پروژه‌ی نهایی درس شبکه‌های کامپیوتری

حسنا سادات آزرم‌سا - ۸۱۰۱۹۶۴۰۸
شکیبا بلبلیان خواه - ۸۱۰۱۹۶۴۲۶

درباره‌ی پیاده‌سازی

از آن‌جایی که پیاده‌سازی ما با صورت پروژه‌ی اول تا حدی پیش رفته بود، به تکمیل پیاده‌سازی آن پرداختیم. برنامه‌ی پیاده‌سازی شده توانایی مدیریت loop در شبکه را ندارد به همین دلیل دو یال از شبکه که باعث ایجاد دور شده بود را علی‌الحساب حذف کرده‌ایم (در کد مربوطه کامنت شده‌است) تا بسته‌های ICMP ارسال شده توسط mininet با مشکل مواجه نشود. در ادامه نحوه‌ی پیاده‌سازی، دلیل مشکل‌ساز بودن وجود حلقه، نحوه‌ی اجرای برنامه و خروجی آن آمده‌است.

پیاده‌سازی موارد مربوط به mininet

شبیه‌سازی و ایجاد توپولوژی، فرستادن بسته‌ها در شبکه، تغییر دادن پهنای لینک‌ها، اجرا برای پنج مرتبه و هر مرتبه به مدت یک دقیقه در بستر mininet صورت می‌گیرد که در ادامه درباره‌ی هر عمل توضیحی آورده شده‌است.

کلاس MyTopo

این کلاس مسئولیت روتین ایجاد یک شبکه با توپولوژی داده شده در صورت پروژه را بر عهده دارد. توابع addSwitch و addLink و addHost به ترتیب مسئولیت اضافه کردن سوییچ، لینک بین دو گره و اضافه کردن هاست را بر عهده دارند.

توابع manageLinks و changeBandwidth

تابع changeBandwidth که به ازای هر گره موجود در شبکه توسط manageLinks صدا زده می‌شود، وظیفه‌ی تغییر پهنای باند لینک‌های موجود را دارد. به این منظور، intfList لیستی از اینترفیس‌های مربوط به هر گره را برمی‌گرداند که از آن لینک‌هایی که به گره مربوطه متصل بوده به دست می‌آیند و در نهایت توسط تابع config پهنای باند آن‌ها تغییر می‌کند. تابع manageLink پس از هر صد بار فرستادن بسته (هر 100ms یک بار) فراخوانی می‌شود. دقت شود از آنجایی که با فراخوانی تابع config در واقع به نوعی کل link مجددا reset می‌شود و به همین دلیل تعدادی packet loss خواهیم داشت که هر ده ثانیه یکبار با تغییر پهنای باند رخ داده و در ترمینال چاپ می‌شود.

- **توابع run و sendTcpPacket**

تابع run وظیفه‌ی فراخوانی sendTcpPacket را در بازه‌های 100ms به منظور ارسال بسته دارد. هم‌چنین همانطور که ذکر شد، هر ده ثانیه یک‌بار پهنای باند مربوط به لینک‌ها را آپدیت می‌کند. تابع sendTcpPacket از سویی دیگر به ازای هر هاست در شبکه، یک هاست دیگر را به عنوان مقصد انتخاب کرده و یک بسته‌ی TCP با اندازه‌ی 100Kbyte را از مبدا به مقصد انتخابی ارسال می‌کند. این کار توسط اجرای دستور زیر صورت می‌گیرد. دقت شود d- برای تعیین سایز بسته و c- برای تعیین تعداد بسته‌ی TCP ارسالی است.

```
hping3 -c 1 -d 100000 target_ip &
```

نکته: در صورتی که بسته‌ها به صورت sequential ارسال شود، هر بار اجرای sendTcpPacket در حدود ۳ ثانیه طول خواهد کشید که زمان‌بندی برنامه را دچار مشکل خواهد کرد. به همین دلیل با اجرای command ها در background خود mininet (با & موجود در انتهای دستور فوق) و ایجاد thread به ازای هر هاست، ارسال بسته‌ها را به thread های جداگانه و در پس‌زمینه منتقل کردیم تا هم‌زمان صورت گیرند

○ پیاده‌سازی موارد مربوط به Ryu Controller

مدیریت بسته‌های ورودی به هر سوییچ و هاست، هدایت بسته‌ها در شبکه بر اساس الگوریتم dijkstra، به روز رسانی جدول flow مربوط به هر سوییچ و هم‌چنین ذخیره‌ی اطلاعات لازم برای تولید خروجی خواسته شده در بستر Ryu Controller صورت می‌گیرد.

- **کلاس dijkstra_switch**

این کلاس یک کلاس روتین Ryu Controller بوده و متدهای متفاوت آن را جهت مدیریت بسته‌ها و ارسال آن‌ها به سوییچ‌های منتخب و ... پیاده‌سازی می‌کند.

○ تابع handler_switch_enter

الگوریتم دایجکسترا شناسه سوییچ بعدی مسیر را مشخص می‌کند. اما ما نیاز داریم پورت خروجی متصل به آن سوییچ را پیدا کنیم. به همین دلیل لازم است لینک‌های شبکه شناسایی شوند که این شناسایی توسط این تابع صورت می‌گیرد. هنگامی که سوییچ به شبکه اضافه

می‌شود، این تابع فراخوانی شده و لینک‌ها و سوییچ‌های شناسایی‌شده تا آن زمان را ثبت می‌کند.
شناسایی لینک‌ها توسط کنترلر با کمک بسته‌هایی خاص با پروتکل LLDP صورت می‌گیرد که با اضافه کردن گزینه --observe-links به دستور اجرای کنترلر همواره ارسال می‌شوند.(در قسمت نحوه اجرا روش اجرای این دستور نشان داده شده‌است).

- **تابع packet_in_handler**

از آن‌جایی که capture کردن بسته‌ها هنگام کار با mininet اصولا توسط Wireshark انجام می‌گیرد و ما آن‌ها را به صورت مستند می‌خواستیم، این عمل به صورت دستی با نوشتن فایل trace برای بسته‌ها در تابع packet_in_handler صورت گرفت. نکته‌ی قابل ذکر آن است که برای دنبال کردن بسته‌ها نیاز به یک شناسه‌ی یکتا برای هر کدام می‌باشد که این داده را از identification موجود در پروتکل ipv4 هر یک از بسته‌ها به دست آوردیم. لازم به ذکر است از آنجایی که اندازه‌ی هر بسته زیاد می‌باشد، TCP آن‌ها را به صورت تکه تکه ارسال می‌کند به همین دلیل برای هر بسته در فایل trace چندین سطر مربوط به یک بسته (با id یکسان) ارسالی بین دو سوییچ مشاهده می‌شود که مربوط به chunk های آن بسته می‌باشد.

- **توابع find_min_distance و dijkstra**

تابع dijkstra مسئولیت پیاده‌سازی الگوریتم را داشته و به ازای هر مبدا و مقصد ورودی، اگر هر دو در شبکه وجود داشته باشند، مسیر موجود بین آن‌ها را برمی‌گرداند. در این میان از تابع find_min_distance برای محاسبه‌ی کوتاه‌ترین مسیر حین پیاده‌سازی الگوریتم بهره می‌گیرد. خروجی تابع dijkstra مسیر بین سوییچ‌ها را تعیین می‌کند؛ این بدان معنی است که کوتاه‌ترین مسیر از سوییچ متصل به هاست مبدا تا سوییچ متصل به هاست مقصد تعیین خواهد شد و ارسال بسته از سوییچ به هاست مربوطه توسط کنترلر صورت می‌گیرد (که اگر سوییچ مقصد پورت متصل به هاست مقصد را آموخته باشد، آن را به همان پورت ارسال می‌کند و در غیر این‌صورت به همه‌ی پورت‌های خروجی به صورت flood ارسال می‌کند و بعد از دریافت ack از هاست مقصد پورت متصل به آن را فرا می‌گیرد تا در ارسال‌های بعدی استفاده کند.) خروجی تابع dijkstra اگر لیست خالی باشد، بدین معنی است که مقصد خواسته شده در شبکه وجود ندارد(برای بسته‌های ICMP استفاده می‌شود.) اگر طول لیست برگردانده شده، برابر یک باشد، یعنی به سوییچ مقصد رسیده و

نیاز است بسته به هاست مقصد ارسال شود؛ اگر بیشتر از یک باشد، مسیر هدایت بسته از سوییچی که در آن قرار دارد تا سوییچی که قرار است به آن برسد مشخص شده است.

○ پیاده‌سازی مربوط به پردازش داده‌های حاصل از شبیه‌سازی

برای به دست آوردن خروجی‌های خواسته شده، controller اجرا شده، اطلاعاتی از بسته‌های ارسالی به همراه داده‌هایشان (مانند identification مربوط به پروتکل IPV4 هر بسته به منظور تشخیص آن، زمان ورود بسته به یک سوییچ، مسیر محاسبه شده برای آن برای رسیدن به مقصد، هاست مبدا و مقصد) را در فایل packetTrace.tr ذخیره می‌کند. هم‌چنین برای محاسبه‌ی نرخ آپدیت شدن جدول Flow اطلاعات آن را در فایل flowRate.txt می‌نویسد. پس از پایان اجرای شبیه‌سازی، کلاس Processor در فایل Processor.py موظف به باز کردن این فایل‌های داده‌های خام، پردازش آن‌ها و تولید خروجی‌های خواسته شده می‌باشد. خروجی‌ها در قالب دو فایل با فرمت json و هفت نمودار به ازای هفت هاست موجود در شبکه می‌باشد.

● چرا وجود حلقه در شبکه، این نوع پیاده‌سازی را با مشکل مواجه می‌کند؟

با اجرای برنامه‌ی Wireshark و capture کردن بسته‌های موجود در شبکه حین اجرای mininet متوجه شدیم که خود شبیه‌ساز mininet تعدادی بسته‌ی ICMP با مقصدهای نامعتبر در ابتدای اجرا تولید می‌کند و آن را به همه‌ی گره‌های موجود می‌فرستد تا بتواند شبکه را discovery کند. هر سوییچ با دریافت این بسته، چون آدرس مقصد بسته با آدرس خود سوییچ برابر نبوده و از طرفی هیچ یک از گره‌های متصل به خود سوییچ نیز چنین آدرسی ندارند، آن را به بقیه‌ی گره‌های متصل به خود flood می‌کند. در چنین شرایطی اگر دوری در شبکه وجود داشته باشد، بسته‌های ICMP بین گره‌های موجود در حلقه‌های گراف دائما جابه‌جا می‌شوند و باعث شلوغی شبکه شده و در نهایت عملکرد مطلوب حاصل نخواهد شد. به همین دلیل از الگوریتم‌هایی مانند spanning tree بهره گرفته می‌شود تا سوییچ‌ها به نحوی تنظیم شوند که در صورت وجود دور بسته‌ها به سوییچ‌هایی که قبلا بسته‌ی موردنظر را دریافت کرده‌اند، flood نشوند که البته این مورد در این پروژه پیاده‌سازی نشده است.

● فرضیات در نظر گرفته‌شده

- **وارد کردن وزن یال‌ها توسط کاربر**

به صورت کلی در الگوریتم dijkstra وزن یال‌های شبکه متناسب با وارون پهنای باند هر لینک در نظر گرفته می‌شود (در واقع هر چه پهنای باند بیشتر باشد، لینک داده‌ها را سریع‌تر انتقال داده و گزینه‌ی بهتری برای مسیر انتخابی خواهد بود) اما در صورت پروژه بیان شده که برنامه از کاربر وزن یال‌ها را دریافت کند (مستقل از پهنای باند متغیر مربوط به هر لینک) هر چند که این مورد چندان با منطق سازگار نیست، برای پیاده‌سازی آن فرض کرده‌ایم کاربر وزن‌های مورد نظر خود را از طریق یک فایل در اختیار ما قرار می‌دهد. به این منظور، یک فایل config در پوشه‌ی پروژه تعریف شده است که در فرمت json یال‌های موجود در شبکه و وزن آن‌ها را تعریف کرده است. کاربر با تغییر وزن یال‌ها در این فایل به مقدار دلخواه، می‌تواند الگوریتم را اجرا کند. Controller تعریف شده در پروژه، با خواندن این فایل، توپولوژی خود را بر اساس آن می‌سازد و با آن کار می‌کند.

- **استفاده از توپولوژی صورت پروژه‌ی اول**

از آن‌جایی که بعد از آپلود پروژه‌ی جدید، موردی درباره‌ی استفاده از توپولوژی جدید به جای توپولوژی موجود در صورت پروژه اول بیان نشده‌بود، از همان توپولوژی ابتدایی استفاده شده است.

- **حذف دورهای موجود در توپولوژی**

به همان دلیل بیان شده در قسمت قبل، کنترلر پتانسیل هندل کردن دور در شبکه را ندارد. به همین منظور دو یال موجود بین سوییچ‌های 1 و 2، و همین طور یال میان سوییچ‌های 2 و 4 در نظر نگرفته شده‌اند. برای جایگزینی هر یک از یال‌های موجود با دیگری، کافیست یال مربوطه با وزن دلخواه در فایل config اضافه شده و در کد مربوط به توپولوژی uncomment شود.

- **نحوه‌ی محاسبه‌ی نرخ آپدیت شدن جدول flow**

برای محاسبه‌ی این نرخ، فرض کرده‌ایم نرخ متوسط آپدیت شدن جدول flow مدنظر می‌باشد. به همین منظور، زمان هر بار آپدیت شدن جدول flow به همراه سوییچ مربوطه‌ش را در فایل خروجی ذخیره کردیم. پس از آن تعداد هر بار آپدیت شدن مربوط به هر سوییچ را شمرده بر زمان کل اجرا (۵ دقیقه در کل) تقسیم کرده و در فایل خروجی flowDetail.json چاپ خواهیم کرد.

- **نحوه‌ی رسم نمودار مربوط به زمان ارسال بسته‌ها**

از آنجایی که ۴۲ زوج ممکن برای هاست‌های مبدا و مقصد بسته‌های ارسالی وجود دارد (برای مسیرها جهت قائل شده‌ایم)، و از طرفی پنج دقیقه کد مربوطه اجرا می‌شود، اگر می‌خواستیم زمان ارسال تک تک بسته‌ها را در نمودار رسم کنیم، با حجم عظیمی از داده‌ها مواجه بودیم. به همین منظور، به ازای هر ده ثانیه اجرای مربوط به هر زوج هاست مبدا و مقصد، میانگین زمان ارسال بسته محاسبه شده و در نمودار به عنوان خروجی آورده شده است. (دلیل انتخاب بازه‌های ده ثانیه، عوض شدن پهنای باند لینک‌ها در هر ده ثانیه می‌باشد.)

نحوه‌ی اجرای برنامه

همانطور که در صورت پروژه بیان شد، ابتدا لازم است، کنترلر اجرا شده و در یک ترمینال جدا، فایل مربوط به توپولوژی اجرا شود.

نحوه‌ی اجرا کردن فایل Controller.py

```
hosna@hosna-X510UQ:~/Documents/term6/CN/CA-final$ ryu-manager  --observe-links --ofp-tcp-l
isten-port 6635 ./controller.py
loading app ./controller.py
loading app ryu.controller.ofp_handler
loading app ryu.topology.switches
loading app ryu.app.ofctl.service
loading app ryu.controller.ofp_handler
instantiating app ./controller.py of dijkstra_switch
instantiating app ryu.controller.ofp_handler of OFPHandler
instantiating app ryu.topology.switches of Switches
instantiating app ryu.app.ofctl.service of OfctlService
```

نحوه‌ی اجرا کردن فایل networkTopology.py

لازم به ذکر است پیش از اجرای شبیه ساز mininet لازم است اطلاعات قبلی موجود در این شبیه‌ساز پاک شود تا در حین اجرا با مشکل مواجه نشود. برای این کار ابتدا دستور زیر اجرا می‌شود.

```
hosna@hosna-X510UQ:~/Documents/term6/CN/CA-final$ sudo mn -c
```

در گام بعدی دستور زیر اجرا خواهد شد.

```
hosna@hosna-X510UQ:~/Documents/term6/CN/CA-final$ sudo -E python networkTopology
.py
*** Starting network
('round ', 0)
*** Add switches
*** Add hosts
*** Add links
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 h7
*** Adding switches:
s1 s2 s3 s4
*** Adding links:
(1.11Mbit) (1.11Mbit) (h1, s1) (3.86Mbit) (3.86Mbit) (h2, s2) (4.03Mbit) (4.03Mb
it) (h3, s3) (1.40Mbit) (1.40Mbit) (h4, s3) (1.94Mbit) (1.94Mbit) (h5, s4) (3.16
Mbit) (3.16Mbit) (h6, s4) (1.79Mbit) (1.79Mbit) (h7, s4) (3.59Mbit) (3.59Mbit) (
s2, s3) (3.18Mbit) (3.18Mbit) (s3, s1) (3.00Mbit) (3.00Mbit) (s3, s4)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7
*** Starting controller
c0
*** Starting 4 switches
s1 s2 s3 s4 ...(1.11Mbit) (3.18Mbit) (3.86Mbit) (3.59Mbit) (4.03Mbit) (1.40Mbit)
```

- نحوه‌ی اجرا کردن فایل **processor.py**

```
hosna@hosna-X510UQ:~/Documents/term6/CN/CA-final$ python3 Processor.py
```

نمونه خروجی برنامه

در ادامه نمونه‌ای از خروجی برنامه به ازای پنج بار اجرای یک دقیقه‌ای آمده است.

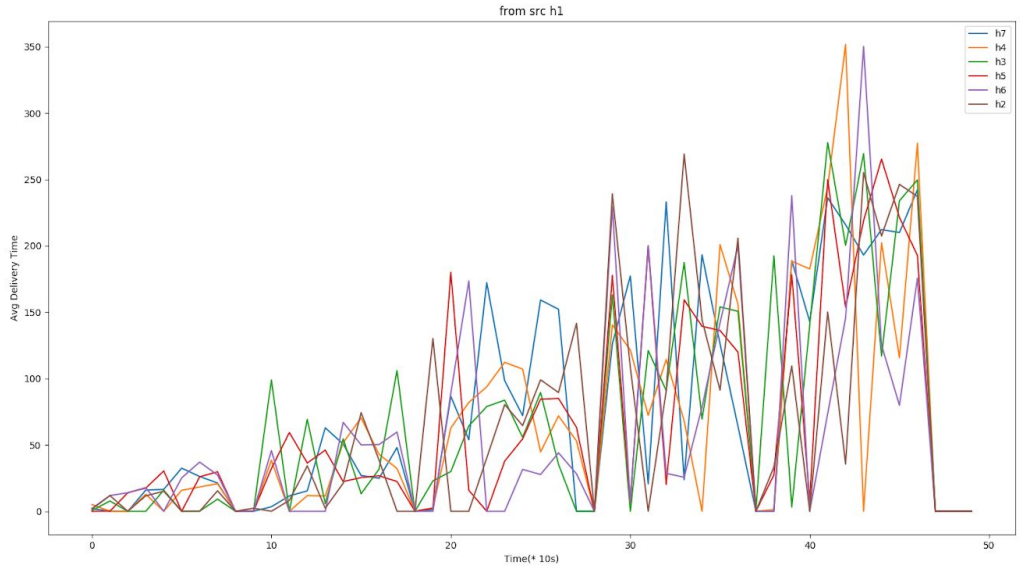
- فایل packetHistory.json


```
{
  "h3 to h1": {
    "106": [
      "[3,1]",
      "353.359"
    ],
    "149": [
      "[3,1]",
      "238.475"
    ],
    "177": [
      "[3,1]",
      "348.414"
    ],
    "190": [
      "[3,1]",
      "348.362"
    ],
    "220": [
      "[3,1]",
      "1.589"
    ],
```

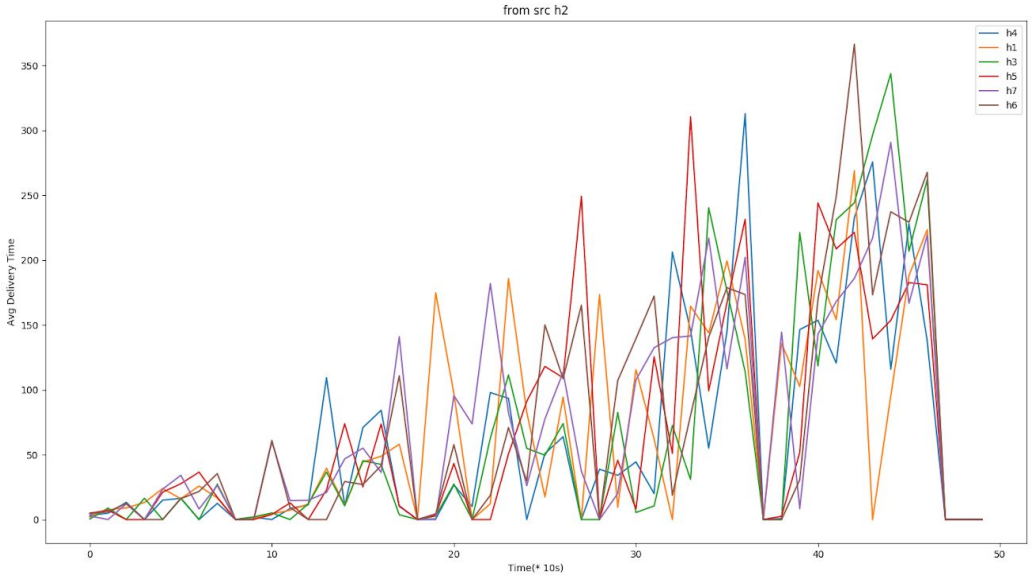

- فایل flowDetail.json


```
{
  "1": 71.96182464741919,
  "4": 238.07899803036258,
  "3": 138.94401265306627,
  "2": 70.10606854420182
}
```

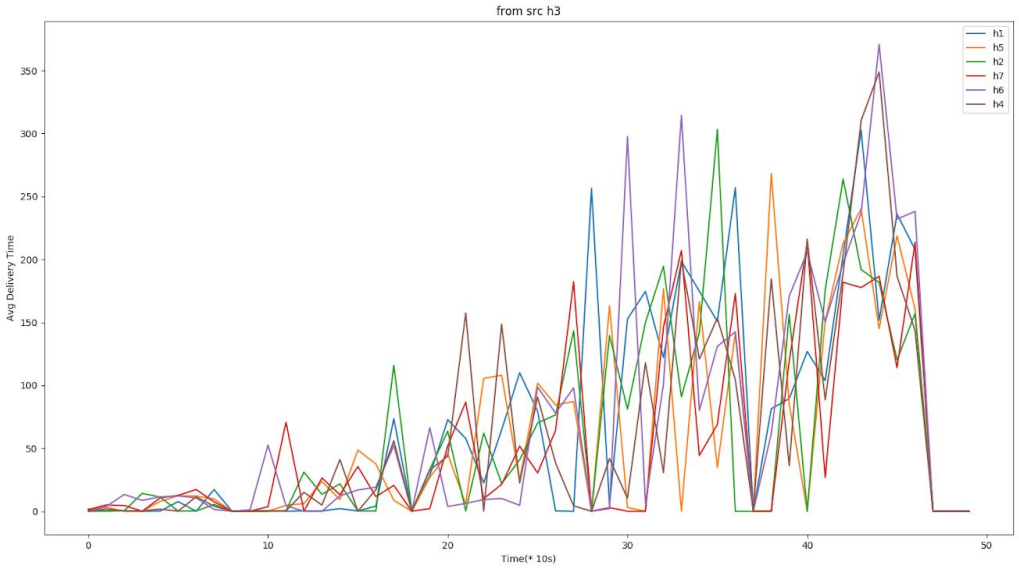
- نمودارهای مربوط به میانگین زمان رسیدن بسته ارسالی به ازای هر host



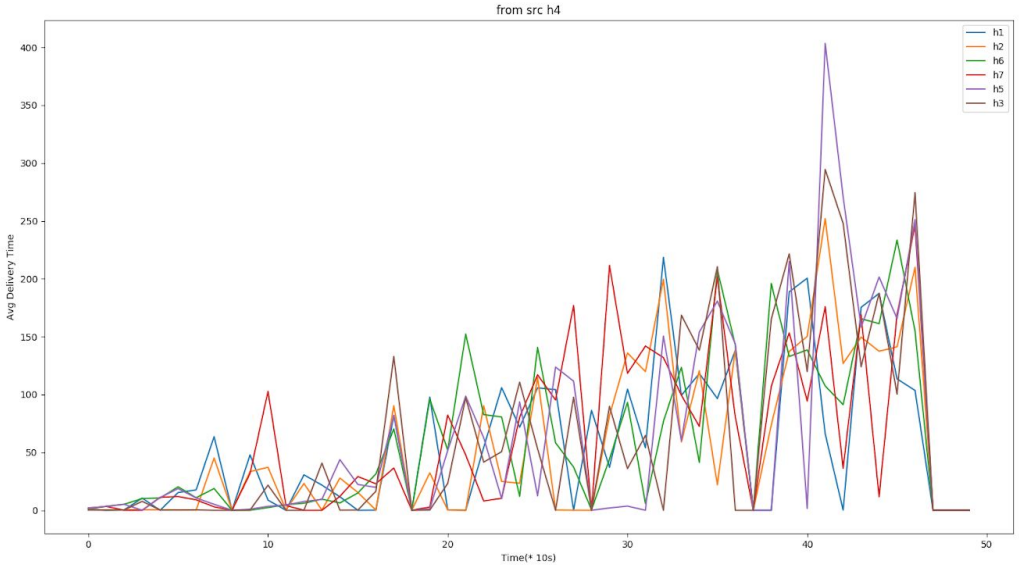
نمودار مربوط به میانگین زمان ارسال بسته از **host1**



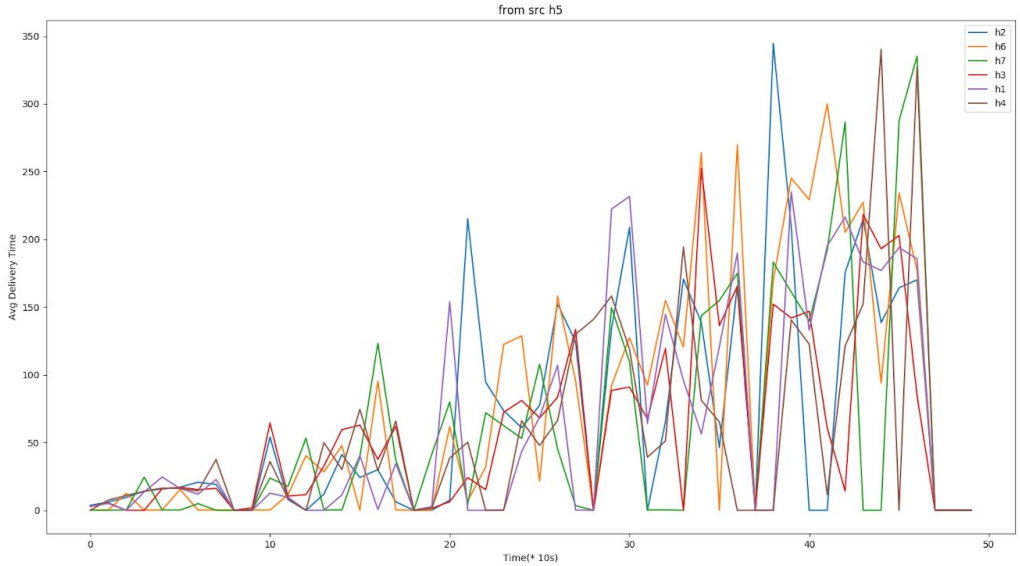
نمودار مربوط به میانگین زمان ارسال بسته از **host2**



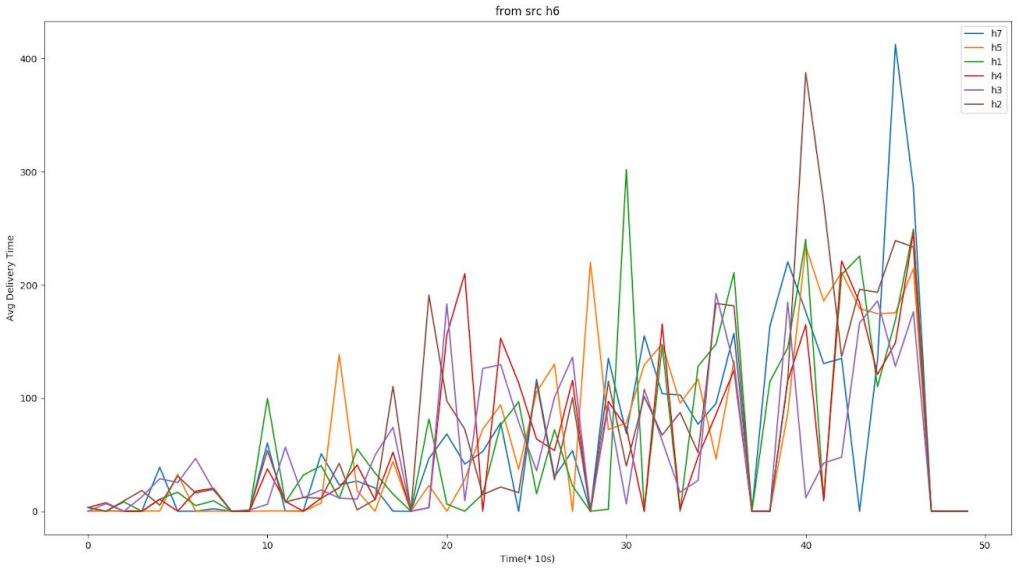
نمودار مربوط به میانگین زمان ارسال بسته از **host3**



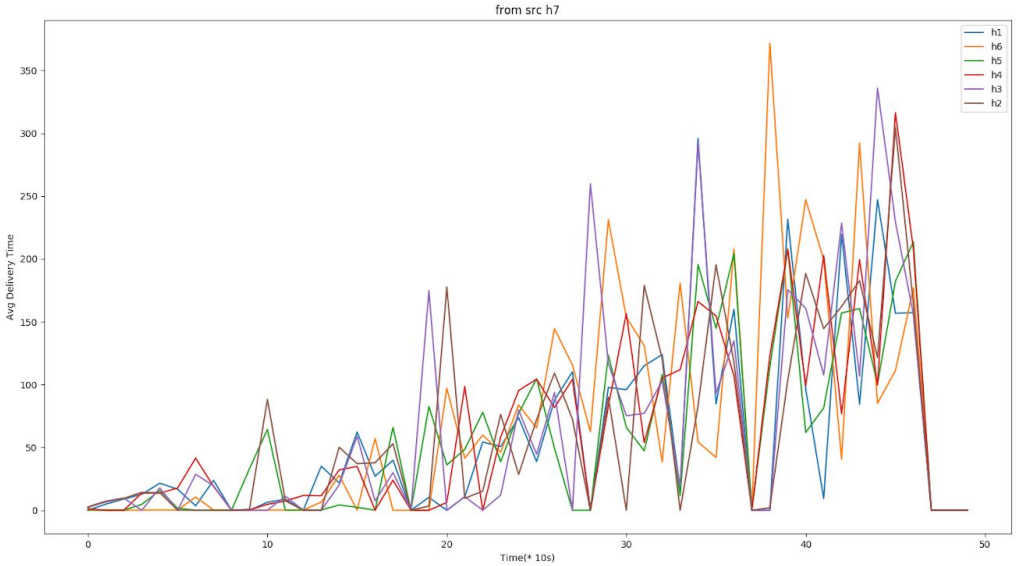
نمودار مربوط به میانگین زمان ارسال بسته از **host4**



نمودار مربوط به میانگین زمان ارسال بسته از **host5**



نمودار مربوط به میانگین زمان ارسال بسته از **host6**



نمودار مربوط به میانگین زمان ارسال بسته از host7