

**OPTIMASI *FLOW ENTRIES* UNTUK MENCEGAH *FLOW TABLE*  
*OVERFLOW* PADA *SERVER LOAD BALANCING* BERBASIS  
*SOFTWARE DEFINED NETWORKING***

**SKRIPSI**

Untuk memenuhi sebagian persyaratan  
memperoleh gelar Sarjana Komputer

Disusun oleh:

Agung Wahyu Setio Budi

NIM: 155150200111070



PROGRAM STUDI TEKNIK INFORMATIKA  
JURUSAN TEKNIK INFORMATIKA  
FAKULTAS ILMU KOMPUTER  
UNIVERSITAS BRAWIJAYA  
MALANG  
2019

## PENGESAHAN

OPTIMASI *FLOW ENTRIES* UNTUK MENCEGAH *FLOW TABLE OVERFLOW* PADA  
*SERVER LOAD BALANCING* BERBASIS *SOFTWARE DEFINED NETWORKING*

### SKRIPSI

Diajukan untuk memenuhi sebagian persyaratan  
memperoleh gelar Sarjana Komputer

Disusun Oleh :  
Agung Wahyu Setio Budi  
NIM: 155150200111070

Skripsi ini telah diuji dan dinyatakan lulus pada  
22 Juli 2019

Telah diperiksa dan disetujui oleh:

Dosen Pembimbing



Achmad Basuki, S.T, M.MG, P.hD.  
NIP: 197411182003121002

Mengetahui  
Ketua Jurusan **Teknik Informatika**



Tri Astoto Kurniawan, S.T, M.T, Ph.D  
NIP: 19710518 200312 1 001

## PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam naskah skripsi ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain untuk memperoleh gelar akademik di suatu perguruan tinggi, dan tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis disitasi dalam naskah ini dan disebutkan dalam daftar referensi.

Apabila ternyata didalam naskah skripsi ini dapat dibuktikan terdapat unsur-unsur plagiasi, saya bersedia skripsi ini digugurkan dan gelar akademik yang telah saya peroleh (sarjana) dibatalkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 23 Juni 2019



Agung Wahyu Setio Budi

NIM: 155150200111070

## KATA PENGANTAR

Puji syukur peneliti panjatkan kepada Tuhan Yang Maha Esa atas limpahan rahmat dan petunjuk-Nya, sehingga peneliti dapat menyelesaikan skripsi yang berjudul “OPTIMASI *FLOW ENTRIES* UNTUK MENCEGAH *FLOW TABLE OVERFLOW* PADA *SERVER LOAD BALANCING* BERBASIS *SOFTWARE DEFINED NETWORKING*”.

Dalam penyusunan dan penelitian skripsi ini tidak lepas dari bantuan moral dan materiil yang diberikan dari berbagai pihak, maka peneliti mengucapkan banyak terima kasih kepada:

1. Bapak Wayan Firdaus Mahmudy, S.Si, M.T, Ph.D. selaku Dekan Fakultas Ilmu Komputer Universitas Brawijaya Malang.
2. Bapak Heru Nurwarsito, Ir., M.Kom. selaku Wakil Ketua I Bidang Akademik Fakultas Ilmu Komputer Universitas Brawijaya Malang.
3. Bapak Tri Astoto Kurniawan, S.T, M.T, Ph.D. selaku Ketua Jurusan Teknik Informatika Universitas Brawijaya Malang.
4. Bapak Agus Wahyu Widodo, S.T, M.Cs. selaku Ketua Program Studi Teknik Informatika Universitas Brawijaya Malang.
5. Bapak Achmad Basuki, S.T, M.MG, Ph.D. selaku dosen pembimbing tunggal yang telah memberikan pengarahan dan bimbingan kepada peneliti, sehingga dapat menyelesaikan skripsi ini dengan baik.
6. Kedua orang tua, kakak, dan seluruh keluarga besar atas segala nasehat, kasih sayang, perhatian, dan kesabarannya memberikan semangat kepada peneliti, serta senantiasa tiada hentinya memberikan doa demi terselesaikannya skripsi ini.
7. Aziza Ainun Assiddiq yang telah memberikan dukungan, semangat, dan doa demi kelancaran serta kemudahan dalam menyelesaikan skripsi ini.
8. Teman – teman saya di TitanSO<sub>4</sub>, KAP, BEM FILKOM UB, Paguyuban KSE UB yang telah memberikan semangat untuk segera menyelesaikan tanggung jawab akademik saya sebagai mahasiswa S1 ini.
9. Seluruh civitas akademika Informatika Universitas Brawijaya dan terkhusus untuk teman-teman Teknik Informatika Angkatan 2015, teman-teman keminatan Komputasi Berbasis Jaringan yang telah banyak memberi bantuan dan dukungan selama peneliti menempuh studi di Teknik Informatika Universitas Brawijaya dan selama penyelesaian skripsi ini.
10. Seluruh pihak yang tidak dapat diucapkan satu persatu, peneliti mengucapkan banyak terima kasih atas segala bentuk dukungan dan doa sehingga laporan skripsi ini dapat terselesaikan.

Peneliti menyadari bahwa dalam penyusunan skripsi ini masih terdapat banyak kekurangan, sehingga saran dan kritik yang membangun sangat peneliti harapkan.

Akhir kata peneliti berharap skripsi ini dapat membawa manfaat bagi semua pihak yang membutuhkannya.

Malang, 23 Juni 2019

Agung Wahyu Setio Budi

agungwahyusb@student.ub.ac.id

## ABSTRAK

**Agung Wahyu Setio Budi, Optimasi *Flow Entries* untuk Mencegah *Flow Table Overflow* pada *Server Load Balancing* Berbasis *Software Defined Networking***

**Pembimbing: Achmad Basuki, S.T, M.MG, Ph.D.**

Konsep dasar SDN adalah manajemen jaringan yang lebih efisien dengan memisahkan *control plane* dan *dataplane*. Salah satu protokol yang menunjang konsep SDN adalah OpenFlow. Permasalahan tentang keterbatasan kemampuan OpenFlow *flow table* dalam menyimpan *flow entries* menjadi salah satu motivasi untuk mencegah terjadinya *flow table overflow*, karena sebagian *switch* yang kompatibel dengan protokol OpenFlow dilengkapi dengan kapasitas *flow table* yang terbatas. Salah satu mekanisme dalam manajemen *flow entries* untuk mencegah *flow table overflow* adalah *flow removal*. Penelitian ini melakukan implementasi 2 mekanisme *flow removal* yaitu, mekanisme *flow expiry* dan *flow modification*. Implementasi mekanisme *flow removal* dilakukan pada sistem *stateless server load balancing* berbasis *software defined networking*, yang menyimpan *flow entries* di dalam *flow table* untuk tujuan penggunaan kembali sebuah *flow entry*. Sistem *stateless server load balancing* mengacu kepada *flow state*, sehingga dapat memicu terjadinya *flow table overflow*. Dari hasil pengujian, diperoleh hasil bahwa sistem mampu mencegah *flow table overflow* sehingga tercapai tujuan optimasi *flow entries*. Sebelum mengimplementasikan mekanisme *flow removal*, 53% dari jumlah paket tidak mendapatkan respon dari server. Setelah mengimplementasikan mekanisme *flow removal*, 100% paket dapat terfasilitasi untuk menempati *flow table* dan berhasil mendapatkan pelayanan dari server. Selain itu, penggunaan kembali *flow entries* dapat memangkas nilai *network latency*.

Kata kunci: *Software Defined Network*, OpenFlow, *Flow Table Overflow*, *Flow Removal*, *Server Load Balancing*

## ABSTRACT

**Agung Wahyu Setio Budi, Flow Entries Optimization to Prevent Flow Table Overflow on Server Load Balancing Based on Software Defined Networking**

**Supervisors: Achmad Basuki, S.T, M.MG, Ph.D.**

*Software Defined Networking (SDN) is new paradigm in computer network management to replace a complex conventional network into the flexible and efficient network by separating the control plane and data plane. This concept lead the control plane define each of flow to be written into data plane. However, the space in the flow table is limited resource, requires careful management to prevent flow table overflow. This research discusses a flow entries management to prevent flow table overflow in SDN by proposing two mechanism of flow removal, that is flow expiry and flow modification. The flow expiry remove an expired flow based on its time, meanwhile the flow modification will be triggered whenever the flow tablespace is nearly full. The implementation of our proposal carried out on a stateless server load-balancing application on SDN. The evaluation results show that the system is able to prevent flow table overflow. Before implements our proposal, 53% of total packet did not get response from server. After implements our proposal, 100% of total packet can accommodate by flow table and get a service from server. In addition, the reuse of flow entries can reduce the value of network latency.*

*Keyword: Software Defined Network, OpenFlow, Flow Table Overflow, Flow Removal, Server Load Balancing*

## DAFTAR ISI

PENGESAHAN .....	ii
PERNYATAAN ORISINALITAS .....	iii
KATA PENGANTAR.....	iv
ABSTRAK.....	vi
ABSTRACT .....	vii
DAFTAR ISI .....	viii
DAFTAR TABEL.....	xi
DAFTAR GAMBAR .....	xii
DAFTAR LAMPIRAN .....	xiv
BAB 1 PENDAHULUAN.....	1
1.1 Latar Belakang.....	1
1.2 Rumusan Masalah.....	3
1.3 Tujuan .....	3
1.4 Manfaat.....	3
1.5 Batasan Masalah .....	3
1.6 Sistematika Pembahasan .....	4
BAB 2 LANDASAN KEPUSTAKAAN .....	5
2.1 Kajian Pustaka .....	5
2.2 Dasar Teori .....	7
2.2.1 <i>Software Defined Network</i> .....	7
2.2.2 Protokol OpenFlow .....	8
2.2.3 Mininet .....	9
2.2.4 Ryu.....	10
2.2.5 <i>Load Balancing</i> .....	12
2.2.6 Algoritme <i>Round-Robin</i> .....	12
2.2.7 <i>Flow Table in OpenFlow</i> .....	13
2.2.8 <i>Flow Removal</i> .....	13
2.2.9 <i>Network Latency</i> .....	14
BAB 3 METODOLOGI PENELITIAN .....	15
3.1 Studi Literatur .....	16



3.1.1 <i>Flow Table Overflow</i> pada OpenFlow Switch.....	17
3.1.2 Mencegah <i>Flow Table Overflow</i> .....	17
3.2 Kebutuhan Lingkungan Pengujian .....	19
3.3 Perancangan Lingkungan Uji.....	19
3.3.1 Topologi Jaringan .....	20
3.3.2 Round-Robin <i>Server Load Balancing</i> .....	22
3.3.3 Kapasitas <i>Flow Table</i> .....	23
3.3.4 <i>Client - Server</i> .....	23
3.4 Implementasi .....	23
3.4.1 Lingkungan Uji.....	23
3.4.2 Mekanisme <i>Flow Removal</i> .....	24
3.5 Pengujian .....	25
3.6 Analisis Hasil.....	25
3.7 Kesimpulan.....	26
BAB 4 IMPLEMENTASI .....	27
4.1 Lingkungan Pengujian .....	27
4.1.1 Instalasi Mininet.....	27
4.1.2 Instalasi Ryu Controller .....	27
4.1.3 Topologi Jaringan .....	28
4.1.4 Konfigurasi <i>Client-Server</i> .....	29
4.1.5 Round-Robin <i>Server Load Balancing</i> .....	30
4.1.6 <i>Flow Limit</i> pada OpenFlow Switch .....	32
4.1.7 <i>Traffic Monitoring</i> .....	32
4.2 Mekanisme <i>Flow Removal</i> .....	33
4.2.1 <i>Idle Timeouts</i> .....	33
4.2.2 <i>Flow Modification</i> .....	34
BAB 5 PENGUJIAN DAN ANALISIS.....	36
5.1 Pengujian <i>Round-Robin Server Load Balancing</i> .....	36
5.2 Pengujian Mekanisme <i>Flow Removal</i> .....	37
5.2.1 <i>Idle Timeouts</i> .....	37
5.2.2 <i>Flow Modification</i> .....	39
5.2.3 <i>Idle Timeouts</i> dan <i>Flow Modification</i> .....	41

5.3 Analisis Hasil Keseluruhan Pengujian .....	43
BAB 6 PENUTUP .....	44
6.1 Kesimpulan.....	44
6.2 Saran .....	44
DAFTAR PUSTAKA.....	46
LAMPIRAN A KODE PROGRAM.....	48

## DAFTAR TABEL

Tabel 2.1 Kajian Pustaka .....	5
Tabel 2.2 Kode Program Ryu Sederhana.....	10
Tabel 3.1 Daftar Komponen Mininet .....	20
Tabel 4.1 Kode Program Konfigurasi Topologi Jaringan .....	28
Tabel 4.2 <i>Pseudocode</i> Program <i>Client Emulation</i> .....	29
Tabel 4.3 <i>Pseudocode</i> Sistem <i>Server Load Balancing</i> .....	30
Tabel 4.4 Kapasitas <i>Flow Table</i> .....	32
Tabel 4.5 <i>Pseudocode</i> Program <i>Traffic Monitoring</i> .....	32
Tabel 4.6 Potongan Kode Program Implementasi <i>Idle Timeouts</i> .....	33
Tabel 4.7 <i>Pseudocode</i> Program <i>Flow Modification</i> .....	34

## DAFTAR GAMBAR

Gambar 2.1 Arsitektur <i>Software Defined Networking</i> .....	8
Gambar 2.2 OpenFlow pada Jaringan .....	8
Gambar 2.3 Arsitektur OpenFlow .....	9
Gambar 2.4 Arsitektur Mininet .....	9
Gambar 2.5 Arsitektur Ryu .....	10
Gambar 2.6 Round-Robin <i>Server Load Balancing</i> .....	13
Gambar 2.7 Ilustrasi Penghitungan Nilai <i>Network Latency</i> .....	14
Gambar 3.1 Alur Metodologi Penelitian .....	15
Gambar 3.2 Ilustrasi <i>Flow Table Overflow</i> .....	17
Gambar 3.3 Mekanisme <i>Flow Expiry</i> .....	18
Gambar 3.4 Ilustrasi Mekanisme Pengiriman Pesan Eksplisit ke <i>Controller</i> .....	18
Gambar 3.5 Topologi Jaringan pada Arsitektur SDN .....	21
Gambar 3.6 Pengaturan <i>controller</i> di Mininet .....	21
Gambar 3.7 Pengaturan <i>preferences</i> di Mininet .....	22
Gambar 3.8 Ilustrasi Round-Robin <i>Server Load Balancing</i> .....	22
Gambar 3.9 Ilustrasi <i>Client - Server</i> .....	23
Gambar 3.10 Mekanisme <i>Flow Removal</i> .....	24
Gambar 4.1 Instalasi Mininet .....	27
Gambar 4.2 Instalasi Ryu <i>Controller</i> .....	28
Gambar 4.3 Konfigurasi <i>Server</i> .....	30
Gambar 4.4 Menjalankan <i>controller</i> Ryu .....	31
Gambar 4.5 Konfigurasi <i>Flow Limit</i> pada OpenFlow <i>Switch</i> .....	32
Gambar 5.1 Hasil <i>Packet Capture</i> Round-Robin <i>Server Load Balancing</i> .....	36
Gambar 5.2 Grafik Kondisi <i>Flow Table</i> Hasil Pengujian <i>Idle Timeouts</i> 60 detik ....	37
Gambar 5.3 <i>Flow Entries</i> Mencapai Batas Maksimal .....	38
Gambar 5.4 Persentase <i>Packet Loss</i> .....	38
Gambar 5.5 Grafik Kondisi <i>Flow Table</i> Hasil Pengujian <i>Flow Modification</i> .....	39
Gambar 5.6 Persentase <i>Packet Loss</i> .....	40
Gambar 5.7 Grafik <i>Network Latency</i> pada Implementasi <i>Flow Modification</i> .....	40
Gambar 5.8 Grafik Hasil Pengujian 2 Mekanisme <i>Flow Removal</i> .....	41

Gambar 5.9 Perbandingan <i>Packet Loss</i> Sebelum dan Sesudah Optimasi.....	41
Gambar 5.10 Grafik <i>Network Latency</i> pada Implementasi 2 Mekanisme <i>Flow Removal</i> .....	42

## DAFTAR LAMPIRAN

LAMPIRAN A KODE PROGRAM.....	48
------------------------------	----

# BAB 1 PENDAHULUAN

## 1.1 Latar Belakang

*Software-Defined Networking* (SDN) merupakan sebuah jaringan komputer terkini yang membawa arsitektur baru dengan memisahkan *network control* dan *forwarding function*. *Network control* atau biasa dikenal dengan *control plane*, memiliki tugas untuk menentukan bagaimana sebuah paket akan diteruskan. Sedangkan *forwarding function* atau biasa disebut dengan *data plane* memiliki tugas untuk meneruskan paket (Al-Najjar, et al., 2016). Salah satu protokol yang digunakan untuk berkomunikasi antara *control plane* dengan *data plane* pada arsitektur *Software-Defined Networking* adalah protokol OpenFlow (Karim, et al., 2019). Dalam sebuah penelitian yang berjudul *Openflow Timeouts Demystified*, Adam Zarek mengatakan bahwa hal yang tidak kalah penting dan perlu dilakukan selain mengatur *flow table* adalah mengatur kapan dan bagaimana menghapus *flow entry*, karena sebagian besar *switch* yang kompatibel dengan protokol OpenFlow dilengkapi dengan ruang *flow table* yang terbatas (Zarek, 2012). Tidak hanya itu, Yang dan Riley juga mengatakan dalam penelitiannya yang berjudul *Machine Learning Based Proactive Flow Entry Deletion for OpenFlow* bahwa kapasitas dari *flow table* terbatas, karena menggunakan *Ternary Content Addressable Memory*(TCAM) yang memiliki konsumsi daya tinggi, biaya *chip* yang mahal, serta kendala *silicon area* (Yang & Riley, 2018).

Sebagai *forwarding function* yang bertugas meneruskan paket, *data plane* mengacu pada *flow state* dari *flow entries* yang disimpan di dalam *flow table*. Ketika *data plane* mengacu pada *flow state*, maka suatu saat terdapat kemungkinan bahwa jumlah *flow entries* yang ada di dalam *flow table* akan mencapai batas maksimalnya. Sedangkan keterbatasan kemampuan *flow table* dalam menyimpan *flow entries* dapat menyebabkan *flow table overflow*. *Flow table overflow* merupakan sebuah kondisi dimana *flow entries* telah mencapai batas maksimalnya, sehingga *active flows* harus digusur untuk dapat mengakomodasi *flow entry* yang baru. Kondisi *flow table overflow* dapat menyebabkan beban *controller* meningkat, karena mengharuskan untuk berulang kali memperbarui *flow table*. Dengan meningkatnya beban *controller* dapat mengganggu aliran yang ada, sehingga menyebabkan menurunnya kinerja jaringan (Guo, et al., 2018).

Untuk menghindari *flow table overflow*, maka diperlukan sebuah langkah untuk melakukan manajemen *flow entries* yang ada di dalam *flow table*. Dalam teori OpenFlow *specification*, metode yang digunakan untuk manajemen *flow table* berupa penghapusan *flow entries* disebut mekanisme *flow removal*. Mekanisme *flow removal* sendiri dapat dibagi menjadi 2, pertama adalah *flow expiry* dan yang kedua adalah *flow modification message*. Mekanisme *flow expiry* akan berjalan dengan cara menghapus *flow entry* ketika nilai *timeouts* yang diberikan telah berakhir. Sedangkan mekanisme *flow modification message* akan

berjalan ketika *switch* mengirimkan pesan kontrol eksplisit kepada *controller* untuk melakukan penghapusan terhadap suatu *flow entry*.

Adam Zarek mengatakan dalam penelitiannya yang berjudul *Openflow Timeouts Demystified* bahwa metode *flow timeouts*, baik *idle timeouts* maupun *hard timeout* untuk mengatur *flow expiry* lebih efektif dan *scalable* daripada mengandalkan sepenuhnya pada *flow modification message*. Namun ia juga mengusulkan untuk melakukan kombinasi antara kedua metode *flow removal* tersebut (Zarek, 2012). Penelitian lain yang berjudul *Machine Learning Based Proactive Flow Entry Deletion for OpenFlow* mengkombinasikan metode *flow modification message* dan *machine learning* dengan harapan dapat mengoptimalkan *flow table* yang terbatas tanpa membebani kinerja *controller* (Yang & Riley, 2018). Selain itu terdapat juga penelitian dengan judul *A Zero Flow Entry Expiration Timeout P4 Switch* yang melakukan penghapusan *flow entry* berdasarkan deteksi paket FIN dan RST untuk protokol TCP (He, et al., 2018).

Dari beberapa penelitian yang telah dibahas di paragraf sebelumnya, permasalahan keterbatasan *flow table* telah banyak mengundang peneliti untuk menemukan solusinya. Namun penelitian yang telah ada hanya berfokus kepada arsitektur SDN yang memiliki kapasitas terbatas dalam menyimpan *flow entries*, sehingga penelitian sebelumnya fokus untuk manajemen *flow entries* di dalam *flow table* agar tidak terjadi *overflow*. Zehua Guo menyarankan perlu dipelajari dan diteliti lagi tentang dampak dari pengaturan *idle-timeout* dalam mekanisme *flow expiry*, misalnya mencari tahu ambang batas yang baik yang tidak membebani CPU dari *switch* dan sekaligus menghindari *overflow* pada *flow table* (Guo, et al., 2018).

Sistem *server load balancing* yang diterapkan pada arsitektur SDN tentunya menjadi salah satu pemicu dari *flow table overflow*, sehingga permasalahan keterbatasan kapasitas *flow table* memungkinkan untuk diteliti lebih lanjut, terlebih jika diterapkan pada sistem *server load balancing* berbasis *software-defined networking*. Dari pembahasan di paragraf sebelumnya, penelitian ini memadukan antara sistem *server load balancing* dan implementasi pengaturan *flow entries* di dalam *flow table*, tentunya dengan mempertimbangkan permasalahan yang sudah disebutkan di paragraf sebelumnya, yaitu keterbatasan kapasitas *flow table* pada sistem *server load balancing* berbasis *software defined networking*. Maka dari itu penelitian ini mengajukan sebuah ide Optimasi *Flow Entries untuk Mencegah Flow Table Overflow pada Server Load Balancing Berbasis Software Defined Networking*.

Dari permasalahan yang telah dipaparkan dalam paragraf sebelumnya serta batasan pada penelitian yang telah ada, dilakukan implementasi 2 mekanisme *flow removal* pada sistem *server load balancing* berbasis *software-defined networking*. Protokol OpenFlow digunakan untuk mengatur komunikasi antara *control plane* dan *data plane*. *Control plane* mengatur logika dari sistem *load balancing*, mengatur *flow expiry* dan *flow modification message*, sedangkan *data plane* menyimpan informasi dari *flow entries*. Pengujian dari implementasi mekanisme *flow removal* dilakukan dengan mengirimkan paket *request* dari



beberapa *source* yang berbeda ke sebuah *server*. Dari serangkaian implementasi, pengujian, dan analisis diharapkan mampu mengetahui pengaruh dari implementasi mekanisme *flow removal* terhadap sistem *server load balancing* serta mengatasi permasalahan *flow table* untuk mencegah *overflow*, sehingga dapat mengoptimalkan kinerja algoritme *round-robin* pada sistem *server load balancing*.

## 1.2 Rumusan Masalah

Berdasarkan latar belakang yang telah dijelaskan, maka rumusan masalah pada penelitian kali ini adalah sebagai berikut:

1. Bagaimana mengimplementasikan mekanisme *flow removal* dengan pengaturan *idle timeouts* dan *flow modification* ?
2. Bagaimana hasil analisis kondisi *flow entries* setelah mengimplementasikan mekanisme *flow removal* untuk mencegah *flow table overflow* ?
3. Bagaimana kinerja sistem *server load balancing* dengan optimasi *flow entries* ?

## 1.3 Tujuan

Keterbatasan kemampuan OpenFlow *flow table* dalam menyimpan *flow entries* dapat menyebabkan terjadinya *flow table overflow*, karena sebagian besar *switch* yang kompatibel dengan protokol OpenFlow memiliki kapasitas *flow table* yang terbatas. Mengacu kepada permasalahan tersebut, tujuan penelitian ini adalah mengoptimalkan *flow entries* untuk mencegah *flow table overflow* dengan mekanisme *flow removal*.

## 1.4 Manfaat

Manfaat yang diharapkan dari penelitian ini adalah sebagai berikut:

1. Bagi penulis, memberikan solusi terhadap permasalahan *flow table overflow* khususnya pada sistem *server load balancing* berbasis *software-define networking*.
2. Bagi pembaca, memberikan wawasan mengenai bagaimana mencegah *flow table overflow* dengan menerapkan mekanisme *flow removal* khususnya pada sistem *server load balancing* berbasis *software-defined networking*.

## 1.5 Batasan Masalah

Penulis memberikan batasan terhadap permasalahan yang akan dibahas agar pembahasan mengenai masalah dapat terarah dengan baik serta tidak menyimpang dari pokok permasalahan. Batasan permasalahan tersebut adalah sebagai berikut :

1. Penelitian ini menggunakan topologi dengan 1 OpenFlow *switch*, 1 *controller*, 3 *server*, dan 1 *client* .
2. Protokol yang digunakan untuk berkomunikasi antara *data plane* dan *control plane* pada arsitektur *software-defined networking* adalah protokol OpenFlow1.3.

3. Pengujian dilakukan pada sistem *server load balancing* algoritme *round-robin*.
4. Pengujian dilakukan menggunakan skenario pengiriman 100 paket HTTP *request* setiap 0.5 detik dengan kapasitas *flow table* 50 *flow entries*.
5. Analisis kinerja sistem *server load balancing* dilakukan dengan melihat kondisi *flow table entries* dan *network latency*.

## 1.6 Sistematika Pembahasan

Penjelasan singkat mengenai struktur dari penulisan pada masing-masing bab adalah sebagai berikut.

### **BAB 1      PENDAHULUAN**

Bab ini menjelaskan latar belakang, rumusan masalah, tujuan penelitian, manfaat penelitian, penentuan batasan penelitian serta sistematika penulisan dari penelitian.

### **BAB 2      LANDASAN KEPUSTAKAAN**

Bab ini mencakup kajian pustaka dari penelitian sebelumnya yang memiliki hubungan dengan penelitian ini. Bab ini juga menyajikan teori dan konsep dari sumber penelitian yang nantinya akan digunakan sebagai panduan dalam penelitian.

### **BAB 3      METODOLOGI**

Bab ini memberikan pemaparan tentang metodologi yang digunakan pada penelitian ini. Bab metodologi berisi tentang penjelasan dari langkah penelitian yang terdiri dari kajian pustaka, perancangan lingkungan pengujian, implementasi sistem, pengujian dan analisis, serta penarikan kesimpulan.

### **BAB 4      IMPLEMENTASI**

Bab implementasi berisi tahap implementasi yang dilakukan untuk membangun sistem *round-robin server load balancing* berbasis *software-defined networking* berkorelasi dengan pengaturan *idle timeouts* serta mekanisme penghapusan *flow entry* secara pro aktif dengan mekanisme *flow modification message*.

### **BAB 5      PENGUJIAN DAN ANALISIS**

Bab ini menjelaskan mekanisme pengujian dan analisis hasil dari implementasi yang telah dilakukan sebelumnya.

### **BAB 6      PENUTUP**

Bab ini memberikan pemaparan tentang pengambilan kesimpulan hasil penelitian dari implementasi sistem, serta pemberian saran pengembangan agar dapat dijadikan referensi pada pengembangan di kemudian hari.

## BAB 2 LANDASAN KEPUSTAKAAN

### 2.1 Kajian Pustaka

Kajian pustaka menampilkan kajian dari penelitian terdahulu yang memiliki hubungan dengan permasalahan pada penelitian ini. Penelitian tersebut berisi teori dan metode yang digunakan pada studi kasus dengan objek yang sama. Tabel Kajian Pustaka dapat dilihat pada Tabel 2.1.

**Tabel 2.1 Kajian Pustaka**

No	Judul, Nama Penulis, Tahun dan Objek	Perbedaan		
		Persamaan	Penelitian Terdahulu	Rencana Penelitian
1	<b>Judul :</b> Openflow Timeouts Demystified (Zarek, 2012)  <b>Objek :</b> OpenFlow controller	Manajemen <i>flow table</i> untuk mencegah <i>flow table overflow</i>	Mengimplementasikan mekanisme pengaturan <i>flow timeouts</i> dan <i>flow modification message</i> untuk mengatasi permasalahan <i>overflow</i> . Belum ada aturan yang efisien untuk melakukan penghapusan <i>flow</i>	Mengimplementasikan mekanisme pengaturan <i>idle timeouts</i> dan <i>flow modification message</i> untuk mengoptimalkan <i>round-robin server load balancing</i> berbasis <i>software defined networking</i>
2	<b>Judul :</b> A Zero Flow Entry Expiration Timeouts P4 Switch (He, et al., 2018)  <b>Objek :</b> OpenFlow controller	Manajemen <i>flow table</i> untuk menghindari <i>Overflow</i> dan <i>overhead</i> pada controller	Mengimplementasikan <i>last packet detection</i> (TCP) FIN dan RST dan <i>zero-timeouts</i> untuk menghapus <i>flow entry</i> dari <i>flow table</i> pada P4 Switch	Mengimplementasikan pengaturan <i>idle timeouts</i> dan <i>flow modification message</i> untuk menentukan mekanisme <i>flow removal</i> dari setiap flow pada <i>Open Flow Swicth</i>

3	<b>Judul :</b> Machine Learning Based Proactive Flow Entry Deletion for OpenFlow (Yang & Riley, 2018)  <b>Objek :</b> OpenFlow controller	Manajemen <i>flow table</i> untuk menghindari <i>Overflow</i> (berdasarkan <i>proactive flow entry deletion</i> )	Mengimplemenasikan <i>machine learning</i> untuk mempelajari pola dari setiap <i>flow entry</i> sebelum menentukan keputusan untuk menghapus <i>flow entry</i> menggunakan mekanisme pesan kontrol eksplisit ( <i>Proactive Deletion</i> )	Mengimplemenasikan pengaturan <i>flow modification message</i> ( <i>Proactive Deletion</i> ) untuk menghapus <i>flow entries</i> yang tidak ada aktivitas
---	---	---	--	---

Pada penelitian sebelumnya yang berjudul *Openflow Timeouts Demystified* meneliti tentang pengaruh dari nilai *timeouts* yang diberikan terhadap kondisi *flow entries* di dalam *flow table*. Selain meneliti tentang *timeouts* dalam penelitian ini Adam Zarek juga menggunakan algoritme FIFO dan *random replacement* untuk menentukan *flow entry* mana yang akan dihapus menggunakan metode *flow modification message*. Meskipun dalam penelitian ini Adam Zarek mengimplementasikan *hybrid flow table management* atau menggunakan mekanisme *flow expiry* dan *flow modification message*, diperoleh hasil bahwa belum ada mekanisme yang efisien untuk melakukan penghapusan *flow entries*.

Penelitian kedua telah disampaikan pada tabel diatas yang berjudul *A Zero Flow Entry Expiration Timeouts P4 Switch* mengimplementasikan mekanisme *flow expiry* dengan nilai *timeouts* 0. Disamping itu penelitian ini juga menambahkan mekanisme untuk mendeteksi paket FIN dan RST pada protokol TCP untuk melakukan penghapusan *flow entry* yang telah selesai melakukan komunikasi atau memutus koneksi dengan mengirimkan paket RST. Cheng-Hung menerapkan metode deteksi paket ini karena menyadari bahwa nilai *timeouts* yang besar dapat menyebabkan kondisi *flow table* penuh, sedangkan nilai *timeouts* yang kecil dapat menyebabkan beban yang berat kepada controller, karena harus setiap saat memperbarui *flow entries*. Dalam penelitian ini dihasilkan bahwa metode *zero timeouts* yang dikombinasikan dengan deteksi paket FIN dan RST dapat mengurangi beban controller dan mencegah terjadinya *flow table overflow*.

Penelitian ketiga pada tabel kajian pustaka yang berjudul *Machine Learning Based Proactive Flow Entry Deletion for OpenFlow* menerapkan metode *flow modification message* atau biasa disebut metode *proactive deletion* dengan

mengirimkan pesan spesifik kepada *controller*. Dalam penelitian ini *proactive deletion* dikombinasikan dengan *machine learning* yang menghapus *flow entry* dengan mempelajari perilaku sebelumnya yang diperoleh dari statistik *flow removed message*, kemudian memprediksi *flow entry* dengan durasi paling pendek dari data sebelumnya. Selain menggunakan *machine learning*, metode *proactive deletion* ini juga diterapkan dengan algoritme FIFO dan *random deletion*. Penelitian ini berfokus kepada menemukan aturan *proactive deletion* yang paling efisien, dan hasilnya menunjukkan dengan komparasi dari beberapa algoritme dan *machine learning* diperoleh hasil bahwa *machine learning* memiliki nilai *table missed* dan *controller overhead* paling kecil serta mengatasi permasalahan *flow table overflows*.

Dari beberapa penelitian diatas yang berkaitan dengan rencana penelitian ini, penulis mengimplementasikan *hybrid flow table management* dengan menerapkan 2 mekanisme *flow removal* yaitu *flow expiry* dan *flow modification message* yang memiliki tujuan sama dengan penelitian sebelumnya yaitu manajemen *flow entries* untuk mencegah *flow table overflows*. Perbedaan dari rencana penelitian ini dengan penelitian sebelumnya adalah pada penerapannya. Rencana penelitian kali ini diterapkan pada sistem *stateless server load balancing* berbasis *software defined networking*.

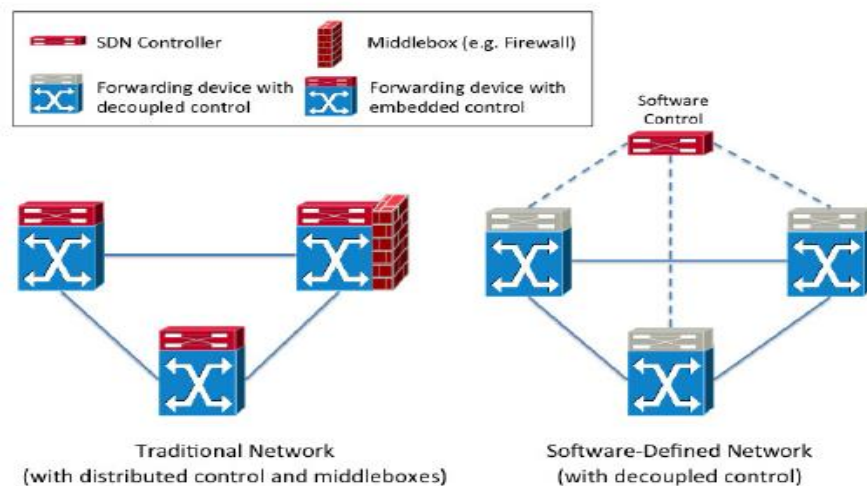
## 2.2 Dasar Teori

Pada sub dasar teori, dijelaskan berbagai teori yang mendukung penelitian ini, yaitu teori untuk mengimplementasikan mekanisme *flow removal* yang terintegrasi dengan sistem *server load balancing* berbasis *software defined networking*.

### 2.2.1 Software Defined Network

*Software-Defined Networking* (SDN) adalah sebuah paradigma baru dalam arsitektur jaringan komputer. SDN memiliki konsep mendesain, mengatur, dan mengontrol jaringan komputer dengan cara memisahkan *control plane* dan *data plane*. *Control plane* memiliki tugas untuk menentukan bagaimana sebuah paket akan diteruskan, sedangkan *data plane* memiliki tugas untuk meneruskan paket (Ardy, et al., 2018).

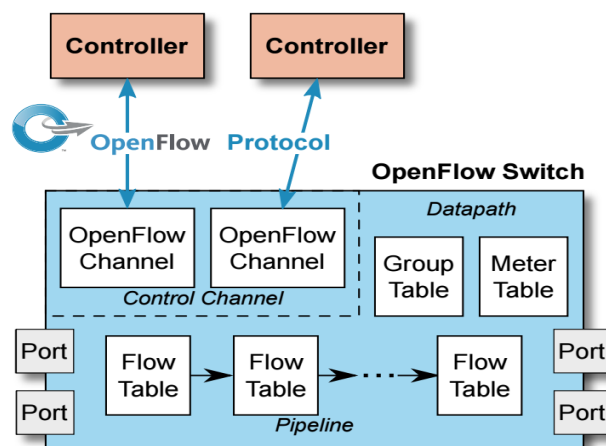
Menurut Nadeu & Gray dalam bukunya yang berjudul "SDN: Software Defined Network", SDN merupakan sebuah pendekatan arsitektur jaringan komputer yang melakukan penyederhanaan operasi sekaligus optimasi pada jaringan. Untuk mencapai hal tersebut, diimplementasikan sebuah *logical network control* yang terpusat (dalam hal ini disebut SDN *controller*) yang memfasilitasi, mengatur, dan memediasi antara perangkat jaringan yang ingin berkomunikasi dengan aplikasi (Nadeau & Gray, 2013).



**Gambar 2.1 Arsitektur *Software Defined Networking***

### 2.2.2 Protokol OpenFlow

Di dalam arsitektur *Software-Defined Networking* (SDN) diperlukan sebuah protokol komunikasi yang digunakan untuk berkomunikasi antara *control plane* dengan *data plane* (Karim, et al., 2019). OpenFlow merupakan protokol standar *Open Network Foundation* (ONF) untuk berkomunikasi antara *control plane* dan *data plane*. OpenFlow controller merupakan bagian dari *network control* SDN yang memberikan perintah ke OpenFlow switch. Sebuah OpenFlow switch mengelola satu atau beberapa *flow table* untuk menangani penerusan paket ke tujuan (McKeown, et al., 2008).



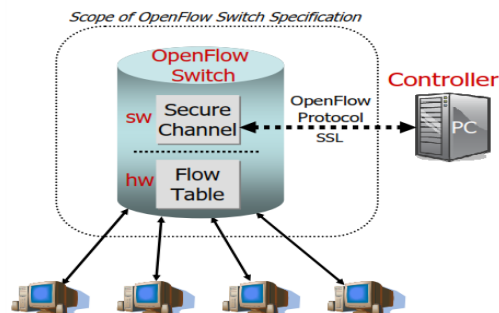
**Gambar 2.2 OpenFlow pada Jaringan**

[Sumber: [opennetworking.org](http://opennetworking.org) (2015)]

OpenFlow memungkinkan untuk memprogram switch pada *data plane*, sehingga melalui antarmuka OpenFlow ini administrator dapat melakukan kontrol secara langsung terhadap lalu lintas paket pada *forwarding function* atau *data*

*plane*. OpenFlow mendefinisikan infrastruktur *Application Programmatic Interface* (API) standar dan *flow-based forwarding* yang memungkinkan *controller* dapat mengarahkan fungsi *switch* melalui saluran yang aman (Sudiyatmoko, et al., 2016).

Protokol OpenFlow bekerja secara terpusat, yang mana ketika sebuah OpenFlow *switch* menerima paket baru yang unik dan tidak memiliki informasi yang sama dengan *flow entries* yang telah ada di dalam *flow table*, maka OpenFlow *switch* akan mengirimkan paket tersebut ke *controller*, kemudian *controller* akan menangani paket tersebut. *Controller* dapat menentukan keputusan untuk membuang paket tersebut atau menambahkan ke dalam *flow table* (Negara, et al., 2018).

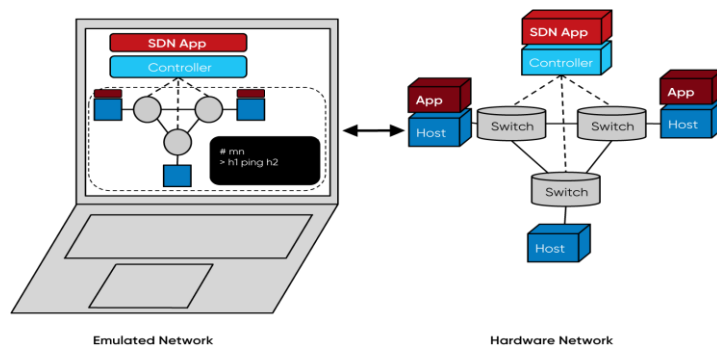


**Gambar 2.3 Arsitektur OpenFlow**

[Sumber: (McKeown, et al., 2008)]

### 2.2.3 Mininet

Mininet merupakan sebuah *emulator* jaringan yang mendukung konsep SDN. Mininet memiliki beberapa komponen yang dapat memvisualisasikan perangkat jaringan diantaranya *host*, *controller*, *switch* sehingga dapat digunakan untuk membangun topologi jaringan sesuai dengan kebutuhan (Mininet Team, 2016). Dalam hal ini mininet berfungsi untuk membuat virtualisasi dari sebuah jaringan. Gambar 2.4 merupakan arsitektur dari mininet. Mininet memanfaatkan *virtual software switch OpenvSwitch* dalam mendukung pembuatan topologi berbasis protokol OpenFlow.

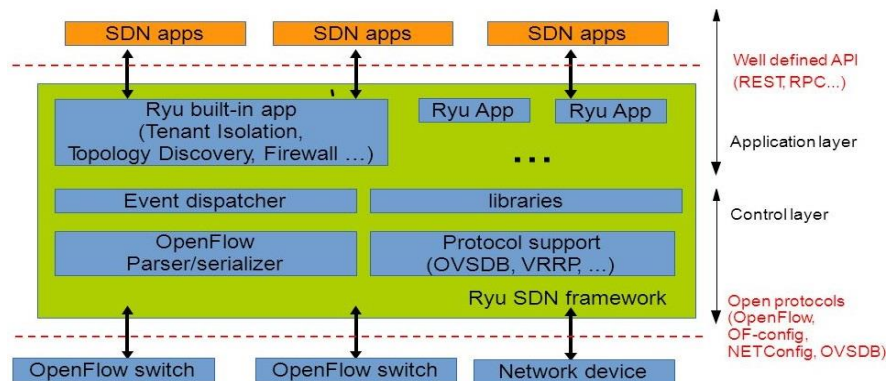


**Gambar 2.4 Arsitektur Mininet**

[Sumber: [opennetworking.org/mininet](http://opennetworking.org/mininet)]

## 2.2.4 Ryu

Ryu merupakan sebuah *framework* berbasis Python pada jaringan *software-defined networking*. Ryu dapat dibangun dengan menggunakan bahasa pemrograman Python maupun dengan cara pengiriman pesan JSON dari *API* yang disediakan. Ryu juga dapat membuat dan mengirim pesan modifikasi OpenFlow yang mengacu pada kejadian asinkron seperti *flow\_removed*, *parse* dan menangani paket yang masuk (Pemberton, et al., 2014). Gambar 2.3 merupakan arsitektur dari *ryu controller*.



**Gambar 2.5 Arsitektur Ryu**

[Sumber: Isaku Yamahata (2013)]

Dalam dokumentasi Ryu, telah disampaikan ketika ingin mengatur perangkat jaringan seperti *switch*, *router*, dll. dengan menggunakan kontroler Ryu, maka kita hanya perlu menuliskan aplikasi Ryu sendiri dengan bahasa pemrograman *python*. Aplikasi / program kita akan dijalankan oleh Ryu dengan melakukan konfigurasi menggunakan protokol OpenFlow. Dibawah ini adalah aplikasi sederhana untuk memprogram *switch* yang mengatur *received packets* untuk diteruskan ke semua port (Nippon Telegraph and Telephone Corporation Revision 56e8fb3f, 2014).

**Tabel 2.2 Kode Program Ryu Sederhana**

First Application	
1	from ryu.base import app_manager
2	from ryu.controller import ofp_event
3	from ryu.controller.handler import MAIN_DISPATCHER
4	from ryu.controller.handler import set_ev_cls
5	from ryu.ofproto import ofproto_v1_0
6	
7	class L2Switch(app_manager.RyuApp):
8	OFP_VERSIONS = [ofproto_v1_0.OFP_VERSION]
9	
10	def __init__(self, *args, **kwargs):
11	super(L2Switch, self).__init__(*args, **kwargs)
12	
13	@set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
14	def packet_in_handler(self, ev):
15	msg = ev.msg



16	dp = msg.datapath
17	ofp = dp.ofproto
18	ofp_parser = dp.ofproto_parser
19	
20	actions = [ofp_parser.OFPACTIONOutput(ofp.OFPP_FLOOD)]
21	out = ofp_parser.OFPPacketOut(
22	datapath=dp, buffer_id=msg.buffer_id,
23	in_port=msg.in_port,
24	actions=actions)
25	dp.send_msg(out)

Kode program pada tabel 2.2 diperoleh dari dokumentasi Ryu, yang mencontohkan bagaimana menuliskan kode program dalam kontroler Ryu. Berikut ini adalah penjelasan dari kode sederhana pada tabel 2.2.

- *Method* 'packet\_in\_handler' ditambahkan ke kelas L2Switch. *Method* ini dipanggil ketika Ryu menerima pesan packet\_in OpenFlow. Caranya menggunakan dekorator 'set\_ev\_cls'. Dekorator ini memberi tahu Ryu kapan fungsi yang didekorasi harus dipanggil.
- Argumen pertama dari dekorator 'ofp\_event.EventOFPPacketIn' menunjukkan jenis *event* apa / kapan fungsi ini harus dipanggil. Fungsi ini akan dipanggil setiap kali Ryu mendapat pesan packet\_in.
- Argumen kedua 'MAIN\_DISPATCHER' menunjukkan keadaan *switch*. Menggunakan 'MAIN\_DISPATCHER' sebagai argumen kedua berarti fungsi ini dipanggil hanya setelah negosiasi antara kontroler Ryu dan *switch* selesai.
- Di bagian pertama dari fungsi 'packet\_in\_handler' terdapat beberapa variabel diantaranya
  - o 'ev.msg' adalah objek yang mewakili struktur data 'packet\_in'.
  - o 'msg.dp' adalah objek yang merepresentasikan 'datapath' (*switch*).
  - o 'dp.ofproto' dan 'dp.ofproto\_parser' adalah objek yang mewakili protokol OpenFlow yang dinegosiasikan Ryu dan *switch*.
  - o Kelas 'OFPACTIONOutput' digunakan dengan pesan 'packet\_out' untuk menentukan *port switch* yang ingin di kirim paket. Aplikasi ini menggunakan *flag* 'OFPP\_FLOOD' untuk menunjukkan bahwa paket tersebut harus dikirim pada semua *port*.
  - o Kelas 'OFPPacketOut' digunakan untuk membangun pesan 'packet\_out'.
  - o Jika memanggil *method* 'send\_msg' kelas 'Datapath' dengan objek kelas pesan OpenFlow, Ryu membangun dan mengirim format data on-wire ke *switch*.

Tidak hanya yang dicontohkan dalam program di atas, Ryu juga menyediakan beberapa *library*, komponen OpenFlow *controller*, dan juga API yang bisa digunakan sesuai dengan kebutuhan, termasuk kebutuhan dalam penelitian ini

yaitu sebuah *virtual load balancing* yang ada di *controller* dan sistem *flow monitoring*.

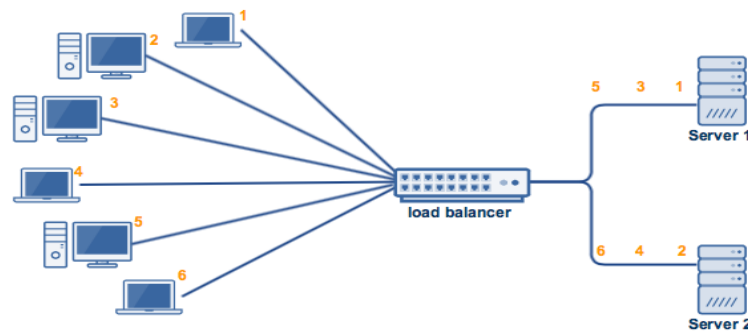
### 2.2.5 Load Balancing

*Load Balancing* adalah sebuah metode untuk membagi beban *traffic* pada jalur koneksi yang ada secara seimbang. Penelitian sebelumnya menyatakan bahwa *load balancing* merupakan sebuah strategi yang digunakan untuk menyebarkan *traffic* yang masuk pada *server* yang tersedia, sehingga *server* dapat menangani *request* dari *client* dengan memberikan *response* yang lebih cepat (Rahman, et al., 2014). Sebuah jaringan dengan distribusi beban yang merata dapat membantu melakukan optimasi terhadap *resource* yang tersedia. Pemeratan distribusi beban memiliki tujuan untuk meminimalisir *response time*, memaksimalkan *throughput*, serta mencegah terjadinya *overload* pada jaringan (Zha, et al., 2010). Dapat dikatakan bahwa *load balancer* berfungsi untuk memangkas *delay* yang tidak diperlukan serta mencegah *congestion* (Boero, et al., 2016).

Implementasi algoritme *load balancing* pada jaringan tradisional terhalang oleh keterbatasan perangkat keras yang menjadi *bottleneck* di dalam sistem. Selain keterbatasan *hardware*, implementasi *load balancing* pada jaringan tradisional memerlukan biaya mahal serta kompleksitas yang tinggi (Zhong, et al., 2016). Untuk mengatasi permasalahan ini, SDN menawarkan berbagai keunggulan untuk melakukan implementasi *load balancing*. Pemisahan antara *control plane* dan *data plane* pada arsitektur SDN memungkinkan untuk melakukan konfigurasi *software* pada *controller* untuk melakukan *load balancing* yang lebih efisien (Zhong, et al., 2016). SDN *controller* yang dipadukan dengan protokol OpenFlow dapat berperan sebagai *load balancer*. Sebagai *load balancer*, *controller* SDN mengumpulkan data yang diperlukan untuk memilih *server* tujuan. Kemudian *server* akan mengganti aturan (*flow table entries*) pada SDN *switch* dengan menggunakan pesan *flow modification* berdasarkan hasil penentuan keputusan. Dalam hal ini SDN *controller* memiliki kekuasaan penuh untuk mengatur *switch* (Al-Najjar, et al., 2016).

### 2.2.6 Algoritme Round-Robin

Algoritme *round-robin* merupakan algoritme yang sederhana dan paling banyak digunakan pada sistem *load balancing*. Algoritme *round-robin* berjalan dengan cara membagi beban secara urut dan bergilir dari satu *server* ke *server* yang lain. Konsep dasar dari algoritme *round-robin* ini adalah menggunakan *time sharing*, pada intinya algoritme ini memproses antrian yang ada secara bergiliran (Ellrod, 2010). Gambar 2.6 merupakan arsitektur dari *load balancing round-robin*.



**Gambar 2.6 Round-Robin Server Load Balancing**

[Sumber: jscape.com]

Dalam sistem *server load balancing*, algoritme round-robin dapat membantu untuk membagi beban dengan mengarahkan setiap *request* dari *source* yang berbeda ke *server* secara bergiliran dan sirkular. Algoritme round-robin yang diterapkan pada sistem *stateless server load balancing* berbasis *software defined networking* dapat menyimpan sebuah *state* dalam sebuah *flow entries*. Penggunaan kembali *flow entries* yang ada di dalam *flow table* OpenFlow switch dapat meminimalisir *delay* dan *response time* dari sebuah *server*.

### 2.2.7 Flow Table in OpenFlow

OpenFlow switches memiliki kapasitas *flow table* yang terbatas. Keterbatasan ruang penyimpanan ini dikarenakan *flow table* terbuat dari *Ternary Content Addressable Memory* (TCAM). TCAM memiliki konsumsi daya yang tinggi dan jejak silikon yang besar, sehingga memerlukan biaya chip yang mahal dan berujung kepada keterbatasan dalam menyimpan *flow entries* (Zarek, 2012).

Keterbatasan ruang penyimpanan untuk memfasilitasi *flow entries* di dalam OpenFlow switch dapat menyebabkan kondisi *flow table overflow*. *Flow table overflow* adalah sebuah kondisi dimana *flow table* tidak lagi dapat memfasilitasi *flow entry* baru. Ketika terjadi *flow table overflow*, maka perlu dilakukan sebuah langkah pengusuran kepada *active flow* untuk dapat memfasilitasi *flow entry* yang baru. Namun untuk meminimalisir gangguan terhadap kinerja *flow entry* dan penambahan beban *controller*, perlu dilakukan mekanisme yang tepat dalam menangani kondisi *flow table overflow* (Guo, et al., 2018).

### 2.2.8 Flow Removal

Di dalam protokol OpenFlow terdapat 2 mekanisme untuk melakukan penghapusan *flow entry* dari sebuah *flow table*. Mekanisme yang pertama adalah menggunakan *flow expiry*, yang mana mekanisme ini dijalankan secara independen oleh switch dari *controller* berdasarkan kondisi dan konfigurasi dari *flow entry*. Setiap *flow entry* memiliki *idle timeout* dan *hard timeout* yang saling berkaitan. Jika keduanya tidak bernilai 0 maka switch harus mencatat waktu

kedatangan dari *flow entry*, karena nantinya mungkin dibutuhkan untuk melakukan penghapusan.

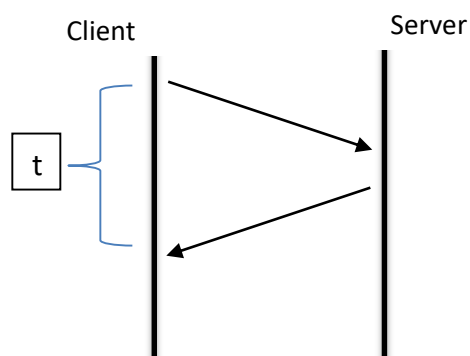
Pemberian nilai pada *hard timeout* menyebabkan *flow entry* dihapus dari *flow table* setelah mencapai nilai atau waktu yang diberikan, tanpa peduli berapa banyak paket yang cocok. Pemberian nilai pada *idle timeout* menyebabkan *flow entry* dihapus dari *flow table* ketika tidak ada paket yang cocok selama nilai atau waktu yang diberikan. Dalam hal ini *switch* harus mengimplementasikan *flow expiry* dan menghapus *flow entry* dari *flow table* ketika salah satu nilai dari *timeout* baik *idle* atau *hard* telah terlampaui.

Mekanisme kedua dalam melakukan penghapusan *flow entry* adalah menggunakan *flow modification message OFPFC\_DELETE* atau *OFPFC\_DELETE\_STRICT*. Mekanisme ini tergolong mekanisme penghapusan yang pro aktif karena tanpa menunggu *flow expiry*, *switch* akan mengirimkan pesan kontrol eksplisit kepada *controller* untuk melakukan penghapusan (Open Networking Foundation, 2012).

### 2.2.9 Network Latency

*Network latency* merupakan waktu yang diperlukan untuk mengirim packet dari ujung jaringan, ke ujung yang lain. Dalam beberapa pengertian *latency* juga bisa dikatakan sebagai *delay* yang merupakan jeda waktu yang dibutuhkan dalam pengiriman paket dari pengirim ke penerima melalui sebuah jaringan. Di dalam sebuah situs web *keycdn* mengatakan bahwa *latency* menentukan seberapa cepat konten dalam jaringan dapat di transfer dari klien ke server dan kembali lagi (KeyCDN Company, 2018).

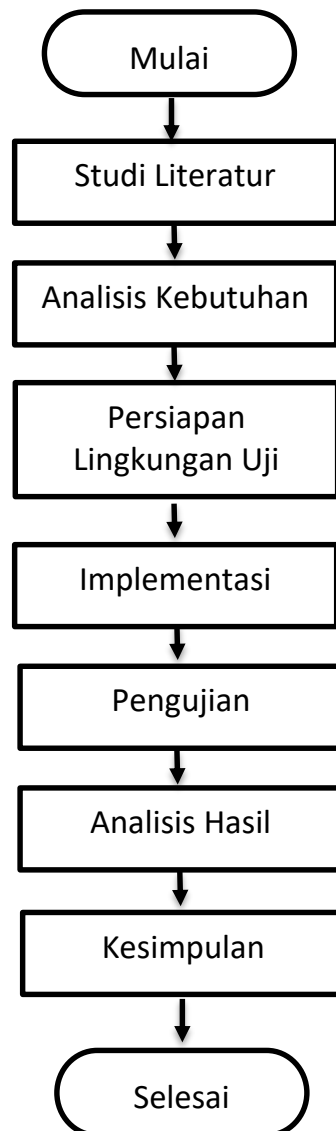
Dalam penelitian ini, *network latency* merupakan salah satu parameter uji yang digunakan untuk melakukan pengujian kinerja *server load balancing*. Nilai *network latency* yang digunakan dalam pengujian adalah total waktu yang dibutuhkan oleh klien ketika mengirim *request* ke *server* hingga *response* dari *server* sampai di *client*. Ilustrasi penghitungan nilai *network latency* dapat dilihat pada Gambar 2.7.



**Gambar 2.7 Ilustrasi Penghitungan Nilai Network Latency**

### BAB 3 METODOLOGI PENELITIAN

Bab ini memaparkan langkah yang dilakukan untuk mencapai tujuan penelitian, yaitu studi kepustakaan, perancangan lingkungan uji, implementasi sistem, pengujian sistem serta analisis hasil pengujian. Kesimpulan dan saran dituliskan sebagai catatan untuk penelitian serupa dimasa yang akan datang. Berikut ini merupakan diagram alur pada penelitian ini :



**Gambar 3.1 Alur Metodologi Penelitian**

Berdasarkan gambar 3.1, dijabarkan langkah-langkah dari metodologi penelitian yang akan dilakukan :

1. Studi kepustakaan penelitian sebelumnya mengenai *software-defined networking*, *flow table*, Mininet, Ryu, dan mekanisme penghapusan *flow entry*.

2. Analisis kebutuhan dari sistem yang mencakup kebutuhan perangkat pada lingkungan pengujian.
3. Persiapan lingkungan uji untuk mengimplementasikan sistem *server load balancing* yang berkorelasi dengan mekanisme *flow removal*.
4. Implementasi pengaturan *idle timeouts* dan *flow modification message delete flow* untuk mengoptimalkan *flow entries* pada sistem round-robin *server load balancing* berbasis *software-defined networking*.
5. Pengujian dan analisis hasil pengujian dari implementasi sistem berdasarkan parameter uji yang telah ditetapkan.
6. Pengambilan kesimpulan didasarkan pada hasil analisis pengujian yang telah dijalankan oleh sistem.

### 3.1 Studi Literatur

Tahapan penelitian kali ini membutuhkan studi *literature* untuk melakukan analisis terhadap pengaruh pada OpenFlow *flow table overflow* terhadap sistem *round-robin server load balancing* berbasis *software-defined networking*. Informasi yang berhubungan dengan penelitian ini didapatkan dari jurnal, buku, situs web, dosen pembimbing, serta rekan mahasiswa. Adapun teori – teori yang dipelajari adalah sebagai berikut :

1. *Software Define Network (SDN)*

Melakukan studi *literature* yang berkaitan dengan *software-defined networking*, mencari dari jurnal yang menjelaskan secara umum bagaimana konsep dari SDN.

2. *Server Load Balancing*

Melakukan survei dan pengamatan serta memahami konsep serta tujuan dari *server load balancing*. Konsep yang harus dipahami dalam hal ini terutama penerapan *load balancing server* pada arsitektur SDN.

3. Algoritme Round-robin

Melakukan survei dan percobaan untuk memahami bagaimana alur logika dari algoritme round-robin. Dalam hal ini tentunya mempelajari penerapan logika round-robin pada sistem *server load balancing*.

4. Protokol OpenFlow

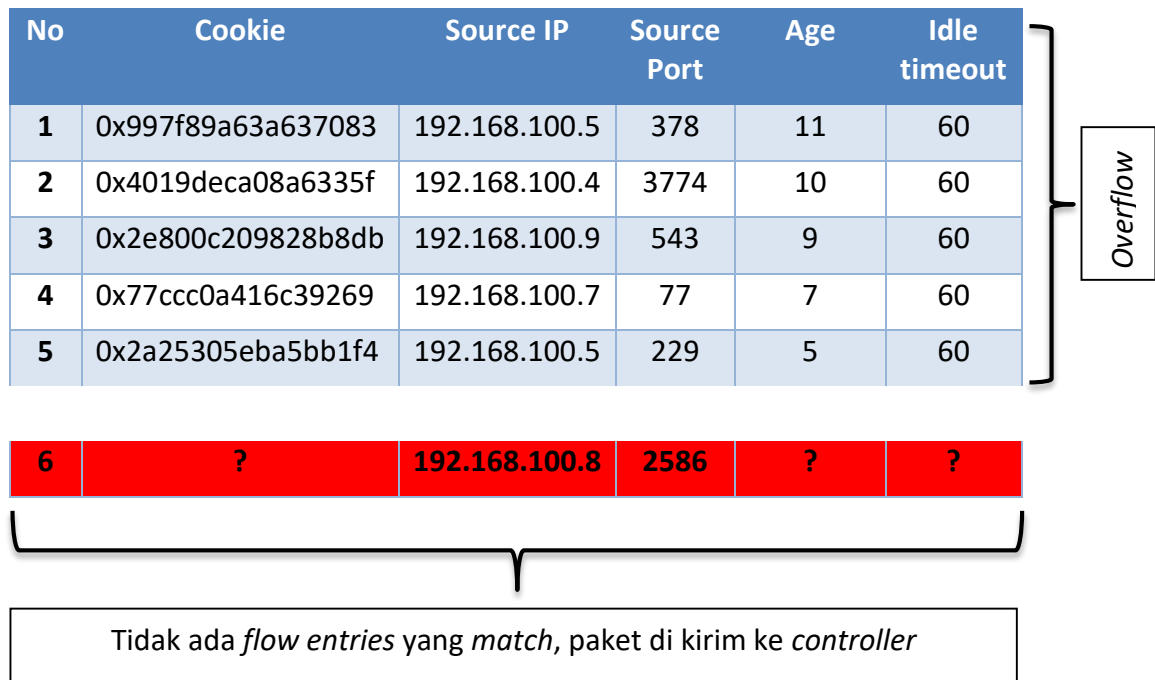
Melakukan studi *literature* yang berkaitan tentang protokol komunikasi OpenFlow, mempelajari dokumentasi mengenai protokol *OpenFlow* terutama OpenFlow 1.3.

5. Mekanisme *Flow Removal* pada OpenFlow Switch

Melakukan studi *literature* dan percobaan untuk memahami bagaimana mekanisme penghapusan *flow entry* dari *flow table*.

### 3.1.1 Flow Table Overflow pada OpenFlow Switch

*Flow table overflow* adalah sebuah kondisi dimana kapasitas kemampuan dari suatu *flow table* dalam menyimpan *flow entries* telah mencapai batas maksimal, sehingga *flow table* tidak dapat memfasilitasi *flow entry* baru yang ingin menempati *flow table*. Berikut ini adalah ilustrasi ketika terjadi *flow table overflow*.



**Gambar 3.2 Ilustrasi Flow Table Overflow**

Dari gambar 3.2 dapat dilihat bahwa ketika terdapat *request* dari *source* yang baru maka *packet* akan di cocok an dengan *flow entries* yang telah ada di dalam *flow table*. Ketika *packet* tidak ada yang cocok dengan *flow entries* yang ada di dalam *flow table*, maka *packet* akan dikirimkan ke *controller* dan akan di definisikan sebagai sebuah *flow entry* baru. Namun ketika *flow entry* baru akan menempati *flow table* yang telah penuh sementara *idle timeout* dari *flow entries* belum mencapai batas waktu, maka *controller* tidak bisa menuliskan sebuah *flow entry* baru di dalam *flow table*.

### 3.1.2 Mencegah Flow Table Overflow

Kondisi *flow table overflow* mengakibatkan *controller* tidak dapat menuliskan *flow entry* baru ke dalam sebuah *flow table*. Maka dari itu agar sebuah *flow entry* dapat terfasilitasi untuk menempati sebuah *flow table*, perlu sebuah langkah untuk mengoptimalkan *flow entries* demi mencegah terjadinya *flow table overflow*. Salah satu cara untuk melakukan optimasi *flow entries* adalah dengan melakukan penghapusan *flow entry* yang sedang menempati *flow table*.

Terdapat 2 mekanisme untuk melakukan penghapusan *flow entry* dari *flow table*. Mekanisme pertama menggunakan *flow expiry*, berikut ini adalah ilustrasi

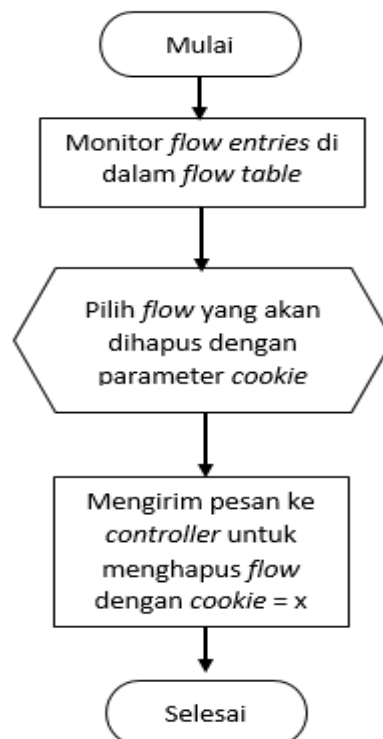
mekanisme penghapusan *flow entry* dengan menggunakan *idle timeout*.

No	Cookie	Source IP	Source Port	Age	Idle timeout
1	0x997f89a63a637083	192.168.100.5	378	11	60
2	0x4019deca08a6335f	192.168.100.6	3774	10	60
3	0x2e800c209828b8db	192.168.100.8	543	9	60
4	0x77ccc0a416c39269	192.168.100.7	77	7	60
5	0x2a25305eba5bb1f4	192.168.100.9	229	5	60

Nilai *idle timeout* menyebabkan *flow entry* akan dihapus dari *flow table* ketika tidak ada paket yang cocok selama nilai atau waktu yang diberikan.

**Gambar 3.3 Mekanisme Flow Expiry**

Mekanisme kedua untuk menghapus *flow entry* dari *flow table* adalah menggunakan *flow modification message delete flow* yaitu dengan cara mengirimkan pesan kontrol eksplisit kepada *controller*. Berikut ini adalah ilustrasi logika pengiriman pesan kontrol eksplisit untuk menghapus *flow entry*.



**Gambar 3.4 Ilustrasi Mekanisme Pengiriman Pesan Eksplisit ke Controller**



### 3.2 Kebutuhan Lingkungan Pengujian

Analisis kebutuhan perangkat dilakukan untuk mencari serta menganalisis hal-hal yang dibutuhkan untuk mendukung penelitian ini. Kebutuhan dalam hal ini mencakup kebutuhan perangkat pendukung dalam melakukan pengujian.

Berikut merupakan kebutuhan fungsional s:

1. Sistem mampu mengatur *idle timeouts* terhadap setiap *flow entry* yang masuk ke *flow table* secara dinamis.
2. Sistem dapat melakukan penghapusan *flow* secara aktif dengan mengirimkan pesan kepada *controller*.
3. Jaringan SDN yang terdiri dari :
  - *Switch* (menjalankan fungsi *packet forwarding* antar *host*)
  - *Controller* (menjalankan *framework Ryu controller* serta mekanisme pengaturan *idle timeouts* dan pengiriman pesan kontrol eksplisit untuk menghapus *flow entry*)
  - *Host* (memerankan sebuah *client* dan *server*, serta mampu mengirim dan menerima paket)
4. Sistem yang dapat mengirim paket HTTP *request* dari beberapa *source* dan 1 *destination* yang sama

Di bawah ini merupakan kebutuhan perangkat lunak dari sistem yang akan dibangun.

1. OpenFlow 1.3: Sistem *controller* yang kompatibel dengan OpenFlow *switch* yang memiliki kemampuan *flow table* terbatas.
2. Linux Ubuntu: Sistem operasi yang menunjang Mininet, Ryu *controller* serta *library* Python.
3. Mininet: Sebagai *emulator* untuk membangun topologi SDN.
4. Ryu: Sebagai SDN *Controller*.

### 3.3 Perancangan Lingkungan Uji

Perancangan lingkungan uji menjelaskan tentang tahapan perencanaan yang merupakan sebuah langkah untuk menyusun lingkungan sistem berdasarkan kebutuhan yang telah dijabarkan pada sub bab sebelumnya. Sistem ini dirancang untuk memenuhi kebutuhan sistem server *load balancing* pada arsitektur *software-defined networking*. Arsitektur SDN dipilih untuk menerapkan sistem *load balancing* karena konsep dari SDN yang memisahkan antara *control plane* dan *data plane*. Dalam hal ini logika dari sistem di arsitektur SDN diatur di dalam *control plane*, termasuk logika dalam mengatur mekanisme *flow removal*, sehingga nantinya peneliti akan mudah melakukan modifikasi dan integrasi antara sistem server *load balancing* yang berkorelasi dengan pengaturan *idle timeouts* dan *flow modification message*. Tahapan perancangan dimulai dari perancangan








topologi jaringan, yang menentukan *host*, *switch*, dan *controller* yang digunakan pada *emulator* Mininet. Selanjutnya perancangan *round-robin server load balancing*, dimana perancangan ini merencanakan skema pembagian beban *server* dari setiap *request* yang ditujukan kepada *server*. Perancangan skenario pengujian merencanakan pemberian batasan kapasitas *flow table* pada emulator Mininet, agar lingkungan uji sesuai dengan permasalahan di keadaan sebenarnya bahwa kemampuan *flow table* pada OpenFlow *switch* terbatas.

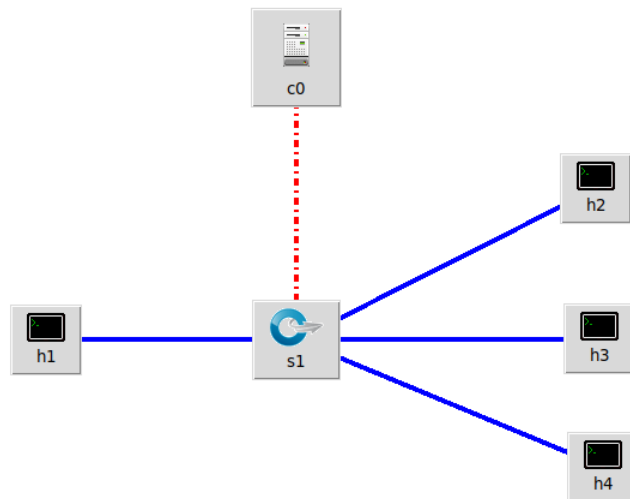
### 3.3.1 Topologi Jaringan

Dalam membangun topologi yang akan digunakan dalam pengujian sistem, terdapat beberapa cara yang dapat dilakukan, salah satunya yaitu dengan memanfaatkan *GUI*. Dalam mininet *GUI* ini dikenal sebagai Miniedit. Berikut ini merupakan langkah-langkah yang dilakukan untuk membangun topologi menggunakan Miniedit.

1. Masuk ke direktori mininet/examples, kemudian miniedit dapat dijalankan dengan memasukkan perintah seperti dibawah ini di dalam *terminal*.  
"\$ sudo python miniedit.py"
2. Menyusun topologi jaringan dengan menggunakan komponen yang telah disediakan oleh miniedit. Tabel 3.1 merupakan daftar komponen yang tersedia di miniedit.

**Tabel 3.1 Daftar Komponen Mininet**

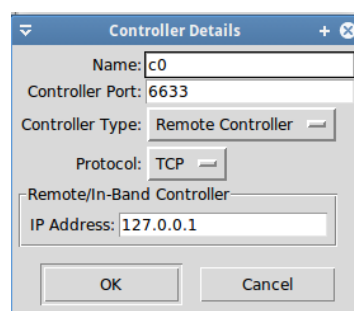
No	Ikon	Nama	Fungsi
1		<i>Cursor</i>	Memilih dan mengatur posisi komponen.
2		<i>Host/end Service</i>	Berperan sebagai <i>host</i> atau <i>end device</i> yang ada di dalam jaringan.
3		OpenFlow <i>switch</i>	Berperan sebagai <i>forwarding function</i> pada perangkat jaringan OpenFlow SDN.
4		<i>Switch</i>	Berperan sebagai <i>switch</i> tradisional yang memiliki fungsi sebagai <i>packet forwarding</i> menggunakan alamat MAC berdasarkan letak bersambungannya <i>port</i> .
5		<i>Router</i>	Berperan sebagai <i>router</i> tradisional yang menentukan keputusan <i>packet forwarding</i> berdasarkan <i>routing protocol</i> .
6		<i>Link</i>	Berperan sebagai penghubung antara satu komponen dengan komponen yang lain.
7		<i>Controller</i>	Berperan sebagai <i>controller</i> SDN yang memiliki fungsi sebagai <i>network control</i> .



**Gambar 3.5 Topologi Jaringan pada Arsitektur SDN**

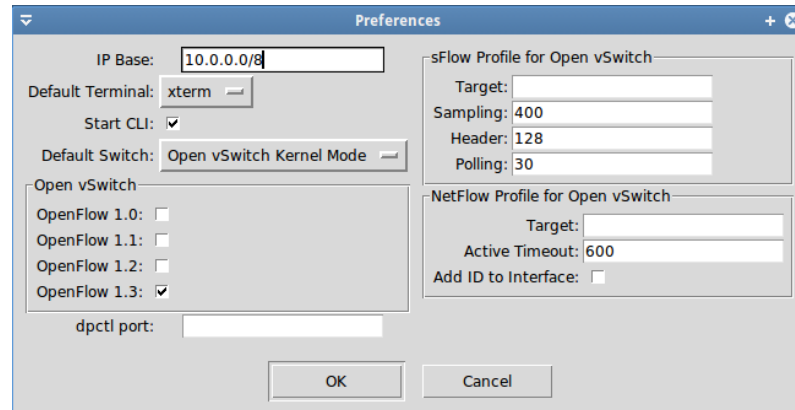
Topologi jaringan yang digunakan dalam penelitian ini yaitu 4 buah virtual *host*, 1 buah *controller* dan 1 buah OpenFlow *switch* di dalam arsitektur SDN. Penyusunan topologi dibuat linier dengan cara menghubungkan tiap - tiap *host* pada OpenFlow *switch* secara langsung. Penentuan bentuk atau tipe topologi sebagaimana pada gambar 3.5 dipilih karena mampu memenuhi lingkungan pengujian dari sistem *server load balancing*.

3. Konfigurasi SDN *controller* pada topologi sesuai dengan gambar 3.5 dengan cara melakukan klik kanan pada entitas *controller* (c0) kemudian memilih pilihan "*properties*", kemudian mengubah tipe *controller* "*Remote Controller*" pada "*Controller Type*". Hal itu dilakukan agar Mininet dapat terhubung dengan program *controller* yang nantinya akan dibangun.



**Gambar 3.6 Pengaturan *controller* di Mininet**

4. Pilih menu "*Edit*", selanjutnya "*Preferences*" untuk melakukan konfigurasi pada *preferences* Mininet. Setelah itu pilih "*Start CLI*" untuk menjalankan fungsi *command line interface* di Mininet. Kemudian pilih versi OpenFlow yang akan digunakan sesuai yang ditunjukkan gambar 3.7.

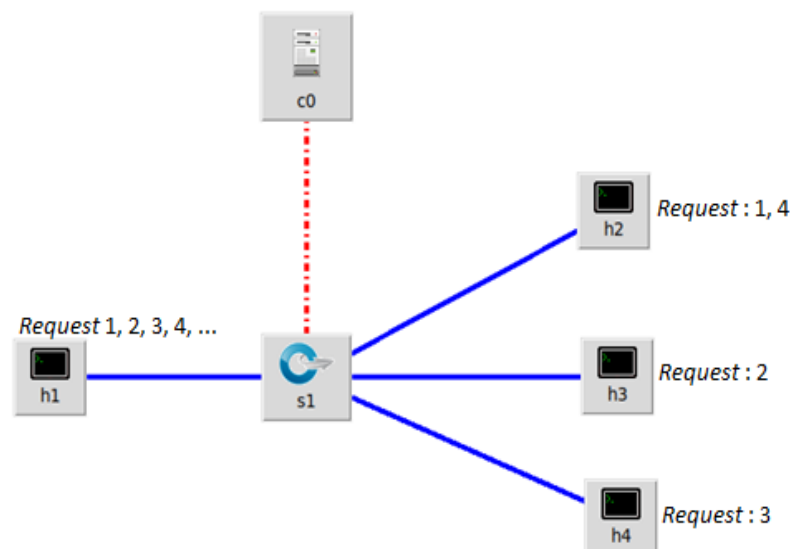


**Gambar 3.7** Pengaturan *preferences* di Mininet

5. Jalankan emulasi topologi yang telah dibangun di dalam Mininet dengan menekan *button* “Run”.

### 3.3.2 Round-Robin Server Load Balancing

Dalam melakukan pembagian beban server, round-robin mengarahkan setiap *request* atau *flow entry* baru ke *server* selanjutnya. Tanpa memperhatikan kondisi *server*, algoritme round-robin selalu berjalan sirkular untuk memenuhi setiap *request* yang didefinisikan sebagai sebuah *flow entry* baru. Ilustrasi pembagian beban server dengan algoritme round-robin dapat dilihat pada gambar dibawah ini.



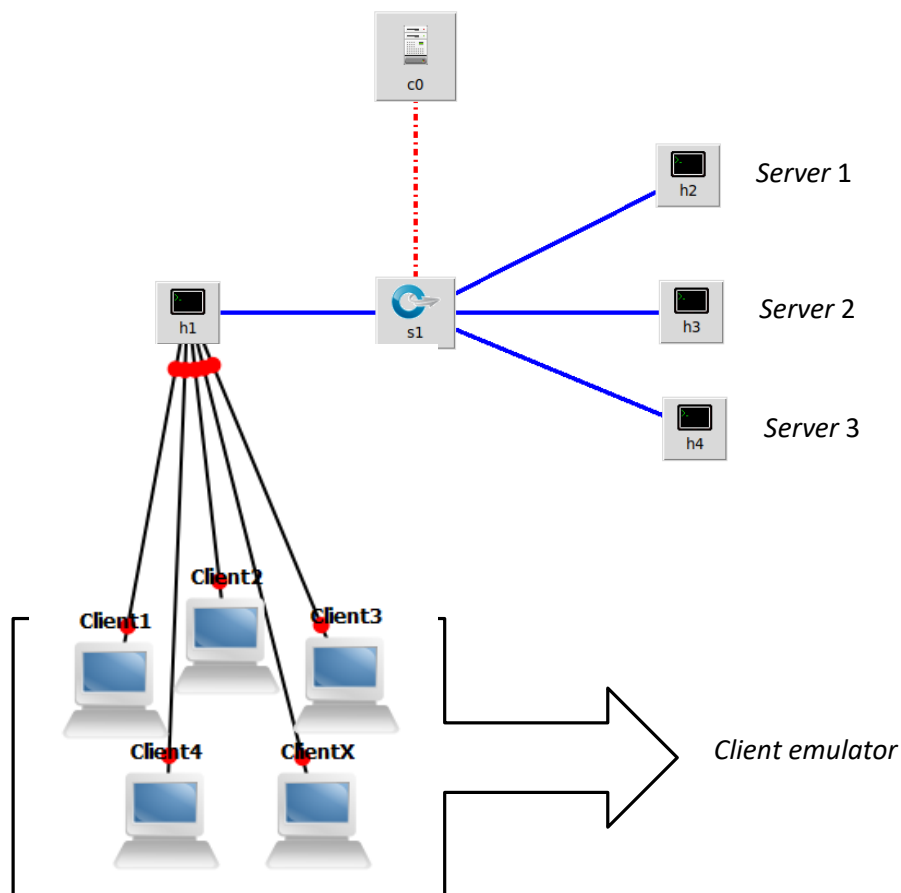
**Gambar 3.8** Ilustrasi Round-Robin Server Load Balancing

### 3.3.3 Kapasitas *Flow Table*

Di dalam emulator Mininet, kapasitas *flow table* dalam OpenFlow switch adalah tidak terbatas, atau dapat menampung sesuai dengan kemampuan CPU yang kita gunakan. Untuk itu dalam rangka mewujudkan kondisi kapasitas *flow table* yang terbatas, perlu disusun skenario untuk membuat batasan pada OpenFlow switch di emulator mininet.

### 3.3.4 *Client - Server*

Untuk membangun sebuah skenario pengiriman paket dari beberapa *source*, maka perlu dilakukan sebuah perancangan sistem *client emulation* yang akan mewakili setiap *request* yang dikirimkan ke *server*. Dibawah ini adalah ilustrasi dari *client emulation* yang dirancang menggunakan sebuah pemrograman.



Gambar 3.9 Ilustrasi *Client - Server*

## 3.4 Implementasi

### 3.4.1 Lingkungan Uji

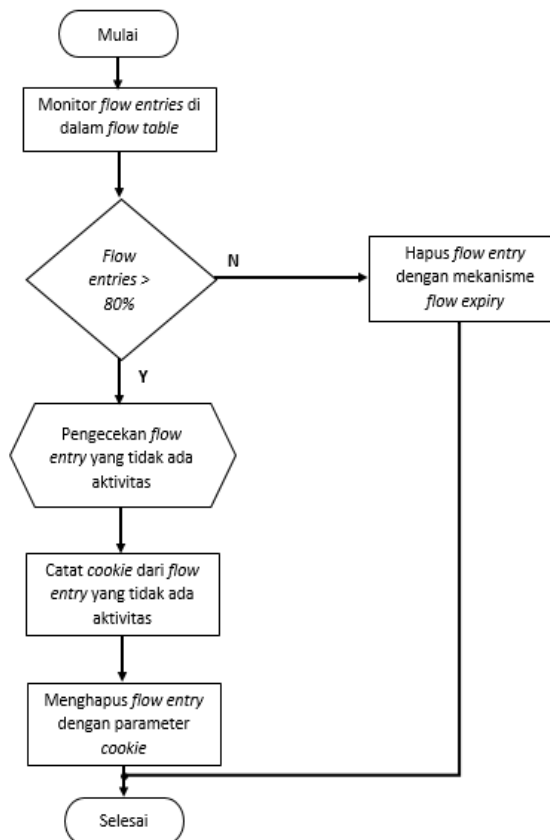
Implementasi meliputi tahapan yang dilakukan untuk membangun sebuah sistem yang sebelumnya telah dirancang. Selain itu juga diterapkan batasan –

batasan dalam membangun sistem. Berdasarkan pemaparan pada sub bab sebelumnya, diperlukan beberapa hal untuk membentuk lingkungan uji, diantaranya adalah sebagai berikut:

1. Instalasi Mininet sebagai Emulator yang mendukung topologi dengan arsitektur SDN.
2. Instalasi Ryu *controller* sebagai *framework* SDN controller.
3. Implementasi *load balancing server* pada Ryu *controller*.
4. Implementasi *flow limit* pada protokol OpenFlow 1.3.
5. Implementasi pengaturan *idle timeouts* pada *flow route to server* dan *reverse from server*.
6. Implementasi *traffic monitoring* pada *switch* untuk melihat kondisi *flow table*.
7. Implementasi pengiriman pesan modifikasi kepada *controller* ketika *flow table entries* mendekati batas kemampuan maksimalnya.
8. Implementasi pemrograman *python* dengan *library request* untuk mengirimkan HTTP *request* ke *server*.

### 3.4.2 Mekanisme Flow Removal

Dalam mengimplementasikan mekanisme *flow removal*, perlu dirancang aturan untuk melakukan penghapusan sebuah *flow entry*. Berikut ini adalah *flow chart* yang menggambarkan alur program dalam melakukan penghapusan terhadap *flow entry*.



**Gambar 3.10 Mekanisme Flow Removal**

Sebelum melakukan penghapusan terhadap suatu *flow entry*, terlebih dahulu dilakukan *monitoring* kondisi *flow entries* di dalam *flow table*. Ketika kondisi *flow entries* lebih dari 80% dari batas maksimal *flow table* maka dilakukan pengecekan terhadap *flow entries* yang tidak ada aktivitas, atau dalam hal ini tidak terjadi penambahan *packet count* ataupun *byte count*. *Cookie* dari *flow entry* yang tidak ada aktivitas akan dijadikan parameter dalam menghapus *flow* ketika kondisi *flow entries* lebih dari 80%. Ketika kondisi *flow entries* kurang dari 80% maka penghapusan *flow entry* menggunakan mekanisme *flow expiry* atau dalam penelitian kali ini menggunakan nilai *idle timeout*.

### 3.5 Pengujian

Pengujian dilakukan untuk mengetahui apakah implementasi sistem telah memenuhi kebutuhan yang diperlukan. Pengujian pertama dilakukan dengan menjalankan topologi dan kontroler yang mengimplementasikan sistem *server load balancing* dan monitoring kondisi *flow table* pada *switch*. Pengujian kedua adalah pengujian sistem *server load balancing* dengan mengirimkan paket ke *server* untuk mengetahui apakah paket *request* dikirimkan ke *server* yang seharusnya sesuai dengan algoritma round-robin *server load balancing*.

Pengujian selanjutnya adalah melakukan pengiriman *request* ke *server* dari alamat IP acak namun dalam batasan tertentu, sehingga memungkinkan untuk terjadi penggunaan kembali sebuah *flow entry*. Pada pengujian ini *client* mengirimkan 100 paket HTTP *request* dari *source* IP yang berbeda beda ke *server*. Setelah mendapatkan *response* dari *server*, *client* akan mengganti alamat IP untuk bertindak sebagai *client* lain. Alamat IP *client* yang digunakan berupa alamat acak yang memungkinkan penggunaan kembali alamat tersebut untuk melakukan *request*. Pengujian selanjutnya dilakukan dengan menganalisis apakah suatu *flow entry* akan dihapus dari *flow table* ketika kondisi *flow table* mendekati batas maksimalnya dengan syarat tertentu, sehingga dapat tercapai tujuan optimasi. Pengujian terakhir dilakukan dengan menguji kinerja *server* melalui parameter *network latency* setelah menerapkan mekanisme *flow removal*.

### 3.6 Analisis Hasil

Analisis hasil pengujian dilakukan setelah seluruh proses pengujian selesai dilaksanakan. Dalam bab ini disajikan data hasil pengujian serta dijabarkan hasil analisis kondisi *flow table* yang telah mengimplementasikan round-robin *server load balancing* dengan pengaturan *flow table*. Data hasil pengujian didapatkan dari jumlah *flow entries* yang merupakan output setiap kali melakukan pengecekan kondisi *flow entries*. Selain itu data hasil pengujian *network latency* didapatkan dari nilai *latency* yang diambil dari hasil *capture packet* melalui wireshark. Data nilai *latency* diolah dalam tabel dan disajikan dalam grafik *cumulative distribution function* (CDF). Hasil dari pembahasan akan memperlihatkan dampak dari pengaturan *flow table* dengan mekanisme *idle timeouts* dan pesan modifikasi *delete flow* terhadap tujuan optimasi berdasarkan skenario pengujian yang sebelumnya telah dilaksanakan.

### 3.7 Kesimpulan

Penarikan kesimpulan dilakukan berdasarkan analisis dan pembahasan mengenai dampak dari pengaturan *flow table* terhadap tujuan optimasi *flow entries*. Optimasi *flow entries* untuk mencegah *flow table overflow* pada *server load balancing* diukur menggunakan parameter pengujian yang telah ditentukan, sehingga dapat disimpulkan apakah pengaturan *idle timeouts* dan pengaturan pesan modifikasi *delete flow* dapat mengoptimalkan *round-robin server load balancing*.



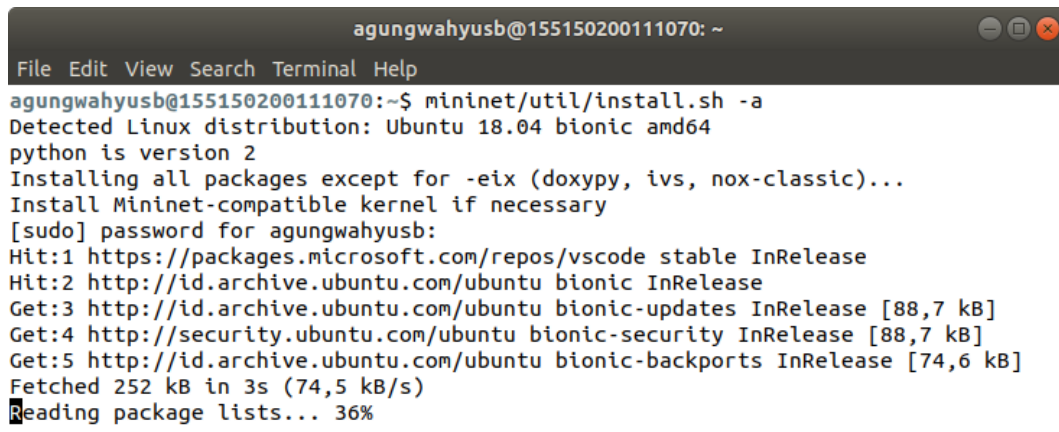
## BAB 4 IMPLEMENTASI

Pada bab ini akan dijelaskan implementasi dari mekanisme *flow removal*. Implementasi mencakup monitoring kondisi OpenFlow *switch* dengan mengirimkan *traffic monitoring request*, memilih data – data yang digunakan dalam melakukan penghapusan *flow entry*, serta implementasi aturan penghapusan. Implementasi dilakukan dengan melakukan modifikasi pada Ryu *controller* yang berjalan pada Mininet dengan protokol OpenFlow.

### 4.1 Lingkungan Pengujian

#### 4.1.1 Instalasi Mininet

Sub bab ini menjelaskan langkah untuk melakukan instalasi mininet yang digunakan sebagai emulator untuk mengabstraksikan sebuah topologi yang telah dirancang sesuai dengan perancangan lingkungan pengujian. Mininet dapat diinstall dengan mengikuti panduan dari situs web resmi mininet di “[mininet.org/download/](http://mininet.org/download/)”. Instalasi dilakukan di terminal dengan menjalankan perintah “`git clone git://github.com/mininet/mininet`” untuk menyalin *source code* mininet. Selanjutnya instalasi dilakukan dengan mengetikkan perintah “`mininet/util/install.sh -a`”.



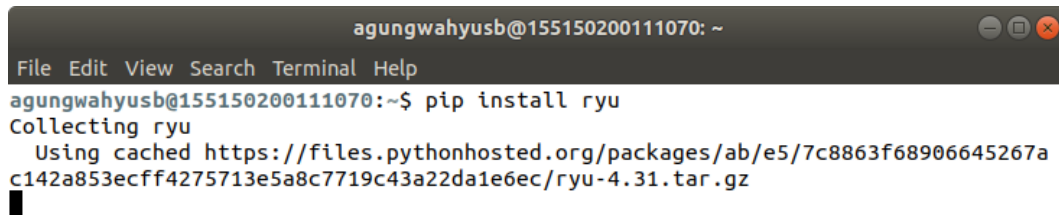
```
agungwahyusb@155150200111070: ~
File Edit View Search Terminal Help
agungwahyusb@155150200111070:~$ mininet/util/install.sh -a
Detected Linux distribution: Ubuntu 18.04 bionic amd64
python is version 2
Installing all packages except for -eix (doxypy, ivs, nox-classic)...
Install Mininet-compatible kernel if necessary
[sudo] password for agungwahyusb:
Hit:1 https://packages.microsoft.com/repos/vscode stable InRelease
Hit:2 http://id.archive.ubuntu.com/ubuntu bionic InRelease
Get:3 http://id.archive.ubuntu.com/ubuntu bionic-updates InRelease [88,7 kB]
Get:4 http://security.ubuntu.com/ubuntu bionic-security InRelease [88,7 kB]
Get:5 http://id.archive.ubuntu.com/ubuntu bionic-backports InRelease [74,6 kB]
Fetched 252 kB in 3s (74,5 kB/s)
Reading package lists... 36%
```

Gambar 4.1 Instalasi Mininet

#### 4.1.2 Instalasi Ryu Controller

Sub bab ini menjelaskan tahapan instalasi *API* Ryu sebagai SDN *controller* dengan memanfaatkan *package-management system* pada *python*. Cara paling mudah dalam menginstal Ryu adalah menggunakan pip yang merupakan sistem manajemen paket yang digunakan untuk menginstal dan mengelola paket

perangkat lunak yang ditulis dengan *python* . Instalasi dilakukan pada terminal dengan mengetikkan perintah “pip install ryu”.



```

agungwahyusb@155150200111070: ~
File Edit View Search Terminal Help
agungwahyusb@155150200111070:~$ pip install ryu
Collecting ryu
  Using cached https://files.pythonhosted.org/packages/ab/e5/7c8863f68906645267ac142a853ecff4275713e5a8c7719c43a22da1e6ec/ryu-4.31.tar.gz

```

**Gambar 4.2 Instalasi Ryu Controller**

Setelah instalasi Ryu selesai dilaksanakan, uji coba dilakukan dengan menjalankan perintah “ryu-manager” untuk melihat versi Ryu. Ryu menyediakan banyak versi *framework* yang dapat disesuaikan untuk pengembangan aplikasi pada protokol OpenFlow.

#### 4.1.3 Topologi Jaringan

Untuk mendukung lingkungan pengujian, perlu disusun topologi jaringan dengan arsitektur yang telah dijelaskan pada perancangan topologi jaringan, yaitu topologi dengan 1 *client*, 3 *server*, 1 OpenFlow *switch*, dan 1 *controller*. Implementasi dari topologi jaringan adalah sebagai berikut.

**Tabel 4.1 Kode Program Konfigurasi Topologi Jaringan**

Topologi Jaringan	
1	from mininet.topo import Topo
2	import SimpleHTTPServer
3	import SocketServer
4	
5	class MyTopo( Topo ):
6	"Simple topology example."
7	
8	def __init__( self ):
9	"Create custom topo."
10	
11	# Initialize topology
12	Topo.__init__( self )
13	
14	# Add hosts and switches
15	client1 = self.addHost( 'h1' )
16	server1 = self.addHost( 'h2' )
16	server2 = self.addHost( 'h3' )
17	server3 = self.addHost( 'h4' )
19	switch1 = self.addSwitch( 's1' )
20	
21	# Add links
22	self.addLink( client1, switch1 )
23	self.addLink( server1, switch1 )
24	self.addLink( server2, switch1 )
25	self.addLink( server3, switch1 )
26	
27	topos = { 'mytopo': ( lambda: MyTopo() ) }

Berdasarkan kode program pada tabel 4.1, alur konfigurasi topologi tersebut dijelaskan pada keterangan di bawah ini :

- Baris 1-3, merupakan langkah untuk *import* salah satu *library* dari mininet yaitu topologi, dan socket
- Baris 4, merupakan deklarasi kelas dengan nama “MyTopo”
- Baris 13-17, merupakan deklarasi sekaligus inisiasi variable *client1*, *server1*, *server2*, *server3*, dan *switch1*
- Baris 20-24, merupakan langkah untuk menambahkan *link* antara *client*, *switch*, dan *server*

#### 4.1.4 Konfigurasi Client-Server

##### 4.1.4.1 Client

Implementasi *client emulation* dilakukan untuk mengabstraksikan sebuah *client* dengan berbagai *source IP* yang melakukan *request* ke sebuah *server*. Berikut ini adalah *pseudocode* untuk melakukan implementasi *client emulation*.

**Tabel 4.2 Pseudocode Program Client Emulation**

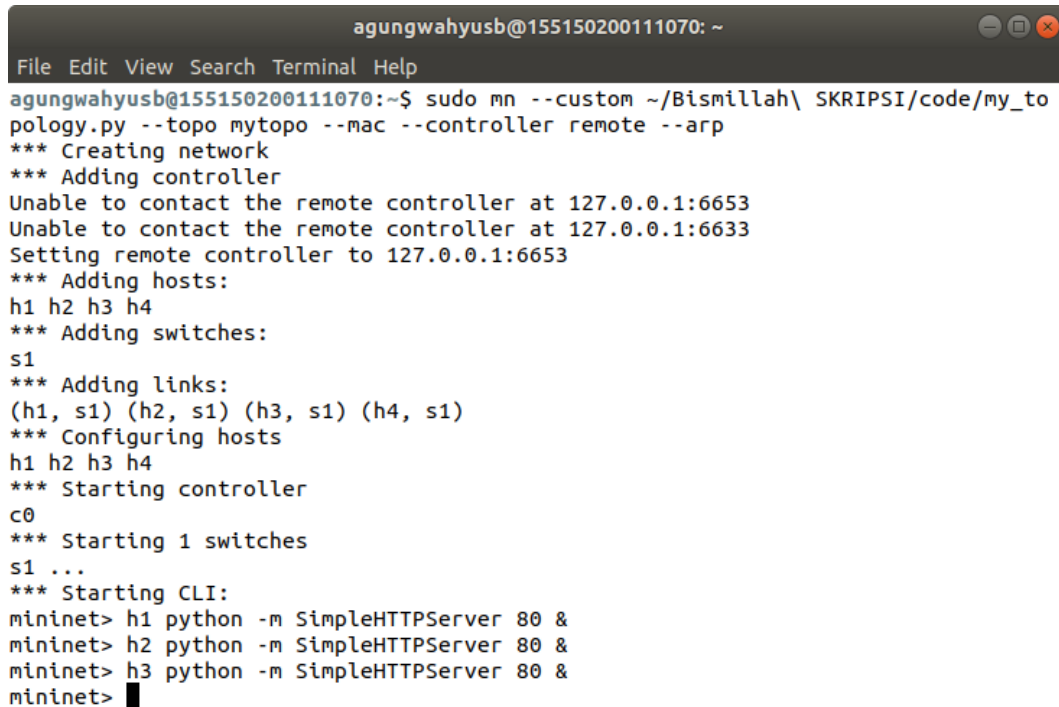
Client Emulation	
1	import library requests, random, os
2	
3	looping 100x:
4	generate random IP Address
5	sleep 0.5 second
6	send request GET to server 10.0.0.100
7	read response from server
8	count += 1

Berdasarkan *pseudocode* pada tabel 4.2, alur berjalannya *client emulation* dijelaskan pada keterangan di bawah ini:

- Baris 1, merupakan langkah untuk *import library request*, *random*, dan *os* yang merupakan *library* yang dibutuhkan dalam program ini
- Baris 3-7, merupakan perulangan dengan mengirimkan paket “request” atau dengan kata lain mengirimkan *request* ke alamat IP *server* dengan mengganti alamat IP *random* sesuai dengan *range* yang telah ditentukan sebelum mengirimkan paket.
- Baris 8, menambahkan nilai 1 pada variabel *count* setelah berhasil mengirimkan paket dan mendapatkan balasan dari *server*.

#### 4.1.4.2 Server

Dalam implementasi *round-robin server load balancing*, *host* dari topologi harus diatur sebagai *client* dan *server* untuk mendukung lingkungan pengujian. *Host* 1, 2, dan 3 dikonfigurasi sebagai *server*, sedangkan *host* 4 di konfigurasi sebagai *client*. Langkah – langkah untuk mengkonfigurasi *host* adalah seperti yang ditunjukkan pada gambar 4.3 dibawah ini. Konfigurasi server dilakukan dengan menjalankan perintah “python –m SimpleHTTPServer 80”.



```
agungwahyusb@155150200111070: ~
File Edit View Search Terminal Help
agungwahyusb@155150200111070:~$ sudo mn --custom ~/Bismillah\ SKRIPSI/code/my_to
pology.py --topo mytopo --mac --controller remote --arp
*** Creating network
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6653
Unable to contact the remote controller at 127.0.0.1:6633
Setting remote controller to 127.0.0.1:6653
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1
*** Adding links:
(h1, s1) (h2, s1) (h3, s1) (h4, s1)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 1 switches
s1 ...
*** Starting CLI:
mininet> h1 python -m SimpleHTTPServer 80 &
mininet> h2 python -m SimpleHTTPServer 80 &
mininet> h3 python -m SimpleHTTPServer 80 &
mininet>
```

Gambar 4.3 Konfigurasi Server

#### 4.1.5 Round-Robin Server Load Balancing

Dalam implementasi, *load balancer* ditempatkan di kontroler Ryu dengan menggunakan algoritme pembebanan round-robin. Setiap *request* yang ditujukan ke *server* akan diteruskan ke beberapa *server* secara sirkular.

Tabel 4.3 Pseudocode Sistem Server Load Balancing

Round-robin Server Load Balancing	
1	Import library
2	
3	configure_virtual_LB(IP, MAC, port)
4	configure_list_server[{1},{2},{3}]
5	
6	request_handler():
7	handle arp reply for virtual server IP
8	return arp_reply packet
9	
10	packet_in_handler():

11	IF server_number == 3:
12	reinitiation server number = 0
13	IF ethernet type = arp:
14	call function request_handler()
15	call function add_flow()
16	
17	IF ethernet type = icmp:
18	call function add_flow()
19	handle packet to prevent icmp flood
20	
21	ELSE: ignore packet
22	
23	#Round Robin Style
24	count = counter %3
25	IF no packet match with flow entries: THEN
26	define as flow_entry_x
27	select server [count]
28	counter + 1
29	
30	ELSE:
31	match packet with flow entry in flow table

Berdasarkan *pseudocode* pada tabel 4.3, alur logika dari sistem *server load balancing* adalah sebagai berikut:

- Baris 1, merupakan persiapan kebutuhan lingkungan dari sistem *server load balancing*, yaitu dengan melakukan import *library*.
- Baris 3-4, merupakan konfigurasi *virtual load balancer* dan *server*.
- Baris 6-8, merupakan *method* untuk melayani *request* dari client dengan membangun paket ARP Reply untuk *virtual load balancer*.
- Baris 10-12, merupakan inisiasi ulang *server number* ketika semua *server* telah dipilih.
- Baris 13-15, merupakan seleksi kondisi paket arp untuk menyiapkan paket arp reply, sekaligus menambahkan flow entry baru.
- Baris 17-19, merupakan seleksi kondisi paket icmp.
- Baris 23-28, merupakan fungsi untuk memilih server yang akan dibebani atau dituju oleh *client* berdasarkan algoritme round-robin.
- Baris 30-31, merupakan fungsi untuk melakukan proses pencocokan paket dengan *flow entries* di dalam *flow table*.

Round-robin *load balancer* dijalankan sebagai kontroler Ryu dengan perintah sebagai berikut: “\$ ryu-manager slb\_rr.py”

```

agungwahyusb@155150200111070: ~/Bismillah SKRIPSI/code
File Edit View Search Terminal Tabs Help
agungwahyusb@155150200111070: ~ x agungwahyusb@155150200111070: ~/Bism... x
agungwahyusb@155150200111070:~/Bismillah SKRIPSI/code$ ryu-manager slb_rr.py
loading app slb_rr.py
loading app ryu.controller.ofp_handler
instantiating app slb_rr.py of loadbalancer
Initialized new Object instance data
instantiating app ryu.controller.ofp_handler of OFPHandler

```

**Gambar 4.4 Menjalankan controller Ryu**

#### 4.1.6 Flow Limit pada OpenFlow Switch

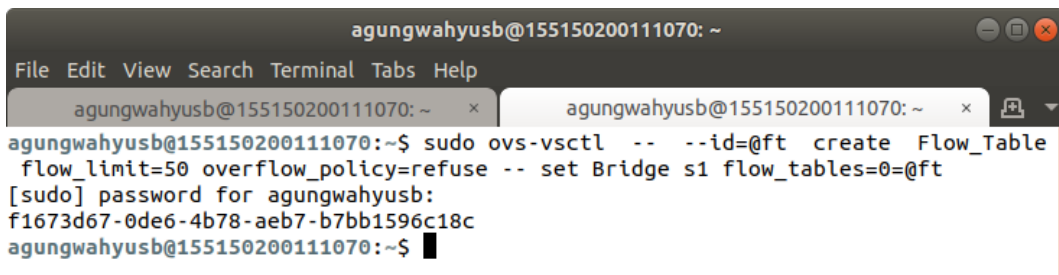
Untuk mendukung lingkungan pengujian agar sesuai dengan kondisi OpenFlow switch yang sebenarnya yaitu memiliki kapasitas yang terbatas dalam menyimpan *flow entry*, maka perlu dilakukan konfigurasi pada topologi yang berjalan pada mininet terutama konfigurasi pada switch. Tabel 4.4 menunjukkan kapasitas maksimal *flow table* dan jumlah paket *request* yang akan dikirimkan sesuai dengan skenario pengujian yang telah ditetapkan sebelumnya.

**Tabel 4.4 Kapasitas Flow Table**

Skenario Pengujian	
Kapasitas Flow Table	50 flow entries
Jumlah Paket Request	100 paket

Untuk melakukan konfigurasi *flow limit*, perintah yang dijalankan di terminal adalah

```
"$ sudo ovs-vsctl -- --id=@ft create Flow_Table flow_limit=50  
overflow_policy=refuse -- set Bridge s1 flow_tables=0=@ft"
```



```
agungwahyusb@155150200111070: ~  
File Edit View Search Terminal Tabs Help  
agungwahyusb@155150200111070: ~ x agungwahyusb@155150200111070: ~ x  
agungwahyusb@155150200111070:~$ sudo ovs-vsctl -- --id=@ft create Flow_Table  
flow_limit=50 overflow_policy=refuse -- set Bridge s1 flow_tables=0=@ft  
[sudo] password for agungwahyusb:  
f1673d67-0de6-4b78-aeb7-b7bb1596c18c  
agungwahyusb@155150200111070:~$
```

**Gambar 4.5 Konfigurasi Flow Limit pada OpenFlow Switch**

#### 4.1.7 Traffic Monitoring

Sebelum mengimplementasikan mekanisme *flow removal*, perlu dilakukan *traffic monitoring* yang digunakan untuk melakukan *monitoring* terhadap kondisi *flow entries* sehingga nantinya dapat dijadikan acuan untuk melakukan penghapusan *flow entry* ketika kondisi *flow table* mendekati kemampuan maksimalnya. Berikut ini adalah implementasi dari *traffic monitoring*.

**Tabel 4.5 Pseudocode Program Traffic Monitoring**

Traffic Monitoring	
1	state_change_handler():
2	see datapath status
3	IF status datapath MAIN DISPATCHER: THEN
4	set switch as target monitor
5	ELSE status datapath DEAD DISPATCHER: THEN
6	delete switch from target monitor
7	
8	monitor():
9	call function request stats

10	request_stats():
11	return response traffic monitoring
12	
13	flow_stats_reply_handler():
14	handle response from traffic monitoring
15	
16	fetch and record message body flow statistic into json

Berdasarkan *pseudo code* pada tabel 4.5, alur *method traffic monitoring* dijelaskan pada keterangan dibawah ini :

- Baris 1-2, merupakan deklarasi *method* “\_state\_change\_handler” yang berfungsi untuk melihat dari status datapath
- Baris 3-6, merupakan seleksi kondisi status datapath
- Baris 8-12, merupakan *method* untuk mengirimkan *request monitoring* ke *switch*
- Baris 13-16, merupakan *method* untuk memfasilitasi balasan dari *method* monitor() yang akan menerjemahkan ke dalam format json.

## 4.2 Mekanisme Flow Removal

### 4.2.1 Idle Timeouts

Implementasi *idle timeouts* dilakukan untuk menjalankan salah satu mekanisme *flow removal* yaitu *flow expiry*. Dalam hal ini sebuah *flow entry* akan dihapus dari *flow table* ketika dalam waktu yang diberikan pada nilai *idle timeouts* telah terlampaui dan selama waktu yang diberikan tidak ada paket yang cocok dengan *flow entry*.

Untuk mengimplementasikan mekanisme *idle timeout*, dilakukan inisiasi pada variabel ‘idle\_timeout’ sebelum *datapath* yang berisi *flow modification message add flow* di eksekusi oleh *controller*. Dibawah ini adalah potongan kode untuk implementasi *idle timeouts*.

**Tabel 4.6 Potongan Kode Program Implementasi Idle Timeouts**

Idle Timeouts	
1	...
2	#Route to server
3	flow_mod = parser.OFPFlowMod(datapath=datapath, match=match,
4	idle_timeout=60, instructions=inst, buffer_id=msg.buffer_id,
5	cookie=cookie)
6	datapath.send_msg(flow_mod)
7	#Reverse route from server
8	flow_mod2 = parser.OFPFlowMod(datapath=datapath, match=match,
9	idle_timeout=60, instructions=inst2, cookie=cookie)
10	datapath.send_msg(flow_mod2)
11	...

Berdasarkan potongan kode program pada tabel 4.6, alur pemberian nilai *idle timeouts* dijelaskan pada keterangan di bawah ini:

- Baris 2–5, merupakan inisiasi variabel ‘flow\_mod’ dengan nilai ‘idle\_timeouts’ 60

- Baris 7–10, merupakan inisiasi variabel 'flow\_mod2' dengan nilai 'idle\_timeouts' 60

#### 4.2.2 Flow Modification

Setelah mengimplementasikan *traffic monitoring* untuk melihat kondisi *flow table* pada *switch*, langkah selanjutnya adalah mengimplementasikan *flow modification message* untuk melakukan penghapusan *flow entries* dengan acuan kondisi *flow table* pada saat itu. Berikut adalah implementasi dari *flow modification message delete flow*.

**Tabel 4.7 Pseudocode Program Flow Modification**

Flow Modification Message	
1	monitor_flow_entries()
2	send request traffic monitoring
3	read response traffic monitoring
4	temporary_flow <- record flow information(cookie, duration,
5	byte_count, packet count)
6	
7	while(flow_entries != 0):
8	IF no flow_entry_x in temporary_flow: THEN
9	ADD flow_entry_x in temporary_flow
10	
11	ELIF flow_entry_x in temporary_flow: THEN
12	record cookie of flow_entry_x
13	match cookie with indexed cookie in temporary_flow
14	
15	IF byte_count.flow_entry_x > byte_count.temporary_flow:
16	update information of temporary_flow with flow_entry_x
17	
18	ELIF byte_count.flow_entry_x == byte_count.temporary_flow:
19	IF amount of flow entries > 80% of maximum capacity: THEN
20	delete flow_entry_x by cookie parameter
21	ELSE:
22	delete flow entry by idle_timeout mechanism

Berdasarkan *pseudocode* pada Tabel 4.7, alur penerapan mekanisme *flow modification message* dijelaskan pada keterangan dibawah ini:

- Baris 1-4, merupakan *method* untuk menyimpan balasan dari *request* ke *switch* yang dikirimkan ketika melakukan *traffic monitoring* ke sebuah variabel penyimpanan sementara.
- Baris 6-8, merupakan perulangan sebanyak *flow entries* yang ada di dalam *flow table* untuk melakukan pengecekan apakah *flow entries* tersebut telah disimpan di variabel penyimpanan sementara atau tidak.
- Baris 10-12, merupakan seleksi kondisi untuk melakukan pengecekan terhadap suatu *flow entry* dan mencocokkan dengan *flow entries* yang sebelumnya telah disimpan di variabel sementara.
- Baris 14–22, merupakan seleksi kondisi yang dilakukan sebelum menentukan apakah *flow entry* akan diperbarui informasi nya, atau akan dihapus dari *flow table*. *Flow entry* akan dihapus ketika tidak ada informasi



perubahan atau penambahan paket. Perintah penghapusan *flow entry* dilakukan ketika kapasitas *flow table* mencapai 80% dari batas maksimal.

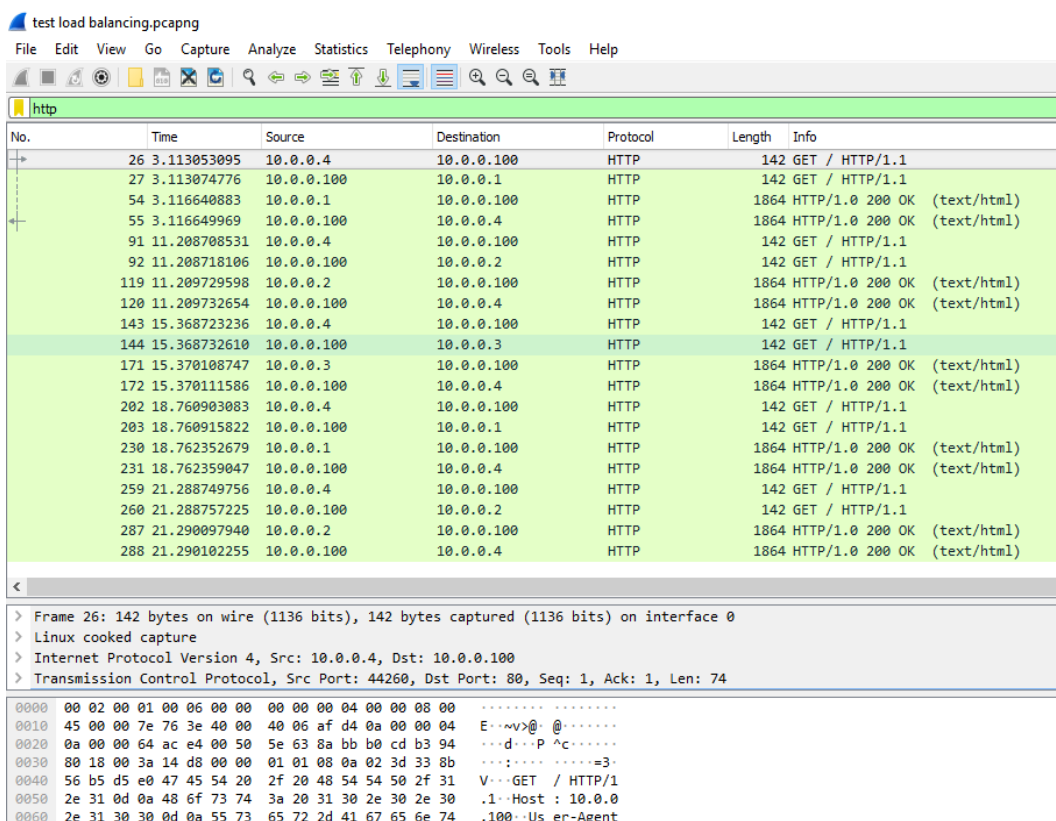
## BAB 5 PENGUJIAN DAN ANALISIS

Bab ini menjelaskan mekanisme dan hasil pengujian dari sistem yang telah diimplementasikan. Pengujian dilakukan dengan tujuan melihat apakah fungsionalitas yang telah dirancang sebelumnya telah terpenuhi. Data dari hasil pengujian akan digunakan untuk menganalisis apakah sistem dapat berjalan sesuai harapan serta dijadikan sebagai bahan acuan dalam mengambil kesimpulan pada penelitian kali ini.

### 5.1 Pengujian *Round-Robin Server Load Balancing*

Pengujian *round-robin server load balancing* dilakukan untuk menguji fungsional dari sistem *server load balancing*. Sistem *server load balancing* akan membagi beban *request* ke server dengan algoritme *round-robin*. Pengujian ini dilakukan untuk mendapatkan informasi apakah sistem *server load balancing* telah berjalan sesuai dengan algoritme *round-robin*.

Untuk melakukan pengecekan apakah *round-robin load balancer* dapat bekerja sesuai dengan yang diharapkan, maka perlu dilakukan percobaan dengan mengirimkan *request* beberapa kali ke server dan melakukan *packets capture* menggunakan *tools* Wireshark.



The screenshot shows a Wireshark packet capture titled "test load balancing.pcapng". The interface displays a list of captured packets, with the selected packet (No. 26) expanded to show its details and raw data. The packet list shows a sequence of HTTP GET requests from source IP 10.0.0.4 to various destination IPs (10.0.0.100, 10.0.0.1, 10.0.0.100, 10.0.0.4, 10.0.0.100, 10.0.0.2, 10.0.0.100, 10.0.0.4, 10.0.0.100, 10.0.0.3, 10.0.0.100, 10.0.0.4, 10.0.0.100, 10.0.0.1, 10.0.0.100, 10.0.0.4, 10.0.0.100, 10.0.0.2, 10.0.0.100, 10.0.0.4) and corresponding HTTP 200 OK responses. The details pane for packet 26 shows it is an HTTP GET request to 10.0.0.100 on port 80, with a length of 142 bytes. The raw data pane shows the hexadecimal and ASCII representation of the packet, including the "GET / HTTP/1.1" request line and the "Host: 10.0.0.100" header.

No.	Time	Source	Destination	Protocol	Length	Info
26	3.113053095	10.0.0.4	10.0.0.100	HTTP	142	GET / HTTP/1.1
27	3.113074776	10.0.0.100	10.0.0.1	HTTP	142	GET / HTTP/1.1
54	3.116640883	10.0.0.1	10.0.0.100	HTTP	1864	HTTP/1.0 200 OK (text/html)
55	3.116649969	10.0.0.100	10.0.0.4	HTTP	1864	HTTP/1.0 200 OK (text/html)
91	11.208708531	10.0.0.4	10.0.0.100	HTTP	142	GET / HTTP/1.1
92	11.208718106	10.0.0.100	10.0.0.2	HTTP	142	GET / HTTP/1.1
119	11.209729598	10.0.0.2	10.0.0.100	HTTP	1864	HTTP/1.0 200 OK (text/html)
120	11.209732654	10.0.0.100	10.0.0.4	HTTP	1864	HTTP/1.0 200 OK (text/html)
143	15.368723236	10.0.0.4	10.0.0.100	HTTP	142	GET / HTTP/1.1
144	15.368732610	10.0.0.100	10.0.0.3	HTTP	142	GET / HTTP/1.1
171	15.370108747	10.0.0.3	10.0.0.100	HTTP	1864	HTTP/1.0 200 OK (text/html)
172	15.370111586	10.0.0.100	10.0.0.4	HTTP	1864	HTTP/1.0 200 OK (text/html)
202	18.760903083	10.0.0.4	10.0.0.100	HTTP	142	GET / HTTP/1.1
203	18.760915822	10.0.0.100	10.0.0.1	HTTP	142	GET / HTTP/1.1
230	18.762352679	10.0.0.1	10.0.0.100	HTTP	1864	HTTP/1.0 200 OK (text/html)
231	18.762359047	10.0.0.100	10.0.0.4	HTTP	1864	HTTP/1.0 200 OK (text/html)
259	21.288749756	10.0.0.4	10.0.0.100	HTTP	142	GET / HTTP/1.1
260	21.288757225	10.0.0.100	10.0.0.2	HTTP	142	GET / HTTP/1.1
287	21.290097940	10.0.0.2	10.0.0.100	HTTP	1864	HTTP/1.0 200 OK (text/html)
288	21.290102255	10.0.0.100	10.0.0.4	HTTP	1864	HTTP/1.0 200 OK (text/html)

Gambar 5.1 Hasil *Packet Capture* Round-Robin Server Load Balancing

Gambar 5.1 menunjukkan hasil dari *packets capture* round-robin *server load balancing*. Dari pengujian sistem *server load balancing*, diperoleh hasil bahwa algoritme round-robin telah berjalan sesuai dengan skema yang mendukung lingkungan uji. Ketika paket *request* pertama dikirimkan, maka akan ditujukan ke *server 1*, *request* selanjutnya dikirimkan ke *server 2*, 3 hingga kembali lagi ditujukan ke *server 1* secara sirkular.

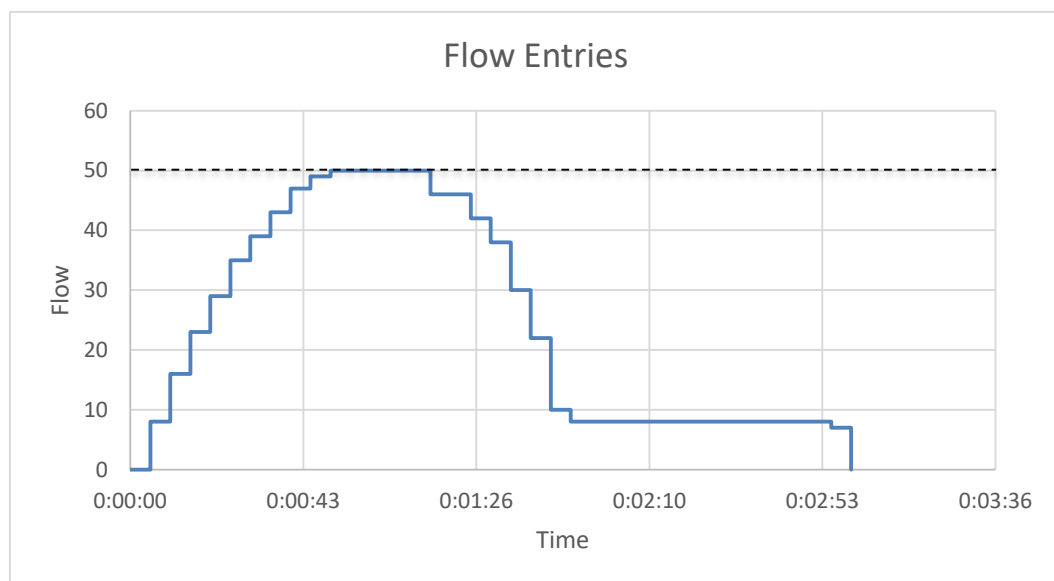
## 5.2 Pengujian Mekanisme *Flow Removal*

### 5.2.1 *Idle Timeouts*

#### 5.2.1.1 Mekanisme Pengujian

Pengujian *idle timeouts* dilakukan dengan cara mengatur *idle timeouts* pada setiap *flow entry* dengan nilai 60 detik. Pengujian dilakukan dengan mengirimkan 100 paket dengan jeda tiap pengiriman paket sebesar 0.5 detik. Dalam pengujian ini, *flow entry* seharusnya dihapus dari *flow table* ketika dalam waktu 60 detik terakhir tidak ada paket yang cocok dengan *flow entry* tersebut. Pengujian ini dilakukan untuk menilai apakah *idle timeout* berjalan sesuai fungsionalitas.

#### 5.2.1.2 Hasil dan Pembahasan



**Gambar 5.2 Grafik Kondisi *Flow Table* Hasil Pengujian *Idle Timeouts* 60 detik**

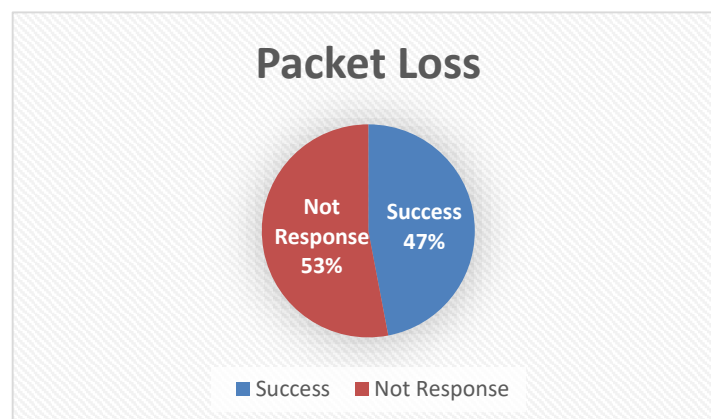
Berdasarkan Gambar 5.2, grafik tersebut menunjukkan bahwa *idle timeouts* berhasil menghapus *flow entry* setelah selama 60 detik terakhir tidak ada aktivitas atau tidak ada paket yang cocok dengan *flow entries* yang ada di dalam *flow table*. Namun karena dalam pengujian ini telah disusun skenario batas maksimal *flow entries*, maka tidak seluruh *request* dapat menempati *flow table* dan mendapat pelayanan dari *server*. Hal ini dapat dilihat dari grafik, yang mana *flow entries* berhenti bertambah ketika telah menyentuh angka 50, karena batas maksimal *flow table* adalah 50 *entries*.

Bukti bahwa tidak semua *request* dapat dilayani oleh *server* dapat dilihat pada gambar 5.3. Di dalam gambar 5.3 menunjukkan bahwa *request* ke 48 tidak dapat terfasilitasi untuk menempati *flow table* karena pada saat itu kondisi *flow entries* telah mencapai batas maksimal kemampuan *flow table*. Dalam pengujian ini terdapat 53% paket tidak dapat mendapatkan pelayanan dari *server*.

```
Request dari alamat : 10.0.0.25
Traceback (most recent call last):
  File "spoofed_packets.py", line 13, in <module>
    r = requests.get('http://10.0.0.100/')
  File "/home/agungwahyusb/.local/lib/python2.7/site-packages/requests/api.py",
line 75, in get
    return request('get', url, params=params, **kwargs)
  File "/home/agungwahyusb/.local/lib/python2.7/site-packages/requests/api.py",
line 60, in request
    return session.request(method=method, url=url, **kwargs)
  File "/home/agungwahyusb/.local/lib/python2.7/site-packages/requests/sessions.
py", line 533, in request
    resp = self.send(prepare, **send_kwargs)
  File "/home/agungwahyusb/.local/lib/python2.7/site-packages/requests/sessions.
py", line 646, in send
    r = adapter.send(request, **kwargs)
  File "/home/agungwahyusb/.local/lib/python2.7/site-packages/requests/adapters.
py", line 516, in send
    raise ConnectionError(e, request=request)
requests.exceptions.ConnectionError: HTTPConnectionPool(host='10.0.0.100', port=
80): Max retries exceeded with url: / (Caused by NewConnectionError('<urllib3.co
nnection.HTTPConnection object at 0x7f94afc850d0>: Failed to establish a new con
nection: [Errno 110] Connection timed out',))
root@155150200111070:~/Bismillah SKRIPSI/code# █
```

**Gambar 5.3 Flow Entries Mencapai Batas Maksimal**

Berdasarkan hasil pengujian dengan menerapkan mekanisme *flow expiry* khususnya *idle timeouts* diperoleh hasil bahwa setiap *flow entry* diberikan *idle timeouts* yang sama yaitu 60 detik. Dari pengujian ini dapat dianalisis bahwa sebuah *flow entry* baru akan dihapus 60 detik setelah *flow* tersebut tidak ada aktivitas atau tidak ada paket yang cocok. Sehingga ketika ada *flow entry* baru yang ingin menempati *flow table* pada saat semua *flow entry* belum melewati waktu *expiry* atau dengan kata lain *flow table* dalam kondisi penuh, maka *flow entry* baru tersebut tidak dapat terfasilitasi. Hal tersebut dapat dilihat pada gambar 5.3. Kondisi *flow table* penuh atau biasa disebut dengan *flow table overflow* menyebabkan *request* tidak dapat sampai ke tujuan.



**Gambar 5.4 Persentase Packet Loss**

Ketika kondisi *flow table* penuh, maka paket selanjutnya tidak dapat menempati *flow table*. Dari 100 paket *request* yang dikirimkan 53% tidak mendapatkan *response* dari *server* karena tidak dapat terfasilitasi untuk menempati *flow table*. Prosentasi *packet loss* dapat dilihat pada gambar 5.4.

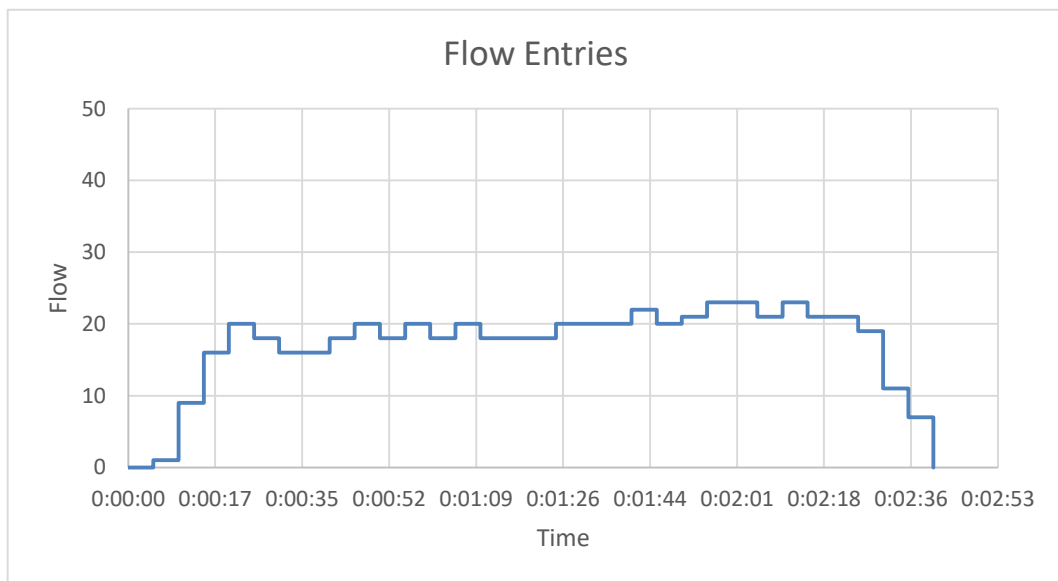
### 5.2.2 Flow Modification

#### 5.2.2.1 Mekanisme Pengujian

Pengujian *flow modification message* dilakukan untuk menghapus *flow entry* dengan mengirimkan pesan ke *controller* ketika tidak ada paket yang cocok. Dalam pengujian ini metode yang digunakan untuk melakukan penghapusan terhadap *flow entry* adalah *flow modification message delete flow*, dengan menggunakan *idle timeouts* 6 detik.

Pengujian dilakukan dengan mengirimkan 100 paket dengan jeda waktu 0.5 detik setiap kali mengirimkan paket. Gambar 5.5 menunjukkan bahwa mekanisme penghapusan *flow entry* menggunakan *flow modification message* telah berjalan dengan baik dengan menghapus *flow entries* yang tidak ada aktivitas atau tidak ada paket yang cocok.

#### 5.2.2.2 Hasil dan Pembahasan



**Gambar 5.5 Grafik Kondisi *Flow Table* Hasil Pengujian *Flow Modification***

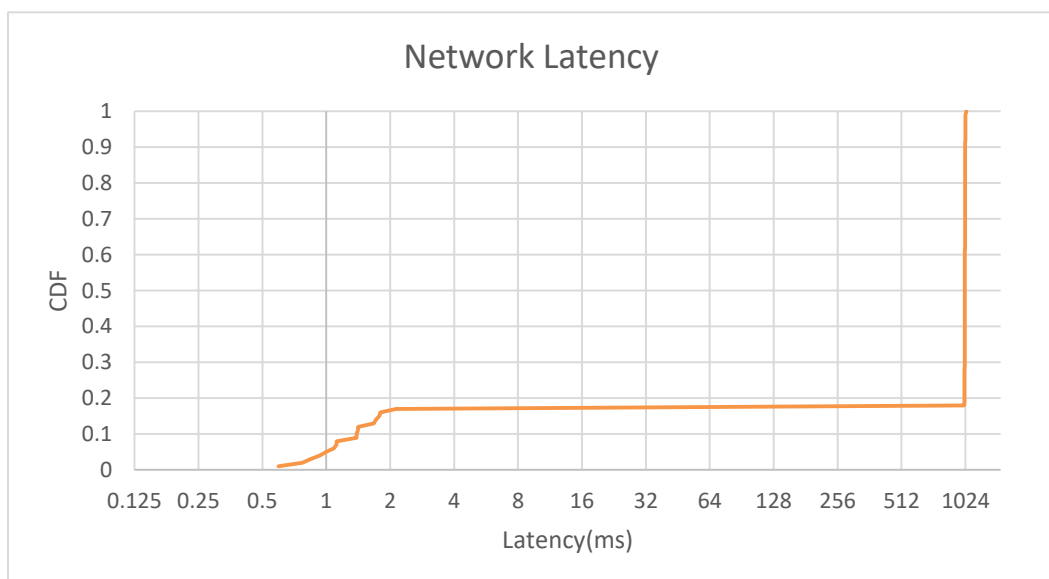
Dalam pengujian ini, 100 paket *request* yang dikirimkan seluruhnya dapat dilayani oleh *server*. Hal ini dapat dilihat pada gambar 5.6. Sedangkan gambar 5.5 menunjukkan bahwa kondisi *flow entries* tidak pernah mencapai batas maksimal, hanya berkisar pada angka 16 – 23 *flow entries*. Berdasarkan hasil pengujian dengan menerapkan mekanisme *flow modification message delete flow* dan dengan memberikan nilai *idle timeout* 6 detik, diperoleh hasil bahwa sebuah *flow entries* akan dihapus ketika tidak ada aktivitas atau tidak ada paket yang cocok. Dari pengujian ini menunjukkan bahwa *flow entries* langsung dihapus dari *flow table* ketika tidak ada paket yang cocok seperti yang dapat dilihat pada Gambar

5.5. Sehingga kondisi ini mengharuskan *controller* untuk selalu memperbarui *flow table* dan mendefinisikan *flow entry* baru setiap menerima *request* dari *source IP* yang baru.



**Gambar 5.6 Persentase *Packet Loss***

Meskipun pada Gambar 5.6 menunjukkan bahwa semua *request* yang dikirimkan ke *server* dapat ditangani dengan baik, Namun grafik *network latency* pada gambar 5.7 menunjukkan bahwa dalam pengujian ini 83% dari jumlah paket yang dikirimkan memiliki nilai *network latency* yang tinggi yaitu antara 1009 – 1035 *millisecond*. Hal ini dikarenakan ketika hanya menerapkan mekanisme *flow modification* tanpa mengatur nilai pada variabel *idle timeouts*, maka setiap *flow entry* yang tidak ada aktivitas akan langsung dihapus tanpa menunggu *idle timeout* mencapai batasnya, baik ketika kondisi *flow entries* mendekati penuh atau jauh dibawah batas maksimal kapasitas *flow table*.



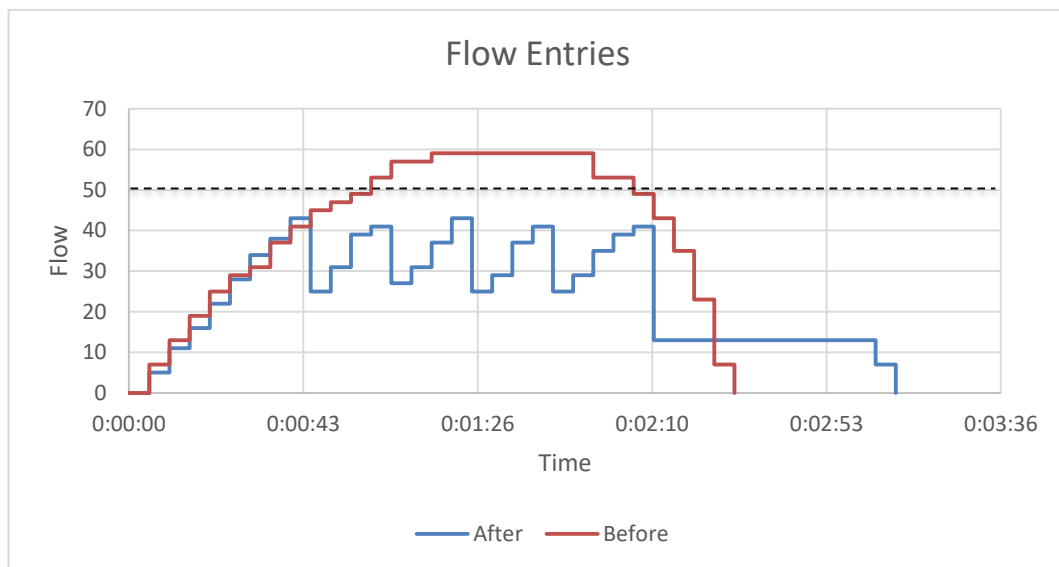
**Gambar 5.7 Grafik *Network Latency* pada Implementasi *Flow Modification***

### 5.2.3 Idle Timeouts dan Flow Modification

#### 5.2.3.1 Mekanisme Pengujian

Pengujian *idle timeouts* dan *flow modification message* dilakukan dengan memadukan antara 2 mekanisme penghapusan *flow entries* yaitu *flow expiry* dan *flow modification message*. Pengujian ini dilakukan dengan skenario yang telah ditetapkan sebelumnya yaitu kapasitas *flow table* adalah 50 *flow entries* dan mengirimkan *request* sebanyak 100 kali dengan jeda pengiriman setiap 0.5 detik.

#### 5.2.3.2 Hasil dan Pembahasan



**Gambar 5.8 Grafik Hasil Pengujian 2 Mekanisme Flow Removal**

Dari Gambar 5.8 dapat dilihat bahwa terdapat perbedaan grafik sebelum dan setelah menerapkan mekanisme *flow modification message delete flow*. Garis warna oranye menunjukkan kondisi dimana jumlah *flow entries* melebihi kapasitas maksimal *flow table* yaitu 50. Hal ini dikarenakan *flow entries* hanya dihapus dengan menggunakan mekanisme *idle timeouts* 60 detik. Garis warna biru menunjukkan kondisi *flow entries* yang stabil dibawah kapasitas maksimal dari *flow table*. Hal ini dikarenakan pengujian dilakukan dengan menerapkan 2 mekanisme *flow removal*, yaitu *idle timeouts* 60 detik dan *flow modification message delete flow*. Mekanisme *flow modification message delete flow* akan berjalan ketika jumlah *flow entries* lebih dari 80% dari batas maksimal kemampuan *flow table*.

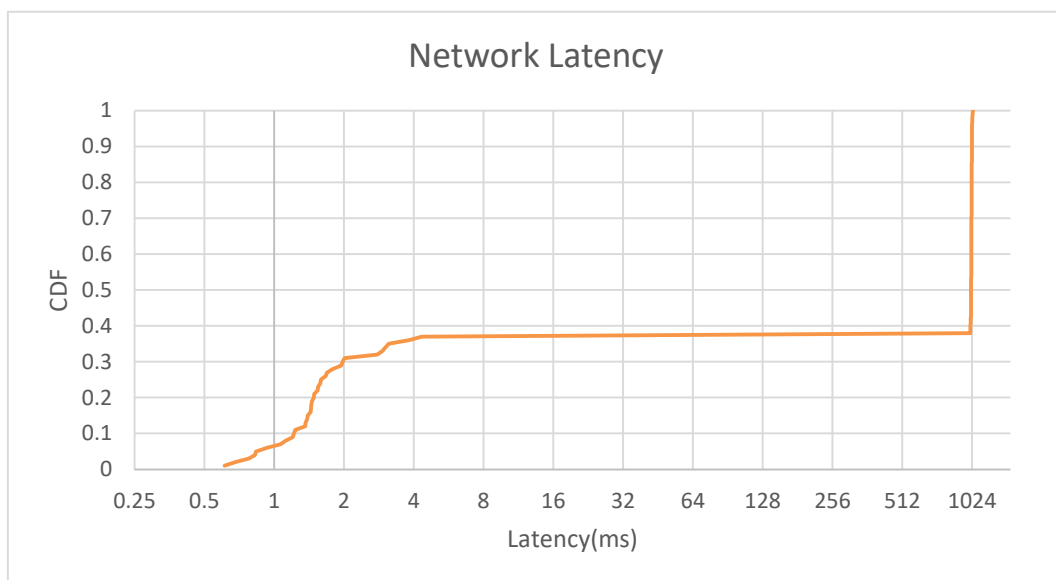
Jumlah Paket	Sebelum	Sesudah
Success	47%	100%
Not Response	53%	0%

**Gambar 5.9 Perbandingan Packet Loss Sebelum dan Sesudah Optimasi**

Gambar 5.9 menunjukkan perbandingan antara *packet loss* sebelum dan sesudah menerapkan 2 mekanisme *flow removal* untuk tujuan optimasi. Sebelum

menerapkan mekanisme *flow removal*, 58% dari jumlah paket yang dikirimkan tidak mendapatkan *response* dari *server* karena tidak memiliki kesempatan untuk menempati *flow table*. Sedangkan setelah menerapkan mekanisme *flow removal*, 100% dari jumlah paket *request* yang dikirimkan dapat menempati *flow table* dan mendapatkan *response* dari *server*.

Pengujian kinerja *server load balancing* juga dilakukan untuk mengetahui apakah setelah menerapkan 2 mekanisme *flow removal* mempengaruhi kinerja dari sistem. Pada gambar 5.10 dapat dilihat hasil kinerja *server* dari nilai *network latency*. Untuk melihat perbedaan yang cukup signifikan antara masing – masing *latency* dapat dilihat dengan membandingkan garis warna oranye pada gambar 5.10. Berdasarkan grafik tersebut diperoleh hasil bahwa 37 dari 100 atau 37% *request* yang dikirimkan ke *server* memiliki nilai *latency* yang kecil yaitu antara 0,6 – 4,3 *millisecond*. Sedangkan sekitar 63% *request* yang dikirimkan ke *server* memiliki nilai *latency* yang cukup besar, yaitu antara 1006 – 1031 *millisecond*.



**Gambar 5.10 Grafik *Network Latency* pada Implementasi 2 Mekanisme *Flow Removal***

Berdasarkan hasil pengujian dengan menerapkan 2 mekanisme *flow removal* dan dengan skenario yang telah ditetapkan, diperoleh hasil bahwa ketika *flow table* mendekati kapasitas maksimal maka mekanisme *flow modification* message akan mengirimkan pesan kepada *controller* untuk melakukan penghapusan terhadap *flow entries* yang tidak ada aktivitas. Ketika kondisi *flow table* normal atau dalam hal ini kurang dari 80% dari kapasitas maksimal, disana peran *idle timeouts* berjalan. Dari hasil pengujian dapat dianalisis bahwa dengan menerapkan 2 mekanisme *flow removal* ini maka tujuan untuk mengoptimalkan *flow entries* dapat diwujudkan, terbukti dengan kapasitas *flow table* yang tidak pernah penuh sehingga semua *flow entry* baru dapat terfasilitasi dan *request* dapat dikirimkan sampai ke *server*. Namun dalam menerapkan 2 mekanisme *flow removal*, hal yang perlu diperhatikan adalah mengatur kapan mekanisme ini



berjalan untuk melakukan pengecekan terhadap ktaondisi *flow entries* secara *real time*, karena pengecekan kondisi *flow entries* menjadi salah satu faktor yang mempengaruhi optimal dan tidaknya implementasi dari 2 mekanisme *flow removal* ini.

### 5.3 Analisis Hasil Keseluruhan Pengujian

Dari hasil pengujian sistem *server load balancing* yang ditunjukkan oleh masing – masing grafik *flow entries*, bahwa penerapan 2 mekanisme *flow removal* dapat mencegah *flow table overflow*. Selain itu dari pengujian *network latency* pada gambar 5.7 dan 5.10 menunjukkan bahwa penerapan mekanisme *flow removal* pada sistem *stateless server load balancing* berbasis *software defined networking* mempengaruhi kinerja dari *server*. Ketika sebuah *client* mengirimkan *request* pertama kali maka sebuah *flow entry* baru akan dituliskan di dalam *flow table* oleh *controller*. Sedangkan ketika *client* mengirimkan *request* berikutnya dan terdapat paket yang cocok dengan salah satu *flow entries*, maka *controller* tidak perlu menuliskan sebuah *flow entry* baru, namun menggunakan kembali *flow entries* yang sudah ada.

Penggunaan kembali *flow entries* ini mempengaruhi nilai *network latency*. Perbedaan yang cukup signifikan terkait pengaruh penggunaan kembali *flow entries* dapat di lihat pada gambar 5.7 dan 5.10 yang menunjukkan bahwa penggunaan kembali *flow entries* dapat memangkas nilai *latency*. Dalam pengujian *flow modification* memang seluruh paket yang dikirimkan dapat dilayani oleh *server*. Namun pada skenario ini, 83% dari jumlah paket *request* yang dikirimkan memiliki nilai dari *network latency* yang besar, yaitu antara 1006 – 1031 *millisecond*.

## BAB 6 PENUTUP

### 6.1 Kesimpulan

Berdasarkan penelitian dengan implementasi 2 mekanisme *flow removal*, maka dapat diambil kesimpulan dari penelitian sebagai berikut.

1. *Flow removal* dapat diimplementasikan dengan 2 cara, yaitu dengan mekanisme *flow expiry* dan mekanisme *flow modification*. Jika hanya menerapkan mekanisme *flow expiry*, maka pada waktu tertentu *flow entries* akan mencapai batas maksimalnya, sehingga menyebabkan *flow table overflow*. Sedangkan jika hanya menerapkan mekanisme *flow modification*, maka seluruh *flow entry* yang baru dapat terfasilitasi untuk menempati *flow table*. Namun hasil pengujian *network latency* pada penerapan mekanisme *flow modification* menunjukkan bahwa beban *controller* berat. Penelitian ini mengimplementasikan 2 mekanisme *flow removal* (*flow expiry* dan *flow modification*). Mekanisme *flow modification* berjalan dengan mengirimkan pesan penghapusan secara eksplisit kepada *controller* ketika kondisi *flow entries* melebihi 80% dari kapasitas maksimal. Mekanisme *flow expiry* diimplementasikan dengan memberikan nilai *idle timeouts* 60 detik. *Idle timeouts* akan bekerja ketika kondisi *flow entries* dibawah 80% dari kapasitas maksimal *flow table*, sehingga *flow entries* akan dihapus dari *flow table* ketika tidak ada paket yang cocok selama waktu yang diberikan pada *idle timeouts* telah terlampaui.
2. Penggunaan 2 mekanisme *flow removal* mampu mengendalikan kondisi *flow entries* agar tetap pada kondisi dibawah 80% dari kapasitas maksimal *flow table*. Dari hasil pengujian dengan mengirimkan *request* setiap 0.5 detik, diperoleh hasil bahwa sistem berhasil menangani 100% paket yang ingin menempati *flow table*, sehingga dalam hal ini tercapai tujuan optimasi *flow entries* untuk mencegah *flow table overflow* pada round-robin *server load balancing* berbasis *software defined networking*.
3. Berdasarkan analisis kinerja sistem *stateless server load balancing*, selain dapat melayani 100% *request* yang dikirimkan oleh *client*, diperoleh hasil bahwa penggunaan kembali *flow entries* dapat memangkas nilai *network latency* dari komunikasi antara *client* dan *server*. Dari 100% paket yang dapat dilayani oleh *server*, 37% *request* yang dikirimkan ke *server* memiliki nilai *network latency* rendah, yaitu antara 0,6 – 4,3 *millisecond*. Sedangkan 63% *request* yang dikirimkan ke *server* memiliki nilai *network latency* yang cukup tinggi, yaitu antara 1006 – 1031 *millisecond*.

### 6.2 Saran

Beberapa saran yang diberikan untuk pengembangan pada penelitian masa mendatang berdasarkan hasil penelitian ini adalah sebagai berikut.

1. Perlu dilakukan analisis kinerja sistem dari implementasi mekanisme *flow removal* berdasarkan jenis paket yang berbeda - beda untuk mengetahui batas kemampuan sistem dalam optimasi.
2. Perlu dilakukan evaluasi terkait dengan aturan penghapusan terhadap *flow entry* yang tidak ada aktivitas.
3. Perlu dilakukan penelitian dengan menerapkan sistem pada jaringan fisik asli (Arsitektur SDN dan OpenFlow *switch*).

## DAFTAR PUSTAKA

- Al-Najjar, A., Layeghy, S. & Portmann, M., 2016. Pushing SDN to The End-Host, Network Load Balancing Using OpenFlow. *IEEE International Conference on Pervasive Computing and Communication Workshops, Percom Workshop*.
- Ardy, L. F. I., Bhawiyuga, A. & Yahya, W., 2018. Implementasi Load Balancer berdasarkan Server Status pada Arsitektur Software Defined Network(SDN). *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, Mei, Volume 2, pp. 2135 - 2143.
- Boero, L. et al., 2016. BeaQoS: Load Balancing and Deadline Management of Queues in an OpenFlow SDN Switch. *Computer Network*.
- Ellrod, C., 2010. *Load Balancing Round-Robin*. [Online] Available at: <https://www.citrix.com/blogs/2010/09/03/load-balancing-round-robin> [Accessed 15 Januari 2019].
- Guo, Z. et al., 2018. Balancing Flow Table Occupancy and Link Utilization in Software Defined Networks. *Future Generation Computer Systems*, Volume 89, pp. Pages 213-223.
- He, C.-H. et al., 2018. A Zero Flow Entry Expiration Timeout P4 Switch.
- Karim, H., Primananda, R. & Yahya, W., 2019. Implementasi Load Balancing Web Server dengan Algoritme Weighted Least Connection pada Software Defined Network. *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, Volume III, pp. 7705-7714.
- KeyCDN Company, 2018. *keycdn*. [Online] Available at: <https://www.keycdn.com/support/what-is-latency> [Accessed 7 25 2019].
- McKeown, N. et al., 2008. OpenFlow: Enabling Innovation in Campus Networks. *Computer Communication Review*, Volume 38, pp. 69-74.
- Mininet Team, 2016. *Mininet Overview*. [Online] Available at: <http://www.mininet.org/overview/> [Accessed 17 February 2019].
- Nadeau, T. D. & Gray, K., 2013. *SDN : Software Defined Networks*. 1st ed. United States of America: O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.
- Negara, L. R. C., Yahya, W. & Primananda, R., 2018. Analisis Dan Implementasi Load Balancing Pada Web Server dengan Algoritme Shortest Delay pada Software Defined Network. *Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer*, Volume II, pp. 2791-2797.
- Nippon Telegraph and Telephone Corporation Revision 56e8fb3f, 2014. *Ryu : Writing Your Application*. [Online]

Available at: [https://ryu.readthedocs.io/en/latest/writing\\_ryu\\_app.html](https://ryu.readthedocs.io/en/latest/writing_ryu_app.html)  
[Accessed 6 5 2019].

Open Networking Foundation, 2012. *Open Networking Organization*. [Online]  
Available at: <https://www.opennetworking.org/wp-content/uploads/.../openflow-spec-v1.3.0.pdf>  
[Accessed 4 Januari 2019].

Pemberton, D., Linton, A. & Russell, S., 2014. <https://nsrc.org/workshops/2014/nznog-sdn/raw-attachment/wiki/WikiStart/Ryu.pdf>. [Online]  
[Accessed 11 Januari 2019].

Rahman, M., Iqbal, S. & Gao, J., 2014. Load Balancer as a Service in Cloud Computing. *IEEE 8th International Symposium on Service Oriented System Engineering*.

Sudiyatmoko, A. R., Hertiana, S. N. & Negara, R. M., 2016. Analisis Performansi Perutingan Link State Menggunakan Algoritma Djikstra pada Platform Software Defined Network(SDN).

Yang, H. & Riley, G. F., 2018. Machine Learning Based Proactive Flow Entry Deletion for OpenFlow. *2018 IEEE International Conference on Communications (ICC)*, pp. 1-6.

Zarek, A., 2012. OpenFlow Timeouts Demystified.

Zha, J., Wang, J., Han, R. & Maoqiang, S., 2010. Research on Load Balance of Service Capability Interaction Management. *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*.

Zhong, H., Fang, Y. & Cui, J., 2016. LBBSRT: An efficient SDN load balancing scheme based on server response time. *Future Generation Computer Systems*.

## LAMPIRAN A KODE PROGRAM

Topologi Jaringan	
1	"""Custom topology example
2	
3	Two directly connected switches plus a host for each switch:
4	
5	host --- switch --- switch --- host
6	
7	Adding the 'topos' dict with a key/value pair to generate our
8	newly defined
9	topology enables one to pass in '--topo=mytopo' from the
10	command line.
11	"""
12	
13	from mininet.topo import Topo
14	import SimpleHTTPServer
15	import SocketServer
16	
17	class MyTopo( Topo ):
18	"Simple topology example."
19	
20	def __init__( self ):
21	"Create custom topo."
22	
23	# Initialize topology
24	Topo.__init__( self )
25	
26	# Add hosts and switches
27	client1 = self.addHost( 'h1' )
28	server1 = self.addHost( 'h2' )
29	server2 = self.addHost( 'h3' )
30	server3 = self.addHost( 'h4' )
31	switch1 = self.addSwitch( 's1' )
32	
33	# Add links
34	self.addLink( client1, switch1 )
35	self.addLink( server1, switch1 )
36	self.addLink( server2, switch1 )
37	self.addLink( server3, switch1 )
38	
39	topos = { 'mytopo': ( lambda: MyTopo() ) }
40	
41	#sudo ovs-vsctl -- --id=@ft create Flow_Table flow_limit=50
42	overflow_policy=refuse -- set Bridge s1 flow_tables=0=@ft
43	

Sistem Server Load Balancing Round Robin	
1	from operator import attrgetter
2	
3	from ryu.base import app_manager
4	from ryu.controller import ofp_event
5	from ryu.controller.handler import CONFIG_DISPATCHER,
6	MAIN_DISPATCHER
7	from ryu.controller.handler import set_ev_cls
8	from ryu.ofproto import ofproto_v1_3

```

9  from ryu.lib.packet import packet
10 from ryu.lib.packet import ethernet
11 from ryu.lib.packet import ether_types
12 import logging
13 import random
14 from ryu.lib.packet import ipv4
15 from ryu.lib.packet import tcp
16 from ryu.lib.packet import arp
17 from ryu.lib.packet import icmp
18
19 from ryu.controller.handler import DEAD_DISPATCHER
20 from ryu.lib import hub
21 import json
22 import time
23
24 #Pingall requiried before trying load balancing functionality
25
26 class loadbalancer(app_manager.RyuApp):
27
28     OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
29
30     def __init__(self, *args, **kwargs):
31         super(loadbalancer, self).__init__(*args,
32 **kwargs)
33         self.mac_to_port = {}
34         self.serverlist = []
35         self.serverlist.append({'ip':"10.0.0.1",
36 'mac':"00:00:00:00:00:01", 'server_port' : "1"})
37         self.serverlist.append({'ip':"10.0.0.2",
38 'mac':"00:00:00:00:00:02", 'server_port' : "2"})
39         self.serverlist.append({'ip':"10.0.0.3",
40 'mac':"00:00:00:00:00:03", 'server_port' : "3"})
41         self.virtual_lb_ip = "10.0.0.100"
42         self.virtual_lb_mac = "AB:BC:CD:EF:F1:12"
43         self.serverNumber = 0
44         self.logger.info("Initialized new Object
45 instance data")
46
47         self.flow_monitor = 0
48         self.datapaths = {}
49         self.monitor_thread = hub.spawn(self._monitor)
50
51         self.ctemp = []
52         self.dtemp = []
53         self.ptemp = []
54         self.btemp = []
55         self.cookie_temp = 0
56         self.longest_duration = 0
57         self.cookie_idx0 = 0xffffffffffffffff
58         self.clongest_dur = 0xffffffffffffffff
59
60         @set_ev_cls(ofp_event.EventOFPSwitchFeatures,
61 CONFIG_DISPATCHER)
62         def switch_features_handler(self, ev):
63             datapath = ev.msg.datapath
64             ofproto = datapath.ofproto
65             parser = datapath.ofproto_parser
66
67             # install table-miss flow entry

```

```

68 #
69 # We specify NO BUFFER to max_len of the output
70 action due to
71 # OVS bug. At this moment, if we specify a
72 lesser number, e.g.,
73 # 128, OVS will send Packet-In with invalid
74 buffer_id and
75 # truncated packet data. In that case, we cannot
76 output packets
77 # correctly. The bug has been fixed in OVS
78 v2.1.0.
79 match = parser.OFPMatch()
80 actions =
81 [parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,
82 ofproto.OFPCML_NO_BUFFER)]
83 self.add_flow(datapath, 0, match, actions)
84 # self.logger.info("Set Config data for new
85 Object Instance")
86
87 def add_flow(self, datapath, priority, match, actions,
88 buffer_id=None):
89 # self.logger.info("Now adding flow")
90 ofproto = datapath.ofproto
91 parser = datapath.ofproto_parser
92
93 inst =
94 [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
95 actions)]
96
97 if buffer_id:
98 mod =
99 parser.OFPFlowMod(datapath=datapath, buffer_id=buffer_id,
100 priority=priority, match=match, instructions=inst)
101 else:
102 mod =
103 parser.OFPFlowMod(datapath=datapath, priority=priority,
104 match=match, instructions=inst)
105 datapath.send_msg(mod)
106 # self.logger.info("Done adding flows")
107
108 def delete_flow(self, ev):
109 msg = ev.msg
110 datapath = msg.datapath
111 ofproto = datapath.ofproto
112 parser = datapath.ofproto_parser
113
114 cookie = self.cookie_temp
115 cookie_mask = self.cookie_temp
116 table_id = 0
117 idle_timeout = hard_timeout = 0
118 priority = 32768
119 buffer_id = ofproto.OFP_NO_BUFFER
120
121 req = parser.OFPFlowMod(datapath, cookie,
122 cookie_mask,
123 table_id,
124 ofproto.OFPFC_DELETE,
125

```



```

127
128 idle_timeout, hard_timeout,
129
130 priority, buffer_id,
131
132 ofproto.OFPP_ANY, ofproto.OFPG_ANY,
133
134 ofproto.OFPFF_SEND_FLOW_REM,)
135
136         datapath.send_msg(req)
137         print("flow dengan cookie " +
138 hex(self.cookie_temp) + " telah dihapus")
139
140         def handle_arp_for_server(self, dmac, dip):
141             # self.logger.info("Handling ARP Reply for
142 virtual Server IP")
143             #handle arp request for virtual Server IP
144             #checked Wireshark for sample pcap for arp-reply
145             #build arp packet - format source web link
146 included in reference
147             hrdw_type = 1 #Hardware Type: ethernet 10mb
148             protocol = 2048 #Layer 3 type: Internet Protocol
149             hrdw_add_len = 6 # length of mac
150             prot_add_len = 4 # lenght of IP
151             opcode = 2 # arp reply
152             server_ip = self.virtual_lb_mac #sender address
153             server_mac = self.virtual_lb_ip #sender IP
154             arp_target_mac = dmac #target MAC
155             arp_target_ip = dip #target IP
156
157             ether_type = 2054 #ethertype ARP
158
159             pack = packet.Packet()
160             eth_frame = ethernet.ethernet(dmac, server_ip,
161 ether_type)
162             arp_rpl_frame = arp.arp(hrdw_type, protocol,
163 hrdw_add_len, prot_add_len, opcode, server_ip, server_mac,
164 arp_target_mac, arp_target_ip)
165             pack.add_protocol(eth_frame)
166             pack.add_protocol(arp_rpl_frame)
167             pack.serialize()
168             # self.logger.info("Done handling ARP Reply")
169             return pack
170
171
172         @set_ev_cls(ofp_event.EventOFPPacketIn, MAIN_DISPATCHER)
173         def _packet_in_handler(self, ev):
174             # self.logger.info("Entered main mode event
175 handling")
176             # # If you hit this you might want to increase
177             # # the "miss_send_length" of your switch
178             # if ev.msg.msg_len < ev.msg.total_len:
179             #     self.logger.debug("packet truncated: only
180 %s of %s bytes",
181 #                                     ev.msg.msg_len,
182 ev.msg.total_len)
183             if self.serverNumber == 3:
184                 self.serverNumber = 0
185

```

```

186         # self.logger.info("Will print data
187 now")
188         #print event data
189
190         #fetch all details of the event
191         msg = ev.msg
192         datapath = msg.datapath
193         ofproto = datapath.ofproto
194         parser = datapath.ofproto_parser
195         in_port = msg.match['in_port']
196         dpid = datapath.id
197
198         pkt = packet.Packet(msg.data)
199         eth = pkt.get_protocols(ethernet.ethernet)[0]
200
201         dst = eth.dst
202         src = eth.src
203
204         dpid = datapath.id
205         self.mac_to_port.setdefault(dpid, {})
206
207         # self.logger.info("packet in %s %s %s %s",
208 dpid, src, dst, in_port)
209
210         # learn a mac address to avoid FLOOD next time.
211         self.mac_to_port[dpid][src] = in_port
212
213         # self.logger.info("Ether Type: %s",
214 eth.ethertype)
215         if eth.ethertype == ether_types.ETH_TYPE_LLDP:
216             # ignore lldp packet
217             return
218
219         if eth.ethertype == 2054:
220             arp_head = pkt.get_protocols(arp.arp)[0]
221             if arp_head.dst_ip == self.virtual_lb_ip:
222                 #dmac and dIP for ARP Reply
223                 a_r_ip = arp_head.src_ip
224                 a_r_mac = arp_head.src_mac
225                 arp_reply =
226 self.handle_arp_for_server(a_r_mac, a_r_ip)
227                 actions =
228 [parser.OFPActionOutput(in_port)]
229                 buffer_id = msg.buffer_id #id
230 assigned by datapath - keep track of buffered packet
231                 port_no = ofproto.OFPP_ANY #for
232 any port number
233                 data = arp_reply.data
234                 out =
235 parser.OFPPacketOut(datapath=datapath, buffer_id=buffer_id,
236 in_port=port_no, actions=actions, data=data)
237                 datapath.send_msg(out)
238                 # self.logger.info("ARP Request
239 handled")
240                 return
241             else:
242                 dst = eth.dst
243                 src = eth.src
244

```

```

245                                     dpid = datapath.id
246                                     self.mac_to_port.setdefault(dpid,
247 {}))
248
249                                     # self.logger.info("packet in %s
250 %s %s %s", dpid, src, dst, in_port)
251
252                                     # learn a mac address to avoid FLOOD next time.
253                                     self.mac_to_port[dpid][src] =
254 in_port
255
256                                     if dst in self.mac_to_port[dpid]:
257                                         out_port =
258 self.mac_to_port[dpid][dst]
259                                     else:
260                                         out_port =
261 ofproto.OFPP_FLOOD
262
263                                     actions =
264 [parser.OFPActionOutput(out_port)]
265
266                                     # install a flow to avoid
267 packet_in next time
268                                     if out_port !=
269 ofproto.OFPP_FLOOD:
270                                         match =
271 parser.OFPMatch(in_port=in_port, eth_dst=dst)
272                                         # verify if we have a
273 valid buffer_id, if yes avoid to send both
274                                         # # flow_mod & packet_out
275                                         if msg.buffer_id !=
276 ofproto.OFP_NO_BUFFER:
277
278                                         self.add_flow(datapath, 1, match, actions,
279 msg.buffer_id)
280                                         return
281                                     else:
282
283                                         self.add_flow(datapath, 1, match, actions)
284                                         data = None
285                                         if msg.buffer_id ==
286 ofproto.OFP_NO_BUFFER:
287                                             data = msg.data
288
289                                         out =
290 parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
291 in_port=in_port, actions=actions, data=data)
292                                         datapath.send_msg(out)
293                                         return
294
295
296                                     try:
297                                         if pkt.get_protocols(icmp.icmp)[0]:
298
299                                         #if ip_head.proto == inet.IPPROTO_ICMP:
300                                             dst = eth.dst
301                                             src = eth.src
302
303                                     dpid = datapath.id

```

```

304                                     self.mac_to_port.setdefault(dpid,
305 {}))
306
307                                     # self.logger.info("packet in %s
308 %s %s %s", dpid, src, dst, in_port)
309
310                                     # learn a mac address to avoid FLOOD next
311 time.
312                                     self.mac_to_port[dpid][src] =
313 in_port
314
315                                     if dst in self.mac_to_port[dpid]:
316 out_port =
317 self.mac_to_port[dpid][dst]
318                                     else:
319 out_port =
320 ofproto.OFPP_FLOOD
321
322                                     actions =
323 [parser.OFPActionOutput(out_port)]
324
325                                     # install a flow to avoid
326 packet_in next time
327                                     if out_port !=
328 ofproto.OFPP_FLOOD:
329 match =
330 parser.OFPMatch(in_port=in_port, eth_dst=dst)
331                                     # verify if we have a
332 valid buffer_id, if yes avoid to send both
333                                     # # flow_mod & packet_out
334                                     if msg.buffer_id !=
335 ofproto.OFP_NO_BUFFER:
336
337                                     self.add_flow(datapath, 1, match, actions,
338 msg.buffer_id)
339                                     return
340                                     else:
341
342                                     self.add_flow(datapath, 1, match, actions)
343                                     data = None
344                                     if msg.buffer_id ==
345 ofproto.OFP_NO_BUFFER:
346                                     data = msg.data
347                                     out =
348 parser.OFPPacketOut(datapath=datapath, buffer_id=msg.buffer_id,
349 in_port=in_port,
350 actions=actions, data=data)
351                                     datapath.send_msg(out)
352                                     return
353
354                                     except:
355                                     pass
356
357
358
359                                     ip_head = pkt.get_protocols(ipv4.ipv4)[0]
360                                     # tcp_head = pkt.get_protocols(tcp.tcp)[0]
361
362

```

```

363         # pingall before executing load balancer
364     functionality
365         # self.logger.info("Trying to map ports and
366     serverlist")
367         for server in self.serverlist:
368             try:
369                 if server['mac'] in
370     self.mac_to_port[dpid]:
371                     try:
372
373                         server['server_port'] =
374     self.mac_to_port[dpid][server['mac']]
375                                     #
376     self.logger.info("Port mapping successful for Server: %s ---
377     check--- %s", server['ip'], server['server_port'])
378                                     except Exception as e:
379
380                         self.logger.info("Internal Exception: %s", e)
381                                     except Exception as e:
382                                         self.logger.info("External
383     Exception: %s", e)
384
385                         # self.logger.info("If there is no failure of
386     mapping then we are good to go...")
387                         #server choice for round robin style
388
389
390                         choice_ip =
391     self.serverlist[self.serverNumber]['ip']
392                         choice_mac =
393     self.serverlist[self.serverNumber]['mac']
394                         choice_server_port =
395     self.serverlist[self.serverNumber]['server_port']
396                         choice_server_port = int(choice_server_port)
397                         # self.logger.info("Server Choice details: \tIP
398     is %s\tMAC is %s\tPort is %s", choice_ip, choice_mac,
399     choice_server_port)
400
401
402
403                         # self.logger.info("Redirecting data request
404     packet to one of the serverlist")
405                         #Redirecting data request packet to Server
406                         match = parser.OFPMatch(in_port=in_port,
407     eth_type=eth.ethertype, eth_src=eth.src, eth_dst=eth.dst,
408     ip_proto=ip_head.proto, ipv4_src=ip_head.src,
409     ipv4_dst=ip_head.dst)
410                         # self.logger.info("Data request being sent to
411     Server: IP: %s, MAC: %s", choice_ip, choice_mac)
412                         actions =
413     [parser.OFPActionSetField(eth_dst=choice_mac),
414     parser.OFPActionSetField(ipv4_dst=choice_ip),
415     parser.OFPActionOutput(choice_server_port)]
416                         instruction1 =
417     [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
418     actions)]
419                         cookie = random.randint(0, 0xffffffffffffffff)
420
421

```

```

422         flow_mod = parser.OFPFlowMod(datapath=datapath,
423 match=match, idle_timeout=60, instructions=instruction1,
424 buffer_id = msg.buffer_id, cookie=cookie)
425         datapath.send_msg(flow_mod)
426
427         # self.logger.info("Redirection done...1")
428         # self.logger.info("Redirecting data reply
429 packet to the host")
430         #Redirecting data reply to respecitve Host
431         match =
432 parser.OFPMatch(in_port=choice_server_port,
433 eth_type=eth.ethertype, eth_src=choice_mac, eth_dst=eth.src,
434 ip_proto=ip_head.proto, ipv4_src=choice_ip,
435 ipv4_dst=ip_head.src)
436         # self.logger.info("Data reply coming from
437 Server: IP: %s, MAC: %s", choice_ip, choice_mac)
438         actions =
439 [parser.OFPActionSetField(eth_src=self.virtual_lb_mac),
440 parser.OFPActionSetField(ipv4_src=self.virtual_lb_ip),
441 parser.OFPActionOutput(in_port) ]
442
443         instruction2 =
444 [parser.OFPInstructionActions(ofproto.OFPIT_APPLY_ACTIONS,
445 actions)]
446         cookie = random.randint(0, 0xffffffffffffffff)
447         flow_mod2 = parser.OFPFlowMod(datapath=datapath,
448 match=match, idle_timeout=60, instructions=instruction2,
449 cookie=cookie)
450         datapath.send_msg(flow_mod2)
451
452         self.serverNumber = self.serverNumber + 1
453         # self.logger.info("Redirecting done...2")
454
455         #Method untuk melakukan monitoring flow table setiap x
456 detik
457         @set_ev_cls(ofp_event.EventOFPStateChange,
458 [MAIN_DISPATCHER, DEAD_DISPATCHER])
459         def _state_change_handler(self, ev):
460             datapath = ev.datapath
461             if ev.state == MAIN_DISPATCHER:
462                 if datapath.id not in self.datapaths:
463                     self.logger.debug('register
464 datapath: %016x', datapath.id)
465                     self.datapaths[datapath.id] =
466 datapath
467             elif ev.state == DEAD_DISPATCHER:
468                 if datapath.id in self.datapaths:
469                     self.logger.debug('unregister
470 datapath: %016x', datapath.id)
471                     del self.datapaths[datapath.id]
472
473             def _monitor(self):
474                 while True:
475                     for dp in self.datapaths.values():
476                         self._request_stats(dp)
477                     hub.sleep(5)
478
479             def _request_stats(self, datapath):
480

```

```

481         self.logger.debug('send stats request: %016x',
482 datapath.id)
483         ofproto = datapath.ofproto
484         parser = datapath.ofproto_parser
485
486         req = parser.OFPFlowStatsRequest(datapath)
487         datapath.send_msg(req)
488
489         req = parser.OFPPortStatsRequest(datapath, 0,
490 ofproto.OFPP_ANY)
491         datapath.send_msg(req)
492
493         @set_ev_cls(ofp_event.EventOFPFlowStatsReply,
494 MAIN_DISPATCHER)
495         def _flow_stats_reply_handler(self, ev):
496             body = ev.msg.body
497             ctemp_idx = 0
498
499             self.flow_monitor = len(body)
500             print("Flow Entry saat ini : " +
501 str(self.flow_monitor))
502             self.logger.info(' Cookie          '
503                             '          Duration          '
504                             ' Packets
505 Bytes')
506             self.logger.info('-----'
507                             '-----'
508                             '-----')
509             -----')
510
511             flow_table = ev.msg.to_jsondict()
512             for i in range (self.flow_monitor):
513                 # print(i)
514                 cookie =
515 (flow_table["OFPFlowStatsReply"]["body"][i]["OFPFlowStats"]["co
516 okie"])
517                 duration =
518 (flow_table["OFPFlowStatsReply"]["body"][i]["OFPFlowStats"]["du
519 ration_sec"])
520                 packet_count =
521 (flow_table["OFPFlowStatsReply"]["body"][i]["OFPFlowStats"]["pa
522 cket_count"])
523                 byte_count =
524 (flow_table["OFPFlowStatsReply"]["body"][i]["OFPFlowStats"]["by
525 te_count"])
526                 # print('{:016x} {:8d} {:15d}
527 {:12d}'.format(cookie, duration, packet_count, byte_count))
528
529                 # print("longest duration : " +
530 str(longest_duration))
531                 if not cookie in self.ctemp:
532                     self.ctemp.append(cookie)
533                     self.dtemp.append(duration)
534                     self.ptemp.append(packet_count)
535                     self.btemp.append(byte_count)
536                     print('{:016x} {:8d} {:15d}
537 {:12d}'.format(cookie, duration, packet_count, byte_count))
538
539                     elif cookie in self.ctemp and cookie !=0:

```

```

540                                     ctemp_idx =
541 self.ctemp.index(cookie)
542
543                                     if byte_count >
544 self.btemp[ctemp_idx] and packet_count > self.ptemp[ctemp_idx]:
545                                     self.ctemp[ctemp_idx] =
546 cookie
547                                     self.btemp[ctemp_idx] =
548 byte_count
549                                     self.dtemp[ctemp_idx] =
550 duration
551                                     self.ptemp[ctemp_idx] =
552 packet_count
553                                     # print("Flow Entry saat
554 ini : " + str(self.flow_monitor))
555                                     print('{:016x} {:8d}
556 {:15d} {:12d}'.format(cookie, duration, packet_count,
557 byte_count))
558
559                                     elif byte_count ==
560 self.btemp[ctemp_idx]:
561                                     if self.flow_monitor >
562 (0.8*50):
563                                     print("hapus flow
564 dengan cookie " + hex(cookie))
565                                     self.cookie_temp =
566 cookie
567                                     # call function
568 delete_flow(cookie)
569
570                                     self.delete_flow(ev)
571
572                                     else:
573                                     print('{:016x}
574 {:8d} {:15d} {:12d}'.format(cookie, duration, packet_count,
575 byte_count))
576                                     else:
577                                     # print("Flow Entry saat ini
578 NORMAL : " + str(flow_monitor))
579                                     print('{:016x} {:8d} {:15d}
580 {:12d}'.format(cookie, duration, packet_count, byte_count))
581
582                                     # function to check longest duration
583 if self.longest_duration <
584 self.dtemp[ctemp_idx]:
585                                     self.longest_duration =
586 self.dtemp[ctemp_idx]
587                                     self.clongest_dur =
588 self.ctemp[ctemp_idx]
589                                     else:
590                                     self.longest_duration =
591 self.longest_duration
592
593                                     # print("longest duration : " +
594 str(self.longest_duration))
595                                     self.longest_duration = 0
596                                     # print(hex(self.cookie_idx0))
597                                     # print(hex(self.clongest_dur))
598 if self.flow_monitor >= (0.9*50):

```



599	print("Flow Table PENUH!!!! Perlu
600	dilakukan penghapusan paksa !!")
601	self.cookie_temp = self.clongest_dur
602	print("menghapus flow entry dengan 'total
603	duration' paling lama . . . ." + hex(self.clongest_dur))
604	self.delete_flow(ev)
605	

Client Emulation	
1	from scapy.all import *
2	import requests
3	import random
4	import os
5	
6	count = 0
7	for i in range(0, 100):
8	acak = random.randint(4, 20)
9	ip = "10.0.0." + str(acak)
10	os.system("ifconfig h4-eth0 " + ip)
11	time.sleep(0.5)
12	print("Request dari alamat : " + ip)
13	r = requests.get('http://10.0.0.100/')
14	#print(type(r))
15	print(r.status_code)
16	#print(r.headers)
17	print(r.headers['content-type'])
18	#print(r.text)
19	print("Req ke : " + str(count))
20	count += 1
21	