# **BAB I**

# **PENDAHULUAN**

## **Latar Belakang**

Virtualisasi merupakan suatu teknologi yang sedang berkembang di dunia jaringan komputer saat ini. Teknologi virtualisasi ini mensimulasikan *hardware* perangkat keras ke dalam perangkat lunak sehingga seolah-olah kita sedang bekerja pada *hardware* yang sebenarnya. Salah satu teknologi virtualisasi tersebut adalah virtualisasi jaringan.

Saat ini berkembang gagasan baru dalam mengelola jaringan komputer, yang disebut *Software-Define Networking* (SDN). *Software-Define Networking* (SDN) adalah sebuah konsep pendekatan baru untuk mendesain, membangun, dan mengelola jaringan komputer dengan memisahkan *control plane* dan *data plane* (Ummah, 2016).

Konsep dasar dari SDN ini adalah dengan melakukan pemisahan eksplisit antara forwarding plane dan control serta melakukan abstraksi sistem dan mengisolasi kompleksitas yang terdapat pada kompenen antar muka standar (Open Networking Foundation, 2014).

SDN juga mampu memberikan solusi untuk permasalahan-permasalahan jaringan yang ada sekarang ini, seperti sulitnya mengintegrasikan teknologi baru karena masalah perbedaan platform perangkat keras, kinerja yang buruk karena ada beberapa operasi yang berlebihan pada protokol layer dan sulitnya menyediakan layanan-layanan baru. Konsep dari SDN sendiri dapat mempermudah dan mempercepat inovasi pada jaringan sehingga diharapkan muncul ide-ide baru yang lebih baik dan dapat dengan cepat diimplementasikan pada *real network environment* (Tulloh, 2017).

Pada jaringan konvensional yang non-SDN, perangkat keras jaringan (*router* dan *switch*) serta perangkat lunak jaringan (*firmware*) selama ini berada di bawah kendali vendor-vendor perusahaan yang memproduksi perangkat tersebut. *Programmer* tidak bisa mengujicobakan program mereka langsung ke *device* tersebut, tetapi harus menggunakan simulator, dimana kode program simulasi sangat berbeda dengan kode program pada *real* *network environment*. Dengan menggunakan SDN dan *OpenFlow*, maka *firmware* dapat di-*switch* secara *remote* dan diakses dari jarak jauh. *OpenFlow* dapat mengakses dan memanipulasi secara langsung *forwarding* *plane* (*data plane*) dari perangkat-perangkat jaringan tersebut, baik secara fisik maupun virtual (Tulloh, 2017).

Pada saat sekarang ini tidak semua perangkat jaringan mendukung *OpenFlow*. Oleh karena itu untuk melakukan simulasi dengan protokol ini diperlukan simulator SDN yang mendukung protokol *OpenFlow*. Pada penelitian ini simulasi jaringan SDN dilakukan dengan menggunakan *Mininet*, *Pox Controller*, dan sistem operasi yang digunakan adalah *UBUNTU 18.04 LTS*

## **Rumusan Masalah**

Berdasarkan latar belakang masalah maka rumusan masalah pada penelitian ini adalah:

1. Bagaimana cara mengimplementasikan jaringan berbasis *Software Defined Network* dengan menggunakan emulator *Mininet*?
2. Bagaimana kinerja *Pox Controller* dan *Ryu Controller* pada jaringan berbasis *Software Defined Network*?

## **Tujuan Penelitian**

Tujuan akhir yang ingin dicapai dari penelitian ini adalah :

1. Untuk mengetahui implementasi *Sotware Defined Network* dengan menggunakan emulator *mininet*.
2. Untuk mengetahui nilai *Quality of Service* dari implementasi jaringan berbasis *Software Sefined Network* dengan menggukan *Pox* dan *Ryu* sebagai *controller*.

## **Manfaat Penelitian**

Manfaat yang diharapkan dalam penelitian ini adalah :

1. Bagi institusi pendidikan, dapat menjadi bahan pembelajaran mengenai *Software Defined Network*, *Pox Controller*, *Ryu Controller* dan *Mininet*.
2. Bagi mahasiswa, dapat membantu dalam memahami cara kerja *software defined network* dapat menjadi referensi penelitian selanjutnya*.*
3. Bagi peneliti, Memberikan kontribusi kepada para peneliti selanjutnya untuk mengembangkan prototipe berbasis jaringan *software defined network.*

## **Batasan Masalah**

Yang menjadi batasan masalah dalam penelitian ini adalah :

1. Menggunakan emulator *mininet*.
2. Menggunakan topologi jaringan Linear.
3. Menggunakan *Virtual Box* untuk menjalankan *Controller* dan *Mininet*
4. Sistem operasi yang digunakan adalah *Ubuntu 18.04 LTS*
5. Parameter yang diujikan adalah *Delay, Jitter, Throughput, Packet Loss* dan waktu singkronisasi *switch* dengan *controller*.
6. Pada peneliatan ini menggunakan eksternal *controller* dan tidak menggunkan *Controller* bawaan *Mininet*. Eksternal *Controller* yang digunakan adalah *Pox Controller* dan *Ryu Controller*

## **Sistematika Penulisan**

Untuk memberikan gambaran singkat mengenai isi tulisan secara keseluruhan, maka akan diuraikan beberapa tahapan dari penulisan secara sistematis, yaitu :

**BAB I PENDAHULUAN**

Bab ini menguraikan secara umum mengenai hal yang menyangkut latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat, dan sistematika penulisan.

**BAB II TINJAUAN PUSTAKA**

Bab ini berisi teori-teori dasar dan berbagai penelitian terkait dari berbagai literatur mengenai hal-hal yang berhubungan dengan *Software Defined Network*, protokol *OpenFlow*, *Mininet dan Controller.*

**BAB III METODOLOGI PENELITIAN**

Bab ini berisi tentang perencanaan dan penerapan *Software Defined Network* menggunkan emulator mininet serta teknik pengolahan data *Quality of Service*, waktu singkronisasi *switch* dan *controller* oleh *OpenFlow*.

**BAB IV HASIL DAN PEMBAHASAN**

Bab ini berisi tentang analisis kebutuhan. gambaran umum sistem dan hasil pengujian sistem yang telah dibuat.

**BAB V PENUTUP**

Bab ini berisi tentang kesimpulan yang didapatkan serta saran pengembagan dari pengerjaan tugas akhir ini untuk pengembangan lebih lanjut.

# 

# **BAB II**

# **TINJAUAN PUSTAKA**

## **Software Defined Network**

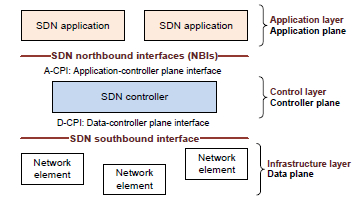
Konsep *Sofware-Defined Network* (SDN), pertama sekali diperkenalkan oleh Martin Casado di Universitas Stanford pada tahun 2007 dengan tulisan pada jurnalnya berjudul “*Ethane: Taking Control of* *the Enterprise*”. Pada jurnal tersebut dikatakan bahwa ethane adalah sebuah arsitektur baru untuk perusahaan, yang mengizinkan manajer mendefinisikan luasan dari sebuah jaringan, kebijakan jaringan dan kemudian menjalankannya secara langsung. Ethane adalah sebuah penyederhanaan yang ekstrim dari ethernet *switch* dengan sebuah kontroler terpusat yang mengatur hak masuk dan aliran routing (Kartadie, 2015).

Adapun tujuan utama SDN adalah menyediakan manajemen kontrol bagi pengguna untuk mengatur *forwarding hardware* dari elemen jaringan. Adapun gagasan dibalik munculnya SDN adalah sentralisasi *control plane,* dan memisahkan *data plane* dari *control plane.* Dengan demikian, perangkat keras jaringan tetap menggunakan *switching fabric (data plane),* tetapi menyerahkan fungsi *intelligence (switching* dan *routing* *information)* ke pada *controller*. Dengan hal ini memungkinkan administrator jaringan unruk mengkonfigurasi elemen jaringan langsung dari *controller*, dimana dengan *control* terpusat ini akan membuat jaringan menjadi sangat fleksibel (Budi, 2016).

SDN adalah sebuah Sebuah arsitektur *network* baru dimana kontrol jaringan dipisahkan dari *forwarding* dan diprogram secara langsung. Gambar menggambarkan bagaimana gambaran *logical* dari SDN arsitektur, yang merupakan jaringan pintar yang secara logikal tersentralisasi berdasarkan *software* (*software-base*). Selain itu ONF menyatakan bahwa dengan SDN tidak lagi membutuhkan protokol standar, tetapi cukup hanya menerima intruksi dari sebuah SDN kontroler *(*Tulloh, 2017).

### **Arsitektur Software Software Defined Network**

Untuk mempromosikan perkembangan SDN, telah dibentuk sebuah organisasi yaitu *Open Networking Foundation* (ONF). Gugus-gugus kerja di ONF telah menghasilkan berbagai rancangan konsep, *framework*, arsitektur, dan standar terkait SDN [Ummah, 2016].

*Open Networking Foundation* (ONF) adalah lembaga non-profit yang mengembangkan, membuat standar, dan membuat komersil dari SDN. Dengan adanya pemisahan antara *control plane* dan *data plane,* SDN menawarkan kontrol jaringan yang lebih baik melalui pemrograman, sehingga dapat meningkatkan konfigurasi, performansi, dan peningkatan arsitektur serta operasi jaringan (Tulloh, 2017).

**Gambar 2.1.** Arsitektur *Software Defined Network* (Open Networking Foundation, 2014).

Arsitektur SDN terdiri dari tiga layer yaitu sebagai berikut, seperti tampak pada Gambar 2.1 (Ummah, 2016) :

1. *Application Plane*: berada pada lapisan teratas, berupa aplikasi yang dapat secara langsung dan eksplisit mendefinisikan *network requirement* dan *network behavior* yang diinginkan. Layer ini berkomunikasi dengan *Control Layer* melalui *NorthBound Interface* (NBI).
2. *Controller Plane*: yaitu entitas kontrol yang memiliki tugas yaitu mentranslasikan *network requirement* yang telah didefinisikan oleh *Application Layer* menjadi instruksi-instruksi yang sesuai untuk *Infrastructure Layer*, dan memberikan *abstract view* yang dibutuhkan bagi *Application Layer* (*abstract view* meliputi informasi statistik dan *event* yang terjadi di jaringan).
3. *Data Plane*: terdiri dari elemen jaringan yang dapat menerima instruksi dari *Control Layer. Interface* antara *Controller Plane* dan *Data Plane* disebut *SouthBound Interface* (SBI), atau *Control-To-Data-Plane Interface* (CDPI).

## **Protokol Openflow**

*Data Plane Layer* dapat berkomunikasi dengan *Control Plane Layer*, membutuhkan protokol yang dapat menjembatanikomunikasi antara kedua layer tersebut. Protokol yangdimaksud adalah *OpenFlow.* Sedangkan yang menjembatanikomunikasi antara *Control Plane Layer* dengan *Application Layer* adalah API (*Application Programming Interface*) (Heryanto, 2017).

*OpenFlow* merupakan protokol yang digunakan untuk mengelola *Southbound Interface* dalam arsitektur SDN. *OpenFlow* adalah standar *interface* pertama yang ditetapkan untukmemfasilitasi interaksi antara *control plane* dan *data plane* dalam arsitektur SDN. *OpenFlow* menyediakan akses berbasis *software* untuk *flow table* dalam menginstruksikan *switch* dan *router* bagaimana mengarahkan lalu lintas *trafick* dalam jaringan. Dengan menggunakan *flow table,* administrator dapat dengan cepat mengubah *layout* jaringan dan lalu lintas jaringan. *OpenFlow* juga menyediakan sebuah *management tool* yang dapat digunakan untuk mengontrolbeberapa fitur, seperti perubahan topologi dan *packet filtering* *(*Budi, 2017).

Untuk bisa menggunakan *OpenFlow* diperlukan *controller* SDN yang mendukung jalannya protokol *OpenFlow*. POX *controller* adalah salah satu *controller* SDN yang mendukung protokol *OpenFlow*. POX adalah *controller* yang berbasis bahasa *Python*, dan dapat dijalankan pada sistem operasi Windows, MacOS dan Linux. POX masih dalam tahap pengembangan, dan merupakan perkembangan dari *controller* terdahulu yaitu NOX (berbasis bahasa C). Dalam jaringan SDN, *controller* harus diaktifkan dahulu sebelum menjalankan jaringan itu sendiri (Ummah, 2016).

Spesifikasi dan standar dari *OpenFLow* ditentukan dan dikontrol oleh *Open Network Foundation* (ONF), yang dipimpin oleh seorang direksi dari 7 perusahaan yang memiliki danmengoperasikan beberapa jaringan terbesar di dunia, yaitu *Verizon, Google, Microsoft, Facebook, Deutsche Telekom, NTT* dan *Yahoo.* Beberapa vendor perangkat keras terkemuka, seperti IBM,HP, dan Cisco telah menawarkan beberapa *switch* dan *router* yang telah mendukung protokol *OpenFLow*. Terdapat dua jenis *OpenFlow* *switch,* yaitu *OpenFLow-Only* dan *OpenFlow- Hybrid*, dimana dalam *OpenFlow-only*, semua paket diolah oleh pipa *OpenFlow*, sedangkan *OpenFlow-hybrid*, dalam *switch* tersebut mendukung operasi *OpenFLow-only* dan juga operasi *switching* Ethernet normal (tradisional L2 dan L3 *switching* dan *routing*) *(*Budi, 2017).

### **2.2.1. Openflow Switch**

*OpenFlow* *switch* merupakan perangkat *forwarding* yang meneruskan paket berdasarkan *flow table. OpenFlow* *switch* mirip dengan perangkat *forwarding* tradisional, tetapi *flow table* tidak dikelola oleh *switch,* melainkan terhubung dengan *controller* melalui *secure channel* antara *switch* dengan *controller* *(*Budi, 2017)*.*

*OpenFlow switch* dibagi dalam dua kategori, yaitu *Commercial Switch* dan *Software Switch. Commercial Switch* adalah *switch* fisik dengan perangkat keras yang telah mendukung *OpenFlow.* Beberapa vendor yang menawarkan *Commercial Switch* yang telah mendukungOpenFlow antara lain Cisco, HP, dan NEX. Sedangkan *Software Switch* adalah perangkat lunakyang telah mendukung OpenFlow dan bisa diinstal dalam sebuah *general purpose hardware*,seperti Raspberry. Adapun beberapa *Software Switch* yang tersedia antara lain *OpenVswitch* (OVS) dan *OpenWRT* *(*Budi, 2017).

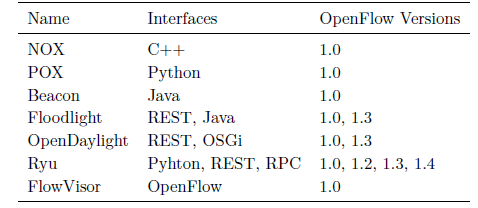
### **2.2.2. Openflow Controller**

*OpenFlow Controller* adalah jenis *Controller Software Defined Network* yang menggunakan *OpenFlow* Protokol. Sebuah *OpenFlow* *Controller* menggunakan protokol *OponFlow* untuk menghubungkan dan mengkonfigurasi perangkat jaringan (*Router, switch, dll*) untuk menentukan jalur terbaik untuk lalu lintas aplikasi (Open Networking Foundation, 2014).

Untuk mengendalikan jaringan yang diprogram *(programmable network)*, diperlukan sebuah perangkat lunak yang memungkinkan untuk mengirim instruksi untuk dikirim ke perangkat jaringan. Perangkat lunak tersebut diimplementasikan dalam sebuah *controller* yang merupakan otak dari jaringan, yang mampu menjalankan beberapa tugas. *OpenFlow controller* adalah aplikasi atau perangkat lunak independen yang berjalan di server khusus, dimana bertanggung jawab untuk mengelola *OpenFlow* *switch* *(*Budi, 2017)*.*

*Controller* bertanggung jawab penuh terhadap apa yang terjadi dalam jaringan. *Controller* dapat menambah, menghapus, atau memperbarui *flow table,* dengan menggunakan protokol *OpenFlow*. *Flow table* merupakan *database* yang menyimpan semua *flow entries* yang terkait dengan *action.,* sehingga *switch* dapat menerapkan tindakan tertentu terhadap *flow* tertentu *(*Budi, 2017).

Berikut adalah jenis – dari *controller* dalam *Software Defined Network* :



**Gambar 2.2** Jenis *Controller* SDN dan Dukungan *OpenFlow*

### **2.2.3. Keuntungan OpenFlow**

Protokol *OpenFlow* memiliki beberapa keuntungan diantaranya sebagai berikut (Putra, 2018):

1. Dapat mengendalikan jaringan secara terpusat pada berbagai vendor Penggunaan *controller* berbasis *software* pada SDN dapat mengendalikan berbagai *network devices* seperti *OpenFlow switch,* *router*. Hal tersebut dapat diterapkan pada berbagai vendor, sehingga dapat mempercepat perkembangan jaringan, konfigurasi dan *update* perangkat secara otomatis.
2. Dapat mengurangi kesalahan secara otomatis. *OpenFlow* berbasis SDN menawarkan jaringan dan manajemen *framework* yang otomatis dan fleksibel sehingga membuat perkembangan *tools* menjadi otomatis. *Tools* yang otomatis akan mengurangi biaya operasional, mengurangi *human error*, dan mendukung kemunculan *IT as a Service*. Dengan adanya SDN, aplikasi berbasis *cloud* dapat mengatur sistem yang cerdas, mengurangi biaya operasional sehingga meningkatkan bisnis yang cerdas.
3. Dapat meningkatkan kehandalan dan keamanan jaringan. SDN menerapkan konfigurasi dan kebijakan tingkat tinggi, yang diterjemahkan melalui *OpenFlow*. *OpenFlow* berbasis arsitektur SDN membutuhkan konfigurasi perangkat jaringan setiap waktu, kebijakan layanan dapat berubah, sehingga dapat mengurangi kesalahan konfigurasi jaringan dan kebijakan yang tidak konsisten.

### **2.2.4. Switch Agent**

Berikut adalah komponen dari *switch agent* (Putra, 2018):

1. *Protocol Openflow*: Ini merupakan bagian *switch* dari protokol *Openflow*.
2. *Core Logic*: Komponen ini mengatur *switch*, menjalankan perintah terhadap *data plane* sesuai dengan kebutuhan, mengatur *offload* dari *data plane*, dan lain sebagainya.
3. *Offload Data plane*: Seringkali *control plane* melakukan *offload* untuk beberapa fungsi pada *OpenFlow* tetapi tidak terdapat pada implementasi *data plane*.
4. *Data plane Protocol*: Protokol *internal* yang digunakan untuk mengatur keadaan *data plane*.

### **2.2.5. Data Plan**

Berikut adalah komponen dari *Data plane* (Flowgramable, 2018):

1. *Port:* Paket masuk dan keluar melalui *switch* pada suatu *port*. Pada setiap versi protokol yang berbeda mendukung tipe *port*, *properties,* dan *configurations*.
2. *Classifiers:* Mempunyai peranan untuk mencocokkan paket untuk *Flow Entry* pada table *Flow*.
3. *Instructions and Actions*: Mengatur bagaimana paket ini diproses waktu *Flow Entry* sesuai dengan yang ada pada tabel *Flow*.

## **Mininet**

Mininet merupakan sebuah emulator jaringan yang dapat menggambarkan jaringan yang besar dengan hanya menggunakan sebuah PC/laptop. Mininet bersifat *open-source*, sehingga proyek yang telah dilakukan berupa *source* *code*, *scripts*, dan dokumentasi yang dapat dikembangkan oleh siapa saja (P. Purnama, 2016). Mininet digunakan untuk mengemulasi *data plan* atau *infrastructure* layer dalam perancangan jaringan SDN (Open Networking Foundation, 2014).

Mininet adalah solusi yang dianggap paling unggul dalam hal kemudahan penggunaan, performansi, akurasi, dan skalabilitas. Ia mampu menyediakan lingkungan yang realistis dan nyaman (*convenience*) dengan harga yang murah (*low cost*). Kita dapat menggunakan alternatif lain seperti hardware *test-bed* untuk simulasi jaringan, yang mana dapat berjalan cukup kencang dan akurat, namun harganya mahal dan harus di-*shared* dengan pengguna lain. Begitu pula, kita dapat menggunakan simulator yang harganya murah, namun seringkali kode simulasi akan harus dimodifikasi lagi bila akan dijalankan di *real network environment*. Mininet menggunakan pendekatan *light-weight virtualization* menggunakan fitur virtualisasi level OS mencakup proses-proses dan *namespace* jaringan, sehingga memungkinkan dilakukannya simulasi jaringan dengan skala sangat besar (hingga ratusan node). Mininet dapat menciptakan jaringan virtual yang realistis, menjalankan *real kernel*, *switch* dan kode aplikasi, pada *single machine* (baik berupa *physical machine*, *virtual machine*, atau *cloud*). Mininet sangat berguna untuk pengembangan riset, pengajaran, serta penelitian (Ummah. ).

Interaksi antara pengguna dengan mininet pada perangkat komputer melalui *Command Line Interface* (CLI) dapat melakukan konfigurasi pada seluruh jaringan yang dirancang dalam satu *console*. Pengembangan jaringan pada mininet secara manual melalui satu *console* akan memakan waktu dan ketelitian yang tinggi, untuk mempermudah pembuatan jaringan dapat digunakan *script* python pada sebuah *file* sehingga pembuatan jaringan akan lebih mudah dan jika kegagalan, penambahan node, penambahan *fiture*, dan lain sebagainya dapat langsung mengedit *file script* tersebut (Effendy).

Mininet sendiri memiliki kelebihan sebagai berikut (P. Purnama, 2016):

1. *Prototype* jaringan yang telah dibuat dengan menggunakan emulator Mininet ini memiliki perilaku jaringan yang sama dengan jaringan yang sebenarnya.
2. Karena bersifat *open-source*, *prototype* yang telah dibuat dengan menggunakan Mininet ini dapat dikembangkan oleh semua orang secara bebas dan bisa juga dikembangkan bersama-sama.
3. Dengan menggunakan Mininet, *prototype* jaringan yang dibuat dapat menggambarkan jaringan dengan jumlah *switch* yang banyak hanya dalam satu perangkat PC/laptop.

## **2.4. Pox Controller**

Untuk bisa menggunakan *OpenFlow* diperlukan *controller* SDN yang mendukung jalannya protokol *OpenFlow*. POX *controller* adalah salah satu *controller* SDN yang mendukung protokol *OpenFlow*. POX adalah *controller* yang berbasis bahasa Python, dan dapat dijalankan pada sistem operasi Windows, MacOS dan Linux. POX masih dalam tahap pengembangan, dan merupakan perkembangan dari *controller* terdahulu yaitu NOX (berbasis bahasa C). Dalam jaringan SDN, *controller* harus diaktifkan dahulu sebelum menjalankan jaringan itu sendiri.

**# ./pox.py samples.pretty\_log forwarding.l2\_learning**

Perintah di atas adalah perintah untuk meng-*invoke* atau mengaktifkan *POX*. **./pox.py** digunakan untuk menjalankan program utama *controller POX*, **semples.pretty\_log** digunakan untuk melihat setiap log dalam perubahan komunikasi. Sedangkan **forwarding.l2\_learning** digunakan untuk mengoperasikan *OpenFlow switch* (Ummah, 2016).

## **2.5. Ryu Controller**

RYU sering disebut sebagai komponen dasar, perangkat lunak *open source* pada kerangka kerja jaringan (*networking framework*). RYU diimplementasikan sepenuhnya dengan *phyton*, dan didukung oleh laboratorium NTT. Seperti *controller* SDN lainnya, *ryu controller* juga menyediakan komponen perangkat lunak dengan API yang tak tersembunyi untuk memungkinkan para pengembang untuk membuat sebuah pengelolaan jaringan baru dan mengontrol aplikasi (Putra, 2018).

## **2.6. Virtual Box**

Teknik Virtualisasi merupakan istilah yang mengacu pada pembuatan suatu versi maya/virtual (daripada versi aktual), termasuk *virtual machine*, yang menyerupai mesin yang dapat menjalankan program layaknya seperti mesin aktual/fisik sesungguhnya .

Oracle VM Virtual Box adalah salah satu aplikasi virtualisasi (*Hypervisor*), dimana dapat di-*install* pada komputer baik *Physical,* baik yang berbasis Intel maupun AMD, tidak membutuhkan fitur *processor* yang dibangun dalam *hardware* baru seperti Intel Vt-x atau AMD-V. Bahkan Oracle VM Virtual Box dapat digunakan pada *hardware/processor* lama yang tidak mendukung *hardware virtulization*.

**Gambar 2.3.** Arsitektur *Virtual Machine*

Oracle VM Virtual Box dapat di-*install* pada Sistem Operasi Windows, Mac, Linux atau Solaris, baik yang 32 bit maupun 64 bit (*Host OS*), dan menjalankan berbagai Sistem Operasi (*Guest OS*) sebanyak yang dikehendaki, berdampingan dengan aplikasi lainnya. Keterbatasan secara praktisnya adalah *Disk Space* dan *Memory* (Gratianus).

## **2.7. Quality of Service**

*Quality of Service* (QoS) merupakan metode pengukuran tentang seberapa baik jaringan dan merupakan suatu usaha untuk mendefinisikan karakteristik dan sifat dari satu servis. QoS digunakan untuk mengukur sekumpulan atribut kinerja yang telah dispesifikasikan dan diasosiasikan dengan suatu servis . Pada penelitian ini menggunakan Model Monitoring QoS pada Gambar 2.3 (Rika Wulandari).

Model Monitoring QoS terdiri dari komponen *monitoring application*, QoS *monitoring*, *monitor*, dan *monitored objects* (Rika Wulandari).

1. *Monintoring Application*

Merupakan sebuah antarmuka bagi administrator jaringan. Komponen ini berfungsi mengambil informasi lalu lintas paket data dari monitor, menganalisanya dan mengirimkan hasil analisis kepada pengguna. Berdasarkan hasil analisis tersebut, seorang administrator jaringan dapat melakukan operasi-operasi yang lain.

2. *Qos Monitoring*

Menyediakan mekanisme monitoring QoS dengan mengambil informasi nilai-nilai parameter QoS dari lalu lintas paket data.

3. *Monitor*

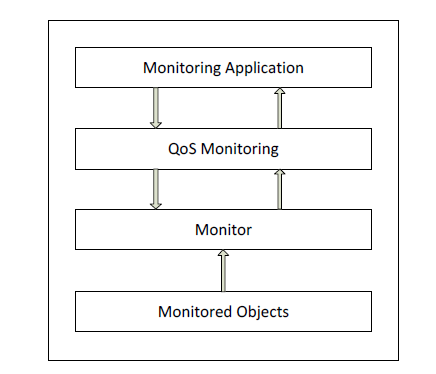
Mengumpulkan dan merekam informasi lalu lintas paket data yang selanjutnya akan dikirimkan kepada monitoring *application*. Monitor melakukan pengukuran aliran paket data secara waktu nyata dan melaporkan hasilnya kepada *monitoring application*.

4. *Monitoring Object*

Merupakan informasi seperti atribut dan aktifitas yang dimonitor di dalam jaringan. Di dalam konteks QoS monitoring, informasi-informasi tersebut merupakan aliran-aliran paket data yang dimonitor secara waktu nyata. Tipe aliran paket data tersebut dapat diketahui dari alamat sumber (*source)* dan tujuan *(destination)* di layer-layer IP, *port* yang dipergunakan misalnya UDP atau TCP, dan parameter di dalam paket RTP.

Menurut informasi QoS yang dapat diperoleh, *monitoring* QoS dapat diklasifikasikan ke dalam dua kategori yaitu *monitoring* QoS dari ujung ke ujung (*end to end* QoS *monitoring* (EtE QM)) dan *monitoring distribusi* QoS per *Node* (*distribution monitoring* (DM)). Di dalam EtE QM, *monitoring* QoS dilakukan dengan cara mengukur parameter-parameter QoS dari pengirim kepada penerima. Sedangkan di dalam DM, proses *monitoring* QoS dilakukan di segmen-segmen jalur pengiriman atau antara *node-node* tertentu yang dikehendaki disepanjang jalur pengiriman paket data dapat dilihat pada Gambar 2.4.

.



**Gambar 2.4.** Model monitoring Qos

**2.8.** **Parameter-parameter QoS****(Quality of Service)**

1. ***Throughput***

Throughput adalah ukuran seberapa banyak unit informasi yang dapat diproses oleh sistem dalam jumlah waktu tertentu. Ini diterapkan secara luas untuk sistem mulai dari berbagai aspek sistem komputer dan jaringan untuk organisasi. Langkah-langkah terkait produktivitas sistem meliputi, kecepatan penyelesaian beberapa beban kerja tertentu, dan waktu respons, jumlah waktu antara satu permintaan pengguna interaktif tunggal dan penerimaan respons (Pakawat Pupatwibul, 2014).

***2. Packet Loss***

*Packet Loss* merupakan suatu parameter yang menggambarkan suatu kondisi yang menunjukkan jumlah total paket yang hilang dapat terjadi karena *collision* dan *congestion* pada jaringan. Indeks dan kategori *Packet Loss* ditunjukkan pada Tabel 2.2 (Rika Wulandari, 2015).

|  |  |  |
| --- | --- | --- |
| Paccket Loss | Packet Loss (%) | Indeks |
| Sangat bagus | 0 | 4 |
| Bagus | 3 | 3 |
| Jelek | 15 | 2 |
| Sangat jelek | 25 | 1 |

**Tabel 2.1.** Tabel indeks *packet loss*

Persamaan perhitungan *Packet Loss :*

*Packet Loss* =

***3. Delay (Latency)***

*Delay* *(Latency)* merupakan waktu yang dibutuhkan data untuk menempuh jarak dari asal ke tujuan. *Delay* dapat dipengaruhi oleh jarak, media fisik, congesti atau juga waktu proses yang lama. Pada Tabel 2.3. diperlihatkan kategori dari *delay* dan besar *delay* (Rika Wulandari, 2015).

|  |  |  |
| --- | --- | --- |
| Delay | Delay (ms) | Indeks |
| Sangat baik | <150 | 4 |
| Baik | 150 ms s/d 300 ms | 3 |
| Jelek | 300 ms s/d 450 ms | 2 |
| Delay | Delay (ms) | Indeks |
| Sangat jelek | >450 | 1 |

**Tabel 2.2.** Indeks *delay*

Persamaan perhitungan *delay* (*Latency*) :

*Delay* =

**4*. Jitter* atau Variasi *Delay***

*Jitter* diakibatkan oleh variasi-variasi dalam panjang antrian, dalam waktu pengolahan data, dan juga dalam waktu penghimpunan ulang paket-paket diakhir perjalanan *jitter*. *Jitter* lazimnya disebut variasi *delay*, berhubungan erat dengan *latency*, yang menunjukkan banyaknya variasi *delay* pada transmisi data di jaringan yang diperlihatkan pada Tabel 2.4 (Rika Wulandari, 2015).

|  |  |  |
| --- | --- | --- |
| Jitter | Jitter | Indeks |
| Sangat baik | 0 ms | 4 |
| Baik | 0 ms s/d 75 ms | 3 |
| Jeek | 75 ms s/d 125 ms | 2 |
| Sangat jelek | 125 ms s/d 225 ms | 1 |

**Tabel 2.3.** *Index jitter*

Persamaan perhitngan *Jitter* :

*Jitter* =

Total variasi *delay* = Delay – (Rata – rata delay)

## **2.9. Topologi Linear**

Topologi jaringan adalah susunan atau pemetaan interkoneksi antara *node*, dari suatu jaringan, baik secara fisik (*riil*) dan logis (*virtual*). Topologi pada Gambar 2.4. menggambarkan metode yang digunakan untuk melakukan pengabelan secara fisik dari suatu jaringan. Topologi fisik jaringan adalah cara yang digunakan untuk menghubungkan *workstation-workstation* di dalam LAN tersebut.

Topologi bus merupakan topologi yang banyak dipergunakan pada masa penggunaan kabel *coaxial* menjamur. Karakteristik topologi ini yaitu satu kabel yang kedua ujungnya ditutup dimana sepanjang kabel terdapat *node-node*, paling *prevevalent* karena sederhana dalam instalasi, *signal* melewati kabel 2 arah dan mungkin terjadi *collision* lihat pada Gambar 2.4 (Rika Wulandari, 2015).

**Gambar 2.5**. Topologi Linear atau topologi Bus

## **2.10*. Iperf***

*Iperf* dikembangkan oleh NLANR / DAST sebagai alternatif *modern* untuk mengukur *bandwidth* TCP dan kinerja UDP secara maksimal*. Iperf* memungkinkan *tuning* berbagai parameter dan karakteristik UDP. *Iperf* melaporkan hasil *bandwidth, delay, jitter* dan *packet loss datagram* disetiap hasil pengukurannya (Iperf.rf, 2018).

*Iperf*adalah aplikasi yang digunakan untuk menghitung *bandwidth* dan kualitas *link* suatu jaringan komputer. *Tool* ini juga bersifat *freeware*. *Tool* ini hanya dapat dijalankan melalui *command prompt* dan tidak memiliki tampilan GUI. Parameter QoS yang dapat diukur melalui *tool* ini adalah *bandwidth, jitter*, dan *packet loss* (Muzawi).

**2.11.** **Perbandingan TCP/IP dengan Software Defined network**

|  |  |  |
| --- | --- | --- |
| TCP/IP Protocol Suite | TCP/IP | SDN |
| FTP, SMTP, Telnet,  HTTP, …… | Application | Application Application |
| TCP, UDP | Transport | Control Layer |
| IP, ARP, ICMP | Internet |
| Network Interface | Network  Access | Physical |

**Tabel 2.4.** Perbandingan TCP/IP dengan SDN

Seperti halnya dengan ISO model, fungsi TCP/IP sendiri terbagi menjadi 4 lapisan. (1) : *Network Access Layer* berisi antarmuka jaringan yang menyediakan akses ke perangkat fisik. (2) : *Internet Layer* membangun jaringan untuk komunikasi jaringan dan oleh karena itu menghubungkan internetworking. (3) : *Transport Layer* menangani komunikasi ujung ke ujung (host-to-host). (4) : *Application layer* menawarkan pengguna dengan antarmuka untuk komunikasi dan memberi jalan bagi aplikasi untuk memiliki akses ke layanan jaringan.

# **BAB III**

# **METODOLOGI PENELITIAN**

## **Tempat dan Waktu Penelitian**

Penelitian ini dilaksanakan di *Laboratorium Cloud Computing and Internet Engineering* Departemen Teknik Informatika Fakultas Teknik Universitas Hasanuddin. Waktu penelitan dimulai sejak disetujuinya proposal penelitian ini pada akhir bulan April 2018 hingga pelaporan hasil penelitian pada Oktober 2018.

## **Instrumen Penelitian**

Instrumen yang digunakan pada penelitian ini adalah:

1. Perangkat Keras
   1. Laptop Asus X550ZE dengan spesifikasi sebagai berikut:
      1. Prosessor AMD FX-7600P
      2. RAM 8GB
2. Perangkat Lunak
   1. Ubuntu 18.04 LTS AMD-64bit
   2. *Mininet*
   3. *Iperf tools*
   4. *Oracle* VM Virtual Box
   5. *Atom Text Editor*
   6. *Pox Controller*
   7. *Ryu Controller*

## **Tahapan Penelitian**

Penelitian ini terdiri dari beberapa tahapan sebagaimana ditunjukkan pada Gambar 3.1.

Studi literatur terkait

Menentukan kebutuhan simulasi

Instalasi tools dan aplikasi simulasi

Konfigurasi tools dan aplikasi simulasi

Penentuan skenario pengujian jaringan

Pengujian *quality of service* jaringan

Pengujian waktu singkronisasi *controller* dengan *switch*

Kesimpulan Penelitian

**Gambar 3.1.** Diagram tahapan penelitian

Secara garis besar, tahapan penelitian yang dilakukan dijelaskan sebagai berikut:

1. Studi Literatur Terkait

Pada bagian studi literatur dibahas mengenai dasar teori dan penelitian-penelitian yang terkait serta relevan terhadap penelitian yang dilakukan oleh penulis.

1. Menentukan kebutuhan simulasi

Sesuai dengan konsep utama *software defined network* yang memisahkan antara *control plane* dan *data plane*, maka *host* untuk menginstall *controller* S*oftware defined network* dipisahkan dengan *host* untuk menginstal *mininet.*

1. Instalasi *tools* dan aplikasi simulasi

Instalasi *mininet*, *pox controller* dan *openvswitch* dilakukan dengan cara *native installation* dari *github*, yaitu dengan cara melakukan *clonning* dari *github*.

1. Konfigurasi *tools* dan aplikasi simulasi

Konfigurasikan *mininet* agar dapat terhubung dengan *controllerSDN* (*Pox Controller dan Ryu controller*). Indeks konfigurasi sudah berjalan dengan baik, yaitu saat *Mininet* menjalankan program/suatu topologi jaringan, pada *controller* seharusnya terdeteksi *MAC Address* dari *switch-switch* yang dijalankan oleh *Mininet.*

1. Penentuan skenario pengujian jaringan

Topologi jaringan yang digunakan dalam simulasi jaringan ini adalah topologi jaringan linear. *Node* dalam jaringan berupa *host*, *switch* dan *controller.*

1. Pengujian *quality of service*

Pada tahap ini parameter yang diuji adalah *delay, jitter, packet loss dan troughput,* waktu singkronisasi *switch* dengan *controller*.

1. Pengujian waktu singkronisasi *controller* dengan *switch*

Pada tahap ini pengujian dilakukan dengan cara menjalankan *pox controller* dan *ryu controller* menjalankan *pox* *controller* dan *ryu controller* bersamaan dengan dijalankannya *script* yang telah dibuat pada *mininet*.

1. Kesimpulan penelitian

Kesimpulan dilakukan setelah seluruh tahap-tahap penelitian selesai dilakukan. Pada tahap ini dijelaskan mengenai kesimpulan yang diperoleh dari hasil perbandingan performansi pada *ryu controller* dan *pox controller* berdasarkan skenario yang digunakan.

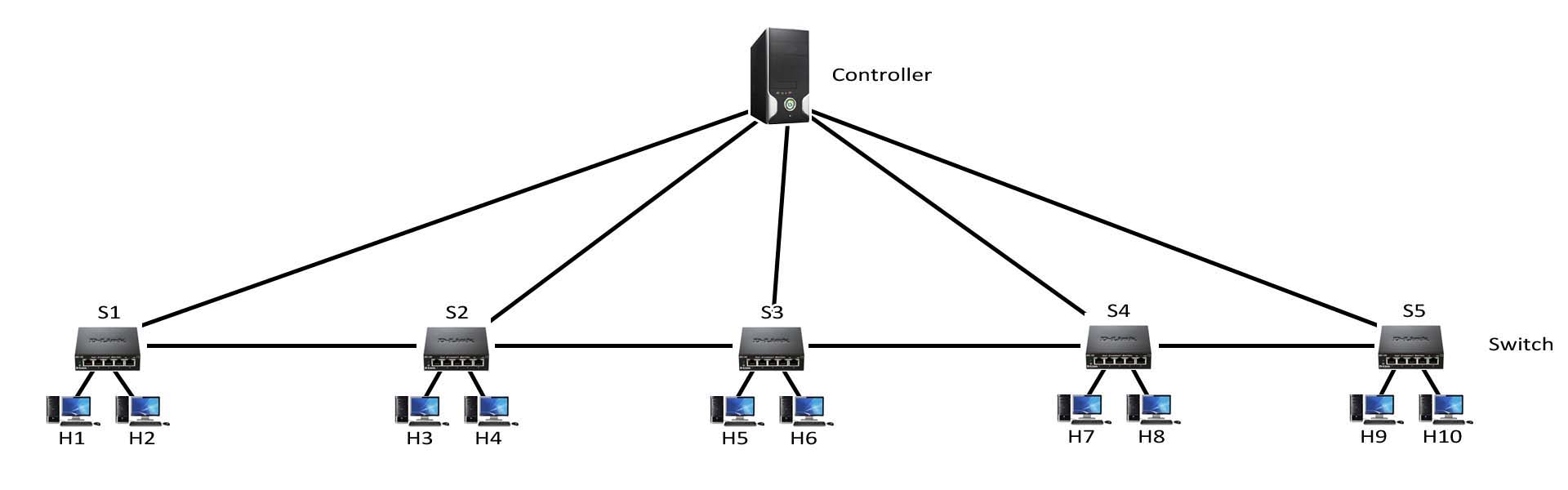
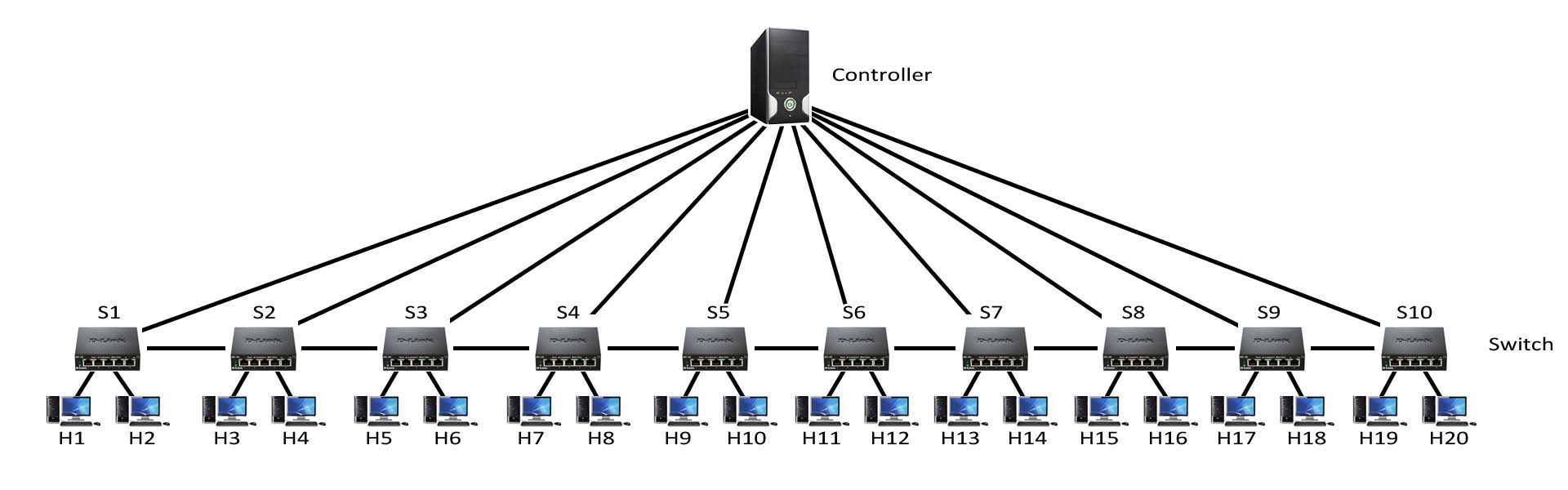
## **Perancangan Sistem**

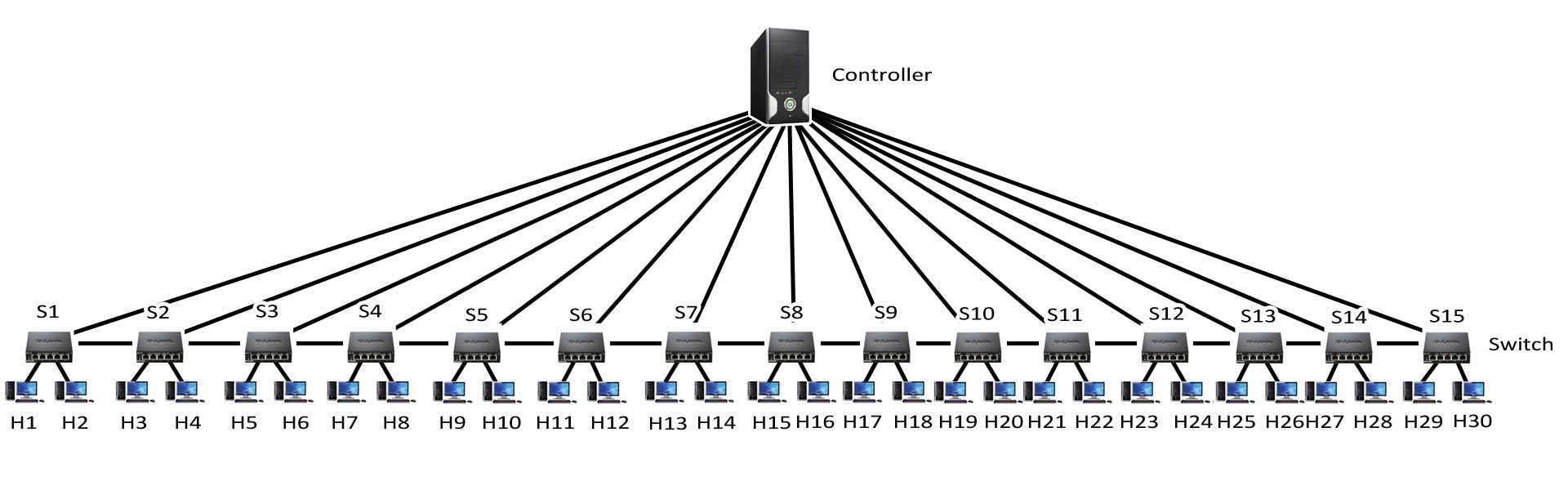
Perancangan sistem dilakukan sesuai desain rancangan yang telah ditentukan. Dengan beberapa hal yang perlu disiapkan untuk menerapkan sistem, antara lain mempersiapkan perangkat keras yang akan digunakan dan juga instalasi perangkat lunak diperlukan.

Implementasi *ryu controller* dan *pox controller* pada rancang jaringan *openflow* yang akan dimulai dengan proses instalasi *controller*. Jika semua *controller* telah berhasil terinstal maka akan dilanjutkan pada tahap pengujian konektivitas *controller* dengan emulator *mininet.*

### **Perancangan Topologi jaringan**

Skenario percobaan yang adalah topologi linear dengan topologi dasar 5 *switch*, 10 *switch* dan 15 *switch*. Setiap *switch* akan terhubung dengan 2 *host* seperti yang ditunjukan pada Gambar 3.2, Gambar 3.3 dan Gambar 3.4. Tujuan dari pengujian *quality of service* ini yaitu agar dapat mengetahui pola perilaku jaringan *Software Defined Network* dan bagaimana tingkat kemampuan *software defined detwork* dalam menangani skala jaringan yang semakin lama semakin besar dengan menggunakan *pox controller* dan *ryu controller*.

**Gambar 3.2.** Topologi jaringan linear 5 *switch*

**Gambar 3.3.** Topologi jaringan linear 10 *switch*

**Gambar 3.4.** Topologi jaringan linear 15 *switch*

Pada seluruh skenario, digunakan IP *address private* dengan *subnet* 10.0.0.0/8 untuk *switch* dan *host*, sedangkan untuk *node* *controller* digunakan *IP private* 127.0.0.1.

### **Instalasi Kebutuhan Simulasi**

[*Mininet*](http://www.zikrillah.com/2014/11/install-mininet-pada-ubuntu.html) merupakan *software open source* yang digunakan untuk melakukan simulasi jaringan, yang sengaja diciptakan untuk memudahkan dalam melakukan *research* dan penjelasan tentang sistem *Software Define Network* (SDN). *Mininet* menciptakan simulasi jaringan yang memamfaatkan *software real* dari komponen jaringan sehingga dapat digunakan secara interaktif untuk melakukan uji coba *software* jaringan.

*Install Ubuntu 18.0*4 pada *virtualBox* dengan spesifikasi ram 4 GB, dan Hard disk 10 GB, lalu *install git* pada ubuntu yang telah terinstall tadi dengan perintah :

#sudo apt-get install git

*Clone mininet* yang terdapat pada *github repository*

#git clone git://github.com/mininet/mininet

*Install mininet* dengan menggunakan perintah

#mininet/util/install.sh -a

Keterangan :

1. install.sh -a : menginstall apapun termasuk *Mininet VM*, depedensi seperti *Open vSwitch*, *wireshark* dan *Pox controller*. Secara *default* setelah penginstalan direktori dari paket tersebut akan berada pada direktori *home*.

Install *Ryu controller* dengan menjalankan perintah :

#mininet/util/install.sh -y

### **Konfigurasi Controller**

Pada tahap ini mengkonfigurasi *Pox controller* dan *Ryu controller* agar dapat terhubung dengan program yang telah dibuat dengan *Atom text editor*. *Pox controller* bisa dijalankan dengan perintah :

# ./pox.py forwarding.l2\_learning

Keterangan :

1. ./pox.py digunakan untuk menjalankan *Pox controller*
2. Forwarding.l2\_learning digunakan untuk mengoperasikan *OpenFlow switch*

*Ryu controller* bisa dijalankan dengan perintah :

# ryu-manager ryu/app/simple\_switch.py

Keterangan :

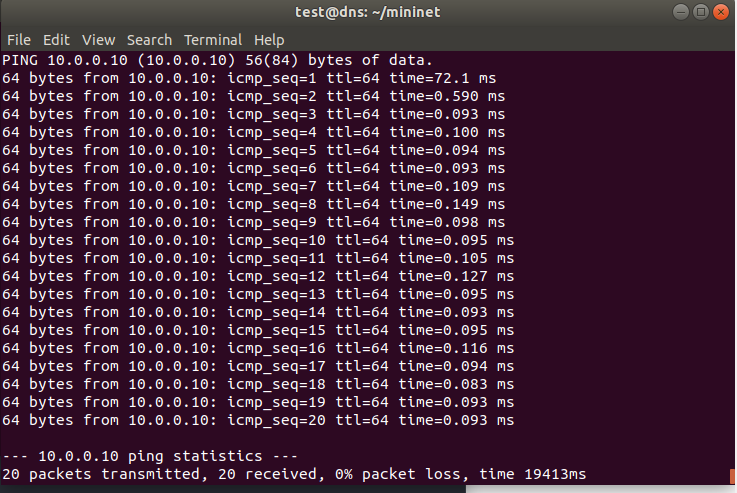
1. Ryu-manager digunakan untuk menjalankan *ryu controller*
2. Ryu/app/simple\_switch.py digunakan mengoperasikan *OpenFlow switch*

Indeks konfigurasi sudah berjalan dengan baik, yaitu saat *Mininet* menjalankan program/suatu topologi jaringan, pada *controller* seharusnya terdeteksi *MAC Address* dari *switch-switch* yang dijalankan oleh *Mininet.*

## **Pengujian dan Analisis**

Pada tahap ini akan dilakukan pengujian dan analisis terhadap performansi *throughput,* *latency, jitter* dan waktu *pembentukan flow tabel* dari masing-masing *controller* yang diuji. Pada penelitan ini terdapat beberapa skenario untuk menguji performansi c*ontroller* dan menganalisis hasil yang diperoleh dari simulasi dengan parameter yang meliputi rata-rata *throughput,* *latency, jitter, packet loss* dan waktu pembentukan *flow tabel*. Pengujian terdiri dari beberapa skenario dengan beban 5, 10, dan 15 *switch*. Masing - masing *switch* nantinya akan dihubungkan dengan 2 *host.*

### **Pengujian Latency dan jitter dengan switch bervariasi**

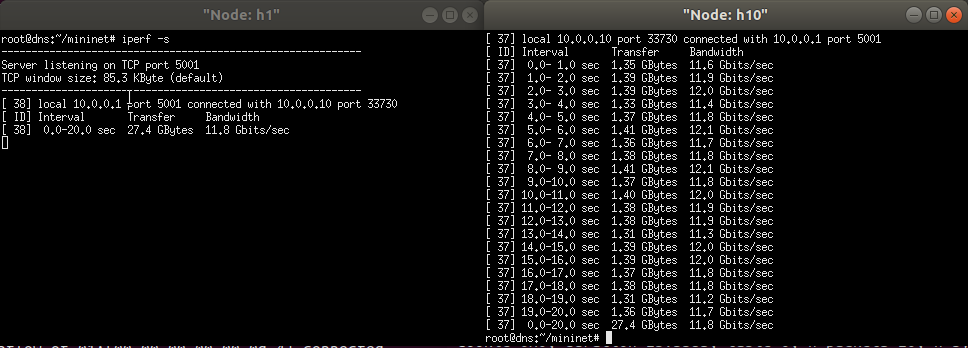
Pengujian *latency dan jitter* dilakukan untuk mengetahui banyak waktu yang dibutuhkan *controller* untuk memberikan respon dalam setiap detiknya. Dalam pengujian untuk mencari nilai *latency* pada setiap *controller* juga akan

**Gambar 3.5.** pengujian menggunakan *protocol icmp*

dilakukan sebanyak duapuluh detik dengan jumlah *switch* bervariasi pada setiap tahap percobaan yang akan dilakukan.

Pengujian ini dilakukan dengan menggunakan Internet Control Message Protocol (ICMP). ICMP adalah [protokol jaringan](https://dosenit.com/jaringan-komputer/teknologi-jaringan/protokol-jaringan) internet yang berfungsi untuk memberikan kiriman pesan-pesan ke dalam sebuah jaringan, mulai dari mengirimkan *pesan error*, pesan diterima, hubungan putus atau *connection lost*, dan sebagainya. Dengan adanya ICMP ini, maka jaringan akan mengetahui respon – respon yang terjadi salami *konektivitas* di dalam jaringan itu berlangsung.

### **Pengujian Throughput dan Packet loss dengan Switch Bervariasi**

Pengujian *throughput* dan *packet loss* ini dilakukan dengan menggunakan *iperf tools.* Dalam pengujian untuk mencari nilai *latency* pada setiap *controller* juga akan dilakukan sebanyak duapuluh detik dengan jumlah *switch* bervariasi pada setiap tahap percobaan yang akan dilakukan.

**Gambar 3.6.** pengujian dengan *iperf tolls*

Pada *host* yang bertindak sebagai *server* dijalankan dengan perintah :

Iperf -s

Pada *host* yang bertindak sebagai *client* dijalankan dengan dengan perintah:

Iperf -c 10.0.0.1 -t 20 -i 1

Keterngan :

1. Iperf -s : perintah yang digunakan untuk menjadikan *host* atau *node* menjadi *server*
2. Iperf -c : perintah yang digunakan untuk menjadikan *host* atau *node* sebagai *client*
3. 10.0.0.1 : merupakan *ip server*
4. -t : waktu selama 20 detik
5. -i 1 : merupakan *interval* setiap 1 detik

# **BAB IV**

# **HASIL DAN PEMBAHASAN**

## **Hasil Penelitian**

### **Hasil Pengujian Pox Controller**

Pada tahap pengujian ini akan dilakukan pengujian pada pox *controller*. Untuk pengujian performansi akan digunakan lima parameter, yaitu *Throughput,* *latency, jitter, packet loss* dan waktu pembentukan *flow table.*

#### Hasil Pengujian Skenario 5 Switch

Pada skenario ini terdapat lima *switch* dengan masing – masih *switch* terhubung dengan dua *host*. *Switch* dan *host* dihubungkan menggunakan kabel *ethernet* sehingga membentuk topologi linear, selanjutnya *pox controller* dijalankan dengan perintah ./pox.py forwarding.l2\_learning. Selanjutnya *pox controller* akan membaca *mac address* setiap *switch* yang terhubung.

Pengujian dilakukan dengan dua cara yaitu dengan menggunakan protokol *ICMP* dan *iPerf tools*. Pengujian menggunakan protokol ICMP dilakukan dengan cara *node* h1 melakukan *request* kepada *node* h10 sebanyak 20 detik dengan mengetikan perintah h1 *ping* -c 20 h10 di *Command Line Interface (CLI)* pada *mininet*.

**Tabel 4.1.** Data hasil ICMP *Pox controller*

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 1 | 64 | 72.1 |
| 2 | 64 | 0.590 |
| 3 | 64 | 0.093 |
| 4 | 64 | 0.100 |
| 5 | 64 | 0.094 |
| 6 | 64 | 0.093 |
| 7 | 64 | 0.109 |
| 8 | 64 | 0.149 |
| 9 | 64 | 0.098 |
| 10 | 64 | 0.095 |
| 11 | 64 | 0.105 |
| 12 | 64 | 0.127 |
| 13 | 64 | 0.095 |
| 14 | 64 | 0.093 |
| 15 | 64 | 0.095 |
| 16 | 64 | 0.116 |
| 17 | 64 | 0.094 |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 18 | 64 | 0.083 |
| 19 | 64 | 0.093 |
| 20 | 64 | 0.093 |
| Jumlah | 1280 | 76.198 |

Selanjutnya data ICMP digunakan untuk menghitung nilai *delay* dan *jitter* pada skenario lima *switch* dengan persamaan :

*Delay* = *= 0.059 ms*

*Jitter* = = 0.002 ms

Total Variasi *Delay* = Rata - rata delay - Delay = 0.059 – 3.809 = 3.75 ms

Rata-Rata *Delay* = = = 3.809 ms

Dari hasil perhitungan diperoleh nilai *delay* sebesar 0,059 ms dan nilai *jitter* sebesar 0.002 ms.

Selanjutnya dilakukan pengujian dengan *iPerf Tools*, *iPerf* dijalankan dengan cara *node* h1 dijadikan sebagai server dan *node* h10 dijadikan sebagai *client* yang melakukan *download* *file* dari *server* selama 20 detik. Pada terminal *node* h1 dijalankan perintah iperf -s dan pada *node* h10 dijalankan perintah iperf -c 10.0.0.1 -t 20 -i 1. Pengujian dengan *iPerf Tools* ini bertujuan untuk mendapatkan nilai *throughput* dan *packet loss.*

**Tabel 4.2.** Data iPerf 5 switch

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 0-1 | 1.35 | 11.6 |
| 1-2 | 1.39 | 11.9 |
| 2-3 | 1.39 | 12.0 |
| 3-4 | 1.33 | 11.4 |
| 4-5 | 1.37 | 11.8 |
| 5-6 | 1.41 | 12.1 |
| 6-7 | 1.36 | 11.7 |
| 7-8 | 1.38 | 11.8 |
| 8-9 | 1.41 | 12.1 |
| 9-10 | 1.37 | 11.8 |
| 10-11 | 1.40 | 12.0 |
| 11-12 | 1.38 | 11.9 |
| 12-13 | 1.38 | 11.9 |
| 13-14 | 1.31 | 11.3 |
| 14-15 | 1.39 | 12.0 |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 15-16 | 1.39 | 12.0 |
| 16-17 | 1.37 | 11.8 |
| 17-18 | 1.38 | 11.8 |
| 18-19 | 1.31 | 11.2 |
| 19-20 | 1.36 | 11.7 |
| Jumlah | 27.4 |

*Packet Loss* = x100%

= 0 %

Dari hasil perhitungan diperoleh nilai *throughput* sebesar 1.37 Gbyte dari percobaan selama 20 detik, sedangkan nilai *packet loss* diperoleh 0%.

#### Skenario 10 Switch

Pada skenario ini terdapat sepuluh *switch* dengan masing – masih *switch* terhubung dengan dua *host*. *Switch* dan *host* dihubungkan menggunakan kabel ethernet sehingga membentuk topologi linear, selanjutnya *Pox controller* dijalankan dengan perintah ./pox.py forwarding.l2\_learning. Selanjutnya *pox controller* akan membaca *mac address* setiap *switch* yang terhubung.

Pengujian dilakukan dengan dua cara yaitu dengan menggunakan protokol *ICMP* dan *iPerf tools*. Pengujian menggunakan protokol ICMP dilakukan dengan cara *node* h1 melakukan request kepada *node* h20 sebanyak 20 detik dengan mengetikan perintah h1 *ping* -c 20 h20 di *Command Line Interface (CLI)* pada *mininet*.

**Tabel 4.3.** Data hasil *ICMP* 10 *switch pox Controller*

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 1 | 64 | 95.9 |
| 2 | 64 | 1.10 |
| 3 | 64 | 0.118 |
| 4 | 64 | 0.119 |
| 5 | 64 | 0.121 |
| 6 | 64 | 0.133 |
| 7 | 64 | 0.122 |
| 8 | 64 | 0.127 |
| 9 | 64 | 0.124 |
| 10 | 64 | 0.125 |
| 11 | 64 | 0.141 |
| 12 | 64 | 0.131 |
| 13 | 64 | 0.121 |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 14 | 64 | 0.124 |
| 15 | 64 | 0.123 |
| 16 | 64 | 0.123 |
| 17 | 64 | 0.119 |
| 18 | 64 | 0.109 |
| 19 | 64 | 0.119 |
| 20 | 64 | 0.302 |
| Jumlah | 1280 | 99.401 |

Selanjutnya data ICMP digunakan untuk menghitung nilai *delay* dan *jitter* pada skenario sepuluh *switch* dengan persamaan :

*Delay* = *= 0.077 ms*

*Jitter* = = 0.003 ms

Total variasi *Delay* = Rata - rata delay - delay = 4.970 – 0.077 = 4.893 ms

Rata-Rata *Delay* = = = 4.970 ms

Dari hasil perhitungan diperoleh nilai *delay* sebesar 0.077 ms dan nilai *jitter* sebesar 0.003 ms.

Selanjutnya dilakukan pengujian dengan *iPerf Tools*, *iPerf* dijalankan dengan cara *node* h1 dijadikan sebagai *server* dan *node* h20 dijadikan sebagai *client* yang melakukan *download* *file* dari *server* selama 20 detik. Pada terminal *node* h1 dijalankan perintah iperf -s dan pada *node* h20 dijalankan perintah iperf -c 10.0.0.1 -t 20 -i 1. Pengujian dengan *iPerf Tools* ini bertujuan untuk mendapatkan nilai *throughput* dan *packet loss.*

**Tabel 4.4** Data iPerf 10 *switch Pox Controller*

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 0-1 | 1.15 | 9.89 |
| 1-2 | 1.19 | 10.2 |
| 2-3 | 1.15 | 9.84 |
| 3-4 | 1.19 | 10.3 |
| 4-5 | 1.19 | 10.2 |
| 5-6 | 1.19 | 10.3 |
| 6-7 | 1.19 | 10.3 |
| 7-8 | 1.20 | 10.3 |
| 8-9 | 1.19 | 10.2 |
| 9-10 | 1.18 | 10.2 |
| 10-11 | 1.20 | 10.3 |
| 11-12 | 1.18 | 10.2 |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 12-13 | 1.19 | 10.2 |
| 13-14 | 1.17 | 10.0 |
| 14-15 | 1.18 | 10.2 |
| 15-16 | 1.16 | 9.92 |
| 16-17 | 1.17 | 10.1 |
| 17-18 | 1.12 | 9.59 |
| 18-19 | 1.10 | 9.45 |
| 19-20 | 1.16 | 9.94 |
| Jumlah | 23.5 |

*Packet Loss* = x100% = 0 %

Dari hasil perhitungan diperoleh nilai *throughput* sebesar 1.37 Gbytes dari percobaan selama 20 detik, sedangkan nilai *packet loss* diperoleh 0%.

#### Skenario 15 Switch

Pada skenario ini terdapat lima belas *switch* dengan masing-masing *switch* terhubung dengan dua *host*. *Switch* dan *host* dihubungkan menggunakan kabel ethernet sehingga membentuk topologi linear seperti pada gambar 4.3, selanjutnya *Pox controller* dijalankan dengan perintah ./pox.py forwarding.l2\_learning. Selanjutnya *pox controller* akan membaca *mac address* setiap *switch* yang terhubung.

Pengujian dilakukan dengan dua cara yaitu dengan menggunakan protokol *ICMP* dan *iPerf tools*. Pengujian menggunakan protokol icmp dilakukan dengan cara node h1 melakukan request kepada node h30 sebanyak 20 detik dengan mengetikan perintah h1 ping -c 20 h30 di *Command Line Interface (CLI)* pada *mininet*.

**Tabel 4.5.** Data hasil *ICMP 15 Switch pox Controller*

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 1 | 64 | 142 |
| 2 | 64 | 1.27 |
| 3 | 64 | 0.141 |
| 4 | 64 | 0.142 |
| 5 | 64 | 0.142 |
| 6 | 64 | 0.140 |
| 7 | 64 | 0.176 |
| 8 | 64 | 0.195 |
| 9 | 64 | 0.176 |
| 10 | 64 | 0.280 |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 11 | 64 | 0.155 |
| 12 | 64 | 0.160 |
| 13 | 64 | 0.161 |
| 14 | 64 | 0.152 |
| 15 | 64 | 0.163 |
| 16 | 64 | 0.156 |
| 17 | 64 | 0.159 |
| 18 | 64 | 0.156 |
| 19 | 64 | 0.154 |
| 20 | 64 | 0.155 |
| Jumlah | 1280 | 146.233 |

Selanjutnya data *ICMP* digunakan untuk menghitung nilai *delay* dan *jitter* pada skenario limabelas *switch*  dengan persamaan :

*Delay* = *= 0.114 ms*

*Jitter* = = 0.005 ms

Total variasi *Delay* = Rata - rata delay - Delay = 7.311 – 0.114 = 7.197 ms

Rata – Rata *Delay* = = = 7.311 ms

Dari hasil perhitungan diperoleh nilai *delay* sebesar 0.114 ms dan nilai *jitter* sebesar 0.005 ms.

Selanjutnya dilakukan pengujian dengan *iPerf Tools*, *iPerf* dijalankan dengan cara *node* h1 dijadikan sebagai *server* dan *node* h30 dijadikan sebagai *client* yang melakukan download *file* dari *server* selama 20 detik. Pada terminal *node* h1 dijalankan perintah iperf -s dan pada *node* h20 dijalankan perintah iperf -c 10.0.0.1 -t 20 -i 1. Pengujian dengan iPerf Tools ini bertujuan untuk mendapatkan nilai *throughput* dan *packet loss.*

**Tabel 4.6.** Data iPerf 15 *switch pox Controller*

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 0-1 | 0.841 | 7.06 |
| 1-2 | 0.892 | 7.48 |
| 2-3 | 0.882 | 7.39 |
| 3-4 | 0.915 | 7.68 |
| 4-5 | 0.920 | 7.72 |
| 5-6 | 0.802 | 6.73 |
| 6-7 | 0.732 | 6.14 |
| 7-8 | 0.744 | 6.24 |
| 8-9 | 0.712 | 5.97 |
| 9-10 | 0.710 | 5.95 |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 10-11 | 0.742 | 6.62 |
| 11-12 | 0.733 | 5.50 |
| 12-13 | 0.799 | 5.86 |
| 13-14 | 0.790 | 6.86 |
| 14-15 | 0.656 | 7.54 |
| 15-16 | 0.699 | 5.54 |
| 16-17 | 0.818 | 6.86 |
| 17-18 | 0.899 | 7.54 |
| 18-19 | 0.660 | 5.54 |
| 19-20 | 0.816 | 6.85 |
| Jumlah | 15.4 |

*Packet Loss* = x100% = 0 %

Dari hasil perhitungan diperoleh nilai *throughput* sebesar 0.77 Gbytes dari percobaan selama 20 detik, sedangkan nilai *packet loss* diperoleh 0%.

### **Hasil Pengujian Ryu Controller**

Parameter yang diujikan adalah *delay*, *jitter, throughput, packet loss* dan waktu pembentukan *flow table*.

#### Hasil Pengujian Skenario 5 switch

*Ryu controller* dijalankan dengan perintah ./ryu-manager . setelah dijalankan *Ryu controller* akan membaca *mac address* setiap *switch* yang terhubung. Pengujian menggunakan protokol ICMP dilakukan dengan cara *node* h1 melakukan request kepada *node* h10 sebanyak 20 detik dengan mengetikan perintah h1 *ping* -c 20 h10 di *Command Line Interface (CLI)* pada *mininet*.

**Tabel 4.7.** Data ICMP 5 *Switch Ryu controller*

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 1 | 64 | 79.9 |
| 2 | 64 | 0.478 |
| 3 | 64 | 0.105 |
| 4 | 64 | 0.095 |
| 5 | 64 | 0.101 |
| 6 | 64 | 0.101 |
| 7 | 64 | 0.105 |
| 8 | 64 | 0.110 |
| 9 | 64 | 0.110 |
| 10 | 64 | 0.116 |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 11 | 64 | 0.094 |
| 12 | 64 | 0.117 |
| 13 | 64 | 0.116 |
| 14 | 64 | 0.113 |
| 15 | 64 | 0.113 |
| 16 | 64 | 0.111 |
| 17 | 64 | 0.112 |
| 18 | 64 | 0.107 |
| 19 | 64 | 0.093 |
| 20 | 64 | 0.108 |
| Jumlah | 1280 | 82.302 |

Selanjutnya data *ICMP* digunakan untuk menghitung nilai *Delay* dan *Jitter* pada skenario lima *switch*  dengan persamaan :

*Delay* = *= 0.064 ms*

*Jitter* = = 0.003 ms

Total variasi *Delay* = Rata - rata delay - Delay = 4.115 – 0.064 = 4.051 ms

Rata-rata *Delay* = = = 4.115 ms

Dari hasil perhitungan diperoleh nilai *Delay* sebesar 0.064 ms dan nilai *Jitter* sebesar 0.005 ms.

Selanjutnya dilakukan pengujian dengan *iPerf Tools*, *iPerf* dijalankan dengan cara *node* h1 dijadikan sebagai server dan *node* h10 dijadikan sebagai *client* yang melakukan download *file* dari *server* selama 20 detik.

**Tabel 4.8**. Data iPerf 5 *Switch Ryu Controller*

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 0-1 | 1.28 | 11.0 |
| 1-2 | 1.36 | 11.7 |
| 2-3 | 1.37 | 11.8 |
| 3-4 | 1.36 | 11.7 |
| 4-5 | 1.34 | 11.5 |
| 5-6 | 1.31 | 11.2 |
| 6-7 | 1.27 | 10.9 |
| 7-8 | 1.32 | 11.3 |
| 8-9 | 1.28 | 11.0 |
| 9-10 | 1.14 | 9.81 |
| 10-11 | 1.27 | 10.9 |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 11-12 | 1.30 | 11.1 |
| 12-13 | 1.28 | 11.0 |
| 13-14 | 1.29 | 11.1 |
| 14-15 | 1.28 | 11.0 |
| 15-16 | 1.23 | 10.6 |
| 16-17 | 1.32 | 11.3 |
| 17-18 | 1.31 | 11.2 |
| 18-19 | 1.30 | 11.2 |
| 19-20 | 1.31 | 11.2 |
| Jumlah | 25.9 |

*Packet Loss* = x100%

= 0 %

Dari hasil perhitungan diperoleh nilai *throughput* sebesar 1.295 Gbytes dari percobaan selama 20 detik, sedangkan nilai *packet loss* diperoleh 0%.

#### Skenario 10 Switch Ryu Controller

Pengujian menggunakan protokol ICMP dilakukan dengan cara *node* h1 melakukan *request* kepada *node* h20 sebanyak 20 detik dengan mengetikan perintah h1 *ping* -c 20 h20 di *Command Line Interface (CLI)* pada *mininet*.

**Tabel 4.9**. Data ICMP 10 *Switch Ryu controller*

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 1 | 64 | 133 |
| 2 | 64 | 0.643 |
| 3 | 64 | 0.116 |
| 4 | 64 | 0.139 |
| 5 | 64 | 0.119 |
| 6 | 64 | 0.124 |
| 7 | 64 | 0.144 |
| 8 | 64 | 0.128 |
| 9 | 64 | 0.122 |
| 10 | 64 | 0.123 |
| 11 | 64 | 0.107 |
| 12 | 64 | 0.147 |
| 13 | 64 | 0.129 |
| 14 | 64 | 0.129 |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 15 | 64 | 0.126 |
| 16 | 64 | 0.131 |
| 17 | 64 | 0.118 |
| 18 | 64 | 0.125 |
| 19 | 64 | 0.120 |
| 20 | 64 | 0.118 |
| Jumlah | 1280 | 136.192 |

Selanjutnya data *ICMP* digunakan untuk menghitung nilai *Delay* dan *Jitter* pada skenario lsepuluh *switch*  dengan persamaan :

*Delay* = *= 0.106 ms*

*Jitte*r = = 0.005 ms

Total variasi *Delay* = Rata - rata delay - Delay = 6.809 – 0.106 = 6.698 ms

Rata-Rata *Delay* = = = 6.809 ms

Dari hasil perhitungan diperoleh nilai *Delay* sebesar 0.106 ms dan nilai *Jitter* sebesar 0.005 ms.

Selanjutnya dilakukan pengujian dengan *iPerf Tools*, *iPerf* dijalankan dengan cara *node* h1 dijadikan sebagai server dan *node* h10 dijadikan sebagai *client* yang melakukan download *file* dari *server* selama 20 detik.

**Tabel 4.10.** Data iPerf 10 *Switch Ryu Controller*

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 0-1 | 1.06 | 11.0 |
| 1-2 | 1.06 | 11.7 |
| 2-3 | 1.07 | 11.8 |
| 3-4 | 1.00 | 11.7 |
| 4-5 | 1.05 | 11.5 |
| 5-6 | 1.01 | 11.2 |
| 6-7 | 1.01 | 10.9 |
| 7-8 | 1.01 | 11.3 |
| 8-9 | 1.00 | 11.0 |
| 9-10 | 1.00 | 9.81 |
| 10-11 | 1.03 | 10.9 |
| 11-12 | 1.02 | 11.1 |
| 12-13 | 1.02 | 11.0 |
| 13-14 | 1.01 | 11.1 |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 14-15 | 1.01 | 11.0 |
| 15-16 | 0.94 | 10.6 |
| 16-17 | 1.01 | 11.3 |
| 17-18 | 1.00 | 11.2 |
| 18-19 | 0.96 | 11.2 |
| 19-20 | 1.01 | 11.2 |
| Jumlah | 20.2 |

*Packet Loss* = x100%

= 0 %

Dari hasil perhitungan diperoleh nilai *throughput* sebesar 1.01 Gbytes dari percobaan selama 20 detik, sedangkan nilai *packet loss* diperoleh 0%.

#### Skenario 15 Switch Ryu Controller

Pengujian menggunakan protokol ICMP dilakukan dengan cara *node* h1 melakukan *request* kepada *node* h30 sebanyak 20 detik dengan mengetikan perintah h1 *ping* -c 20 h30 di *Command Line Interface (CLI)* pada *mininet*.

**Tabel 4.11.** Data icmp 15 *switch* *Ryu controller*

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 1 | 64 | 201 |
| 2 | 64 | 0.969 |
| 3 | 64 | 0.131 |
| 4 | 64 | 0.119 |
| 5 | 64 | 0.133 |
| 6 | 64 | 0.133 |
| 7 | 64 | 0.128 |
| 8 | 64 | 0.157 |
| 9 | 64 | 0.139 |
| 10 | 64 | 0.138 |
| 11 | 64 | 0.135 |
| 12 | 64 | 0.137 |
| 13 | 64 | 0.141 |
| 14 | 64 | 0.136 |
| 15 | 64 | 0.154 |
| 16 | 64 | 0.144 |
| 17 | 64 | 0.139 |
| **Waktu (s)** | **Data request (Bytes)** | **TTL (ms)** |
| 18 | 64 | 0.135 |
| 19 | 64 | 0.133 |
| 20 | 64 | 0.134 |
| Jumlah | 1280 | 204.454 |

Selanjutnya data *ICMP* digunakan untuk menghitung nilai *Delay* dan *Jitter* pada skenario limabelas *switch*  dengan persamaan :

*Delay* = *= 0.159 ms*

*Jitter* = = 0.007 ms

Total variasi *Delay* = Rata - rata delay - Delay = 10.222 – 0.159 = 10.063 ms

Rata-Rata *Delay* = = = 10.222 ms

Dari hasil perhitungan diperoleh nilai *Delay* sebesar 0.159 ms dan nilai *Jitter* sebesar 0.007 ms.

Selanjutnya dilakukan pengujian dengan *iPerf Tools*, *iPerf* dijalankan dengan cara *node* h1 dijadikan sebagai server dan *node* h30 dijadikan sebagai *client* yang melakukan download *file* dari *server* selama 20 detik.

**Tabel 4.12**. Data iPerf 15 *Switch Ryu Controller*

|  |  |  |
| --- | --- | --- |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 0-1 | 0.980 | 8.22 |
| 1-2 | 0.874 | 7.33 |
| 2-3 | 0.904 | 7.59 |
| 3-4 | 0.909 | 7.63 |
| 4-5 | 0.885 | 7.43 |
| 5-6 | 0.852 | 7.15 |
| 6-7 | 0.840 | 7.05 |
| 7-8 | 0.825 | 6.92 |
| 8-9 | 0.850 | 7.13 |
| 9-10 | 0.866 | 7.26 |
| 10-11 | 0.999 | 8.38 |
| 11-12 | 0.965 | 8.10 |
| 12-13 | 1.02 | 8.73 |
| 13-14 | 0.958 | 8.04 |
| 14-15 | 0.890 | 7.47 |
| 15-16 | 0.913 | 7.66 |
| 16-17 | 0.906 | 7.60 |
| **Waktu (s)** | **Transfer (GigaBytes)** | **Bandwith (Gbits/second)** |
| 17-18 | 0.883 | 7.41 |
| 18-19 | 0.838 | 7.03 |
| 19-20 | 0.858 | 7.20 |
| Jumlah | 17.6 |

*Packet Loss* = x100%

= 0 %

Dari hasil perhitungan diperoleh nilai *throughput* sebesar 0.88 Gbytes dari percobaan selama 20 detik, sedangkan nilai *packet loss* diperoleh 0%.

## **Pembahasan**

**Tabel 4.13** Hasil pengujian pada *Pox controller*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Jumlah switch** | **Delay (ms)** | **Jitter (ms)** | **Throughput (Gigabytes)** | **Packet loss (%)** |
| 5 switch | 0.059 | 0.002 | 27.4 | 0 |
| 10 switch | 0.077 | 0.003 | 23.5 | 0 |
| 15 switch | 0.144 | 0.005 | 15.4 | 0 |

**Tabel 4.14**. Hasil pengujian *Ryu controller*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Jumlah switch** | **Delay (ms)** | **Jitter (ms)** | **Throughput (Gigabytes)** | **Packet loss (%)** |
| 5 switch | 0.064 | 0.003 | 25.9 | 0 |
| 10 switch | 0.106 | 0.005 | 20.2 | 0 |
| 15 switch | 0.159 | 0.007 | 17.6 | 0 |

### **Delay**

Pola nilai *delay* yang diperoleh dari pengujian dua *controller* yang berbeda dengan jumlah switch yang berbeda cenderung naik, namun pada pada pengujian dengan *Pox controller* terjadi penurunan. Penurunan terjadi pada topologi linear dengan jumlah *switch* 5 ke topologi dengan jumlah *switch* 10. Hal ini disebabkan oleh waktu yang dibutuhkan *OpenFlow* untuk membentuk *flow table* pada setiap *switch* yang ada. *OpenFlow* pada *Pox Controller bekerja* lebih cepat dengan waktu rata – rata dibawah 10 detik disetiap switch yang ada pada setiap topologi, sedangkan *OpenFlow* pada *Ryu Controller* memiliki waktu rata – diatas 10 detik (Lihat Lampiran 4, 5 dan 6) . Nilai *delay* tertinggi terjadi pada topologi dengan 15 *switch* untuk pengujian pada *rox controller,* sedangkan pada *ryu controller* nilai tertinggi terdapat pada topologi 15 *switch*. Untuk lebih jelasnya dapat dilihat pada Gambar 4.1.

**Gambar 4.1.** Perbandingan *delay* *pox controller* dan *ryu controller*

### **Jitter**

Pola nilai *jitter* yang diperoleh dari pengujian dua *controller* yang berbeda dengan jumlah *switch* yang berbeda cenderung naik. Kenaikan terjadi pada semua skenario. Hal ini disebabkan oleh waktu yang dibutuhkan *OpenFlow* untuk membentuk *flow table* pada setiap *switch* yang ada. *OpenFlow* pada *Pox Controller bekerja* lebih cepat dengan waktu rata – rata dibawah 10 detik disetiap switch yang ada pada setiap topologi, sedangkan *OpenFlow* pada *Ryu Controller* memiliki waktu rata – diatas 10 detik (Lihat Lampiran 4, 5 dan 6) . Nilai *Jitter* tertinggi terjadi pada topologi dengan 15 *switch* untuk pengujian pada *pox controller,* sedangkan pada *ryu controller* nilai tertinggi terdapat pada topologi 15 *switch*. Untuk lebih jelasnya dapat dilihat pada Gambar 4.2.

**Gambar 4.2.** Perbandingan *jitter* *pox controller* dan *ryu controller*

### **Throughput**

Nilai *throughput* dari kedua *controller* mengalami penurunan. Hal ini disebabkan oleh bertambahnya jumlah *node* yang ada pada topologi jaringan. *throughput* tertinggi diperoleh pada skenario dengan 5 *switch* dan 10 *host*, sedangkan *throughput* paling rendah diperoleh pada skenario dengan 15 *switch* 30 *host*. Untuk lebih jelasnya bisa dilihat pada Gambar 4.3dibawah ini.

**Gambar 4.3.** Perbandingan *throughput Pox* *controller* dan *Ryu controller*

### **Packet Loss**

Pada pengujian paket loss dengan menggunakan iperf tools diperoleh nilai packet loss 0% dari setiap skenario. Hal ini disebabkan oleh data yang di *download* dari *server* berhasil terdownload semua oleh *client*. Untuk lebih jelasnya bisa dilihat pada **Gambar 4.4.**

**Gambar 4.4** Perbandingan *packet loss Pox controller* dengan *Ryu controller*

|  |  |  |
| --- | --- | --- |
| **Skenario** | **Pox Controller** | **Ryu Controller** |
| 5 switch | 6 s | 5 s |
| 10 switch | 7 s | 7 s |
| 15 switch | 10 s | 10 s |

* + 1. **Waktu Sinkronisasi Switch dan Controller**

**Tabel 4.15.** Hasil perbandingan sinkronisasi oleh *controller*

Proses pengujian dilakukan dengan mengaktifkan aplikasi *simple*\_*switch.py* pada *Ryu controller* sedangkan pada *Pox controller* dijalnkan perintah forwarding.l2\_learning. Waktu sinkronisasi diperoleh menggunakan aplikasi *wireshark.*

**Gambar 4.5** Perbandingan waktu sinkronisasi *switch* dengan *controller*

Pada Gambar 4.5 waktu yang dibutuhkan oleh kedua *controller* untuk membentuk singkronisasi dengan *switch* cenderung naik. Nilai terendah diperoleh pada *ryu controller* dengan nilai sebesar 5 detik.

# **BAB V**

# **PENUTUP**

## **Kesimpulan**

Dari hasil analisis yang telah dilakukan dalam pengujian sistemberbasis *Software Defined Network*, maka dapat disimpulkan bahwa :

1. Topologi jaringan virtual berbasis SDN dapat dirancang dan dapat dijalankan pada mininet, dengan maksimum topologi 15 switch menggunakan sumber daya dengan spesifikasi RAM 8 GB.
2. Kinerja *Pox controller* dan *Ryu controller* memiliki nilai *delay* yang tetap meskipun ada beberapa peningkatan sesuai dengan jumlah *switch* yang semakin meningkat, nilai *jitter* cenderung meningkat sesuai dengan jumlah *switch* yang ada pada skenario. Sedangkan untuk nilai *throughput* cenderung turun sesuai dengan beban *switch* yang diberikan.

## **Saran**

Sehubungan dengan selesainya proses pembuatan skripsi ini, penulis bermaksud menyampaikan beberapa saran kepada para pembaca yakni :

1. Menambah jumlah node dalam topologi jaringan.
2. Sebaiknya dilakukan pengujian dengan topologi jaringan seperti topologi tree, mesh dan sebagainya.
3. Sebaiknya dilakukan pengujian dengan parameter quality of service lainnya*.*

# **DAFTAR PUSTAKA**

Budi, B, & Haryadi, S., 2016. *Simulasi Mobility pada Software Defined Networking*. Bandung Institude of Technogy, 2-3.

Effendy, M. R., Eki, A. Z. H., & Andriansyah S. *Implementasi GRE Tunneling Menggunakan Open vSwitch Pada Jaringan Kampus.*

Florian Sieberz, ed. 2014. *Masterprojekt Software Defined Networking.* Kaiserslautern (GE) : Technische Universitat Kaiserslautern.

Flowgramable. 2018. *Openflow.*

[*http://flowgrammable.org/sdn/openflow/#tab\_protocol*](http://flowgrammable.org/sdn/openflow/#tab_protocol), diakses tanggal 17 Oktober.

Gratianus, S., & Nafiri, L. *Pemanfaatan Virtual Box Dalam Praktikum Administrasi Server Menggunakan Teknik Dhcp Pada Mikrotik Router Os.* Universitas Methodist Indonesia.

<https://www.sdxcentral.com/sdn/definitions/sdn-controllers/openflow-controller/>, diakses pada tanggal 17 Oktober.

Heryanto, A., & Afrilia. 2017. *Software Defined Network Menggunakan Simulator Mininet*. Universitas Sriwijaya, 1 – 2.

Iperf.rf. <https://iperf.fr/>, diakses pada tanggal 17 oktober 2018.

Kartadie, R., & Satya, B., 2015. *Uji Performa Implementasi Software-Based Openflow Switch Berbasis Openwrt Pada Infrastruktur Software-Defined Network*. STIMIK AMIKOM Yogyakarta, 15, 87 – 89.

Muzawi, R., & Romi, R. *Perancangan Server Dan Analisis Quality of Service (QoS) Jaringan Diskless PXE Linux Pada Laboratorium Komputer STMIK-Amik-Riau*. STIMIK-Amik-Riau. Riau

Open Networking Foundation, 2014*, SDN Architecture*. [University of Washington](https://www.coursehero.com/sitemap/schools/2505-University-of-Washington/) , 1 , 16-17.

P. Purnama, Depe, R. Rumani M., Sofia Naning Hertania. 2016. *Perancangan Sistem Monitoring Pada Sdn (Software Defined Network) Berbasis Web Dengan Menggunakan Protokol Rest*. Universitas Telkom,

Pupatwibul, Pakawat, Ameen Banjar, Robin Braun.2014*. Performance Evaluation of TCP/IP vs. OpenFlow in INET Framework Using OMNeT++, and Implementation of Intelligent Computational Model to Provide Autonomous Behaviour.* University of Technology Sydney.

Putra, Moh.W, Eko S. P., & Widhi Y. (2018). *Analisis Perbandingan Performansi Kontroler Floodlight, Maestro, RYU, POX Dan ONOS Dalam Arsitektur Software Defined Network (SDN)*. Universitas Brawijaya, 2, 3779-3787.

Rika Wulandari. *Analisis Qos (Quality Of Service) Pada Jaringan Internet (Studi Kasus Upt Loka Uji Teknik Penambangan Jampang Kulon – Lipi).*

*Searchnetworking.* [*https://searchnetworking.techtarget.com/definition/throughput*](https://searchnetworking.techtarget.com/definition/throughput)*. Diakses tanggal 25 januari*

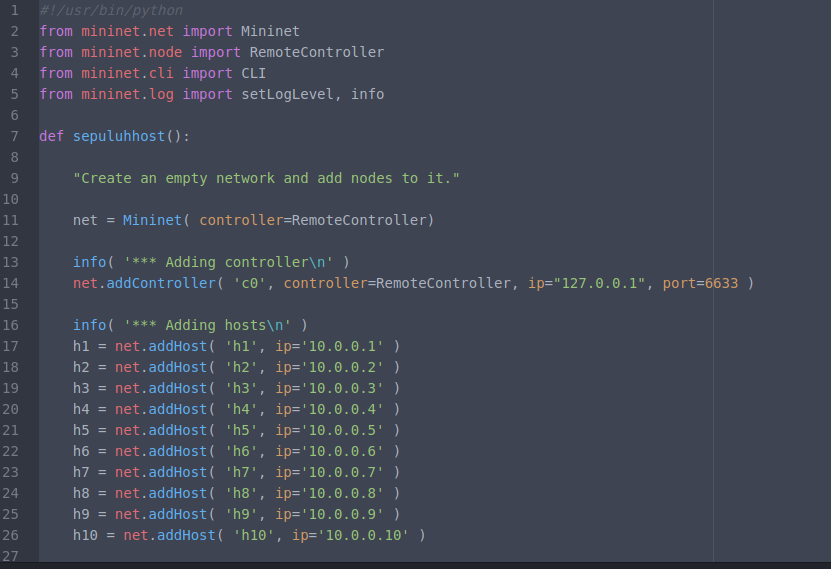
Tulloh, R., 2017. *Analisis Performansi VLAN Pada Jaringan Software Defined Network (SDN).* Universitas Telkom, 9, 407-408.

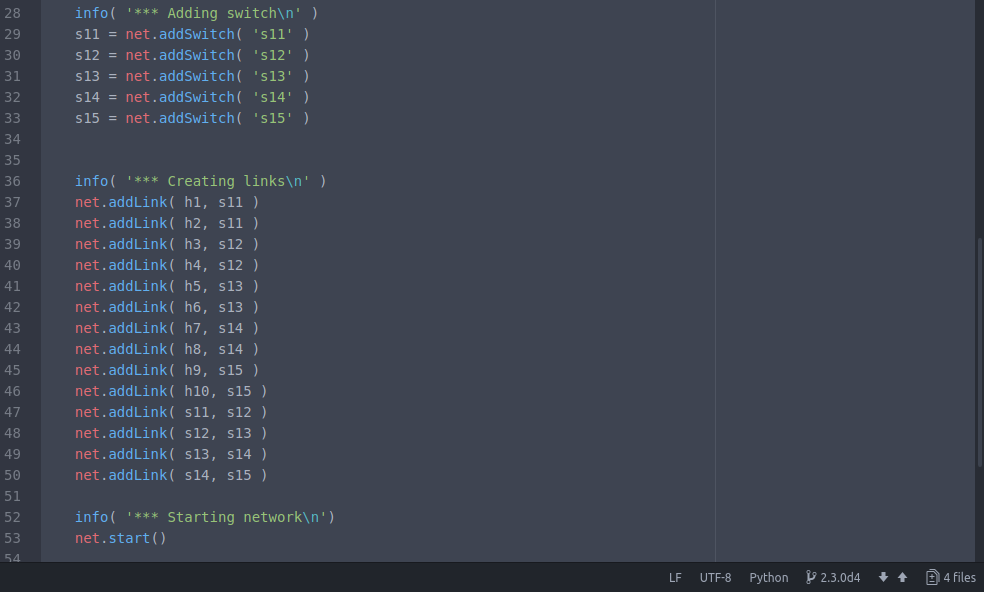
Tulloh, Rohmat, Ridha Muldina Negara, Arif Nurhidayat. 2015.  *Simulasi Virtual Local Area Network (VLAN) Berbasis Software Defined Network (SDN) Menggunakan POX Controller*. Universitas Telkom,

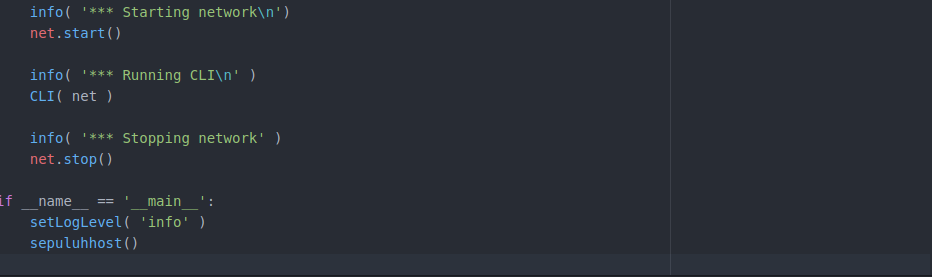
Ummah, I., & Abdillah, S., 2016. *Perancangan Simulasi Jaringan Virtual Berbasis Software-Define Networking*. Telkom University, 1, 96-97.

**LAMPIRAN**

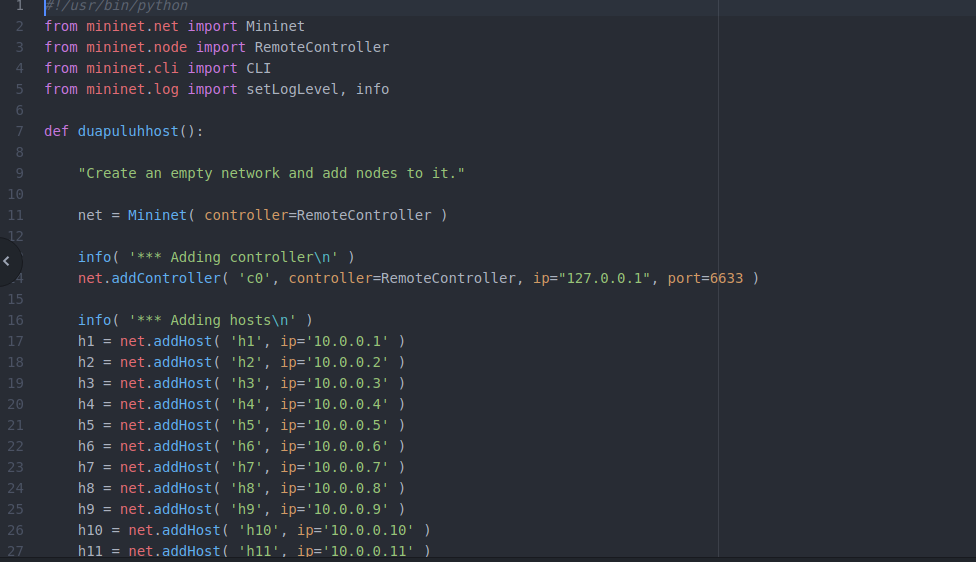
* + - 1. **Souce code 5 switch**

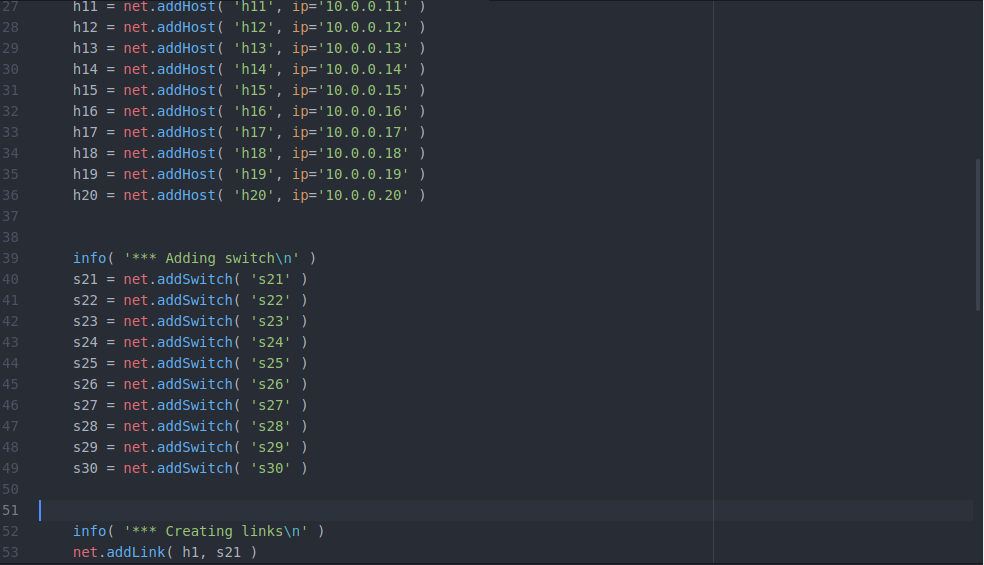


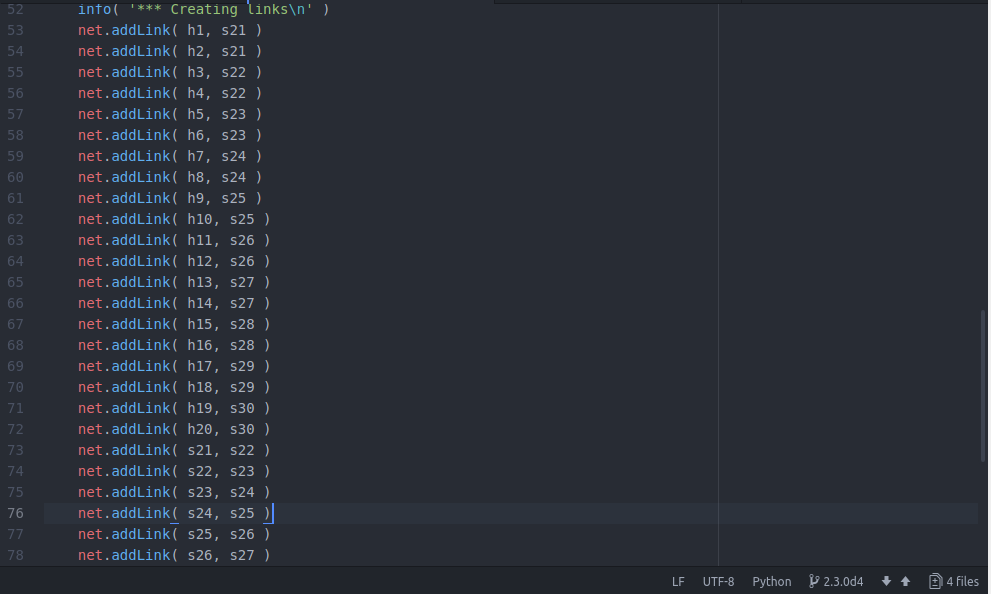


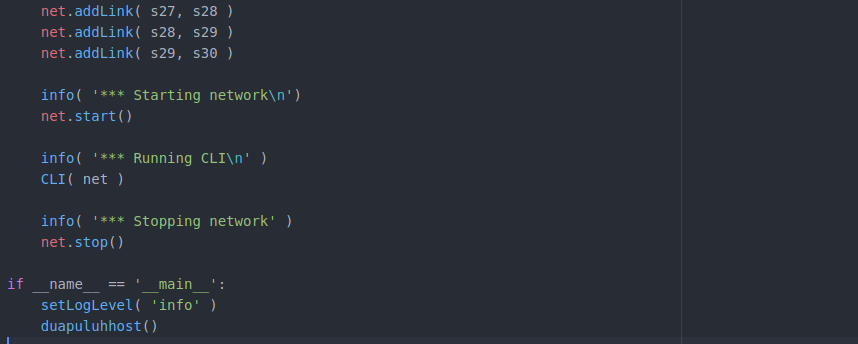


* + - 1. **Source code 10 switch**

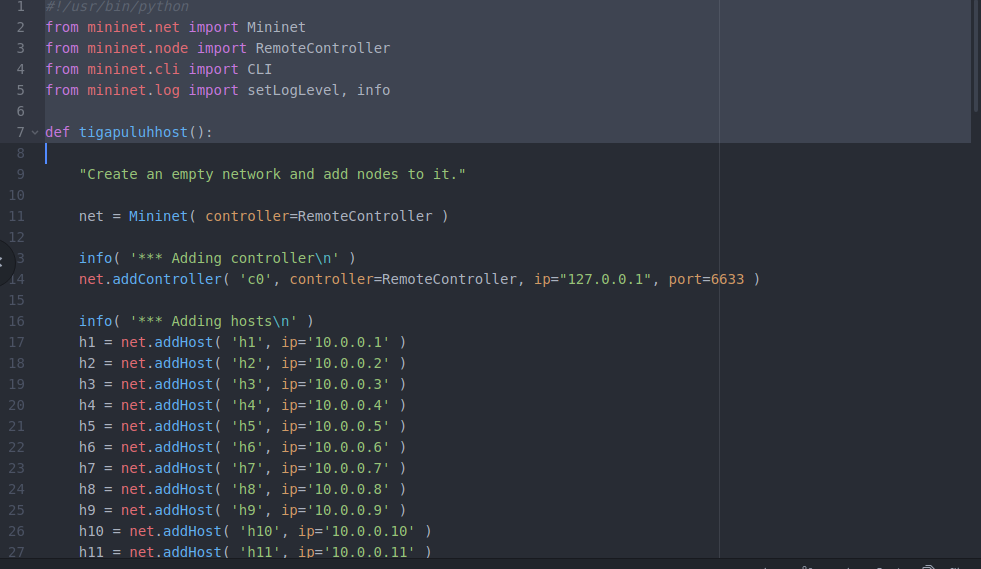


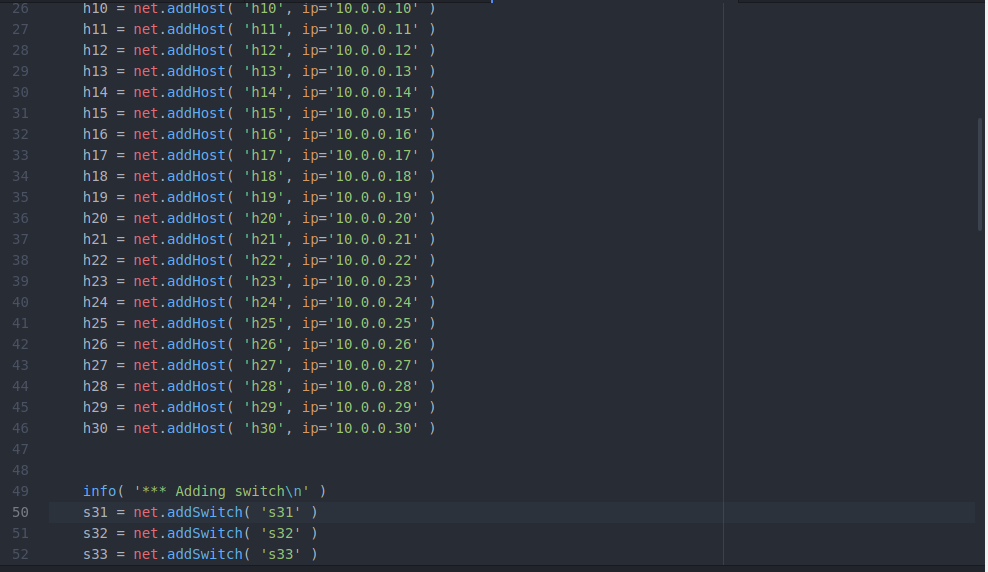


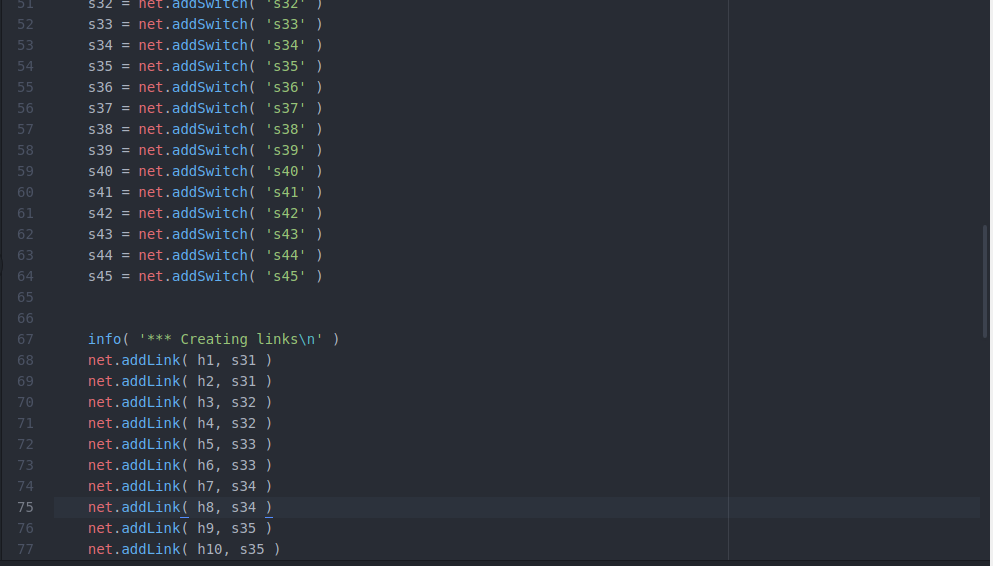


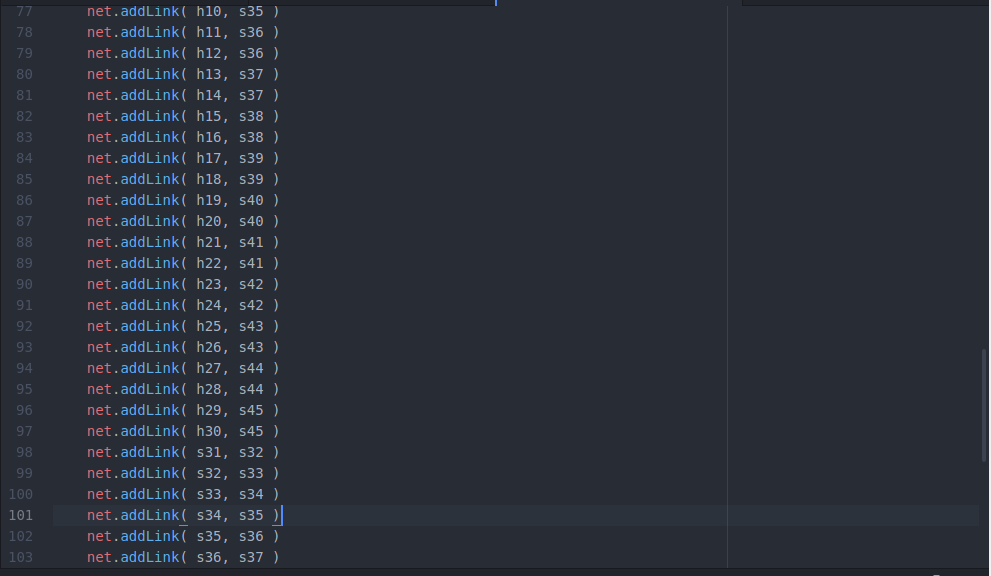


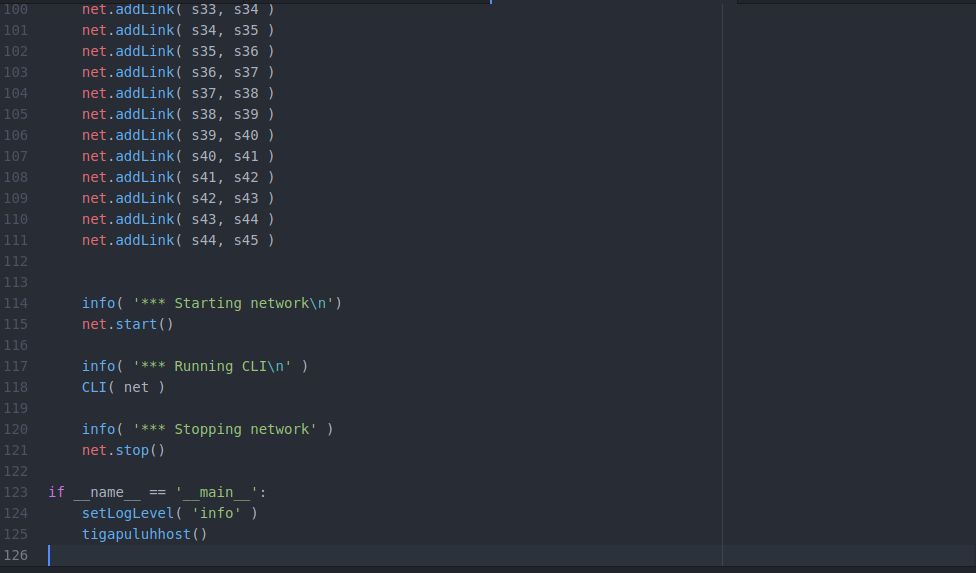
* + - 1. **Source code 15 switch**











**4. Waktu Pembentukan Flow Table pada topologi 5 switch**

|  |  |  |
| --- | --- | --- |
| **Switch** | **Pox controller** | **Ryu Controller** |
| Switch 11 - switch 12 | 4 s | 14 s |
| Switch 12 – switch 13 | 7 s | 10 s |
| Switch 13 -switch 14 | 0 s | 14 s |
| Switch 14 – switch 15 | 5 s | 10 s |

**5. Waktu Pembentukan Flow Table topologi 10 Switch**

|  |  |  |
| --- | --- | --- |
| **Switch** | **Pox controller** | **Ryu Controller** |
| **Switch 11 - switch 12** | **5 s** | **11 s** |
| **Switch** | **Pox controller** | **Ryu Controller** |
| Switch 22 - switch 23 | 10 s | 9 s |
| Switch 23 – switch 24 | 1 s | 15 s |
| Switch 24 - switch 25 | 5 s | 10 s |
| Switch 25 – switch 26 | 5 s | 13 s |
| Switch 26 – switch 27 | 1 s | 14 s |
| Switch 27 – switch 28 | 5 s | 13 s |
| Switch 28 – switch 29 | 6 s | 13 s |
| Switch 29 – switch 30 | 6 s | 14 s |

**6. Waktu Pembentukan Flow Table Topologi 15 Switch**

|  |  |  |
| --- | --- | --- |
| **Switch** | **Pox controller** | **Ryu Controller** |
| Switch 31 - switch 32 | 6 s |  |
| Switch 32 – switch 33 | 3 s | 11 s |
| Switch 33 - switch 34 | 3 s | 17 s |
| Switch 34 – switch 35 | 2 s | 16 s |
| **Switch** | **Pox Controller** | **Ryu Controller** |
| Switch 35 – Switch 36 | 4 s | 9 s |
| Switch 36 – switch 37 | 5 s | 13 s |
| Switch 37 – Switch 38 | 3 s | 14 s |
| Switch 38 – Switch 39 | 1 s | 8 s |
| Switch 39 – Switch 40 | 4 s | 10 s |
| Switch 40 – Switch 41 | 11 s | 15 s |
| Switch 41 – Switch 42 | 14 s | 9 s |
| Switch 42 – switch 43 | 14 s | 15 s |
| Switch 43 – Switch 44 | 7 s | 12 s |
| Switch 44 – Switch 45 | 7 s | 5 s |