ANALISIS *OPENFLOW LOAD BALANCING WEB SERVER* DENGAN ALGORITMA *LEAST CONNECTION* PADA *SOFTWARE DEFINED NETWORK*

SKRIPSI

KEMINATAN TEKNIK KOMPUTER

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
Erine Karantha Denny Laksana
NIM: 125150301111029

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2016

PENGESAHAN

ANALISIS *OPENFLOW LOAD BALANCING WEB SERVER* DENGAN ALGORITMA *LEAST CONNECTION* PADA *SOFTWARE DEFINED NETWORK*

SKRIPSI

KEMINATAN TEKNIK KOMPUTER

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Erine Karantha Denny Laksana
NIM: 125150301111029

Skripsi ini telah diuji dan dinyatakan lulus pada
28 Juli 2016
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Rakhmadhany Primananda, S.T, M.Kom	R.Arief Setyawan, S.T, M.T
NIK: -	NIP. 19750819 199903 1 001

Mengetahui
Ketua Program Studi 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 pustaka.

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, 28 Juli 2016

Erine Karantha Denny Laksana

NIM: 125150301111029

KATA PENGANTAR

Puji syukur kehadirat Allah SWT, berkat rahmat dan karunia-Nya penulis dapat menyelesaikan penyusunan tugas akhir dengan baik. Penulisan tugas akhir ini diajukan untuk memenuhi salah satu syarat untuk mendapat gelar sarjana pada Program Studi Informatika Keminatan Teknik Komputer, Fakultas Ilmu Komputer, Universitas Brawijaya Malang. Judul tugas akhir yang penulis ajukan adalah **"**Analisis *Openflow Load Balancing Web Server* Dengan Algoritma *Least Connection* Pada *Software Defined Network***"**.

Terima kasih pula Penulis sampaikan kepada pihak-pihak yang telah membantu dalam penyelesaian tugas akhir ini. Pihak-pihak tersebut antara lain:

1. Bapak Rakhmadhany Primananda, S.T., M.Kom selaku Dosen Pembimbing I yang telah banyak memberikan bimbingan, ilmu dan saran dalam penyusunan tugas akhir ini.
2. Bapak R. Arief Setyawan, S.T., M.kom selaku pembimbing II skripsi yang telah banyak memberikan bimbingan, ilmu dan saran dalam penyusunan tugas akhir ini.
3. Sabriansyah Rizqika Akbar, S.T., M.Eng selaku Ketua Program Studi Teknik Komputer serta segenap Bapak/Ibu Dosen, Staf Administrasi dan Perpustakaan Program Studi Teknik Komputer Universitas Brawijaya.
4. Bapak Tri Astoto Kurniawan, S.T, M.T, Ph.D selaku Ketua Program Studi Informatika
5. Seluruh civitas akademika di PTIIK Universitas Brawijaya yang telah memberi bantuan dan dukungan selama peneliti menempuh studi dan melakukan penelitian skripsi ini.
6. Semua teman-teman Teknik Komputer angkatan 2012 segala bantuannya sehingga terselesaikannya skripsi ini dan bantuan selama menjadi mahasiswa.

Semoga jasa dan amal baik mendapatkan balasan dari Allah SWT. Ibarat tak ada gading yang tak retak, dengan segala kerendahan hati, penulis menyadari sepenuhnya bahwa skripsi ini masih belum sempurna. Oleh karena itu kritik dan saran untuk kesempurnaan skripsi ini, senantiasa penulis harapkan dari berbagai pihak.

Malang, 28 Juli 2016

Penulis
dennykarantha@gmail.com

ABSTRAK

Pada perkembangan jaringan komputer saat ini terdapat sebuah teknologi dimana sistem pengontrol dari arus data dipisahkan dari perangkat kerasnya. *Software defined network* merupakan jaringan komputer yang memisahkan antara *control plane* dan data *plane* pada *switch. Software defiened network* juga merupakan jaringan komputer yang sangat *flesibel* karena dikonfigurasikan dan dikendalikan melalui sebuah *software* terpusat. Cara komunikasi antara perangkat dan kontroller menggunakan sebuah protokol yang disebut dengan *Openflow.*

Sebagai konsep jaringan yang sangat komplek *software defined network* menawarkan *scability* dan *programmability* seperti *load balancing web server. Load balancing* merupakan sebuah metode pembagian beban yang diberikan ke suatu *web server* ketika melayani *request* yang dilakukan oleh *user*. Pembagian beban ke *web server* dilakukan berdasarkan dengan algoritma *load balancing. Least connection* merupakan algoritma *load balancing* yang akan menyalurkan koneksi jaringan ke *server* yang memiliki koneksi aktif paling sedikit.

Melalui Implementasi sistem *load balancing* dengan algoritma *least connection* sebagai pembagi beban ke *server.* Sistem *load balancing* memberikan nilai rata – rata koneksi per detik yang cukup rendah yakni 1.28 conn/s dan akan turun ketika koneksi yang dilakukan *user* semakin tinggi. Berbeda dengan nilai *Reply time* yang akan meningkat ketika jumlah permintaan *user* juga semakin tinggi dengan nilai rata- rata 199.58 ms. Untuk nilai *error* memberikan hasil yang baik dengan nilai *error* 0 yang berarti tidak terjadi *error* dari beberapa skenario pengujian pada sistem.

Kata kunci : *load balancing, web server, least connection, software defined network.*

ABSTRACT

Today there is a technology that the system controller of the data flow is separated from the hardware. Software defined network is a computer networking that separates the control plane and data plane on switch. Software defiened network is also a highly flexible network because it was configured and controlled via a centralized software. OpenFlow is a method to communicating between device and controller.

Software defined network is a computer networking it provide a scability and programmability concept as load balancing on web server. Load balancing is a method to sharing the load were leading to a web server when served user. Load sharing in the web server carried based on load balancing algorithms. Least connection is a load balancing algorithm that will distribute the network connection to the server with the fewest active connections.

Load balancing with least connection algorithm as a sharing the load that leading to the server. This system provides a fairly low average value of connections per second with the result 1,28 conn/ s and will down when the connection of user go higher. It different with the reply time value, it will increase when the number of user requests are also higher with the result of average value 199.58 ms and not an error occur with the result of value 0 in several test scenarios on system.

Keywords: *load balancing, web server, least connection, software defined network*

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

DAFTAR LAMPIRAN

BAB 1 PENDAHULUAN

1.1 Latar belakang

Perkembangan teknologi saat ini berkembang sangat pesat khususnya pada pemanfaatan Internet. Dimana semua orang di dunia terhubung dengan satu jaringan kompter yang luas, yang bisa saling berkomunikasi satu dengan yang lainnya. Salah satu bukti dari perkembangan jaringan yang terjadi munculah sebuah teknologi yang merupakan ternologi jaringan masa depan dikenal dengan *software define network* (SDN).

Software defined networking adalah sebuah pendekatan jaringan komputer yang dimana sistem pengontrol dari arus data dipisahkan dari perangkat kerasnya. SDN merupakan pemisah antara *control* dan data pada *switch* yang dikontrol melalui aplikasi. Perangkat networking secara umum terdiri dari dua bagian yaitu *control plane* dan *data plane*. *Control plane* merupakan bagian yang berfungsi sebagai pengatur logika pada perangkat sedangkan *data plane* berfungsi untuk meneruskan paket yang masuk ke suatu *port* menuju *port* tujuan dengan komunikasi pada *control plane.*

Software defined network juga merupakan suatu jaringan komputer yang sangat fleksibel karena dikonfigurasi dan dikendalikan melalui sebuah software terpusat, dimana konfigurasi jaringan dapat dilakukan sendiri tanpa menunggu perkembangan dari vendor. Pada *software defined network* antara *control* jaringan dipisahkan dari sistem forwadingnya dan *controller* tersebut dapat kita program secara langsung. Cara komunikasi antara perangkat dan *controller* menggunakan sebuah protocol yang disebut dengan *Openflow.*

Sebagai konsep jaringan *software defined network* menawarkan *scability* dan *programmability* untuk penggunaan jaringan yang semakin komplek seperti *load balancing web server* (Senthil, Ranjani S., 2015). *Load balancing* merupakan sebuah metode pembagian beban yang diberikan ke suatu *web server*. Ketika suatu situs *web* memiliki kunjungan yang tinggi, membuat *web server* mendapatkan peningkatan layanan sehingga dapat membebani kinerja dari *server*. Untuk mengatasi hal tersebut maka beban *server* harus dibagi dengan *server* yang lain sehingga kinerja dari *server* menjadi lebih optimal. Dalam melakukan pendistribusian ke *web server load balancing* memiliki beberapa algoritma salah satunya adalah *least connection.*

Load balancing least connection akan menyalurkan koneksi jaringan ke *server* yang memiliki koneksi aktif paling sedikit. Pada *server* yang memiliki kemampuan pemrosesan yang sama, algoritma penjadwalan *least connection* akan mendistribusikan beban permintaan dengan baik karena permintaan yang panjang tidak akan disalurkan ke sebuah *server* (Yogi, Kurniawan.,2014). Sehingga dengan algorima *least connection* ini *server* dapat memberikan respon yang baik terhadap permintaan yang dilakukan oleh *user* dan juga dapat digunakan sebagai penyeimbang beban pada *server.*

Kinerja dari *web server* dipengaruhi oleh permintaan yang dilakukan oleh *user.* Semakin banyak permintaan terhadap suatu halaman *web* maka beban terhadap *web server* akan semakin tinggi sehingga respon yang akan diberikan juga semakin lambat dan dapat menyebabkan *server* utama mati. Sehingga diperlukan penambahan *server* untuk tujuan melakukan pembagian beban *server* mengunakan algoritma *least connection* agar tidak terjadi *overload.* Algoritma *least connection* juga dapat mendistribusikan beban permintaan dengan baik karena permintaan yang panjang tidak akan disalurkan ke sebuah *server.* Dengan demikian sistem *load balancing* dengan mengunakan algoritma *least connection* ini dapat memberikan solusi terhadap beban *server* dalam memberikan respon permintaan *user*.

Berdasarkan penjelasan di atas maka penulis akan menggunakan metode algoritma *least connection* untuk melakukan implementasi dan pengujian sistem *load balancing* pada *software defined network*. Arsitektur jaringan yang akan digunakan adalah *software defined network* karena memberikan manajemen jaringan yang *fleksibel, dinamis* dan kemudahan dalam melakukan monitoring jaringan yang dapat dilakukan terpusat pada *controller*. Algoritma *least connection* dipilih sebagai penyeimbang beban *server* karena mampu menyalurkan koneksi jaringan ke *server* yang memiliki koneksi aktif paling sedikit. Algoritma penjadwalan *least connection* akan mendistribusukan beban permintaan dengan baik karena permintaan yang panjang tidak akan disalurkan ke sebuah *server.*

1.2 Rumusan masalah

Berdasarkan latar belakang yang telah dipaparkan, maka rumusan masalah dalam penelitian ini adalah sebagai berikut:

1. Bagaimana cara melakukan implementasi *load balancing web server* dengan algoritma *least connection* pada *software defined network*?
2. Bagaimana cara melakukan pengujian *load balancing* pada *software defined network* dengan parameter *connection rate, reply time, cpu time* dan juga *error* pada sistem *load balancing?*
3. Bagaimana melakukan analisa *connection rate, reply time, cpu time* dan juga *error* pada sistem *load balancing?*

1.3 Tujuan

Berdasarkan latar belakang yang telah dipaparkan, maka tujuan dalam penelitian ini adalah sebagai berikut:

1. Untuk melakukan Implementasi *load balancing web server* dangan algoritma *least connection* pada *software defined network*.
2. Untuk dapat melakukan pengujian dengan menggunakan parameter *connection rate, reply time, cpu time* dan juga *error* pada sistem *load balancing.*

3. Untuk dapat melakukan analisa *connection rate, reply time, cpu time* dan juga *error* pada sistem *load balancing.*

1.4 Manfaat

1. Dapat melakukan implementasi *load balancing* dengan algoritma *least connection* pada *web server* dengan menggunakan *software defined network* sehingga dapat diketahui hasil kinerja dari *server*.
2. Dapat melakukan pengujian pada *web server* pada *software defined network* dengan menggunakan algoritma *least connection*, sehingga performa dari *server* dapat lebih maksimal.
3. Dapat mengetahui performa suatu sistem *load balancing web server* dari hasil analisis *connection rate, reply time, cpu time* dan juga *error.*

1.5 Batasan masalah

Batasan masalah yang dapat dipetik berdasarkan rumusan masalah adalah sebagai berikut:

1. Pada penelitian ini hanya fokus terhadap Implementasi *load balancing web server* dengan algoritma *least connection* pada *software defined network*
2. Menggunakan bahasa pemrograman python dan *library pox*
3. Menggunakan *mininet server* sebagai media implementasi *web server*
4. Melakukan pengujian peforma *server* dengan melakukan *request client* ke *web server* dan mengunakan jumlah *rate* tertentu

1.6 Sistematika pembahasan

Langkah-langkah atau tahapan-tahapan yang akan dilakukan dalam menyelesaikan skripsi ini adalah sebagai berikut:

BAB I : **Pendahuluan**
Menguraikan latar belakang, rumusan masalah, batasan masalah, tujuan, manfaat dan sistematika penulisan.

BAB II : **Landasan Kepustakaan**
Menguraikan kajian pustaka dan dasar teori yang mendasari *load balancing* dan juga *software defined network*

BAB III : **Metodologi**
Menguraikan tentang metode dan langkah kerja yang terdiri dari studi literatur, perancangan sistem, implementasi dan analisis serta pengambilan kesimpulan

BAB IV : **Implementasi**
Membahas penerapan sistem *load balancing least connection* pada *software defined network* dengan menggunakan *pox* sebagai *controller* dan *mininet* sebagai simulasi.

BAB V : **Pengujian**
Memuat proses pengujian dan analisa terhadap *server* yang

dilakukan pada sistem *load balancing software defined network.*

BAB VI : Penutup

Memuat kesimpulan dan saran yang didapat dari hasil pengujian dari sistem *load balancing software defined network.*

BAB 2 LANDASAN KEPUSTAKAAN

2.1 Tinjauan Pustaka

Pada penelitian ini tinjauan pustaka diambil dari penelitian yang pernah dilakukan sebelumnya dan dijadikan sebagai pedoman dalam pelaksaan penelitian. Berikut Tabel perbandingan penelitian terdahulu dan sekarang :

Tabel 2.1 Kajian Pustaka

No	Nama Penulis, Tahun dan Judul	Persamaan	Perbedaan	
			Penelitian Terdahulu	Rencana Penelitian
1	Yogi Kurniawan [2014] Analisis Kinerja Algoritma *Load* Balancer dan Implementasi pada layanan *web server*	Melakukan penerapan sistem *load balancing* pada *web server*	Menjelaskan implementasi sistem *load balancing* pada *web server* pada jaringan *convensional*	Mengimplementasi sistem *load balancing* pada *web server* pada jaringan *software defined network*
2	Dependral Dhakal.; Bishal Pradhan.; Sunil Dhima. *Campus Network Using Software defined network.* International Journal Of Computer Applications. 2016 (0975-8887)	Mengunakan jaringan *Software defined network*	Mengunakan *sorfware defined network* untuk melakukan pengujian terhadap bandwidth dan Jitter	Mengunakan *Software defined network untuk melakukan sistem load balancing*
3	Suckhveer Kaur.; Japinder S.; Navtej Sigth G. N*etwork Programibility using pox controller. International Conference on Comunication, computing and sistem* (ICCCS-2014)	Menggunakan pox sebagai *controller*	Menjelaskan implementasi dan *running* file mengunakan *pox controller*	Mengunakan *pox controller* untuk sistem *load balancing* mengunakan algoritma *least connection*

Berdasarkan apa yang telah dipaparkan di atas ada beberapa teori yang akan diulas untuk menganalisis pengimplementasian *openflow Load balancing web server* dengan algoritma *least connection* pada *software defined network*. Dasar teori yang diperlukan untuk menyusun penelitian yang diusulkan diantaranya yaitu :

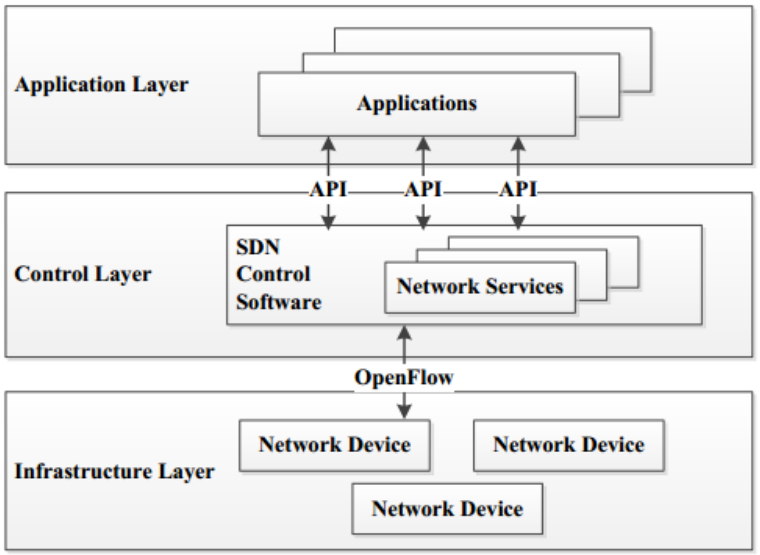
2.2 Software Defnined Network

Software defined network (SDN) adalah istilah yang merujuk pada konsep/paradigma baru dalam mendisain, mengelola dan mengimplementasikan jaringan, terutama untuk mendukung kebutuhan dan inovasi di bidang jaringan yang semakin lama semakin kompleks. Konsep dasar SDN adalah dengan melakukan pemisahan *eksplisit* antara *control* dan *data plane*, serta kemudian melakukan abstraksi sistem dan meng-isolasi kompleksitas yang ada pada komponen atau sub-sistem dengan mendefinisikan antar-muka (*interface*) yang standard.

Beberapa aspek penting dari SDN adalah:

1. Adanya pemisahan secara fisik/eksplisit antara *forwarding/data plane* dan *control-plane.*
2. Antarmuka standard (*vendor-agnosic*) untuk memprogram perangkat jaringan
3. *Control-plane* yang terpusat (secara logika) atau adanya sistem operasi jaringan yang mampu membentuk peta logika (*logical map*) dari seluruh jaringan dan kemudian mempresentasikannya melalui (sejenis) API (*Application Programming Interface*)
4. Virtualisasi dimana beberapa sistem operasi jaringan dapat mengkontrol bagian-bagian (*slices* atau*substrates*) dari perangkat yang sama.

Software defined network adalah suatu arsitektur jaringan dimana control network dipisahkan dari sistem *forwadingnya* dan *controller* tersebut dapat diprogram secara langsung. Pada umumnya perangkat jaringan yang ada saat ini terdiri dari *control plane* dan data *plane* yang mana keduanya tertanam dalam satu perangkat. Sedangkan dalam SDN antara *control plane* dan data *planenya* dipisahkan dalam suatu perangkat yang berbeda.

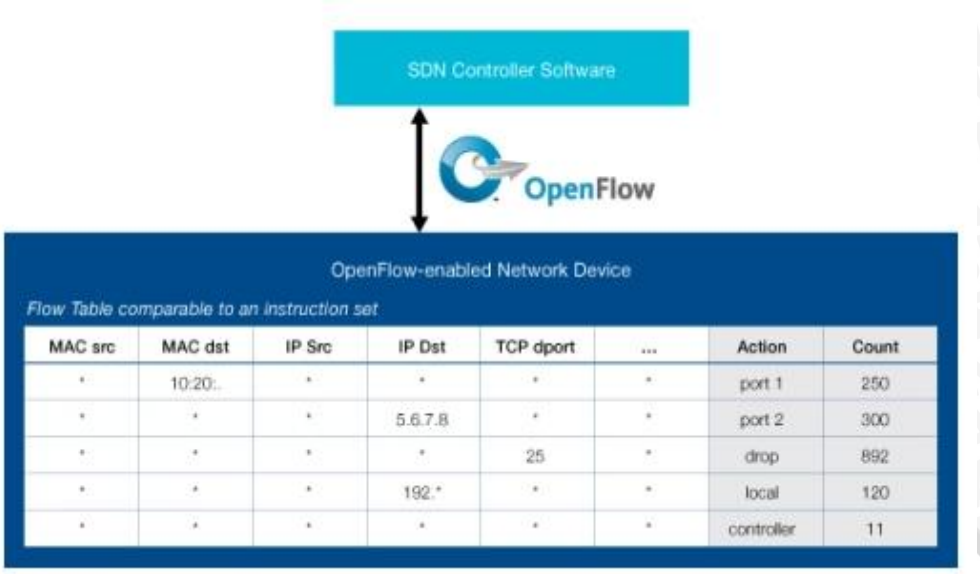


Gambar 2.1 Arsitektur dari Sofware Defined Network
Sumber : R. Bayshore,Alto Palo (2012)

Pada Gambar 2.1 di atas dapat dilihat bahwa *control layer* memegang peran penting mengendalikan sistem jaringan, setiap perangkat dari berbagai macam vendor dapat beroprasi sesuai perintah dari *controllernya*. Cara komunikasi antara perangkat dan *controller*nya sendiri menggunakan suatu protocol yang disebut dengan *Openflow*. *Openflow* sendiri bisa diibaratkan seperti sebuah CPU pada computer. (Astuto, et al. 2014).

2.2.1 Openflow

Openflow adalah komunikasi *software defined network* yang didefinisikan antara kontrol dan data *plane* pada *arsitektur software defined network*. *Openflow* memungkinkan akses langsung dan dapat digunakan untuk memanipulasi data *plane* pada perangkat jaringan seperti *switch* dan router, baik secara fisik maupun virtual. *Controller* melalui *openflow* memperbolehkan memanipulasi data secara langsung seperti menambah, menghapus maupun melakukan modifikasi pada *flow table*. Pada paket *openflow* berisikan informasi tentang source MAC, destination MAC source IP, destination IP dan TCP *Port* yang digunakan untuk menentukan arah paket akan diteruskan. Berikut Gambar 2.2 tabel dari *openflow* pada *software defined network.*



Gambar 2.2 Open*flow* SDN
Sumber : R. Bayshore,Alto Palo (2012)

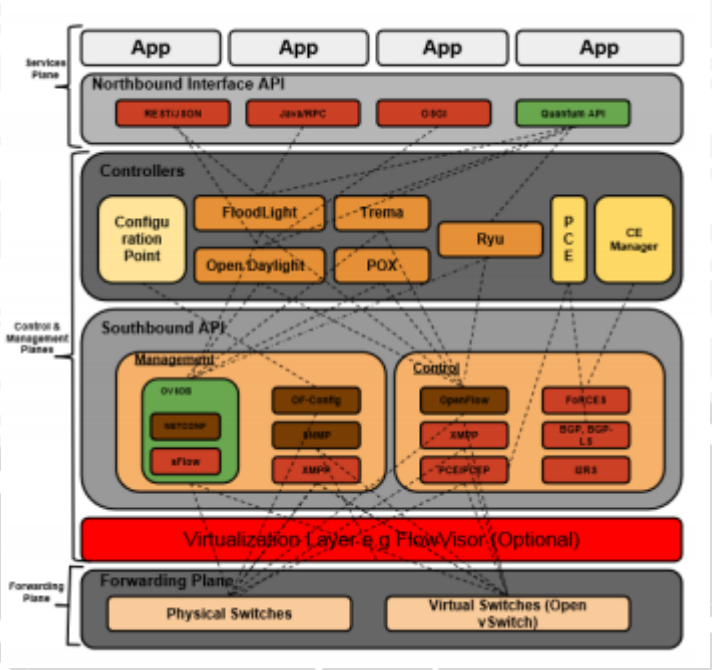
Mekanisme kerja dari protokol *openflow* adalah pada saat *switch openflow* menerima sebuah paket dan tidak memiliki kecocokan dengan tabel *flow* yang sudah ada maka *switch openflow* akan meneruskan paket menuju ke *controller openflow*. Kemudian *controller* akan memberikan respon kepada paket yang datang tersebut berdasarkan tabel *flow.* Sehingga komunikasi yang terjadi antara *control plane* dengan data *plane* melalui *controller* bisa memberikan respon yang sesuai terhadap setiap paket yang datang.

Protokol *openflow* sendiri terdiri dari komponen penting yaitu *header fields, action,* dan *couter. Header fields* merupakan paket *header* yang mendefinisikan suatu *flow* dangan *fields* yang terdiri dari enkapsulasi seperti enkapsulasi segmen pada protokol VLAN *ethernet.* Pada paket dan *byte* disetiap *flow* digunakan untuk membantu membuang dan me-nonaktifkan *flow* pada *couter.* Sehingga protokol *openflow* tidak hanya melakukan *forwarding* tetapi juga dapat melakukan pengaturan pergerakan paket terhadap jaringan tersebut.

2.2.2 *Controller*

Controller merupakan salah satu bagian dari arsitektur jaringan yang ada pada *software defined network* yang berfungsi sebagai pusat kontrol dan logika jaringan. *Software defined network* memiliki beberapa *controller* yang dapat membantu melakukan komunikasi dan konfigurasi antara *application layer* dan *infrastrukture layer.* Semua kebijakan yang ada pada jaringan seperti *routing*, *switching* dan juga pengaturan jaringan lainnya terdapat pada *controller* sehingga tugas *controller* pada *sofware defined network* sangatlah penting.

Ada banyak *controller* pada SDN yang dapat digunakan yaitu *floodlight, trema,POX , RYU* dan juga *Open daylight.*



Gambar 2.3 *Controller* Software Defined Network
Sumber : lara et al (2013)

Berdasarkan Gambar 2.3 dapat dijelaskan implementasi *load balancing controller* yang digunakan adalah *POX controller* yang berfungsi untuk mengatur aplikasi yang ada pada jaringan. Untuk melakukan *setting* komunikasi *openflow interface POX controller* ini menggunakan bahasa pemograman python. Pada *POX controller* juga tersedia *support modules* untuk *openflow.* Untuk menangani konektifitas dan juga metode untuk berinteraksi antara *application interface* dengan *openflow pada POX* juga telah tersedia.(lara et al, 2013).

Controller sendiri memiliki banyak sekali kelebihan dan kekurangan dan telah dibandingkan berdasarkan *interface* yang mereka sediakan, *support* terhadap *virtual switching, Graphical User Interface (GUI), support* untuk API REST, *support* bahasa pemograman, dan juga *support* terhadap jaringan *Openstack*. Hasil dari perbandingan ditunjukan oleh Gambar 2.4, dan dapat disimpulkan bahwa tidak ada *controller* yang memiliki *support* terhadap semua sifat- sifat dari *controller.*

	POX	Ryu	Trema	FloodLight	OpenDaylight
Interfaces	SB (OpenFlow)	SB (OpenFlow) +SB Management (OVSDB JSON)	SB (OpenFlow)	SB (OpenFlow) NB (Java & REST)	SB (OpenFlow & Others SB Protocols) NB (REST & Java RPC)
Virtualization	Mininet & Open vSwitch	Mininet & Open vSwitch	Built-in Emulation Virtual Tool	Mininet & Open vSwitch	Mininet & Open vSwitch
GUI	Yes	Yes (Initial Phase)	No	Web UI (Using REST)	Yes
REST API	No	Yes (For SB Interface only)	No	Yes	Yes
Productivity	Medium	Medium	High	Medium	Medium
Open Source	Yes	Yes	Yes	Yes	Yes
Documentation	Poor	Medium	Medium	Good	Medium
Language Support	Python	Python-Specific + Message Passing Reference	C/Ruby	Java + Any language that uses REST	Java
Modularity	Medium	Medium	Medium	High	High
Platform Support	Linux, Mac OS, and Windows	Most Supported on Linux	Linux Only	Linux, Mac & Windows	Linux
TLS Support	Yes	Yes	Yes	Yes	Yes
Age	1 year	1 year	2 years	2 years	2 Month
OpenFlow Support	OF v1.0	OF v1.0 v2.0 v3.0 & Nicira Extensions	OF v1.0	OF v1.0	OF v1.0
OpenStack Networking (Quantum)	NO	Strong	Weak	Medium	Medium

Gambar 2.4 Tabel *controller SDN*
Sumber : R. Bayshore,Alto Palo (2012)

Dari Gambar 2.4 dapat kita lihat bahwa setiap *controller* memiliki kelebihan dan juga kekurangan pada masing- masing karakteristik yang ada. Oleh karena itu untuk memilih *controller* kita perlu mempertimbangkan beberapa kriteria seperti *Interface, modularitas,* dan juga *GUI*. Dalam hal ini digunakan AHP (*Analytic Hierarchy Process)* untuk memilih *controller* terbaik. Pendekatan pilihan mengunakan AHP memiliki beberapa ke unggulan yakni:

- Prioritas berpasangan merupakan persyaratan sebagai *input*
- Konsisten melakukan *Checking*
- Manfaat prioritas relatif lebih baik dibandingkan dengan prioritas linier (dellay, throughput dan loss)

Analytic Hierarchy Process juga digunakan untuk menentukan prioritas dari berbagai alternatif. Langkah pertama yaitu menentukan tujuan dan kriteria seleksi untk mencapai hasil. Kemudian prioritas penugasaan dari kriteria seleksi yang ditetapkan oleh manager dan prioritas kriteria bertugas untuk *controller.*

2.3 *Mininet*

Mininet adalah merupakan sebuah emulator jaringan yang mensimulasikan koleksi dari *host-end, switch, router,* dan *link* pada *single kernel Linux*. Masing-masing elemen ini disebut sebagai "*host*" menggunakan virtualisasi ringan untuk membuat sistem tampilan tunggal sehingga terlihat seperti jaringan yang lengkap, menjalankan kernel yang sama, sistem, dan *user code*. *Mininet* penting

bagi komunitas *open-source* SDN. *Mininet* biasanya digunakan sebagai simulasi, verifikasi, *testing tool*, dan *resource*. (O'Reilly, 2013).

Pada *mininet sebuah host* akan beroperasi seperti sebuah mesin yang sebenarnya dan bisa menjalankan *code* secara bersama-sama, *host Mininet* merupakan *shell* pada mesin yang programnya dapat diubah-ubah. Program kustom ini dapat mengirim, menerima, dan memroses paket melalui program yang tampaknya seperti *ethernet* nyata tetapi sebenarnya adalah *switch virtual.*

2.4 *Load Balancing*

Load balancing adalah suatu teknik yang digunakan untuk memisahkan antara dua atau banyak *network link*. Dengan mempunyai banyak *link* maka optimalisasi utilitas sumber daya, *throughput* atau *response time* akan semakin baik karena mempunyai lebih dari satu *link* yang bisa saling mem-backup pada saat *network down* dan menjadi cepat pada saat *network* normal. Jika memerlukan realibilitas tinggi yang memerlukan 100% koneksi *uptime* dan yang menginginkan koneksi *upstream* yang berbeda dan dibuat saling mem-backup (Lukitasari dan Oklilas,2010).

Load balancing adalah teknik untuk mendistribusikan beban trafik pada dua atau lebih jalur koneksi secara seimbang, agar *traffic* dapat berjalan optimal, memaksimalkakn *throughput*, memperkecil waktu tanggap dan menghindari *overload* pada salah satu jalur koneksi (Sirajuddin,2012).

2.4.1 Implementasi *Least Connection*

Pada mekanisme pembagian beban *server* yang digunakan pada sistem *load balancing* pada *software defined network* dilakukan dengan pembagian *traffic* berdasarkan *request* yang dilakukan oleh *client.* Pembagian beban dilakukan dengan melakukan pengecekan pada semua *server* yang sedang aktif, diman *server* yang sedang melayani *request* aktif paling sedikit yang akan diberikan beban *traffic* baru yang masuk.

Mekanisme pemilihan *server* dengan algoritma *least connection* dapat dilihat pada Gambar 2.5 berikut.

```
for (m = 0; m < n; m++) {
  if (W(Sm) > 0) {
    for (i = m+1; i < n; i++) {
      if (W(Si) <= 0)
        continue;
      if (C(Si) < C(Sm))
        m = i;
    }
    return Sm;
  }
}
return NULL;
```

Gambar 2.5 Implementasi *Least Connection*

Berdasarkan Gambar 2.5 di atas dapat dijelaskan mekasnisme pemilihan *server* menggunakan algoritma *least connection.* Algoritma *least connection* akan menyalurkan koneksi jaringan kepada *server* yang memiliki koneksi aktif paling sedikit. Pada *server* yang memiliki kemampuan pemrosesan yang sama, algoritma penjadwalan *least connection* akan mendistribusikan beban permintaan dengan baik karena permintaan yang panjang tidak akan disalukan kepada *server* (Yogi Setiawan, 2014). Sehingga algoritma *least connection* ini saat baik digunakan sebagai penyeimbang beban pada *server.* Pembagian beban yang seimbang dapat membantu *server* dalam memberikan *respon* terhadap permintaan *user.*

2.5 *Web Server*

Web server dapat diartikan sebagai sebuah perangkat keras ataupun perangkat lunak yang menyediakan layanan kepada pengguna *web* dengan menggunakan protokol HTTP ataupun HTTPS atas berkas-berkas yang terdapat dalam sebuah situs *web* dalam layanan ke pengguna menggunakan aplikasi tertentu seperti *web browser*. Penggunaan paling umum dari sebuah *web server* adalah untuk menempatkan sebuah situs *web*, namun pada kenyataannya penggunaannya dapat diperluas sebagai tempat penyimpanan data ataupun untuk menjalankan sejumlah aplikasi bisnis (Kadir,2003).

Berdasarkan penjelasan diatas dapat disimpulkan fungsi dari *web server* adalah untuk mentransfer ataupun untuk memindahkan data yang diminta oleh *user* dengan menggunakan protokol tertentu. *Web server* akan memberikan data yang diminta oleh *user* dengan memberikan respon berupa tampilan halaman *web* yang akan ditampilkan pada *user*. Kemudian data yang diberikan *web server* kepada *user* dapat berupa halaman *html*.

2.6 Httperf

Ketika melakukan pengujian jaringan banyak *tool* yang dapat digunakan salah satunya adalah *httperf.* *Httperf* sendiri merupakan sebuah *tool* yang dapat digunakan untuk melakukan pengujian terhadap suatu *web server* dengan menggunakan beberapa parameter tertentu. Parameter yang digunakan antara lain *reply time, connection rate, error time, cpu time* dan juga *standart deviasi* pada suatu *web server*.

1. *Connection rate* adalah bayaknya koneksi yang dapat dilakukan oleh user dalam satu waktu (*rate*). Semakin tinggi nilai *connection rate* maka semakin baik kualitas dari suatu sistem, tetapi jika semakin kecil nilai dari *connection rate* maka semakin buruk kualitas dari suatu sistem.
2. *Reply time* adalah jumlah total kedatangan paket yang sukses yang diamati pada tujuan selama interval waktu tertentu dibagi oleh durasi interval waktu tersebut. Semakin besar nilai *reply time* dari sebuah sistem maka semakin baik kualitas dari sistem tersebut, tetapi jika

semakin kacil nilai dari *reply time* maka semakin buruk kualitas dari suatu sistem.

3. Cpu time adalah waktu yang dibutuhkan sistem untuk melakukan pemrosesan semua permintaan yang dilakukan oleh user. Semakin kecil nilai dari *cpu time* maka kualitas dari sebuah sitem semakin baik, tetapi jika nilai *cpu time* semakin tinggi maka kualitas dari sistem semakin buruk.
4. *Error adalah kegagalan sistem* yang dikarenakan banyaknya permintaan yang dilakukan oleh user tidak dapat diproses oleh sistem. Semakin tinggi nilai *error* maka semaikin buruk kualitas dari sistem, tetapi jika semakin kecil nilai *error* dari sistem maka kualitas sistem semakin baik.

Pada pengujian *client* akan melakukan *request* kepada *web server. Request* digunakan untuk memberikan beban permintaan *client* kepada *server* dalam satu waktu (*rate*) sehingga dapat diketahui kemampuan *server* dalam melakukan respon terhadap *client*. *Rate* pada pengujian ini adalah digunakan untuk memberikan ukuran kecepatan bit data transmisi dalam satu waktu.

Uji performansi terhadap sistem *web server* yang ada digunakan untuk tujuan menentukan seberapa baik performanya, sehingga dapat dilakukan suatu perubahan-perubahan untuk meningkatkan performansinya (Obaidat,2010). Uji performansi memiliki bentuk-bentuk tersendiri sebagai berikut:

1. *Performance Test*
Digunakan untuk mengujian setiap bagian dari suatu *web server* untuk menemukan teknik terbaik untuk mencapai optimasi ketika *traffic web* meningkat.

2. *Load Test*
Dilakukan untuk melakukan pengujian *website* menggunakan estimasi *traffic* dari sebuah *website* yang mampu dilayani. Caranya dengan mendefinisikan waktu maksimum sebuah halaman *web* dimuat. Pada akhir pengujian dilakukan perbandingan seberapa maksimum waktu yang dibutuhkan untuk membuka *web* sebuah *web server.*

3. *Stress Test*
Merupakan sebuah simulasi serangan yang menjalankan muatan atau permintaan secara berlebihan menuju *web server.* Tujuan *stress test* adalah untuk estimasi muatan maksimum sebuah *web server* sanggup menanganinya.

Pada saat melakukan sebuah pengujian performansi *web server,* tujuan utamanya adalah mengetahui tingkat kejenuhan dari sebuah *web server.* Komponen yang dibutuhkan untuk pengujian *web server* adalah sebuah *server* yang menjalankan perangkat lunak *web server.* Kemudian satu atau lebih *client* yang menjalankan perangkat lunak pemberi paket dan jaringan yang membutuhkan *client* dan *server.*

Sebuah pengujian *web server* sederhana, *client* membuat sejumlah besar permintaan dari *web server.* Sehingga akan terukur troughput yang berupa jumlah *reply* per *second* dari *web server* tersebut. Pengujian dilakukan beberapa kali dengan meningkatkan permintaan secara monoton hingga pada *rate* tertentu. Hasil pengukuran dari *httperf* memberikan data mentah tentang kinerja dari *web server,* namun kondisi ini berbeda pada pengujian nyata. Hal ini dikarenakan pada pengujian *web server* ini hanya satu file yang dilakukan permintaan sehingga *web server* tidak teruji secara keseluruhan.

2.7 Library POX

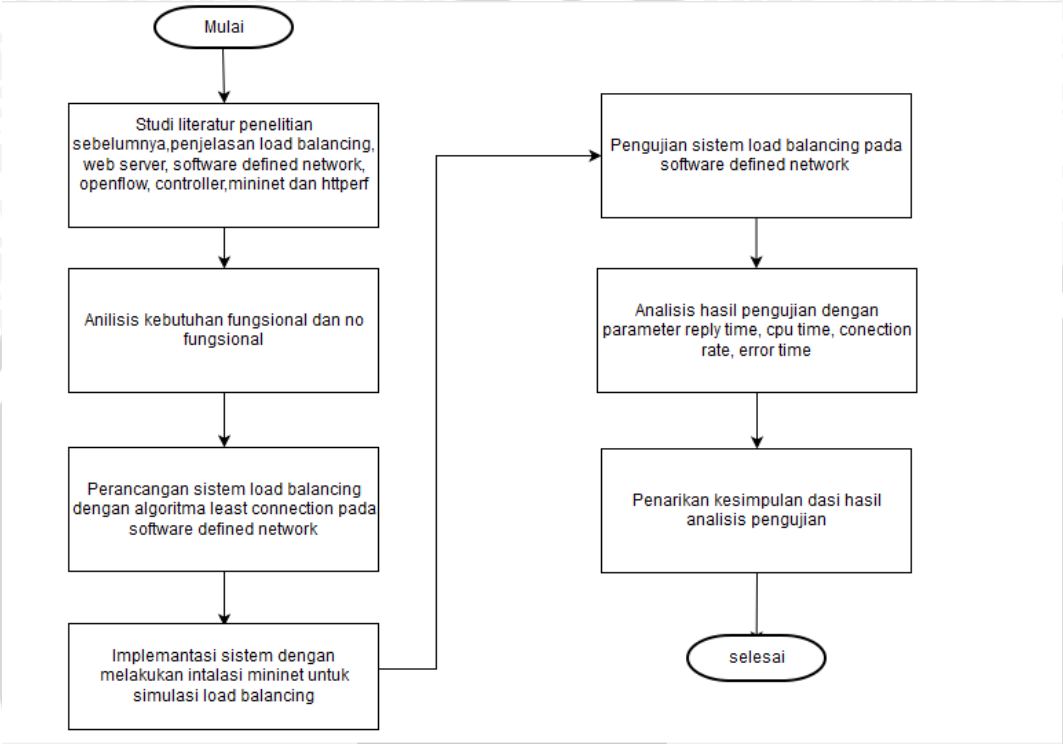
POX adalah library yang digunakan untuk memudahkan pengguna melakukan konfigurasi pada jaringan *software defined network.* POX juaga menyediakan *modules* khusus untuk komunikasi *OpenFlow* dan juga menyediakan metode-metode dan API untuk dapat berinteraksi dengan *Openflow switch*. Dimana Komponen-komponen yang ditambahan akan menghubungkan API yang telah tersedia termasuk *host tracking, routing, topology* (LLDP), dan *Python interface* yang diimplementasikan sebagai pembungkus untuk komponen API. Berikut Gambar 2.6 teori yang digunakan pada library POX.

Events generated from messages from switch	Packets parsed by pox/lib
FlowRemoved, FeaturesReceived ConnectionUp, FeaturesReceived RawStatsReply, PortStatus PacketIn, BarrierIn, SwitchDescReceived, FlowStatsReceived, QueueStatsReceived, AggregateFlowStatsReceived, TableStatsReceived, PortStatsReceived	arp, dhcp, dns eapol, eap ethernet, icmp igmp, ipv4 llc, lldp, mpls rip, tcp, udp, vlan

Gambar 2.6 library POX

BAB 3 METODOLOGI

Pada bagian ini akan dijelaskan tentang langkah yang dilakukan dalam pengerjaan tugas akhir. Berikut merupakan tahapan-tahapan metodologi penelitian yang digambarkan dengan diagram alir.



Gambar 3.1 Diagram Alir Metode Penelitian

Berdasarkan Gambar 3.1 di atas maka dapat dijelaskan langkah-langkah metodologi yang digunakan dalam tugas akhir ini yaitu:

1. Studi literatur penelitian sebelumnya, *software defined network, openflow, controller, mininet, web server* serta *httperf*.
2. Analisis kebutuhan sistem yang meliputi kebutuhan fungsional dan non fungsional.
3. Perancangan sistem *load balancing* dengan *algoritma least connection* pada *software defined network.*
4. Implementasi sistem dengan melakukan instalasi *mininet* untuk melakukan simulasi pada *software defined network*.
5. Pengujian sistem *load balancing* pada *software defined network*.
6. Analisis pengujian dengan *parameter reply time, cpu time, connection rate,* dan *error time*.
7. Penarikan kesimpulan berdasarkan hasil analisis pengujian yang dilakukan terhadap sistem

3.1 Studi Literatur

Pada bagian ini membahas tentang dasar teori yang mendukung segala kebutuhan dalam mengimplementasikan *load balancing* pada *software defined network* dengan metode algoritma *least connection.* Adapun yang dapat dijadikan bahan studi literatur meliputi :

1. *Software defined network*
   Merupakan insfrastruktur jaringan yang menggunakan protokol *openflow* yang berfungsi untuk memisahkan *control plane* dengan *data plane* dan juga komunikasi antara keduanya.
2. *Controller*
   Merupakan salah satu bagian dari arsitektur jaringan yang ada pada *software defined network* yang berfungsi sebagai pusat *control* dan logika jaringan. Dan juga bertugas untuk menjalankan sistem *load balancing*
3. *Openflow*
   Merupakan komunikasi pada *software defined network* yang didefinisikan antara kontrol dan data pada arsitektur *software defined network. Openflow* memungkinkan akses langsung dan dapat digunakan untuk memanipulasi *forwarding* pada perangkat jaringan seperti *switch* dan *router*, baik secara fisik maupun virtual.
4. *Mininet*
   Merupakan sebuah *Emulator* jaringan yang mensimulasikan koneksi dari *host-end, switch, router,* dan *link* pada *single kernel Linux*. Masing-masing elemen ini disebut sebagai *"host"* menggunakan virtualisasi ringan untuk membuat sistem tampilan tunggal sehingga terlihat seperti jaringan yang lengkap, menjalankan *kernel* yang sama, sistem, dan *user code.*
5. *Algoritma Least Connection*
   Merupakan algoritma pemilihan *server* untuk melakukan penyeimbangan beban *server*. Sehingga kinerja dari sebuah *web server* bisa lebih efisien.
6. *Web server*
   Merupakan sekumpulan halaman yang memberikan data ataupun transfer data informasi terhadap permintaan yang dilakukan oleh *user*.
7. Parameter pengujian
   Merupakan parameter atau ukuran yang digunakan untuk mengetahui performa dari suatu *web server*.
8. *Load balancing*
   Suatu teknik yang digunakan untuk memisahkan antara dua atau banyak *network link*. Dengan mempunyai banyak *link* maka optimalisasi utilitas sumber daya, *throughput* atau *response time* akan semakin baik karena mempunyai lebih dari satu link yang bisa saling mem-backup pada saat *network down* dan menjadi cepat pada saat *network* normal.

3.2 Analisis Kebutuhan

Analisis kebutuhan yang diperlukan untuk menganalisis apa saja yang dibutuhkan oleh sistem pada penelitian yang dilakukan, sehingga dapat dilakukan sesuai dengan yang diharapkan. Pada sistem ini terdiri dari kebutuhan fungsional dan kebutuhan non-fungsional.

3.2.1 Kebutuhan Fungsional

Kebutuhan fungsional merupakan kebutuhan yang dapat dilakukan oleh sistem, informasi apa saja yang harus ada dan bisa dihasilkan oleh sistem. Kebutuhan fungsional pada sistem ini antara lain:

1. Sistem dapat melakukan *service* terhadap *client* dengan jumlah *request* tertentu dalam satu waktu.
2. Sistem dapat melakukan pembagian beban *server* berdasarkan algoritma *least connection*.
3. Sistem dapat melakukan pengontrolan terhadap *web server* secara terpusat melalui *controller* pada *software defined network.*
4. Sistem dapat melakukan log terhadap jumlah koneksi setiap *server* dan juga log pada koneksi yang *flow*.

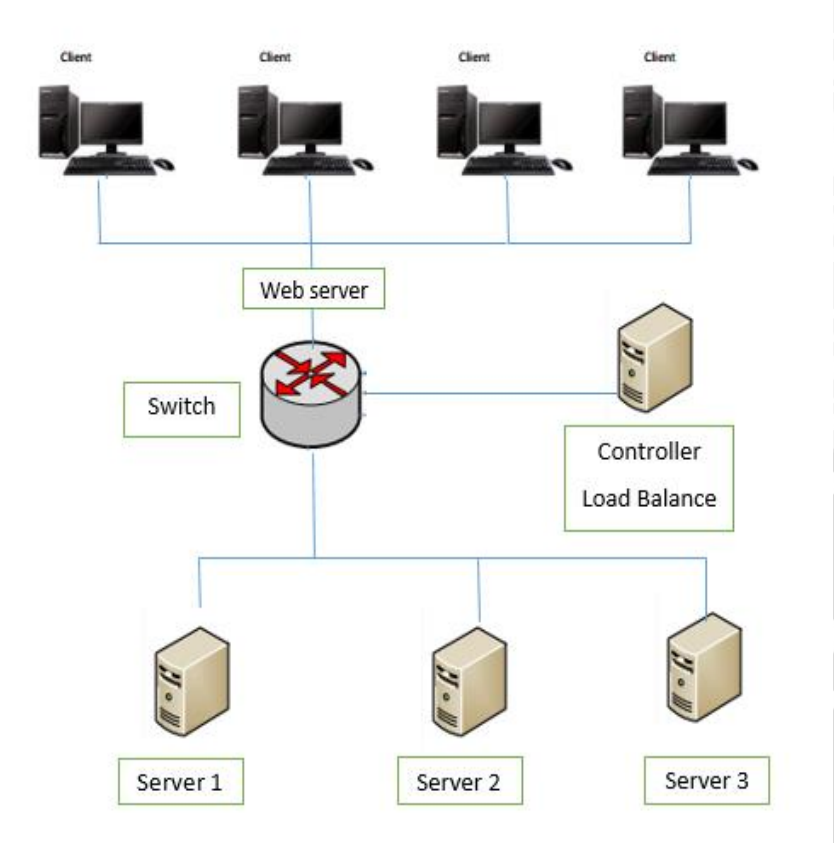
3.2.2 Kebutuhan Non-Fungsional

Kebutuhan non-fungsional merupakan kebutuhan tentang perangkat keras maupun lunak yang digunakan untuk membangun sistem agar sesuai dengan yang diharapkan. Kebutuhan non-fungsional pada sistem ini antara lain.

1. Kebutuhan perangkat keras yang dibutuhkan antara lain PC sebagai *controller* dan intalasi *mininet* sebagai simulasi *load balancing* serta *router* dengan instalasi *openflow*, Processor Intel(R) Core(TM) i3-3217U CPU 1.80GHz, 4.00 GB RAM dengan 64-bit *operating system*
2. Kebutuhan perangkat lunak antara lain *mininet*-VM dengan *operating system Ubuntu 14.04, Xming, Putty, python*, *pox controller* dan *httperf.*

3.3 Perancangan Sistem

Tahap ini merupakan tahapan untuk membangun sistem dari penelitian setelah melakukan studi literatur dan analisis kebutuhan sistem. Perancangan sistem dapat dilihat pada diagram blok berikut.



Gambar 3.2 Perancangan sistem

Pada Gambar 3.2 di atas merupakan perancangan yang digunakan pada topologi *mininet server* dengan menggunakan tiga *server* virtual yang masing-masing akan berkomunikasi pada *port* 80. Kemudian *Switch* yang bertugas meneruskan paket ke *controller* untuk selanjutnya diberikan *action* terhadap paket tersebut, lalu *traffic* paket akan dibagi berdasarkan algoritma *least connection*. Setelah itu dilakukan pengujian performa dan kinerja dari *web server* dengan *client* melakukan *request*. Perancangan sistem sendiri terdiri dari tujuh *host* dengan tiga *host* sebagai *server* dan empat *host* sebagai *client* dan juga satu buah *switch* dan satu buat *controller* dan *web server.*

3.3.1 Implementasi

Pada sub bab implementasi ini merupakan penjelasan perancangan sistem mulai dari perancangan kebutuhan sampai hasil akhir yang diperoleh oleh sistem.

Implementasi dilakukan dengan mengacu pada perancangan sistem. Maka implementasi meliputi :

a. Instalasi *mininet*-vm dan sistem operasi *linux*
b. Melakukan instalasi *controller POX*
c. Mebangun arsiterktur *load balancing web server software defined network* pada *mininet server*

d. Membuat algoritma *least connection load balancing* sebagai mekanisme pemilihan *server*
e. Melakukan akses ke halaman *web server*
f. Implementasi pengujian dengan parameter *httperf* untuk mengetahui performa dan kinerja *web server*

3.3.2 Pengujian dan Analisis

Pengujian dan analisis sistem dilakukan untuk mengetahui apakah kinerja sistem telah sesuai dengan kebutuhan atau tidak. Pada penelitian ini terdapat dua pengujian untuk menguji keberhasilan sistem .

1. Melakukan *request* dengan menggunakan 4 *client* secara bersamaan dengan memberikan beban pada *server* 50, 75, 100 dan 150 *request*/sec.
2. Melakukan *request* dengan menggunakan 4 *client* secara bersamaan dengan memberikan *rate* 25, 50, 75, dan 100 *conn/sec*.
3. Melakukan *request* dengan memberikan beban *server* dengan 200 *request*/sec dan 400 *request* /sec
4. Melakukan analisis dengan parameter yang digunakan adalah *connection rate, reply time, cpu time* dan *error.*
5. Melakukan analisis beban *server* saat awal koneksi, ketika terjadi *expired flow,* dan setelah *expired flow.*

3.3.3 Pengambilan Kesimpulan

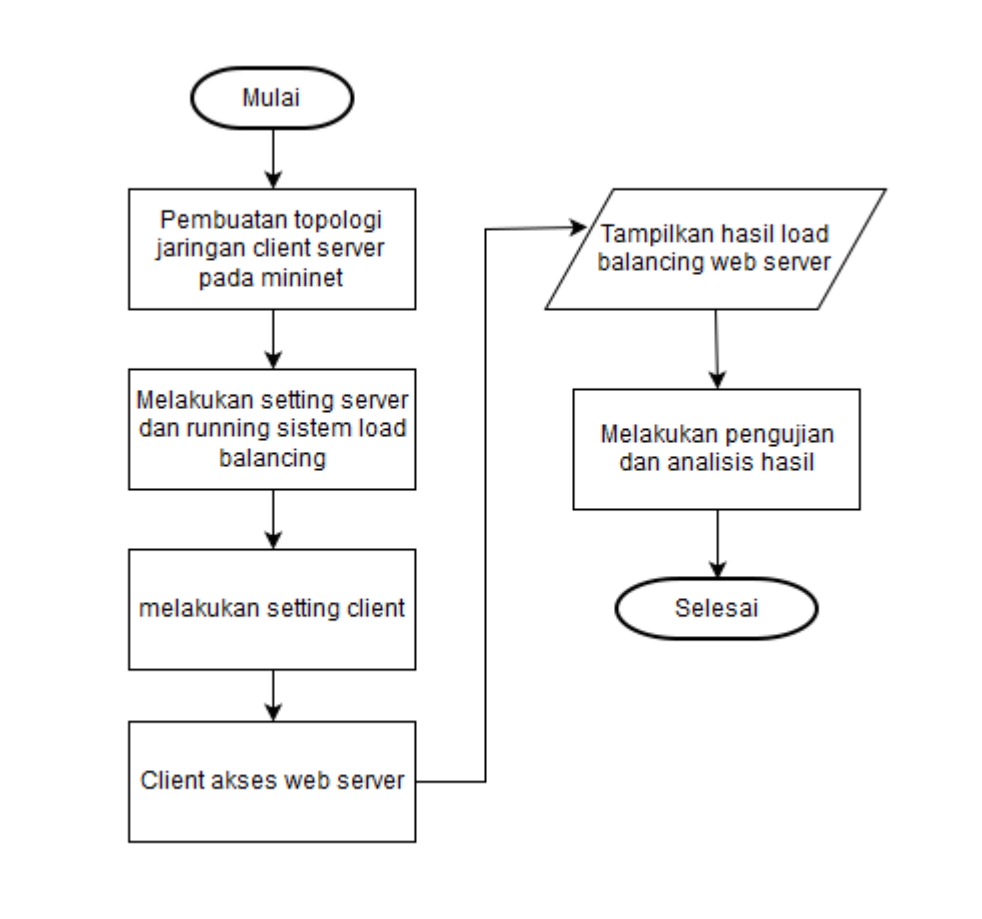
Pengambilan kesimpulan akan dilakukan setelah melakukan semua tahapan perancangan implementasi dan pengujian sistem yang dibuat selesai dilakukan. Kesimpulan diambil dari pengujian sistem yang telah dilakukan terhadap sistem yang telah dibuat. Dan tahap terakhir dalam penulisan adalah pemberian saran untuk pengembangan sistem lebih lanjut.

BAB 4 PERANCANGAN

Pada bagian ini akan dijelaskan tentang perancangan yang dilakukan dalam pengerjaan tugas akhir. Bagian ini menjelaskan lebih detail konsep perancangan *load balancing*, pengecekan paket, alur sistem dan juga diagram blok perancangan sistem serta alur komunikasi.

4.1 Diagram Alir Perancangan Sistem

Pada bagian ini akan dijelaskan perancangan sistem dari tahap pembuatan topologi sampai hasil pengujian dan kesimpulan yang dilakukan pada sistem.



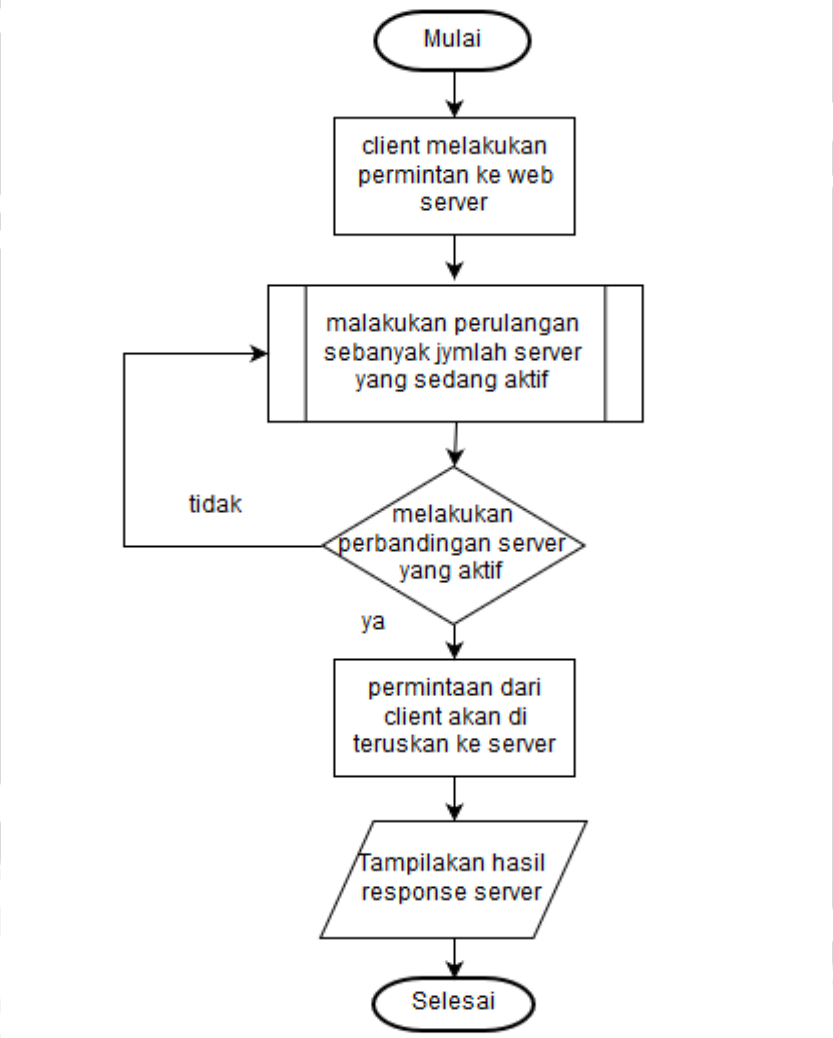
Gambar 4.1 Diagram Alir Perancangan Sistem

Pada Gambar 4.1 di atas dapat dijelaskan alir perancangan sistem yang dilakukan, Pada tahap awal dimulai dengan melakukan pembuatan topologi jaringan *client server* pada *mininet* yang nantinya digunakan untuk melakukan implementasi *load balancing*. Pada tahap ini terdapat tujuh buah *host* yang masing-masing tiga buah *host* sebagai *server* dan empat *host* lainnya bertugas sebagai *client*. Selanjutnya melakukan *setting* pada *server* dan *running* sistem *load balancing* setelah itu dilanjutkan dengan melakukan *setting* pada *client* yang akan digunakan untuk melakukan pengujian. Kemudian *client* melakukan komunikasi dengan melakukan akses ke *web server* yang sudah di*setting*

sebelumnya. Setelah *request client* terhadap *web server* berjalan maka proses selanjutnya akan ditampilkan hasil dari sistem *load balancing* menggunakan algorima *least connection*. Tahap akhir dari perancangan sistem ini adalah dengan melakukan pengujian terhadap sistem dengan melakukan analisis dari hasil pengujian menggunakan parameter yang sudah ada seperti *resply time*, cpu *time*, *connection rate*, *error time*.

4.2 Load Balancing

Pada perancangan sistem ini juga menggunakan mekanisme pemilihan *server* atau *load balancing*. Berikut algoritma yang menjelaskan tentang *load balancing* yang akan diterapkan:



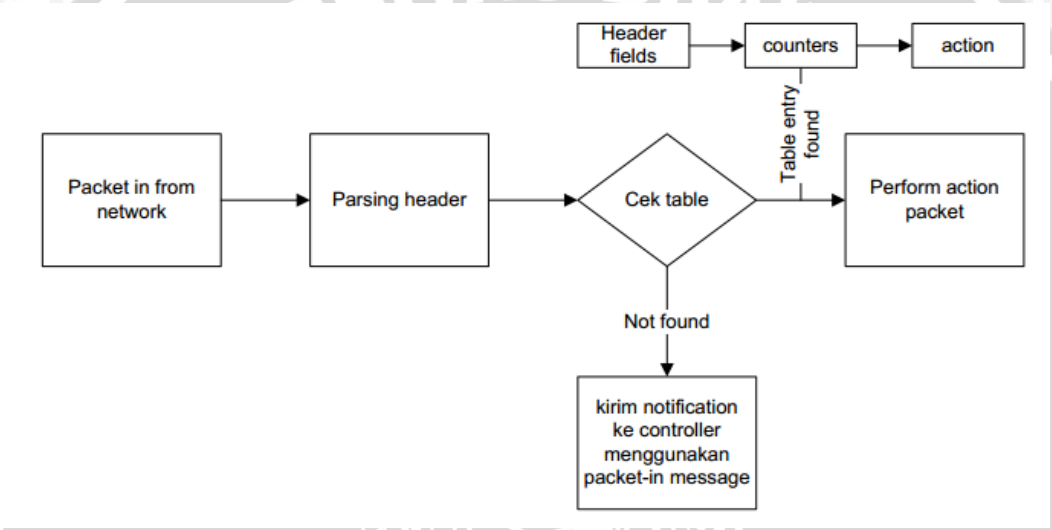
Gambar 4.2 *Flowchart Load Balancing*

Pada Gambar 4.2 di atas dapat dijelaskan proses *load balancing*, *load balancing* sendiri adalah melakukan pembagian beban untuk *server* dengan cara membagi *traffic request* yang dilakukan oleh *client* secara otomatis. Pada

diagram di atas proses pembagian *traffic* diawali dengan *request client* ke *server* yang selanjutnya *request* tersebut akan dilakukan pengecekan untuk pemilihan *traffic* jaringan. Pengecekan dilakukan dengan cara melakukan perbandingan antara *server* yang sedang aktif dimana *server* yang memiliki jumlah permintaan dari *client* paling sedikit maka akan diberikan *request client* yang masuk. Selanjutnya setelah melakukan pengecekan, *server* yang memiliki sedikit permintaan maka paket akan diteruskan ke *traffic server* tersebut. Namun ketika keadaan *server* tidak dalam melanyani *request* dari *client* maka *request* dari *client* yang pertama kali masuk akan diteruskan ke *server* yang pertama.

4.2.1 Pengecekan Paket

Berikut merupakan *flowchart* pengecekan *packet* forwarding dengan *openflow* di *switch :*



Gambar 4.3 flowchart Pengecekan Paket
Sumber: McKeown et al (2008)

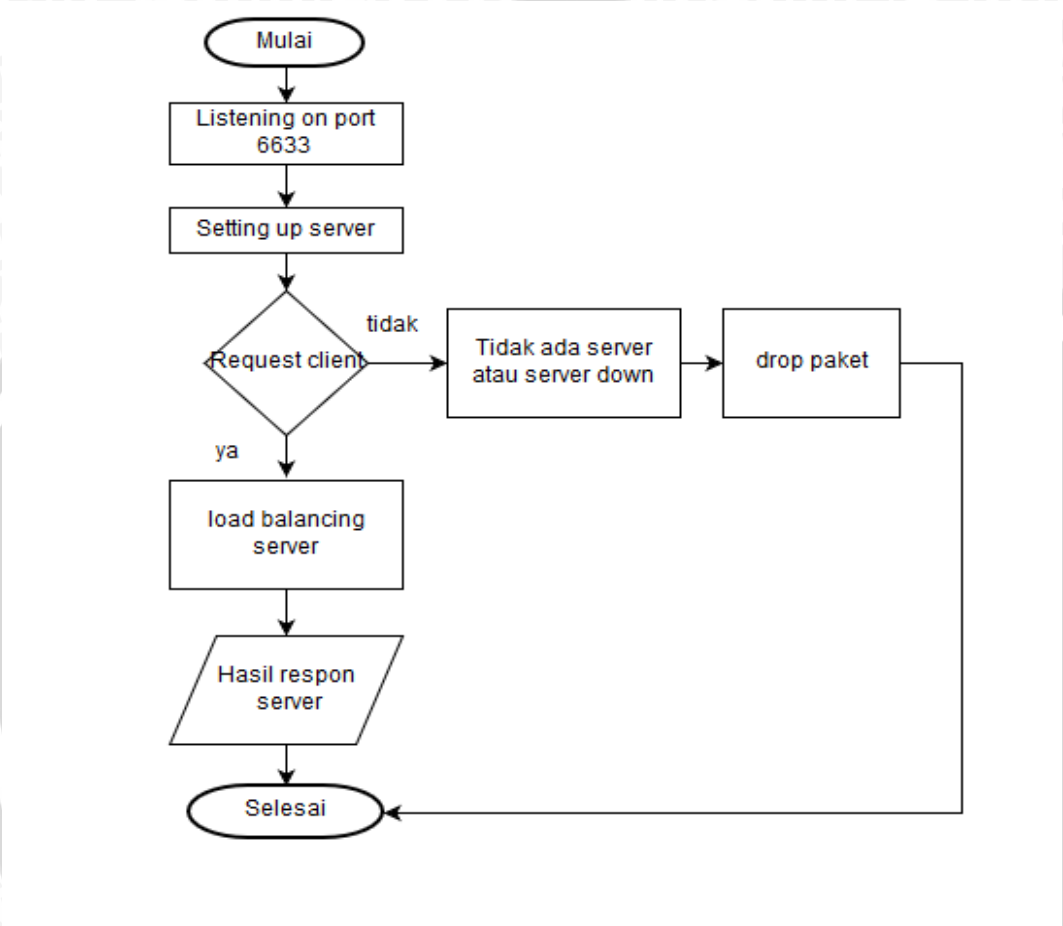
Berdasarkan Gambar 4.3 di atas maka dapat disimpulkan bahwa ketika ada paket dari suatu jaringan maka akan diproses *parsing header* yang dimiliki paket tersebut. *Header* paket berisi tentang informasi paket. Selanjutnya dilakukan pengecekan pada *tabel openflow* untuk dilakukan *action* terhadap paket tersebut. Jika dalam tabel *openflow* paket sesuai maka *switch* akan meneruskan paket menuju *destination*, sedangkan jika tidak ada kesesuaian *header* paket maka *switch* akan mengirimkan pemberitahuan kepada *controller* untuk memberikan *action* terhadap paket tersebut. Setiap paket yang datang dalam suatu jaringan maka diberikan action terhadap paket tersebut sesuai dengan *flow tabel* yang diatur oleh *controller*.

Mekanisme pengecekan tabel berdasarkan tiga komponen yaitu *header fields, counters dan action*. *Header fields* merupakan pengecekan *header* yang dimiliki oleh paket. *Counter* merupakan jumlah paket dan *byte* yang dimiliki oleh setiap *flow.* Sedangkan *action* merupakan aktifitas yang diberikan terhadap

paket. Misalnya paket akan diteruskan menuju ke tujuan ataupun paket akan dibuang karena tidak sesuai dengan tabel *flow* paket

4.3 Alur Sistem

Load balancing pada *software defined network* dengan menggunakan algoritma *least connection* akan dirancang alur sistem sebagai berikut :



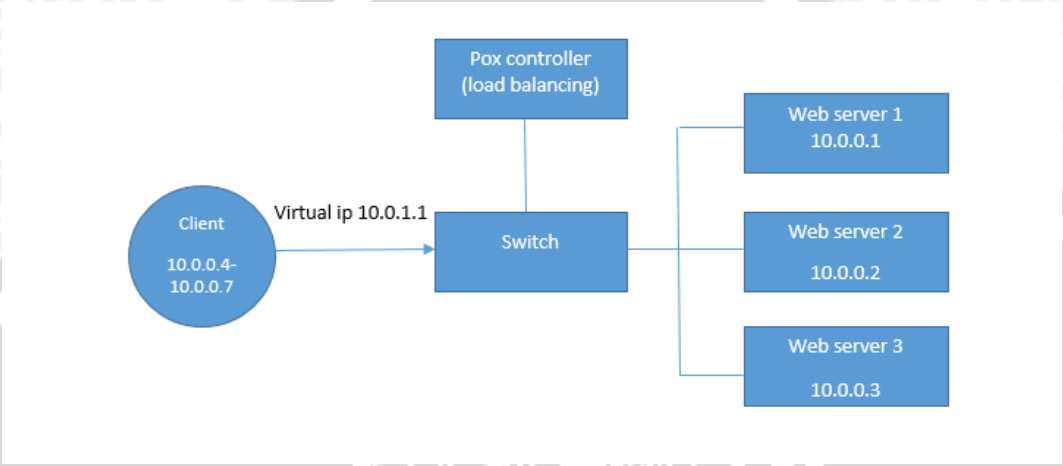
Gambar 4.4 Alur Sistem

Pada Gambar 4.4 di atas dapat dijelaskan ketika sistem dijalankan maka akan melakukan *listening* ke *port* 6633 untuk *pox controller* dan selanjutanya melakukan *setting up server*. Ketika selesai melakukan *setting server* virtual yang akan *listening request* pada *port 80, maka* controller akan *setting up server* tersebut. Saat ada *server* yang *down* maka akan dibuang oleh *controller.* Ketika tidak ada *setting server* maka akan muncul peringatan tidak ada *server.* Sehingga akan dibuang juga permintaan *client* tersebut. Saat selesai melakukan *setting server* virtual, maka *client* yang melakukan *request* akan terhubung langsung ke *server tersebut.*

Ketika ada *request* dari *client* sistem akan melakukan pengecekan terhadap *server* yang sedang up. Pada saat tidak ada *server* yang up atau aktif maka *request* dari *client* akan dibuang dari *traffic,* tetapi jika semua pengecekan menunjukan ada *server* yang sedang aktif maka *request* dari *client* akan

dilanjutkan ke *server*. Sebelum sampai ke *server traffic* yang masuk akan dibagi berdasarkan algoritma *load balancing*. Dalam hal ini algoritma *load balancing* yang digunakan *least connection* dimana pembagian *traffic* berdasarkan *server* yang paling sedikit sedang menangani *request* dari *client.* Setelah melakukan pengecekan *server* dengan *load balancing* maka selanjutnya *trafiic* dari *client* akan diteruskan ke *server* yang selanjutnya akan menunggu respon dari *server* yang dituju untuk kemudian ditampilkan respon *server* tersebut.

4.4 Alur Komunikasi



Gambar 4.5 Alur Komunikasi

Pada Gambar 4.5 di atas dapat diketahui alur komunikasi yang akan dilakukan oleh *user*. Pertama *user* dengan alamat *client* 10.0.0.1 sampai 10.0.0.7 akan mengakses ip virtual *web server* dengan alamat 10.0.1.1. Selanjutnya *client* akan terhubung ke *switch* yang bertugas untuk meneruskan permintaan *user* menuju ke alamat *destination. Switch* akan terhubung ke sebuah *controller*, *controller* bertugas untuk membuat keputusan terhadap paket yang masuk. Kemudian sistem *load balancing* akan membagi *request user* berdasarkan jumlah *server* yang tersedia.

Permintaan dari *client* akan dibagikan ke *server* berdasarkan pada algoritma *least connection*. Alamat *server* pada *ip* 10.0.0.4 sampai ip 10.0.0.7. Algoritma *least connection* akan memilih *server* yang memiliki jumlah koneksi paling sedikit maka permintaan akan diteruskan ke *server* tersebut. Pada saat *user* melakukan permintaan ke *web server*, *controller* akan memberikan keputusan terhadap paket tersebut. Sebelum memberikan keputusan *controller* akan melakukan pengecekan paket pada tabel *openflow.* Jika dalam tabel *openflow* paket sesuai maka *switch* akan meneruskan paket menuju *destination*, sedangkan jika tidak ada kesesuaian *header* paket maka *switch* akan mengirimkan pemberitahuan kepada *controller* untuk memberikan *action* terhadap paket tersebut.

Jumlah *server* yang digunakan pada komunikasi ini berjumlah tiga *server*. Ketika keadaan semua *server* dalam keadaan kosong atau sedang tidak melayani permintaan dari *user*. Koneksi yang pertama kali masuk akan diteruskan ke *server* yang pertama. Selanjutnya ketika ada koneksi yang dibuang dari *server* karena

batas waktu koneksi maka permintaan *user* yang berikutnya masuk akan diteruskan ke *server* yang paling sedikit sedang melayani permintaan dari *user*. Sehingga *server* tidak mengalami *overload* karena banyaknya permintaan dari *user*. Pembagian beban *server* ini diatur melalui *controller* sebagai pengatur komunikasi dalam *software defined network*.Masing-masing *server* mendapatkan respon yang sama yang diberikan oleh *web server*. Karena *server* yang ada memiliki paket data yang sama berupa halaman *html* yang diakses oleh setiap *user*.

Model alur komunikasi yang digunakan menggunakan MAC, dimana host yang tergabung atau terhubung dalam sebuah jaringan LAN saling berkomunikasi menggunakan alamat fisik (*Mac Address*). Komunikasi dimulai pada tahap transfer data sebelum sebuah data diberikan *Mac Address,* terlebih dulu data tersebut diberi alamat logis berupa IP *Address*. Ketika *client* melakukan request *client* akan mengakses alamat IP *Address* dari web server yakni 10.0.1.1. kemudian paket akan masuk ke *switch*. Ketika berada di *switch* paket akan disalurkan ke server berdasarkan algoritma least connection. Ketika algoritma least connection sudah memilih server mana yang akan menerima paket yang dikirimkan oleh *client*. Paket dari *client* akan diberikan alamat IP *host* tujuan yakni 10.0.0.1. IP *Address* yang ditambahkan merupakan IP *Address* dari *host* pengirim dan *host* penerima yakni 10.0.1.1 dan 10.0.0.1. Selanjutnya untuk menentukan alamat fisik atau *Mac Address* dari *host* tujuan digunakan protokol ARP dengan memanfaatkan informasi IP *Address host* tujuan yang ada, maka host pengirim melakukan pencarian dengan menugaskan protokol ARP.

Protokol ARPakan melakukan pengiriman sebuah pesan yang bersifat broadcast yang berisi pesan permintaan sebuah alamat Mac *Address* suatu *host* berdasarkan alamat IP *Address* setiap *host*. Protokol ARP akan melakukan broadcast ke IP server 10.0.0.1, 10.0.0.2 dan 10.0.0.3. Setelah pesan sampai pada Ip *host* tujuan yakni 10.0.0.1, maka IP host 10.0.0.1 akan membalas pesan tersebut dengan sebuah pesan balasan yang berisi alamat fisik atau Mac *Address* yang sesuai dengan IP *Address* yang diminta client. Kemudian paket akan di teruskan ke IP 10.0.0.1 sebagai server 1 dengan menggunakan alamat *Mac Address* yang telah diberikan.

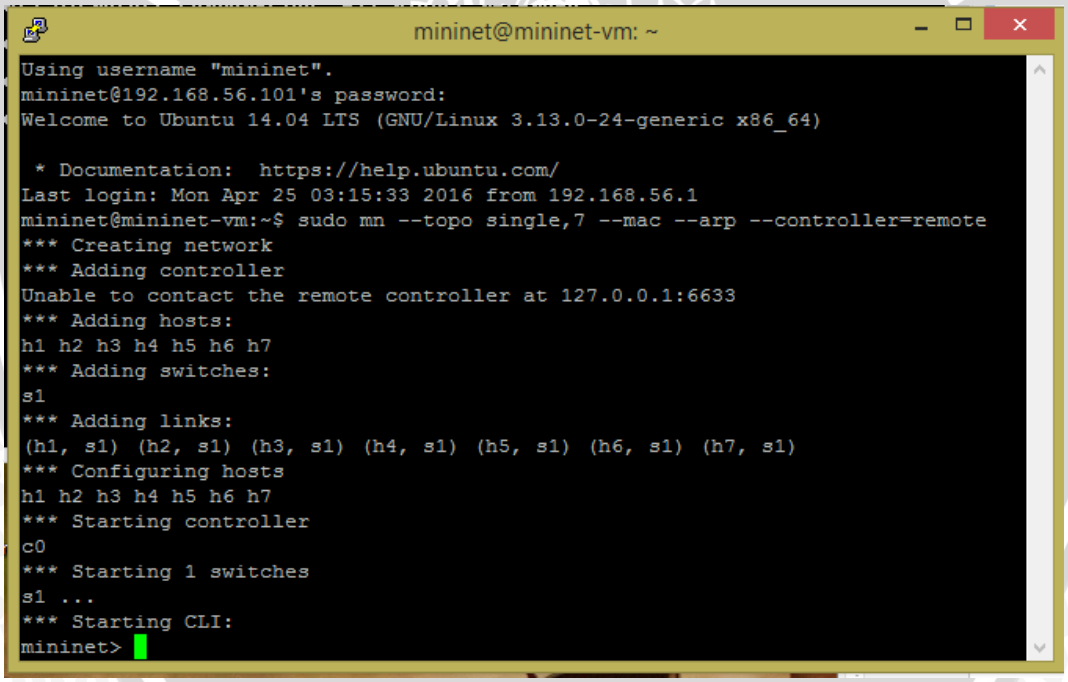
BAB 5 PEMBAHASAN

5.1 Membuat Topologi

Untuk melakukan implementasi *load balancing* di *web server* pada *software defined network*, hal pertama yang dilakukan adalah melakukan inisialisasi topologi pada *mininet*. Topologi yang akan digunakan adalah dengan menggunakan tujuh *host* dan *pox* sebagai remote *controller* serta *single switch*. Berikut *source code* untuk topologi :

```
1   $ sudo mn --topo single,7 --mac --arp --controller=remote
```

Pada baris program di atas dapat dijelaskan membuat topologi dengan *single switch* dengan jumlah *host* sebanyak tujuh *host*, Kemudian mengatur alamat MAC *address* secara otomatis dan juga ARP antar *host* dan *remote contoller* yang akan digunakan sebagai pengatur *traffic* paket. Berikut Gambar 5.1 hasil dari program di atas :



Gambar 5.1 Membuat Topologi Jaringan

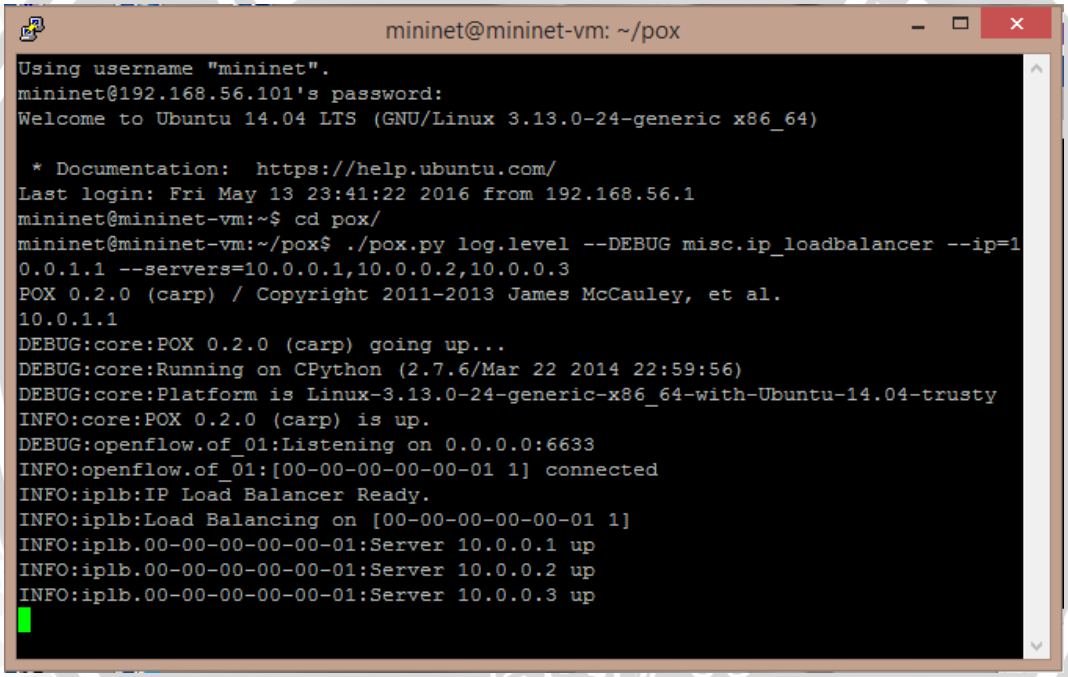
Dari Gambar 5.1 dapat dijelaskan dalam topologi melakukan *create host* sebanyak tujuh *host* yaitu h1 sampai h7. Kemudian menambahkan *controller* yaitu (c0) dan *single switch* (s1). Untuk mengubungkan masing-masing *host* juga dilakukan create links dari (h1,s1) sampai (h7,s1). Setelah semua topologi selesai dibuat *controller* dan *switch* akan melakukan *starting* sistem.

5.2 Menjalankan *Load Balancing*

Untuk melakukan implementasi *load balancing* pada *software defined network* dengan menggunakan algoritma *least connection* terlebih dahulu dilakukan *running* sistem *load balancing* dengan menggunakan *pox* sebagai *controller*. *Source code* program sebagai berikut:

```
1  $ ./pox.py log level DEBUG misc.ip_loadbalancing --ip=10.0.1.1
2  --servers=10.0.0.1,10.0.0.2,10.0.0.3
```

Pada baris ke-1 digunakan untuk melakukan *import module load balancing* dengan direktori pada *pox.* Kemudian dilanjutkan dengan menjalankan file sistem dari *load balancing*. Selanjutnya mengatur alamat *web server* dengan IP (10.0.1.1) dan juga *server* pada *host* dengan IP 10.0.0.1 sampai 10.0.0.3. Berikut Gambar 5.2 hasil dari perintah program di atas:



Gambar 5.2 Menjalankan *Load Balancing*

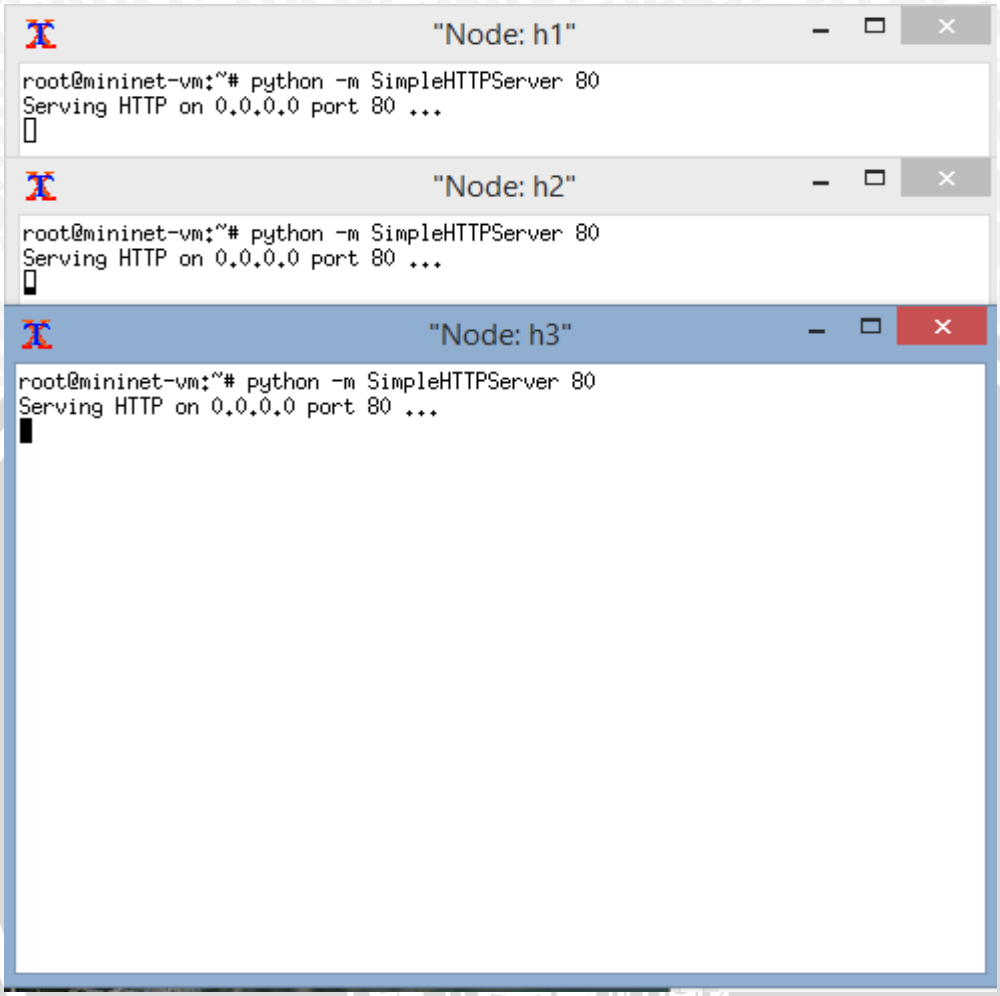
Pada Gambar 5.2 dapat diketahui ketika *controller pox* aktif maka *controller* akan melakukan *listening* pada *port* 6633 dan akan melakukan *setting* up pada *server* 10.0.0.1 sampai 10.0.0.3. Ketika selesai melakukan *setting* maka virtual akan *listening port 80.* Maka sistem *load balancing* siap untuk dijalankan.

5.2.1 *Setting Server*

Untuk melakukan pengaturan *server* pada *host* digunakan perintah sebagai berikut :

```
1  #python -m SimpleHTTPServer 80
```

Perintah di atas digunakan untuk melakukan pengaturan *server* pada *host* dengan menggunakan protokol HTTP dengan *port* 80. Berikut gambar 5.3 hasil dari perintah tersebut :



Gambar 5.3 *Setting Server*

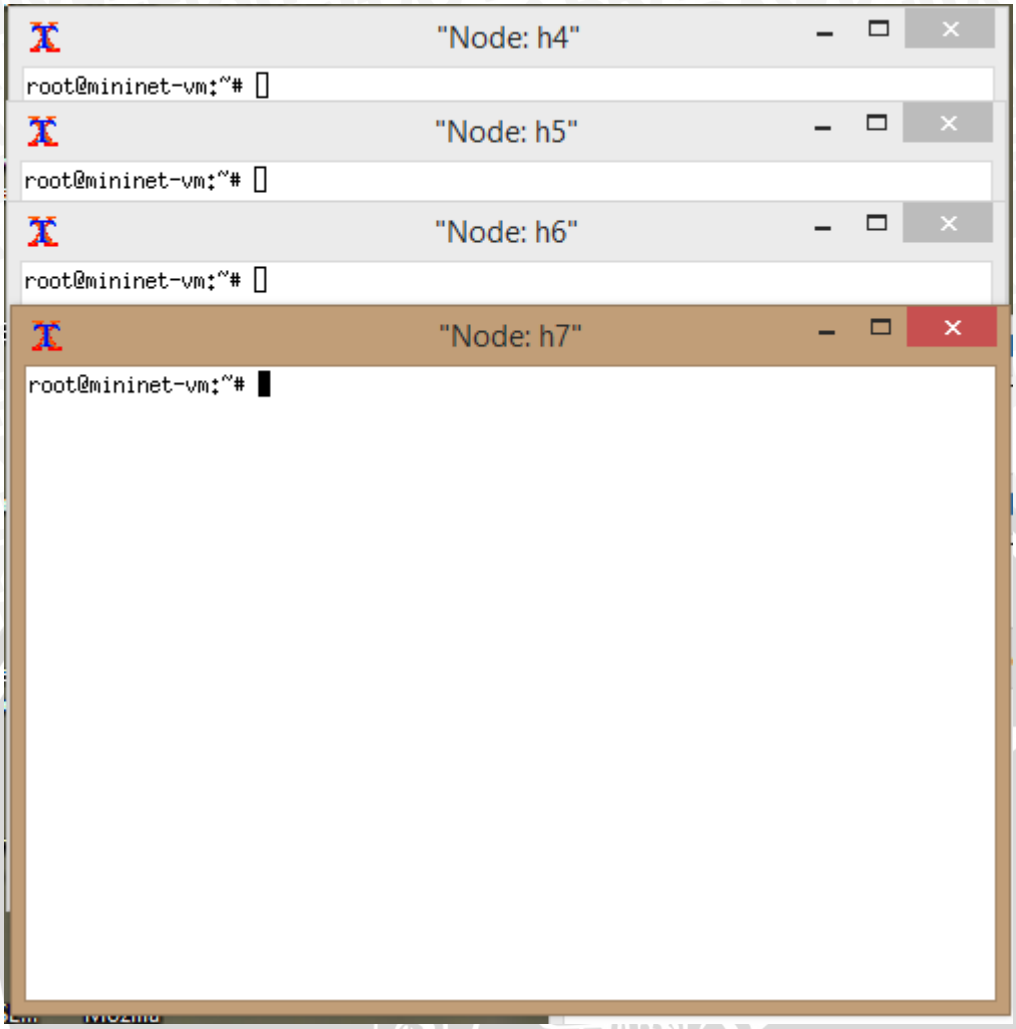
Pada Gambar 5.3 dapat diketahui bahwa pengaturan *server* pada *host* sudah selesai dan *server* siap digunakan pada protokol HTTP *port* 80.

5.2.2 *Setting Client*

Untuk malakukan pengujian *load balancing* pada *web server* diperlukan *client* untuk melakukan *request*. Berikut program untuk melakukan *setting client*.

```
1   xterm h1 h2 h3 h4 h5 h6 h7
```

Pada program diatas xterm h1 sampai dengan h7 digunakan untuk membuat jaringan *host* h1 sampai h7 yang nantinya akan digunakan sebagai *server* dan *client*. Berikut Gambar 5.4 hasil dari program di atas :



Gambar 5.4 *Setting Client*

Pada Gambar 5.4 di atas adalah tampilah jaringan *host* h4 sampai h7. *Host* h4 sampai h7 ini nantinya akan berfungsi sebagai *client* 1 sampai *client* 4 yang akan melakukan permintaan ke *web server*.

5.2.3 *Web Server*

Untuk malakukan implementasi *load balancing* digunakan perintah *#curl* pada alamat IP *10.0.1.1* yang dilakukan pada sisi *client*. Kemudian akan muncul halaman *doctype html* yang nantinya data tersebut digunakan untuk melakukan permintaan ke *server*. Berikut Gambar 5.5 hasil *curl* yang dilakukan oleh *client* sebagai berikut :

Gambar 5.5 *Web Server*

Pada Gambar 5.5 di atas dapat diketahui bahwa permintaan *client* dengan mengunakan *#curl* telah dikirim oleh *server* yang berupa halaman *doctype html.*

5.3 Sistem *Load balancing Least Connection*

Pada sistem laod *balancing* pada *software defined network* untuk melakukan pemilihan *server* dilakukan oleh *controller.* Sistem *load balancing least connection* dengan *source code* sebagai berikut :

```
1   global selected_server,cj
2       k=0;
3       m=1;
4       for i in range(m+1,(len(self.live_servers)+1)):
5           if (len(cj[i])) < (len(cj[m])):
6               m = i
7       k = m
8       cj[k].append(self.memory)
9       print len(cj)
10      j = len(cj)
11      for m in range(1,(len(self.live_servers)+1)):
12        print "Jumlah koneksi server %i =" % m
13        print len(cj[m])
14        f = open("/home/mininet/pox/pox/misc/c_server.txt","a")
15        f.write("koneksi ke %i = "%j)
16        f.write("koneksi server %i = "%m) #\n untuk baris baru
17        f.write(str(len(cj[m])))
18        f.write("\n")
19        f.close()
20      a=self.live_servers.keys()
21      selected_server=k-1
22      b=a[selected_server]
23      return b
```

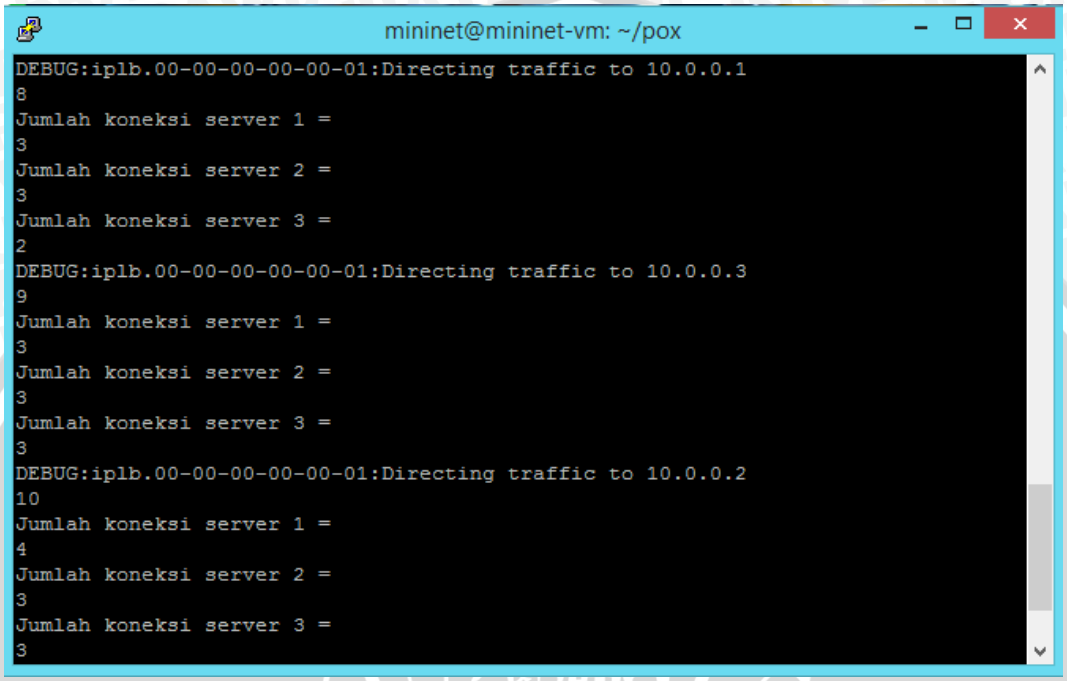
Gambar 5.6 Algoritma *Least Connection*

Berdasarkan Gambar 5.6 di atas dapat dijelaskan jalan program *least connection* sebagai berikut :

- Pada baris ke-1 merupakan inisialisasi variable global untuk selected *server* dan cj. Variable CJ digunakan untuk menampung nilai dari koneksi semua *server*.
- Pada baris ke-2 dan ke-3 digunakan untuk melakukan deklarasi nilai awal
- Pada baris ke-4 sampai ke-7 merupakan algoritma *least connection*, dimana variabel (i) akan melakukan perulangan sebanyak jumlah *server* yang aktif. Selanjutnya nilai dari (m) diinisialisasikan sebagai *server* 1 yang kemudian akan dibandingkan dengan *server* lain (i). Ketika nilai (i) lebih kecil dari (m) maka variabel (m) akan benilai (i) dan nilai dari variabel (k) akan bernilai sama dengan variabel (m).
- Pada baris ke-8 variabel (cj) merupakan array yang digunakan untuk menyimpan nilai dari semua *server* yang aktif dan berisikan alamat IP *server*, source IP, destination IP, source *Port* dan destination *Port*.
- Pada baris ke-9 sampai ke-14 digunakan untuk menampilkan jumlah total semua koneksi dari *client* yang sudah masuk ke *server*.
- Pada baris ke-15 sampai ke-20 digunakan untuk menyimpan nilai dari jumlah koneksi masing masing *server* yang disimpan dalam *.txt.*
- Pada baris ke-21 variabel a diberikan nilai yang menyimpan alamat IP *server*, source IP, destination IP, source *Port* dan destination *Port*.

- Pada baris ke-22 digunakan untuk pemilihan *server*, karena *selected server* = 0 maka nilai dari (K) akan dikurangi 1
- Pada baris ke-23 memberikan nilai baru kepada variabel (b) sesuai dengan nilai (a) dan selected_*server*.
- Pada baris ke-24 program akan kembali ke baris ke-23

Berikut Gambar 5.7 hasil dari program di atas:



Gambar 5.7 *Load Balancing Least Connection*

Pada Gambar 5.7 dapat diketahui bahwa program akan membandingkan jumlah koneksi dari masing- masing *server*. *Server* yang memiliki jumlah koneksi paling sedikit maka akan diberikan beban koneksi berikutnya.

5.4 *Expired Flow*

Pada *software defined network expired flow* adalah batas waktu hidup dari sebuah koneksi. Ketika sebuah koneksi sudah melewati batas *expired flow* maka koneksi tersebuat akan dibuang dari *traffic* jaringan yang sedang berlangsung. Berikut *source code* sistem *expire flow* dari *load balancing* dengan *least connection:*

```
1  # Expire old flows
2      c = len(self.memory)
3      z = 0
4      y = 1
5      self.memory = {k:v for k,v in self.memory.items()
6                     if not v.is_expired}
7      if len(self.memory) != c:
8        self.log.debug("Expired %i flows", c-len(self.memory))
9        a = c-len(self.memory)
10       b = a/2
11       for j in range(0,b):
12         for i in range(1,(len(self.live_servers)+1)):
13           if len(ci[i])!=0 :
14             if len(ci[i]) > len(ci[y]):
15                y = i
16             z = y
17             del ci[z][0]
18             f = open("/home/mininet/pox/pox/misc/c_flow.txt","a")
19             f.write("server %i="%z)
20             f.write(str(len(ci[z])))
21             f.write("\n")
22             f.close()
23           else :
24             return
```

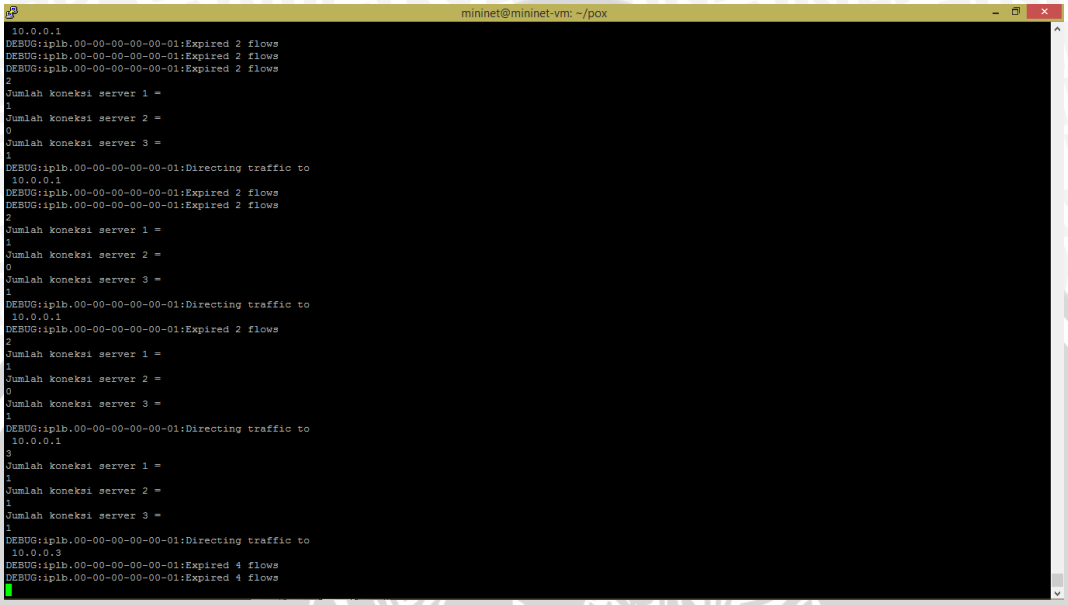
Gambar 5.8 *Expired flow*

Berdasarkan Gambar 5.8 dapat dijelaskan jalan program *expired flow* sebagai berikut:

- Pada baris ke-2 sampai ke-4 melakukan inisialisasi variable yang akan digunakan untuk menyimpan nilai–nilai yang ada pada program
- Pada baris ke-5 dan ke-6 memberi nilai dari *self.memory* apakah sama dengan *self.memory.item()*
- Pada baris ke-7 dan ke-8 melakukan pengecekan apakan nilai dari panjang *self.*memory sama dengan nilai (c) jika tidak sama maka nilai (c) akan dikurangi dengan nilai panjang *self.memory* dan *traffic* akan di buang (drop).
- Pada baris ke-9 dan ke-10 nilai variable (a) akan diganti dengan nilai *expired flow* kemudian nilai dari variable (b) akan diganti dengan nilai variable (a) dan dibagi 2. Karena setiap 1 *flow* memory terdiri dari 2 key
- Pada baris ke-11 melakukan pengedropan *traffic* sebanyak jumlah *expired* yang terjadi
- Pada baris ke-12 melakukan pengedropan berdasarkan jumlah koneksi yang ada pada tiap *server. Server* yang memiliki jumlah koneksi paling banyak maka *expired flow* akan di berikan kepada *server* tersebut
- Pada baris ke-13 sampai ke-32 terjadi pengecekan jika jumlah koneksi yang di layani *server* tidak nol maka kan terjadi *expire flow* pada *server* tersebut.

- Pada baris ke-33 terjadi pengecekan jika jumlah beban di *server* sama dengan nol maka *server* akan mengulang dengan membanding setiap *server* yang sedang altif. Ketika sistem tidak terjadi permintaan yang dilakukan *user* maka beban reqest yang ada di *server* akan kembali menjadi 0.

Berikut Gambar 5.9 hasil dari program di atas:



Gambar 5.9 *Expired flow*

Dari Gambar 5.9 di atas dapat dijelaskan ketika terjadi waktu *expired flow traffic* yang melebihi waktu *expired flow* akan dibuang dan tidak akan di salurkan ke *server*. *Server* yang memiliki jumlah koneksi terbanyak yang akan dicek terlebih dahulu untuk diberikan *expired flow*.

BAB 6 PENGUJIAN

Pada bab ini akan membahas hal yang terkait dengan pengujian seperti tujuan pengujian, proses pengujian, skenario atau prosedur pengujian serta analisis terhadap hasil data dari pengujian yang telah dilakukan.

6.1 Pengujian *Web Server*

6.1.1 Tujuan Pengujian

Tujuan dari pengujian ini untuk mengetahui performa dari *sistem load balancing web server* pada *software defined network.* Pengujian juga dilakukan untuk mengetahui hasil dari nilai rata-rata paramater yang diberikan seperti *request* dan juga *rate*. *Request* digunakan untuk memberikan beban permintaan *client* kepada *server* dalam satu waktu (*rate*) sehingga dapat diketahui kemampuan *server* dalam melakukan respon terhadap *client*. *Rate* pada pengujian ini adalah digunakan untuk memberikan ukuran kecepatan bit data transmisi dalam satu waktu. Selanjutnya hasil dari masing-masing percobaan akan dibandingkan dan dilakukan analisis dari hasil yang didapat. Sehingga dapat diketahui performa dari *web server* dalam melakukan respon terhadap permintaan *client*.

6.1.2 Proses Pengujian

Dalam hal ini pengujian dilakukan dengan memberikan *request* ke *web server* sebanyak 50 kali, 75 kali, 100 kali dan 150 kali. Sedangkan untuk *rate* yang digunakan yaitu 25 conn/s, 50 conn/s, 75 conn/s, dan 100 conn/s. Proses pengujian dilakukan dengan melakukan *request* secara bersaman dengan menggunakan empat *client.* Untuk *source* pengujian sebagai berikut:

```
1  # httperf --server 10.0.1.1 --port 80 --num-conns x --rate y
```

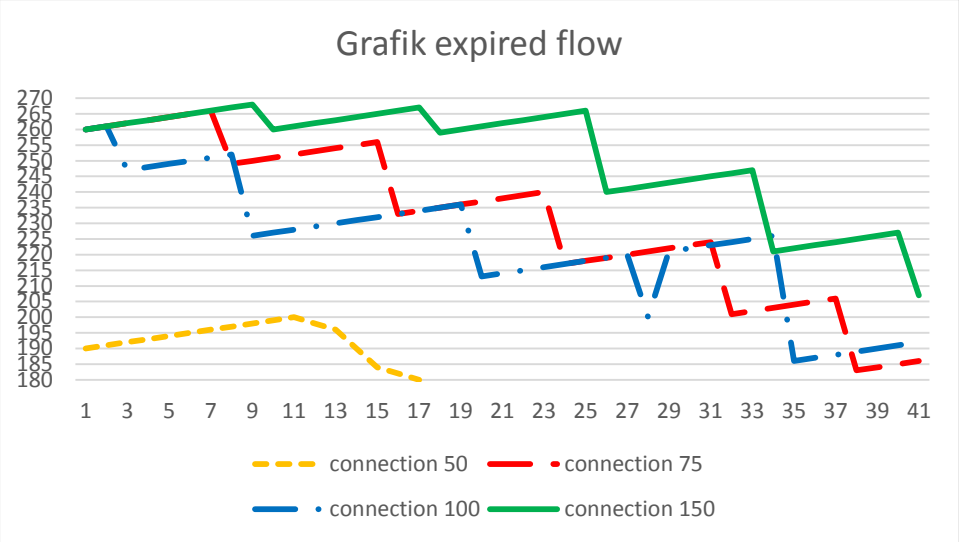
Pada program di atas dijelaskan *tool* yang digunakan untuk melakukan pengujian *load balancing* dengan menggunakan #*httperf* dengan alamat *web server* yang di akses oleh *client* 10.0.1.1. *Port* yang digunakan untuk komunikasi *client server* adalah *port* 80 serta banyaknya permintaan yang akan dilakukan ke *server* dan juga jumlah koneksi yang akan dibuat dalam satu waktu.

6.1.3 Skenario Pengujian

Skenario pengujian dilakukan dengan melakukan pengujian terhadap masing-masing *server*, jumlah permintaan *user* dan juga berdasarkan nilai *fixed rate*. Dari skenario pengujian diatas maka akan didapat parameter pengujian diantaranya jumlah rata-rata *connection rate* dengan satuan conn/s, jumlah rata- rata *cpu time* dengan satuan *s*, jumlah *error rate* dengan satuan paket *error* dan terakhir jumlah rata–rata *reply time server* dengan satuan *ms.*

6.2 Hasil Pengujian

Pada skenario pengujian yang pertama dilakukan pengujian di sisi *web server*. Pengujian ini bertujuan untuk mengetahui pada koneksi ke berapa *server* melakukan *expired flow*. Skenario pengujian dilakukan dengan melakukan permintaan *user,* dengan jumlah permintaan yang dilakukan oleh *user* terhadap *web server* sebanyak 50 kali, 75 kali, 100 kali dan juga 150 kali. *Rate* yang digunakan dalam pengujian ini adalah *rate 0,* sedangkan *server* yang digunakan pada pengujian sistem *load balancing* ini sebanyak 3 *server.* Berikut hasil



pengujian yang dilakukan.

Gambar 6.1 Grafik Hasil *flow* dari Sistem *Load Balancing*

Berdasarkan Gambar 6.1 dapat disimpulkan ketika *user* melakukan *request* ke *server* terjadi dua kondisi yakni kondisi ketika *request* sedang aktif atau diproses *server* dan kondisi ketika *request* down atau dibuang dari *traffic* oleh sistem sehingga terbentung grafik seperti tabel di atas. Berikut penjelasan grafik di atas sebagai berikut :

1. Pada pengujian dengan menggunakan jumlah permintaan *user* sebanyak 50 kali, ternyata pada sistem *load balancing* tidak terjadi *expired flow*. Dari grafik dapat dilihat ketika jumlah koneksi sudah sesuai dengan jumlah *request* yang diminta *user* maka koneksi akan menurun sampai jumlah koneksi ke *server* menjadi 0.
2. Pada pengujian dengan menggunakan jumlah permintaan *user* sebanyak 75 kali, dapat dilihat pada grafik ternyata pada koneksi ke-266 terjadi *expire flow* dan jumlah koneksi ke *server* turun menjadi 249 koneksi kemudian akan naik lagi menjadi 256. Ketika koneksi dari semua *user* sudah selesai di proses maka pada koneksi 300 jika tidak ada lagi koneksi maka jumlah koneksi akan menurun sampai jumlah koneksi ke *server* kembali bernilai 0.

3. Pada pengujian dengan menggunakan jumlah permintaan *user* sebanyak 100 kali, pada grafik dapat dilihat terjadi *expired flow* pada koneksi ke-261 dan jumlah koneksi ke *server* turun menjadi 247 koneksi yang sedang dilayani oleh *server* kemudian pada koneksi ke-263 koneksi kembali mengalami kenaikan. Koneksi akan naik dan turun sampai semua jumlah *request* yang diminta oleh *user* selesai. Ketika tidak ada *request* lagi pada koneksi ke-400 koneksi ke *server* akan turun secara bertahap sampai nilai masing-masing *server* kembali ke 0.
4. Pada pengujian dengan melakukan jumlah permintaan *user* sebanyak 150 kali, pada grafik dapat dilihat terjadi *expired flow* pada koneksi ke-268 dan jumlah koneksi ke *server* turun menjadi 260 koneksi yang sedang dilayani oleh *server* kemudian pada koneksi ke-270 keoneksi kembali mengalami kenaikan. Ketika tidak ada *request* yang dilakukan oleh *user* koneksi ke *server* akan turun sampai nilai koneksi yang dilayani masing- masing *server* kembali bernilai 0.

Dari hasil grafik di atas maka dapat disimpulkan, *request* yang dilakukan oleh *user* akan terjadi *expired flow* pada *range* r*equest* ke-275 hal ini dikarenakan *server* akan melakukan *refresh* sistem setiap 60s. Pada saat *request* sudah berjalan selama 60s maka sistem akan melakukan pemutusan koneksi yang melebihi batas *expired* maka terjadilah *expired flow.* Dengan demikian beban *server* akan tetap stabil dan kemungkinan *overload* pada *server* semakin kecil. Sehingga *server* tetap dapat memberikan respon yang stabil kepada *request* dari *user.*

6.2.2 Pengujian dengan Rate 25 conn/s

Pada pengujian ini dilakukan dengan memberikan rete 25 conn/s dengan *request user* sebanyak 50 kali, 75 kali, 100 kali dan 150 kali. Didapatlah hasil pengujian sebagai berikut :

Tabel 6.1 Hasil pengujian rate 25conn/s

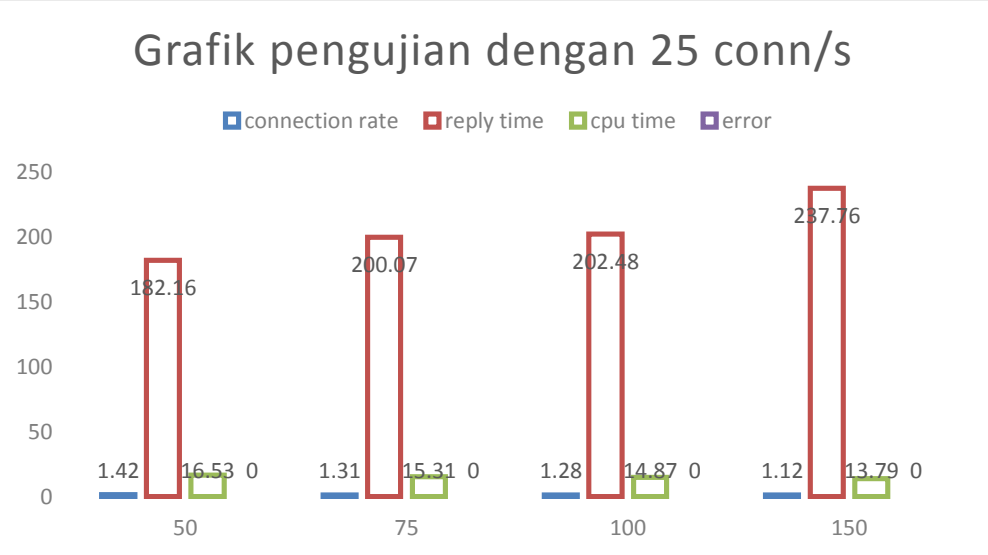
Request	*Connectiong rate* (conn/s)	reply *time*(ms)	cpu *time* (s)	*Error*
50	1,42	182,16	16,53	0
75	1,31	200,07	15,31	0
100	1,28	202,48	14,87	0
150	1,12	237,76	13,79	0

Berdasarkan Tabel 6.1 dapat dijelaskan hasil dari pengujian dengan mengunakan parameter *connection rate, reply time, cpu time* dan *error* sebagai berikut :

- *Connection rate* berdasarkan tabel di atas yaitu dengan *request* 50 didapatkan 1.42 conn/s, *request* 75 didapatkan 1.31 conn/s, *request* 100 didapatkan 1.28 conn/s, dan *request* 150 didapatkan 1.12 conn/s

- *Reply time* dari tabel di atas yaitu pada *request* 50 didapatkan 182.16ms, *request* 75 didapatkan 200,07ms, *request* 100 didaptkan 202,48ms dan *request* 150 didaptkan 237,76ms.
- *Cpu time* yang didapatkan sebesar 16,53 s pada *request* 50, 15,31 s *request* 75, 14,87 s *request* 100 dan *request* 150 didapatkan 13,79 s
- *Error* dari hasil tabel di atas diketahui dari pengujian *request* 50 kali, 75 kali, 100 kali, dan 150 kali didapatkan *error* 0.

Berikut grafik hasil pengujian di atas dengan *rate* 25 conn/s :



Gambar 6.2 Grafik hasil Rate 25 conn/s

Berdasarkan Gambar 6.2 di atas dapat diketahui dari *request* yang dilakukan *connection rate* yang diperoleh menunjukan nilai 1.42 conn/s pada *request* 50 kali dan kemudian mengalami penurunan seiring dengan banyaknya *request* oleh *user.* Semakin banyak *request* yang dilakukan maka *connection rate* yang didapat juga akan semakin kecil dengan nilai rata- rata 1.28 conn/s.

Reply time mengalami peningkatan seiring dengan jumlah *request* yang dilakukan, semakin banyak *request* yang dilakukan *user* maka *reply time* juga semakin tinggi. Nilai rata-rata *reply time* yang didapat pada tabel di atas berdasarkan *request client* ke *server* sebesar 205,61 ms.

Cpu time adalah waktu yang dibutuhkan sistem untuk mengeksekusi semua komunikasi yang terjadi antara *client* dan *server.* Waktu yang dibutuhkan sistem untuk mengeksekusi proses semakin menurun seiring bertambahnya permintaan *client.* Hal ini dikarenakan semakin banyak *request* yang dilakukan maka semakin sering akan terjadi *expired flow* sehingga proses yang dilakukan semakin sedikit karena jumlah *request* semakin berkurang maka waktu yang dibutuhkan juga akan semakin berkurang. Nilai rata-rata *cpu time* berdasarkan tabel di atas adalah 15.12 s sedangkan *error* pada pengujian dengan *rate* 25 conn/s sebesar 0.

6.2.3 Pengujian dengan Rate 50 conn/s

Pada pengujian ini dilakukan dengan memberikan *rate* 50 conn/s dengan *request user* sebanyak 50 kali, 75 kali, 100 kali dan 150 kali. Didapatlah hasil pengujian sebagai berikut :

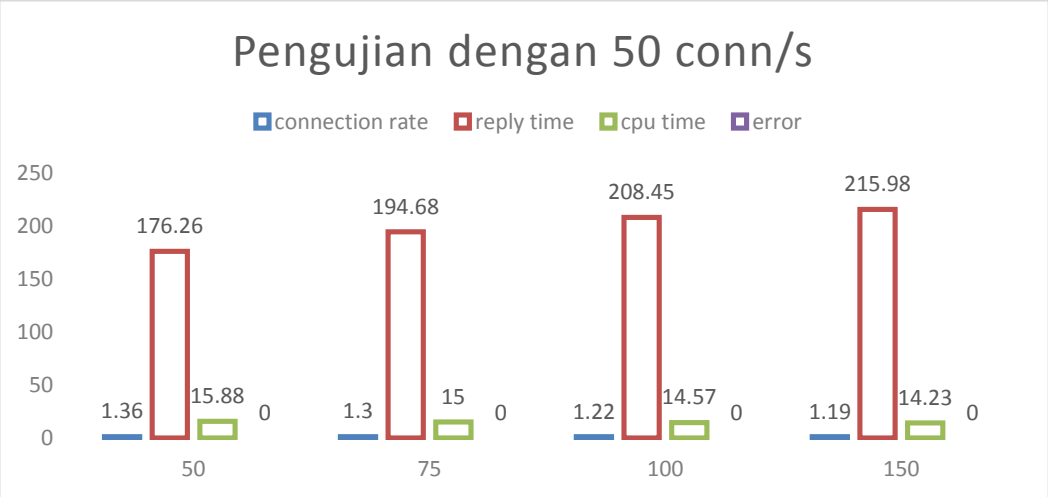
Tabel 6.2 Hasil Pengujian Rate 50 conn/s

Request	*Connectiong rate* (conn/s)	reply *time* (ms)	cpu *time* (s)	*Error*
50	1,36	176,26	15,88	0
75	1,3	194,68	15	0
100	1,22	208,45	14,57	0
150	1,19	215,98	14,23	0

Berdasarkan Tabel 6.2 dapat dijelaskan hasil dari pengujian dengan mengunakan parameter *connection rate, reply time, cpu time* dan *error* sebagai berikut :

- *Connection rate* berdasarkan tabel di atas yaitu dengan *request* 50 didapatkan 1.36 conn/s, *request* 75 didapatkan 1.3 conn/s, *request* 100 didapatkan 1.22 conn/s, dan *request* 150 didapatkan 1.19 conn/s
- *Reply time* dari tabel di atas yaitu pada *request* 50 didapatkan 176,26 ms, *request* 75 didapatkan 194,68 ms, *request* 100 didapatkan 208,45 ms dan *request* 150 didapatkan 215,98 ms.
- *Cpu time* yang didapatkan sebesar 15,88 s pada *request* 50, 15 s *request* 75, 14,57 s *request* 100 dan *request* 150 didapatkan 14,23 s
- *Error* dari hasil tabel di atas diketahui dari pengujian *request* 50 kali, 75 kali, 100 kali, dan 150 kali didapatkan *error* 0.

Berikut grafik hasil pengujian di atas dengan *rate* 50 conn/s :



Gambar 6.3 Grafik hasil Rate 50 conn/s

Berdasarkan Gambar 6.3 di atas dapat diketahui dari *request* yang dilakukan *connection rate* yang di peroleh menunjukan nilai 1.36 conn/s pada *request* 50

kali dan kemudian mengalami penurunan seiring dengan banyaknya *request* oleh *user.* Pada *request* 75 kali *connection rate* menurun menjadi 1.3 conn/s dan semakin banyak *request* yang dilakukan maka *connection rate* yang didapatkan akan semakin kecil.

Sedangkan pada *Reply time* mengalami peningkatan seiring dengan jumlah *request* yang dilakukan, semakin banyak *request* yang dilakukan *user* maka *reply time* juga semakin tinggi. Nilai *reply time* pada *request* 50 kali sebesar 176.26 ms dan naik menjadi 194.68 ms pada *request* 75 kali. Nilai rata- rata *reply time* yang didapat pada tabel di atas berdasarkan *request client* ke *server* sebesar 198.84 ms.

Cpu time dari tabel di atas adalah 15.88 pada *request* 50 kali dan 14.23 s pada *request* 150 kali. Sehingga waktu yang dibutuhkan sistem untuk mengeksekusi proses semakin menurun seiring bertambahnya permintaan *client.* Hal ini dikarenakan semakin banyak *request* yang dilakukan maka semakin sering akan terjadi *expire flow* sehingga proses yang dilakukan semakin sedikit karena jumlah *request* semakin berkurang maka waktu yang dibutuhkan juga akan semakin berkurang. Nilai rata- rata *cpu time* berdasarkan tabel di atas adalah 14.92 s sedangkan *error* pada pengujian dengan *rate* 50 conn/s, tetap sebesar 0.

6.2.4 Pengujian dengan Rate 75 conn/s

Pada pengujian ini dilakukan dengan memberikan *rate* 75 conn/s dengan *request user* sebanyak 50 kali, 75 kali, 100 kali dan 150 kali. Didapatlah hasil pengujian sebagai berikut :

Tabel 6.3 Hasil Pengujian Rate 75 conn/s

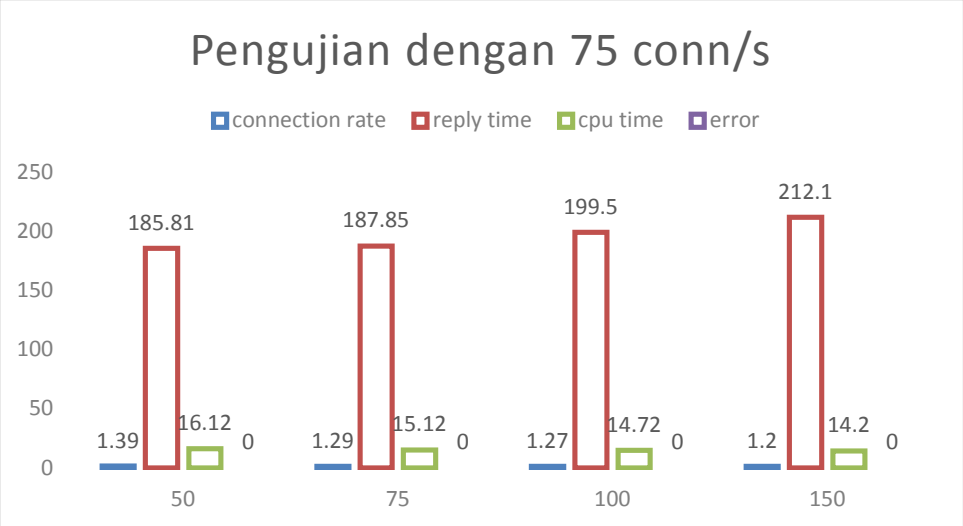
Request	*Connectiong rate* (conn/s)	reply *time* (ms)	cpu *time* (s)	*Error*
50	1,39	185,81	16,12	0
75	1,29	187,85	15,12	0
100	1,27	199,5	14,72	0
150	1,2	212,1	14,2	0

Berdasarkan Tabel 6.3 dapat dijelaskan hasil dari pengujian dengan mengunakan parameter *connection rate, reply time, cpu time* dan *error* sebagai berikut :

- *Connection rate* berdasarkan tabel di atas yaitu dengan *request* 50 didapatkan 1.39 conn/s, *request* 75 didapatkan 1.29 conn/s, *request* 100 didapatkan 1.27 conn/s, dan *request* 150 didapatkan 1.2 conn/s
- *Reply time* dari tabel di atas yaitu pada *request* 50 didapatkan 185.81 ms, *request* 75 didapatkan 187,85 ms, *request* 100 didaptkan 199,5 ms dan *request* 150 didaptkan 212,1 ms.
- *Cpu time* yang didapatkan sebesar 16,12 s pada *request* 50, 15,12 s *request* 75, 14,72 s *request* 100 dan *request* 150 didapatkan 14,2 s

- *Error* dari hasil tabel di atas di ketahui dari pengujian *request* 50 kali, 75 kali, 100 kali, dan 150 kali didapatkan *error* 0.

Berikut grafik hasil pengujian di atas dengan *rate* 75 conn/s :



Gambar 6.4 Grafik hasil Rate 75 conn/s

Berdasarkan Gambar 6.4 di atas dapat diketahui dari *request* yang dilakukan *connection rate* yang diperoleh menunjukan nilai 1.39 conn/s pada *request* 50 kali, 1.29 conn/s *request* ke 75 kali, kemudian turun menjadi 1.27 conn/s dan 1.2 conn/s pada *request* ke 150 kali. *Connection rate* akan mengalami penurunan seiring dengan banyaknya *request* oleh *user.*

Reply time mengalami peningkatan pada *request* 50 kali, sampai *request* 150 kali. Pada *request* 50 kali *reply time* sebesar 185.81 ms semakin banyak *request* yang di lakukan *user* maka *reply time* juga semakin tinggi yakni 212.1 ms pada *request* 150 kali. Nilai rata- rata *reply time* yang di dapat pada tabel di atas berdasarkan *request client* ke *server* sebesar 196.31 ms.

Cpu time dari hasil di atas menunjukan nilai *cpu time* akan turun seiring bertambahnya permintaan *user. Cpu time* adalah waktu yang dibutuhkan sistem untuk mengeksekusi semua komunikasi yang terjadi antara *client* dan *server.* Pada *request* 50 kali *cpu time* bernilai 1.39 s dan terus turun menjadi 14.2 s ketika *request* 150 kali. Nilai rata-rata *cpu time* berdasarkan tabel di atas adalah 15.04 s sedangkan *error* dengan pengujian *rate* 75 conn/s masih sebesar 0 pada semua *request* yang dilakukan.

6.2.5 Pengujian dengan Rate 100 conn/s

Pada pengujian ini dilakukan dengan memberikan *rate* 100 conn/s dengan *request user* sebanyak 50 kali, 75 kali, 100 kali dan 150 kali. Didapatlah hasil pengujian sebagai berikut :

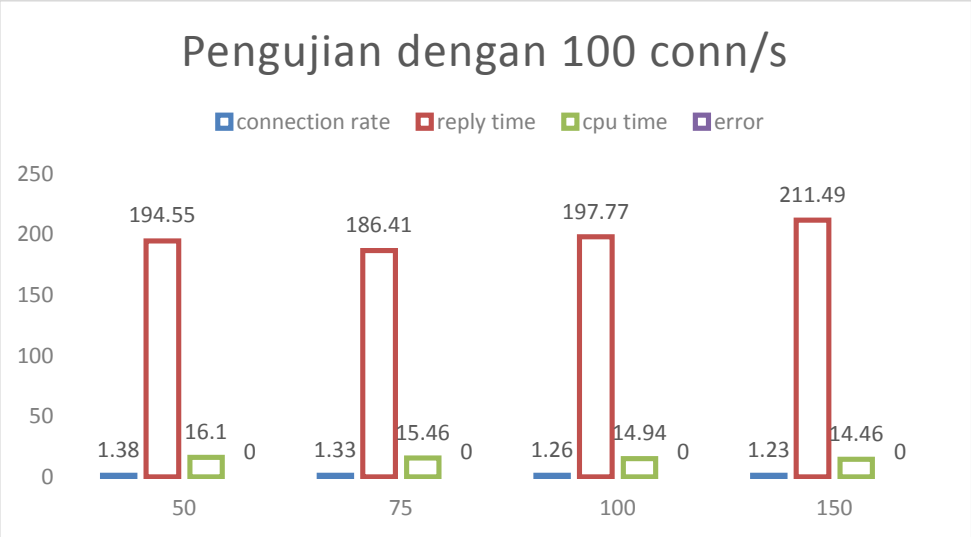
Tabel 6.4 Hasil Pengujian Rate 100 conn/s

Request	*Connectiong rate* (conn/s)	reply *time*(ms)	cpu tie (s)	*Error*
50	1,38	194,55	16,1	0
75	1,33	186,41	15,46	0
100	1,26	197,77	14,94	0
150	1,23	211,49	14,46	0

Berdasarkan Tabel 6.4 dapat dijelaskan hasil dari pengujian dengan mengunakan parameter *connection rate, reply time, cpu time* dan *error* sebagai berikut :

- *Connection rate* berdasarkan tabel di atas yaitu dengan *request* 50 didapatkan 1.38 conn/s, *request* 75 didapatkan 1.33 conn/s, *request* 100 didapatkan 1.26 conn/s, dan *request* 150 didapatkan 1.23 conn/s
- *Reply time* dari tabel di atas yaitu pada *request* 50 didapatkan 194,55 ms, *request* 75 didapatkan 186,41 ms, *request* 100 didaptkan 197,77 ms dan *request* 150 didaptkan 211,49 ms.
- *Cpu time* yang didapatkan sebesar 16,1 s pada *request* 50, 15,46 s *request* 75, 14,94 s *request* 100 dan *request* 150 didapatkan 14,46 s
- *Error* dari hasil tabel di atas di ketahui dari pengujian *request* 50 kali, 75 kali, 100 kali, dan 150 kali didapatkan *error* 0.

Berikut grafik hasil pengujian di atas dengan *rate* 100 conn/s :



Gambar 6.5 Grafik hasil *Rate* 100 conn/s

Berdasarkan Gambar 6.5 di atas dapat diketahui dari *request* yang dilakukan *connection rate* yang di peroleh menunjukan nilai 1.38 conn/s pada *request* 50 kali dan kemudian mengalami penurunan menjadi 1.23 conn/s pada *request* 150 kali. Semakin banyak *request* yang dilakukan maka *connection rate* yang didapat akan semakin kecil dengan nilai rata-rata *connection rate* 1.3 conn/s.

Reply time mengalami penurunan berbeda pada *rate* sebelumnya. Pada *request* 50 kali *reply time* bernilai 194.55 ms dan turun menjadi 186.41 ms pada *request* 75 kali kemudian *request* 100 kali *reply time* kembali naik menjadi 197.77 ms dan 211.49 ms pada *request* 150 kali. Nilai rata-rata *reply time* yang didapatkan pada tabel di atas berdasarkan *request client* ke *server* sebesar 197.55 ms.

Cpu time tetap mengalami penurunan sama seperti *rate* sebelumnya. Bernilai 1.38 s pada *request* 50 kali dan menjadi 1.23 s di *request* 150 kali sehingga waktu yang dibutuhkan sistem untuk mengeksekusi proses semakin menurun seiring bertambahnya permintaan *client.* Nilai rata-rata *cpu time* berdasarkan tabel di atas adalah 15.24 s sedangkan *error* pada pengujian dengan *rate* 100 conn/s sebesar 0. Dari hasil pengujian dengan *rate* 25, 50, 75, dan 100 dapat disimpulkan semakin tinggi *rate* yang digunakan maka nilai *connection rate* juga akan semakin kecil begitu juga dengan *reply time* dan *cpu time.* Kemudian *error* yang didapatkan juga tetap bernilai 0 dengan pengujian dengan *rate* 100 conn/s.

6.2.6 Pengujian 200 dan 400 *Request*

Pada pengujian ini dilakukan untuk mengetahui performa *server* jika *server* diberikan beban melebihi kapasitas beban *server*. *Server* akan diberikan jumlah permintaan sebesar 200 dan 400. Berikut hasil dari pengujian yang dilakukan:

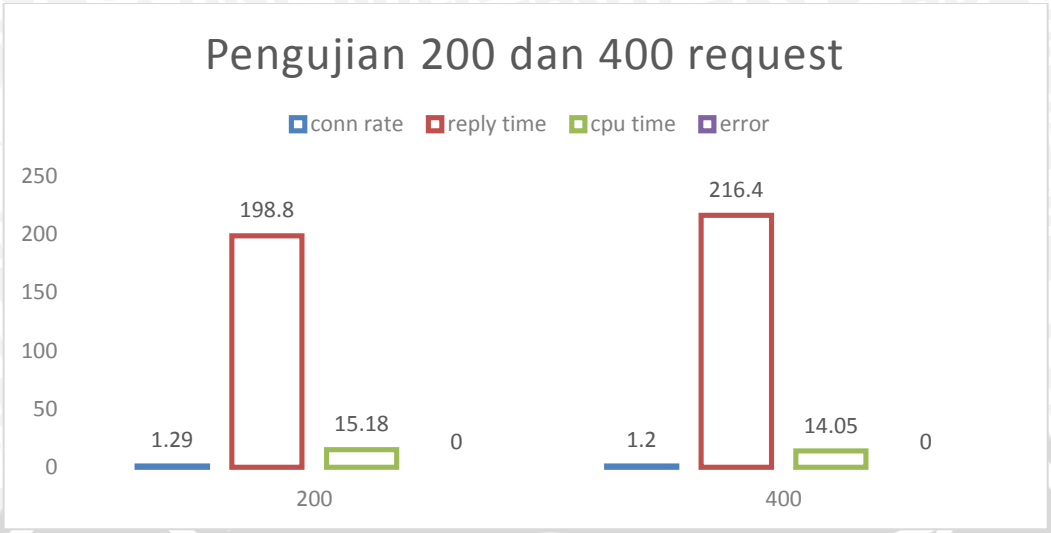
Tabel 6.5 Hasil Pengujian 200 dan 400 *Request*

Request	*Connectiong rate* (conn/s)	reply *time*(ms)	cpu *time* (s)	*Error*
200	1,29	198,8	15,18	0
400	1,2	216,4	14,05	0

Berdasarkan Tabel 6.5 dapat dijelaskan hasil dari pengujian dengan mengunakan parameter *connection rate, reply time, cpu time* dan *error* sebagai berikut :

- *Connection rate* berdasarkan tabel di atas yaitu dengan *request* 200 didapatkan 1.29 conn/s, dan *request* 400 didapatkan 1.2 conn/s
- *Reply time* dari tabel di atas yaitu pada *request* 200 didapatkan 198,8 ms, *request* 400 didaptkan 216,4 ms.
- *Cpu time* yang didapatkan sebesar 15,18 s pada *request* 200, dan *request* 400 didapatkan 14,05 s
- *Error* dari hasil tabel di atas diketahui pada *request* 200 dan *request* 400 tidak ada *error* yang terjadi

Berikut grafik hasil pengujian 200 dan 400 request :



Gambar 6.6 Grafik hasil Pengujian 200 dan 400 *Request*

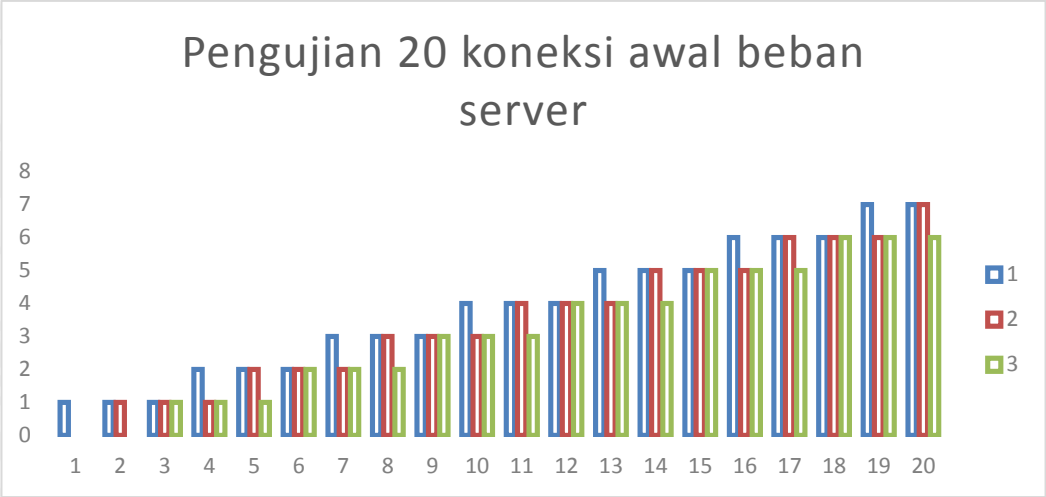
Berdasarkan Gambar 6.6 di atas dapat diketahui dari hasil pengujian dengan 200 *request* dan 400 *request* tidak terjadi *error* pada sistem. Nilai *error* pada sistem *load balancing* tetap bernlai baik yakni menunjukkan nilai 0. Hal ini membuktikan bahwa sistem masih bisa memproses *request* yang dilakukan *client.* Hal ini di karenakan setiap *server* tidak diberikan nilai limit atau batas maksimal koneksi yang mampu diproses setiap *server*.

Pada *connection rate* untuk pengujian pada *request* 200 dan 400 nilai yang diberikan yakni 1.29 conn/s dan 1.2 conn/s lebih rendah dibandingkan nilai *request* 50, 75 ,100 dan 150. Hal ini menunjukkan nilai *request* yang semakin besar maka nilai *connection rate* akan semakin kecil.

Untuk *reply time* pada pengujian dengan jumlah *request* 200 nilai yang diperoleh adalah 198.8 ms sedangkan dengan jumlah *request* 400 bernilai sebesar 216.4 ms. Sehingga semakin besar jumlah *request* yang diberikan maka nilai dari *reply time* juga akan semakin besar. *Cpu time* pada grafik menunjukan nilai yang menurun yakni 15.18 s pada *request* 200 dan 14.05 pada *request* 400. Nilai *cpu time* akan semakin kecil jika jumlah *request* yang diberikan semakin besar.

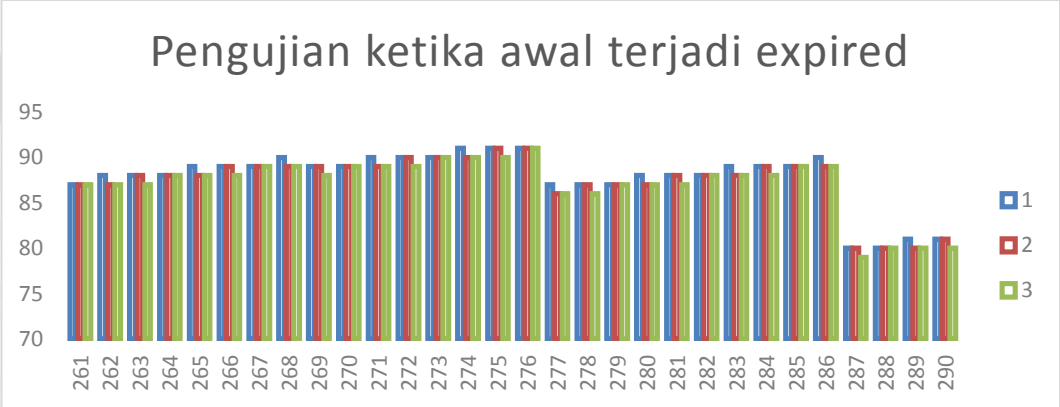
6.2.7 Skenario Pengujian 2

Pada skenario pengujian kedua dilakukan untuk mengetahui beban *server* terhadap *request* yang dilakukan *client* pada *web server.* Melalui skenario ini dapat didapatkan hasil pembagian beban *server* pada setiap *request* yang dilakukan oleh *client.* Pada pengujian ini jumlah *request* yang diberikan ke *web server* sebanyak 50 kali, 75 kali, 100 kali dan 150 kali. Berikut hasil dari pengujian yang dilakukan



Gambar 6.7 Pengujian 20 koneksi awal beban *server*

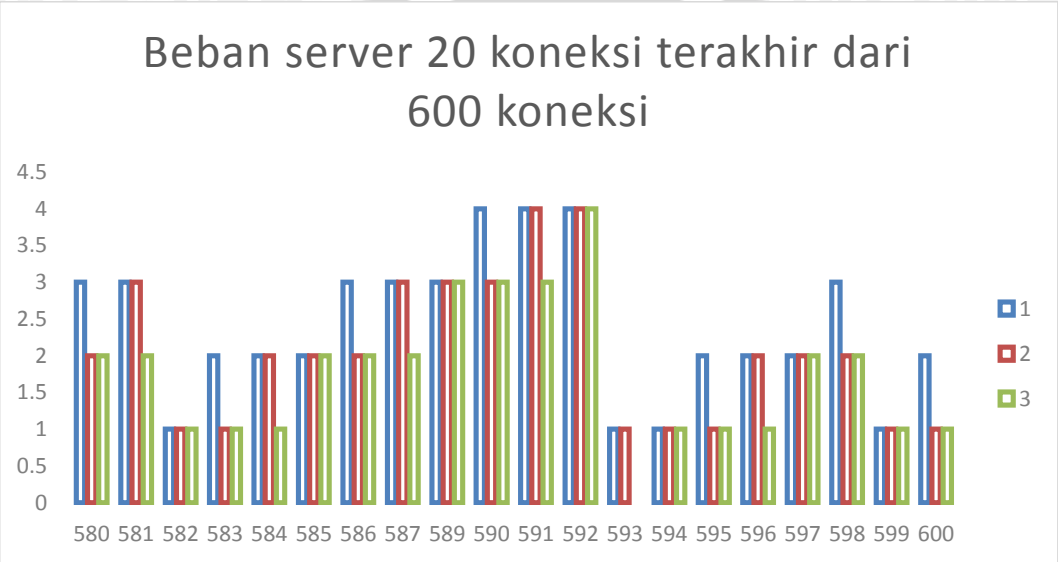
Pada Gambar 6.7 merupakan hasil pengujian 20 koneksi awal yang dilakukan oleh *client.* Dari hasil pengujian di atas dapat dijelaskan pada koneksi 1 sampai dengan 20 pembagian beban *server* dilakukan dengan cara bergantian pada *server* yang sedang aktif mulai dari *server* 1 sampai dengan *server* 3. Pada koneksi yang 1 beban *server* akan diberikan kepada *server* yang 1 kemudian koneksi yang ke 2 beban *server* akan diberikan kepada *server* yang ke 2 dan selanjutnya koneksi yang 3 beban koneksi akan diberikan kepada *server* yang ke 3. Ketika beban *server* 1 sampai dengan *server* 3 memiliki nilai beban yang sama koneksi berikutnya yang masuk yakni koneksi ke 4 akan kembali diberikan kepada *server 1.* Pembagian *server* akan terus bergantian sampai terjadi *expired flow* pada salah satu *server* yang sedang aktif.



Gambar 6.8 Pengujian ketika awal terjadi *Expired*

Pada Gambar 6.8 merupakan hasil pengujian ketika awal terjadi *Expired*. Dari hasil dipengujian di atas dapat dilihat ketika koneksi ke-261 sampai dengan koneksi ke-268 pembagian beban *server* dilakukan bergantian dari *server 1* sampai dengan *server 3,* beban *server* 1 adalah 90 koneksi beban *server* 2 adalah 89 koneksi dan beban *server* 3 adalah 89 koneksi. Ketika pada koneksi ke-269 terjadi *Expired flow* beban *server* menjadi 89 koneksi untuk *server 1, 89 koneksi* untuk *server* 2 dan 88 koneksi untuk *server* 3. Pada hal ini algoritma *least*

connection berperan dalam pemilihan *server. Server* yang memiliki jumlah koneksi paling sedikit akan diberikan beban koneksi selanjutnya yang akan masuk. Sehingga pada koneksi ke-270 beban koneksi diberikan pada *server* ke 3 karena memiliki beban paling sedikit. Pada koneksi ke-271 pembagian beban *server* kembali ke *server* yang 1 karena nilai beban dari *server* 1 sampai dengan *server* 3 adalah sama yakni 89 koneksi. Begitu pula pada koneksi ke-277 dan koneksi ke-287 ketika terjadi *Expired* makan beban koneksi selanjutnya yang masuk akan diberikan pada *server* yang memiliki jumlah koneksi paling sedikit.



Gambar 6.9 Baban *server* 20 koneksi terakhir dari 600 koneksi

Pada Gambar 6.9 merupakan hasil pengujian beban *server* 20 koneksi terakhir dari 600 koneksi. Pada koneksi ke-580 sampai konkesi ke-600 sering terjadi *Expired flow* karena pada koneksi di atas koneksi ke-270 akan terjadi pengecekan *Expired flow* setiap 5 detik sekali. Semakin banyak jumlah *request* yang diberikan oleh *client* maka semakin banyak *Expired* yang terjadi dan beban *server* akan semakin sedikit. Ketika semua *request* yang dilakukan oleh *client* selesai diproses semuanya maka beban *server* akan kembali ke nilai awal yakni 0.

BAB 7 PENUTUP

Berdasarkan metodologi penelitian yang sudah disusun sebelumnya, bab ini merupakan tahap akhir dalam melakukan penelitian setelah melakukan pengujian dan analisis hasil pengujian dari sistem yang sudah dirancang. Berikut merupakan kesimpulan dan saran yang dapat ditarik dari penelitian yang telah dilakukan.

7.1 Kesimpulan

1. Penerapan Algoritma *least connection* terhadap pengujian *load balancing* pada *software defined network* dapat berjalan dengan baik dan membantu meringankan beban *server* dalam memberikan perlayan terhadap permintaan yang dilakukan oleh *user*
2. Pada sistem *load balancing* dengan algoritma *least connection* mampu memberikan performa yang baik terhadap *server*. Berdasarkan pengujian yang telah dilakukan algoritma *least connection* mampu membagi *traffic* jaringan dengan memberikan beban pada *server* yang memiliki jumlah permintaan *user* paling sedikit.
3. Pada skenario pengujian didapatkan nilai *expired flow* yang berbeda di setiap pengujian. Pada pengujian 50 kali tidak terjadi *expire flow,* pengujian 75 kali *expire flow* terjadi pada koneksi ke-266, pengujian 100 kali *expire flow* pada koneksi ke-261, dan pada pengujian 150 kali *expire flow* pada koneksi ke-268.
4. Semakin banyak jumlah *request* yang dilakukan maka nilai dari *connection rate* juga semakin kecil. Nilai rata- rata *connection rate* dari 4 kali pengujian 1.28 conns/s.
5. Nilai *reply time* akan semakin tinggi seiring jumlah *request* yang dilakukan oleh *user* juga semakin tinggi. Rata-rata nilai *reply time* dari semua pengujian yang dilakukan sebesar 199.58 ms.
6. Nilai *cpu time* juga akan berkurang ketika jumlah *request* yang dilakukan oleh *user* juga semakin tinggi. Dari haril pengujian didapatkan nilai rata-rata *cpu time* 15.08 s.
7. Pada pengujian yang dilakukan tidak terjadi *error* baik pada *rate* 25, 50, 75 maupun 100. Hal ini dikarenakan sistem tidak mengalami over*load* karena permintaan yang sangat banyak dari *user*.
8. Pada pengujian dengan jumlah *request* 200 dan 400 sistem tidak mengalami *error*. Hal ini berarti sistem *load balancing* bekerja dengan baik walaupun *request* yang diberikan melebihi kapasitas sistem.

7.2 Saran

Ada beberapa saran untuk para peneliti yang ingin melakukan pengembangan pada penelitian ini antara lain:

1. Melakukan pengujian terhadap sistem *load balancing* dengan jumlah *request* yang lebih banyak
2. Melakukan implementasi dengan beban *server* pada sistem *load balancing* dapat diatur jumlah maksimal masing-masing *server*.
3. Melakukan implementasi dengan spesifikasi *server* yang berbeda.
4. Melakukan pengujian dengan parameter pengujian yang lebih banyak lagi. Tidak hanya *connection rate, cpu time, reply time* dan *error.*

DAFTAR PUSTAKA

Cui Chen-Xiao and Xu Ya-bin. 2016. *Research on Load Balance Method in SDN.* Internasional Journal of Grid Distributed Computing, Volume 9. No. 1 pp.25-36

Dependra, Dhakal., Bishal, Pradhan., Sunil, Dhimal., 2016. Campus Network Using Software defined network*. International Journal Of Computer And Information Technology*, Volume 138 - no 4

Kurniawan, Yogi., 2013. *Analisis Kinerja Algoritma Load Balancer dan Implementasi pada Layana Web.* Malang: Universitas Brawijaya

Lara, A., Kolasani, A., Ramamurthy, B., 2013. Network Innovation Using Open*Flow*: A Survey.*IEEE Commun. Surv. Tutor. 16*, p.1–20

Mukundha., Dr. Chinthagunta., P. Gayatri., Dr.I.Surya, Prabha., 2016. *Load Balance Scheduling Algorithm for Serving of Requests inCloud Network Using Software defined networks.* International Journal of Applied Engineering Research. (ISSN 0973-4562)

Mustafa E. M., Amin Mubark I., 2015. Load Balancing Algorithms Round-Robin, Least-Connection And Least Loaded Efficiency. *International Journal Of Computer And Information Technology,* Volume 04-Issue 02

POX, 2016. Pox controller Tutorial, Tersedia di : < http://sdnhub.org/tutorials/pox/> [Diakses 21 juni 2016]

Qingwei Du and Huaidong Zhuang. 2015. OpenFlow-Based Dynamic Server Cluster Load Balancing with Measurement Support*. Journal Of Communication*, No. 8

Senthil, Ranjani S., 2015. *Load balancing Using Software Defined Networks*. International conference on current trends in advances computing. ICCTAC.SRM University, Chennai

Sukhveer Kaur., Japinder S., Navtej Singth G., 2014. Network Programmability Using Pox Controller. *International Conference on communication, computing and sistem.*

Yuanhao, Zhou.; Li, Ruan.; Rui, Liu. 2014. A Method for *Loadbalancing* based on Software Defined Network, p. *45*

LAMPIRAN DATA HASIL PENGUJIAN SISTEM

1. *Rate* 25 conns/S

Nomor	Connection rate (conns/s)				Reply time (ms)				Cpu (%)				Error			
	50	75	100	150	50	75	100	150	50	75	100	150	50	75	100	150
1	1,4	1,425	1,3	1,225	206,025	207,75	203,65	229,775	16,1	15,925	14,975	14,4	0	0	0	0
2	1,425	1,325	1,3	1,225	180,5	199,875	195,55	209,5	16,475	15,3	15,05	14,05	0	0	0	0
3	1,5	1,325	1,275	1,175	164,35	207,75	199,375	215,475	17,6	15,6	14,7	14,225	0	0	0	0
4	1,425	1,35	1,275	1,075	183,35	163	212,175	247,075	16,625	15,175	14,825	13,7	0	0	0	0
5	1,45	1,25	1,225	1,025	170,575	236,9	215,5	273,65	16,225	14,925	14,625	13,275	0	0	0	0
6	1,325	1,275	1,3	1,1	216,2	208,7	182,575	238,8	16,275	15,05	14,65	13,275	0	0	0	0
7	1,4	1,35	1,275	1,1	195,7	185,775	182,675	237,8	16,625	15,35	14,775	13,625	0	0	0	0
8	1,425	1,275	1,275	1,1	174,925	200,025	237,65	239,175	16,575	14,9	15,275	13,775	0	0	0	0
9	1,45	1,3	1,225	1,1	184,975	195,025	204,3	258,5	17,2	15,625	14,25	13,375	0	0	0	0
10	1,425	1,3	1,35	1,125	145	195,925	191,4	227,925	15,6	15,25	15,65	14,2	0	0	0	0
rata- rata	1,4225	1,3175	1,28	1,125	182,16	200,0725	202,485	237,7675	16,53	15,31	14,8775	13,79	0	0	0	0

2. *Rate* 50 conns/s

Nomor	Connection rate (conns/s)				Reply time (ms)				Cpu (%)				Error			
	50	75	100	150	50	75	100	150	50	75	100	150	50	75	100	150
1	1,35	1,375	1,3	1,15	156,1	158,725	178,75	196,75	15,825	14,875	14,375	14,025	0	0	0	0
2	1,375	1,3	1,2	1,125	163,7	183,7	227,5	242,85	16,175	15,225	14,85	13,85	0	0	0	0
3	1,35	1,3	1,175	1,2	184,575	206,55	229,975	205,45	16	15,2	14,425	14,175	0	0	0	0
4	1,4	1,25	1,225	1,2	184,9	203,65	227,9	218,9	16,175	14,85	14,325	14,175	0	0	0	0
5	1,3	1,3	1,225	1,2	192,55	188,25	213,975	213,85	15,825	15,05	14,25	14,125	0	0	0	0
6	1,4	1,375	1,225	1,25	163,35	186	195,875	208,325	15,075	15,4	14,15	14,7	0	0	0	0
7	1,35	1,3	1,2	1,2	170,925	207,6	212,9	215,85	16,1	14,675	14,4	14,325	0	0	0	0
8	1,375	1,25	1,25	1,225	205,05	178,15	186,6	216,475	16,475	14,375	15,2	14,275	0	0	0	0
9	1,4	1,225	1,225	1,2	149,975	233,7	219,725	229,925	15,25	14,975	14,75	14,275	0	0	0	0
10	1,325	1,325	1,25	1,225	191,5	200,525	191,375	211,425	15,925	15,45	14,975	14,4	0	0	0	0
rata- rata	1,3625	1,3	1,2275	1,1975	176,2625	194,685	208,4575	215,98	15,8825	15,0075	14,57	14,2325	0	0	0	0

3. *Rate* 75 conns/s

Nomor	Connection rate (conns/s)				Reply time (ms)				Cpu (%)				Error			
	50	75	100	150	50	75	100	150	50	75	100	150	50	75	100	150
1	1,325	1,275	1,225	1,2	213,125	214,8	206,7	219,05	15,725	14,575	14,5	14,275	0	0	0	0
2	1,375	1,3	1,225	1,2	203	185,8	217,8	201,85	15,55	15,475	14,725	14,05	0	0	0	0
3	1,45	1,2	1,25	1,2	184,825	220,075	192,725	196,825	16,425	14,625	14,325	14,15	0	0	0	0
4	1,35	1,25	1,275	1,225	219,6	163,475	203,725	212,625	16,15	14,725	14,225	14,575	0	0	0	0
5	1,3	1,45	1,325	1,2	194,15	166,425	180,7	239,475	15,35	16,275	15,25	14,25	0	0	0	0
6	1,3	1,275	1,275	1,2	186,975	208,175	206,6	212,225	16,025	14,925	14,575	14,1	0	0	0	0
7	1,4	1,325	1,275	1,2	180,95	162	208	194,425	15,925	15,15	14,875	14,125	0	0	0	0
8	1,5	1,3	1,275	1,2	159,725	177,625	204,975	207,125	16,875	14,85	14,675	14,425	0	0	0	0
9	1,475	1,3	1,325	1,2	168,625	184,475	185,05	221,475	16,7	15,4	14,975	14,45	0	0	0	0
10	1,425	1,3	1,275	1,2	147,15	195,65	188,75	215,925	16,55	15,2	15,1	13,625	0	0	0	0
rata- rata	1,39	1,2975	1,2725	1,2025	185,8125	187,85	199,5025	212,1	16,1275	15,12	14,7225	14,2025	0	0	0	0

4. *Rate* 100 conns/s

Nomor	Connection rate (conns/s)				Reply time (ms)				Cpu (%)				Error			
	50	75	100	150	50	75	100	150	50	75	100	150	50	75	100	150
1	1,325	1,325	1,225	1,225	223,675	184,675	197	202,5	15,35	15,25	14,6	14,325	0	0	0	0
2	1,3	1,375	1,275	1,2	240,8	185,325	189,375	207,925	16,175	16,15	14,775	14,475	0	0	0	0
3	1,425	1,325	1,3	1,325	175,175	195,15	196,9	195,475	16,3	15,475	15	15,3	0	0	0	0
4	1,375	1,425	1,2	1,225	209,65	169,825	211,425	212,325	16,725	16,5	14,475	14,7	0	0	0	0
5	1,325	1,4	1,225	1,2	175,2	182,1	212,675	216,375	15,05	16,625	14,425	14,125	0	0	0	0
6	1,45	1,325	1,375	1,25	142,65	164,625	176,825	207,4	16,3	14,825	16,15	14,525	0	0	0	0
7	1,4	1,25	1,25	1,2	223,675	213,15	189,875	217,55	16,45	14,5	14,925	13,975	0	0	0	0
8	1,3	1,375	1,225	1,25	226,225	168,925	228,825	216,1	16,65	15,175	14,175	14,1	0	0	0	0
9	1,45	1,275	1,425	1,3	168,625	195	162,6	213,7	16,3	14,975	16,65	14,8	0	0	0	0
10	1,45	1,275	1,15	1,2	159,825	205,375	212,2	225,625	16,425	15,15	14,275	14,35	0	0	0	0
rata- rata	1,38	1,335	1,265	1,2375	194,55	186,415	197,77	211,4975	16,1725	15,4625	14,945	14,4675	0	0	0	0