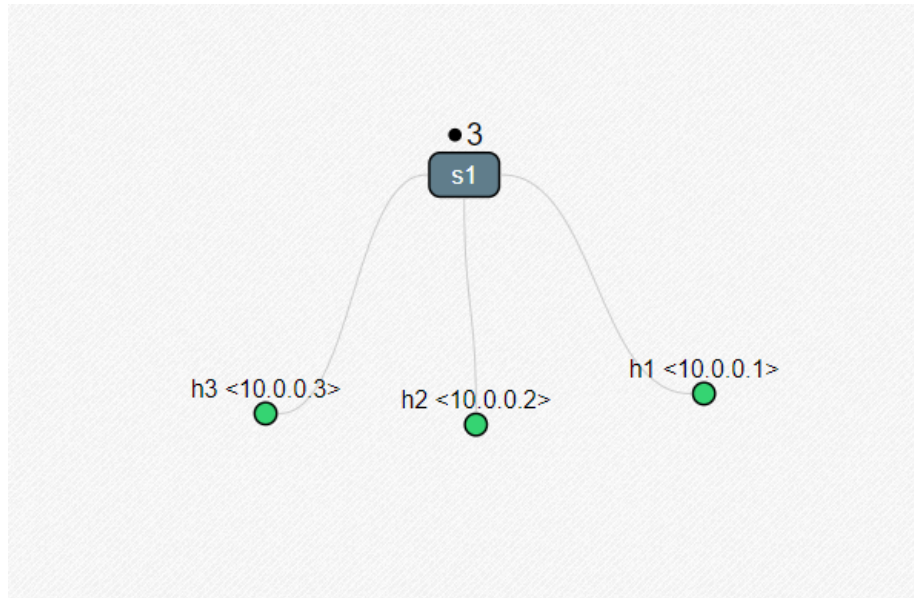
```
iperf -B 192.168.10.100:78 -c 192.168.10.200 -p 5566 -t
```

5

### ۳.۷ معرفی ساختار شبکه برای پیاده سازی سناریو

برای این قسمت ساختار بسیار ساده زیر را داریم. که در شکل ۱۹ مشخص شده است.

در این توپولوژی سه میزبان  $h1<10.0.0.1>$ ,  $h2<10.0.0.2>$ ,  $h3<10.0.0.3>$  را داریم که هر سه به یک سوئیچ متصل اند. بر روی پورت 9999 میزبان  $h1<10.0.0.1>$  یک TCP Server اجرا می نماییم. همچنین بر روی پورت 5566 میزبان  $h2<10.0.0.2>$  نیز این کار را انجام داده و یک TCP Server اجرا می نماییم.



شکل ۱۹: ساختار شبکه دارای ۳ هاست

با استفاده از دستور موجود در الگوریتم ۱۲ دو پنجره جدید ترمینال برای دو میزبان  $h1$  و  $h2$  ایجاد می‌کنیم.

الگوریتم ۱۲: دستور ایجاد پنجره جدید ترمینال

```
xterm h1 h2
```

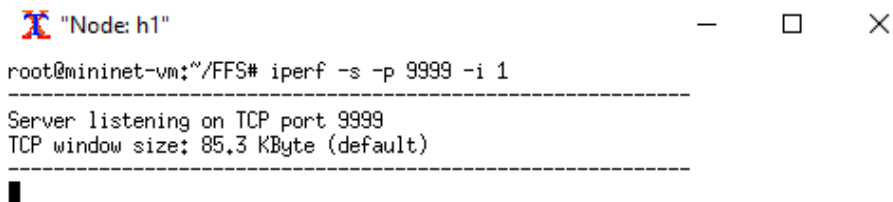
سپس با استفاده از دستور موجود در الگوریتم ۱۳ برای میزبان  $h1$  در شکل ۲۰ و دستور موجود در الگوریتم ۱۴ برای میزبان  $h2$  در شکل ۲۱، سرور TCP را راه‌اندازی می‌کنیم.

الگوریتم ۱۳: دستور ایجاد سرور TCP روی پورت ۹۹۹۹

```
iperf -s -p 9999 -i 1
```

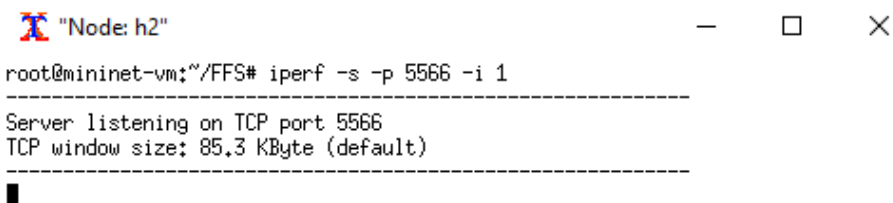
الگوریتم ۱۴: دستور ایجاد سرور TCP روی پورت ۵۵۶۶

```
iperf -s -p 5566 -i 1
```



```
"Node: h1"
root@mininet-vm:~/FFS# iperf -s -p 9999 -i 1
-----
Server listening on TCP port 9999
TCP window size: 85,3 KByte (default)
-----
```

شکل ۲۰: راه‌اندازی TCP Server بر روی میزبان  $h1<10.0.0.1>$



```
"Node: h2"
root@mininet-vm:~/FFS# iperf -s -p 5566 -i 1
-----
Server listening on TCP port 5566
TCP window size: 85,3 KByte (default)
-----
```

شکل ۲۱: راه‌اندازی TCP Server بر روی میزبان  $h2<10.0.0.2>$

#### ۴.۷ معرفی سناریوی مبتنی بر قانون

قصده داریم که از پورت ۸۸ میزبان  $h1<10.0.0.1>$  یک اتصال TCP باز کرده که به TCP Server که روی پورت ۵۵۶۶ میزبان  $h2<10.0.0.2>$  در حال اجراست متصل شود. اما شرایطی وجود دارد که تا زمانی که

از پورت 78 روی میزبان <10.0.0.2>h2 به پورت 9999 روی میزبان <10.0.0.1>h1 اتصال TCP ایجاد نشود، پورت 88 میزبان <10.0.0.1>h1 نمی تواند به پورت 5566 میزبان <10.0.0.2>h2 بسته ای ارسال نماید.

برای پیاده سازی این سناریو، کنترل کننده ی مورد نظر خود را در فایل policyController.py پیاده سازی می کنیم. در این کنترل کننده که با استفاده از رویکرد یادگیری سوئیچ<sup>۱</sup> عمل می کند. بسته های ارسالی را کنترل می کنیم. در این کنترل کننده، با استفاده از تابع \_handle\_PacketIn() بسته هایی که با هیچ کدام از قوانینی که در سوئیچ موجود است مدیریت نمی شوند، به این تابع فرستاده شده تا در مورد آنها تصمیم گیری شود. سیاست<sup>۲</sup> مورد نظر در فایل policy.txt تعریف شده است. محتوای این فایل به صورت الگوریتم ۱۵ است:

الگوریتم ۱۵: فایل تعریف سیاست

```
10.0.0.1 9999 88 10.0.0.2 5566 78
```

<sup>۱</sup> Learning Switch

<sup>۲</sup> Policy

در این فایل، سیاست طوری تعریف شده است که میزبان <10.0.0.1> تا زمانی که بسته‌ی TCP از پورت 78 میزبان <10.0.0.2> به مقصد پورت 9999 خود دریافت نکند نمی‌تواند بسته TCP از پورت 88 خود به پورت 5566 میزبان <10.0.0.2> ارسال نماید.



## ۵.۷ نتایج اعمال سیاست در تصمیم‌گیری سوئیچ

ابتدا دستورهای موجود در الگوریتم ۱۶ را اجرا کرده تا سوئیچ بداند هر یک از میزبان‌ها به کدام شماره پورت متصل است. در واقع در این مرحله سوئیچ یادگیری انجام داده و جدول خود را به‌روزرسانی می‌کند.

الگوریتم ۱۶: دستورهای ایجاد اتصال TCP

```
۱ iperf h1 h3
```

```
۲ iperf h2 h3
```

با اجرای دستور موجود در الگوریتم ۱۷، یک TCP Client روی پورت 88 از میزبان  $h1 < 10.0.0.1 >$  اجرا کرده که به TCP Server که روی پورت 5566 میزبان  $h2 < 10.0.0.2 >$  است، درخواست ارسال می‌کند. نتیجه اجرای این دستور در شکل ۲۲ مشخص است.

الگوریتم ۱۷: دستور ایجاد TCP Client روی پورت ۸۸

```
۱ iperf -B 10.0.0.1:88 -c 10.0.0.2 -p 5566 -t 5
```

```
"Node: h1"
root@mininet-virtual-machine:~# iperf -B 10.0.0.1:88 -c 10.0.0.2 -p 5566 -t 5
connect failed: Operation now in progress
root@mininet-virtual-machine:~#
```

شکل ۲۲: نتیجه راه اندازی TCP Client روی پورت ۸۸ میزبان <10.0.0.1> h1

مشاهده می شود که با اجرای این دستور پیغامی مبنی بر connect failed دریافت می شود که نشان از برقرار نشدن ارتباط TCP میان پورت ۸۸ میزبان <10.0.0.1> h1 و پورت ۵۵۶۶ میزبان <10.0.0.2> h2 است.

همچنین در سمت میزبان <10.0.0.2> h2 که بر پورت ۵۵۶۶ آن TCP Server در حال اجراست نیز پیغامی مبنی بر دریافت بسته از میزبان <10.0.0.1> h1 مشاهده نمی شود. در شکل ۲۳ این قضیه مشخص است.

```
"Node: h2"
root@mininet-virtual-machine:~# iperf -s -p 5566 -i 1
-----
Server listening on TCP port 5566
TCP window size: 85.3 KByte (default)
-----
█
```

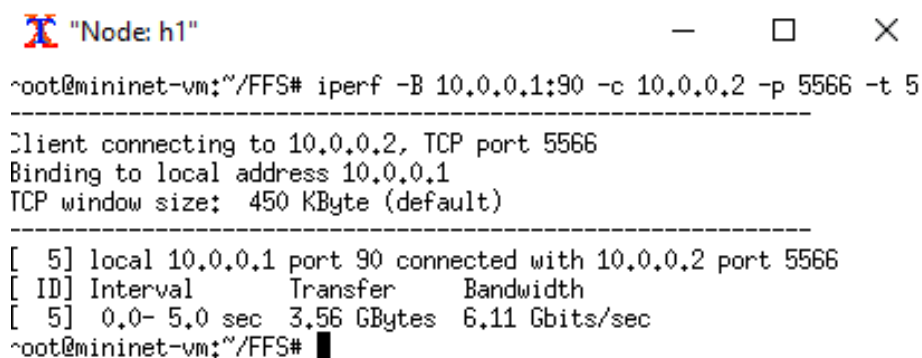
شکل ۲۳: نتیجه راه اندازی TCP Server روی پورت ۵۵۶۶ میزبان <10.0.0.2> h2

برای اطمینان از اینکه تنها پورت 88 از میزبان  $h1<10.0.0.1>$  اجازه ارسال به پورت 5566 از میزبان  $h2<10.0.0.2>$  ندارد، از پورت دیگری از میزبان  $h1<10.0.0.1>$  بسته‌ی TCP به پورت 5566 از میزبان  $h2<10.0.0.2>$  ارسال می‌کنیم. با اجرای دستور موجود در الگوریتم ۱۸ از پورت 90 میزبان  $h1<10.0.0.1>$  اتصال TCP به پورت 5566 میزبان  $h2<10.0.0.2>$  ایجاد می‌کنیم.

الگوریتم ۱۸: دستور ایجاد TCP Client روی پورت ۹۰

```
iperf -B 10.0.0.1:90 -c 10.0.0.2 -p 5566 -t 5
```

در شکل ۲۴ مشاهده می‌شود که نتیجه دستور کاملاً موفقیت‌آمیز بوده و در نتیجه پورت 90 میزبان  $h1$  به پورت 5566 میزبان  $<10.0.0.2>$  متصل شده است.



```
"Node: h1"
root@mininet-vm:~/FFS# iperf -B 10.0.0.1:90 -c 10.0.0.2 -p 5566 -t 5
-----
Client connecting to 10.0.0.2, TCP port 5566
Binding to local address 10.0.0.1
TCP window size: 450 KByte (default)
-----
[ 5] local 10.0.0.1 port 90 connected with 10.0.0.2 port 5566
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0-5.0 sec 3.56 GBytes 6.11 Gbits/sec
root@mininet-vm:~/FFS#
```

شکل ۲۴: نتیجه راه‌اندازی TCP Client روی پورت 90 هاست  $h1<10.0.0.1>$

در شکل ۲۵ مشاهده می‌شود که TCP Server که در پورت 5566 میزبان <10.0.0.2> h2 در حال اجراست، اتصالی از پورت 90 میزبان <10.0.0.1> h1 دریافت کرده و این اتصال موفقیت‌آمیز بوده است.

```
"Node: h2"
root@mininet-vm:~/FFS# iperf -s -p 5566 -i 1
-----
Server listening on TCP port 5566
TCP window size: 85.3 KByte (default)
-----
[ 6] local 10.0.0.2 port 5566 connected with 10.0.0.1 port 90
[ ID] Interval      Transfer    Bandwidth
[ 6] 0.0- 1.0 sec   731 MBytes  6.13 Gbits/sec
[ 6] 1.0- 2.0 sec   797 MBytes  6.69 Gbits/sec
[ 6] 2.0- 3.0 sec   691 MBytes  5.80 Gbits/sec
[ 6] 3.0- 4.0 sec   698 MBytes  5.85 Gbits/sec
[ 6] 4.0- 5.0 sec   726 MBytes  6.09 Gbits/sec
[ 6] 0.0- 5.0 sec   3.56 GBytes 6.10 Gbits/sec
```

شکل ۲۵: نتیجه راه‌اندازی TCP Server روی پورت 5566 هاست <10.0.0.2> h2

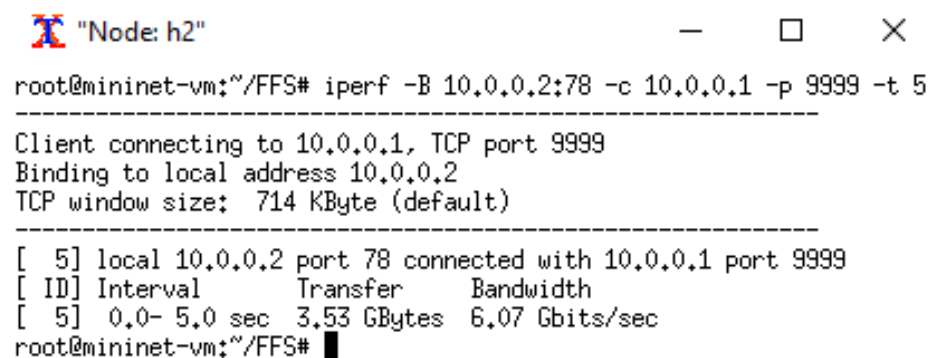
پس نتیجه گرفته می‌شود در حال حاضر تنها پورت 88 از میزبان <10.0.0.1> h1 نمی‌تواند به پورت 5566 میزبان <10.0.0.2> h2 بسته‌ای ارسال نماید.

حال قصد داریم کاری صورت دهیم که از پورت 88 از میزبان  $h1<10.0.0.1>$  توانایی ارسال به پورت 5566 از میزبان  $h2<10.0.0.2>$  را داشته باشیم. در سناریو شرط این کار به این صورت بیان شده است که تنها در صورتی می‌تواند این اتفاق رخ دهد که میزبان  $h2<10.0.0.2>$  از پورت 78 خود به پورت 9999 میزبان  $h1<10.0.0.1>$  ارسال داشته باشد. بنابراین با اجرای دستور موجود در الگوریتم ۱۹ از پورت 78 از میزبان  $h2<10.0.0.2>$  یک TCP Client ایجاد نموده که به پورت 9999 از میزبان  $h1<10.0.0.1>$  که TCP Server روی آن در حال اجراست، بسته TCP ارسال می‌کند.

الگوریتم ۱۹: دستور ایجاد TCP Client روی پورت ۷۸

```
iperf -B 10.0.0.2:78 -c 10.0.0.1 -p 9999 -t 5
```

در شکل ۲۶ مشاهده می‌شود که پورت 78 از میزبان  $h2<10.0.0.2>$  به طور موفقیت‌آمیز به پورت 9999 میزبان  $h1<10.0.0.1>$  متصل شده است.



```
root@mininet-vm:~/FFS# iperf -B 10.0.0.2:78 -c 10.0.0.1 -p 9999 -t 5
-----
Client connecting to 10.0.0.1, TCP port 9999
Binding to local address 10.0.0.2
TCP window size: 714 KByte (default)
-----
[ 5] local 10.0.0.2 port 78 connected with 10.0.0.1 port 9999
[ ID] Interval      Transfer    Bandwidth
[ 5] 0.0- 5.0 sec  3.53 GBytes 6.07 Gbits/sec
root@mininet-vm:~/FFS#
```

شکل ۲۶: نتیجه راه‌اندازی TCP Client روی پورت 78 هاست  $h2<10.0.0.2>$

همچنین در شکل ۲۷ مشاهده می‌شود که در میزبان  $h1<10.0.0.1>$ ، پورت 78 از میزبان  $h2<10.0.0.2>$  به طور موفقیت‌آمیز به پورت 9999 متصل شده است.

```
"Node: h1"
root@mininet-vm:~/FFS# iperf -s -p 9999 -i 1
-----
Server listening on TCP port 9999
TCP window size: 85.3 KByte (default)
-----
[ 6] local 10.0.0.1 port 9999 connected with 10.0.0.2 port 78
[ ID] Interval      Transfer    Bandwidth
[ 6] 0.0- 1.0 sec   680 MBytes  5.71 Gbits/sec
[ 6] 1.0- 2.0 sec   736 MBytes  6.17 Gbits/sec
[ 6] 2.0- 3.0 sec   780 MBytes  6.54 Gbits/sec
[ 6] 3.0- 4.0 sec   776 MBytes  6.51 Gbits/sec
[ 6] 4.0- 5.0 sec   646 MBytes  5.42 Gbits/sec
[ 6] 0.0- 5.0 sec   3.53 GBytes 6.06 Gbits/sec
```

شکل ۲۷: نتیجه راه‌اندازی TCP Server روی پورت 9999 میزبان  $h1<10.0.0.1>$

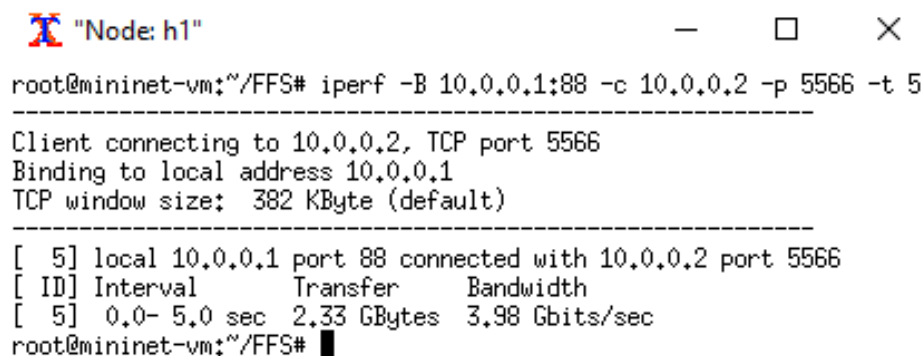
پس از اینکه شرط سناریو را برآورده کردیم، بار دیگر به سراغ پورت 88 از میزبان  $h1<10.0.0.1>$  باز می‌گردیم تا به پورت 5566 میزبان  $h2<10.0.0.2>$  ارسال نماییم.

حال بار دیگر با اجرای دستور موجود در الگوریتم ۲۰ در میزبان  $h1<10.0.0.1>$  بر روی پورت ۸۸ آن یک TCP Client ایجاد نموده که به میزبان  $h2<10.0.0.2>$  که بر روی پورت ۵۵۶۶ آن TCP Server در حال اجراست، درخواست ارسال می کند.

الگوریتم ۲۰: دستور ایجاد TCP Client روی پورت ۸۸

```
iperf -B 10.0.0.1:88 -c 10.0.0.2 -p 5566 -t 5
```

در شکل ۲۸ مشاهده می شود که در سمت کلاینت از پورت ۸۸ میزبان  $h1<10.0.0.1>$  به طور موفقیت آمیز به پورت ۵۵۶۶ میزبان  $h2<10.0.0.2>$  بسته ارسال شده است.



```
root@mininet-vm:~/FFS# iperf -B 10.0.0.1:88 -c 10.0.0.2 -p 5566 -t 5
-----
Client connecting to 10.0.0.2, TCP port 5566
Binding to local address 10.0.0.1
TCP window size: 382 KByte (default)
-----
[ 5] local 10.0.0.1 port 88 connected with 10.0.0.2 port 5566
[ ID] Interval      Transfer    Bandwidth
[ 5]  0.0- 5.0 sec  2.33 GBytes  3.98 Gbits/sec
root@mininet-vm:~/FFS#
```

شکل ۲۸: نتیجه راه اندازی TCP Client روی پورت ۸۸ هاست  $h1<10.0.0.1>$



همچنین در سمت سرور در شکل ۲۹ مشاهده می‌شود که TCP Server  
ایکه روی پورت 5566 از میزبان <10.0.0.2> h2 در حال اجرا بوده  
است، یک اتصال موفقیت‌آمیز از پورت 88 میزبان <10.0.0.1> h1  
دریافت کرده است. این مسئله بیانگر آن است که میزبان <10.0.0.1> h1  
از پورت 88 خود توانایی ارسال به پورت 5566 از میزبان <10.0.0.2> h2  
را داراست.

```
"Node: h2"
root@mininet-vm:~/FFS# iperf -s -p 5566 -i 1
-----
Server listening on TCP port 5566
TCP window size: 85.3 KByte (default)
-----
[ 6] local 10.0.0.2 port 5566 connected with 10.0.0.1 port 88
[ ID] Interval      Transfer    Bandwidth
[ 6] 0.0- 1.0 sec   463 MBytes  3.88 Gbits/sec
[ 6] 1.0- 2.0 sec   430 MBytes  3.61 Gbits/sec
[ 6] 2.0- 3.0 sec   497 MBytes  4.17 Gbits/sec
[ 6] 3.0- 4.0 sec   485 MBytes  4.07 Gbits/sec
[ 6] 4.0- 5.0 sec   498 MBytes  4.18 Gbits/sec
[ 6] 0.0- 5.1 sec   2.33 GBytes  3.94 Gbits/sec
```

شکل ۲۹: نتیجه راه‌اندازی TCP Server روی پورت 5566 هاست <10.0.0.2> h2

در نتیجه پس از برطرف کردن شرط سناریو مشاهده می شود که پورت  
88 از میزبان  $h1 < 10.0.0.1 >$  توانایی ارسال TCP به پورت 5566  
از میزبان  $h2 < 10.0.0.2 >$  را داراست.

## ۸ مراجع

- [۱] V. Moruse and Amrita Manjrekar. Software defined network based firewall technique. *International Journal of Computer Engineering and Technology*, ۴:۵۹۸-۶۰۳, ۲۰۱۳
- [۲] Chalk Talk. *an expedient introduction to mininet*. <https://www.ciena.com/insights/what-is/What-Is-SDN.html> [Accessed: Jan 25 2022].
- [۳] Fahad Nife and Zbigniew Kotulski. Application-aware firewall mechanism for software defined networks. *Journal of Network and Systems Management*, ۲۸-۲۷, ۲۰۲۰
- [۴] David Bombal. *complete, practical sdn and open-flow fundamentals: over 8hrs*. December 18, 2019. <https://downloadfreecourse.com/complete-practical-sdn-and-openflow-fundamentals-over-8hrs-free-download> [Accessed: July 16 2021].
- [۵] Anshuman Chhabra. *an expedient introduction to mininet*. December 31, 2017. <http://www.anshumanc.ml/networks/2017/08/31/mininet/> [Accessed: July 31 2021].
- [۶] Narmox. *mininet topology visualizer*. <http://demo.spear>.

- narmox.com/app/?apiurl=demo/mininet[Accessed: July 11 2021].
- [۷] Connor Craven. *what is openflow? definition and how it relates to sdn*. November 1, 2020. <https://www.sdxcentral.com/networking/sdn/definitions/what-is-openflow/> [Accessed: Jan 26 2022].
- [۸] mininet. *create a learning switch*. <https://github.com/mininet/openflow-tutorial/wiki/Create-a-Learning-Switch>[Accessed: July 11 2021].
- [۹] noxrepo. *pox documentation*. <https://noxrepo.github.io/pox-doc/html/>[Accessed: July 11 2021].
- [۱۰] Anshuman Chhabra. *implementing a layer-2 firewall using pox and mininet*. September 19, 2017. <http://www.anshumanc.ml/networks/2017/09/19/firewall/> [Accessed: September 28 2021].
- [۱۱] Ian Cosden. *instructions to connect to a remote linux server and open a graphical program*. 2018. <https://princetonuniversity.github.io/PUbootcamp/ssh-instructions/> [Accessed: November 9 2021].
- [۱۲] Chih-Heng Ke. *how to user iperf over mininet*. [http://csie.nqu.edu.tw/smallko/sdn/iperf\\_mininet.htm](http://csie.nqu.edu.tw/smallko/sdn/iperf_mininet.htm)[Accessed: November 9 2021].

- [۱۳] Joe D. *how to specify iperf client port*. July 30, 2018. <https://stackoverflow.com/questions/10065379/how-to-specify-iperf-client-port> [Accessed: November 9 2021].

## ۹ پیوست

### پیوست الف)

تمامی برنامه‌های اجرا شده در مخزن گیت‌هاب در دسترس است.