Jurnal Pengembangan Teknologi Informasi dan Ilmu Komputer e-ISSN: 2548-964X
Vol. 2, No. 5, Mei 2018, hlm. 2135-2143 http://j-ptiik.ub.ac.id

Implementasi Load Balancer Berdasarkan Server Status pada *Arsitektur Software Defined Network* (SDN)

Lalu Fani Islahul Ardy[1], Adhitya Bhawiyuga[2], Widhi Yahya[3]

Program Studi Teknik Informatika, Fakultas Ilmu Komputer, Universitas Brawijaya
Email: [1]faniislahul@gmail.com, [2]bhawiyuga@ub.ac.id, [3]widhi.yahya@ub.ac.id

Abstrak

Salah satu permasalahan yang terdapat di dalam jaringan adalah masalah *load balancing*. Arsitektur Software Defined Network (SDN) yang terus berkembang diharapkan menjadi paradigma baru dalam menjawab permasalahan terkait *load balancing* pada jaringan tradisional. Fleksibilitas yang didapatkan dari pemisahan antara *control plane* dan *data plane* pada arsitektur SDN memungkinkan untuk mengembangkan berbagai teknologi yang tidak dapat dilakukan pada jaringan tradisional. Pada penelitian ini dilakukan pengembangan algoritma *load balancing* yang berjalan pada arsitektur SDN. Algoritma tersebut diimplementasikan pada sebuah SDN *controller* yang terhubung dengan sebuah SDN *switch*. SDN *controller* akan mengirimkan pesan kepada setiap *server* berdasarkan interval waktu yang ditentukan untuk mendapatkan nilai CPU dan *memory utilization* serta *response time* dari masing-masing *server*. Parameter tersebut digunakan untuk menentukan *load* dari masing-masing *server*. SDN *switch* kemudian akan membagi *request* yang berasal dari *client* menuju *server* dengan *load* paling kecil. Algoritma *load balancing* tersebut kemudian akan diibandingkan performanya dengan algoritma sejenis pada *server cluster* dengan spesifikasi yang sama (homogen) dan *server cluster* dengan spesifikasi berbeda (heterogen). Hasilnya ditemukan bahwa algoritma yang dikembangkan mampu untuk menditribusikan request yang berasal dari *client* menuju *server* berdasarkan status dari *server* tersebut serta memiliki performa yang lebih baik dari algoritma sejenis pada *server* sengan spesifikasi yang berbeda (heterogen).

Kata kunci : *Software Defined Network* (SDN), *Load Balancer*, *Client-Server*, Ryu NOS, Open vSwitch

Abstract

One of the problems in computer network is load balancing. The emergence of Software Defined Network (SDN) architecture is expected to be a new paradigm in addressing issues related to load balancing on traditional networks. The flexibility gained from separation between control plane and data plane on the SDN architecture makes it feasible to develop technologies that hard to implement on traditional networks. In this research, a load balancing algorithm is developed based on the SDN architechture. An SDN controller that is connected to an SDN switch will sends messages to each server based on the specified time interval to get the CPU and memory utilization and response time from each server. Those parameters are used to determine the load of each server. The SDN switch will then split request from client to server with smallest load. Load balancing algorithm's performance will then be compared to similar algorithms on server clusters with same specification (homogeneous) and server clusters with different specifications (heterogeneous). The result found that the proposed algorithm is able to distribute requests from clients to servers based on status of the server and perform better than similar algorithms on server clusters with different spesifications (heterogeneous).

***Keywords** : Software Defined Network (SDN), Load Balancer, Client-Server, Ryu NOS, Open vSwitch*

1. PENDAHULUAN

Salah satu permasalahan yang terdapat di dalam jaringan adalah masalah *load balancing*. Teknologi *load balancing* pertama kali diterapkan menggunakan Domain Name Server (DNS) yang memungkinkan beberapa *client* yang berbeda untuk mengakses *server* yang berbeda agar dapat menyeimbangkan *load* dari *server* (Shang et al. 2013). Teknologi *load balancing* kemudian berkembang menjadi

berbagai macam algoritma. Beberapa algoritma yang biasa diterapkan untuk melakukan *load balancing* antara lain Random (R), Round-Robin (RR), Weighted Round Robin (WRR), Least Connection (LC), Least Loaded Server (LLS), dll (Mahmood & Rashid 2011). Sebuah jaringan yang memiliki distribusi *load* yang merata akan membantu optimasi terhadap *resource* yang tersedia untuk dapat memaksimalkan *throughput*, meminimalisasi *response time*, dan mencegah terjadinya *overload* pada jaringan (Zha et al. 2010). Jadi dapat dikatakan bahwa *load balancer* memiliki fungsi utama untuk mencegah *congestion* serta memangkas *delay* yang tidak diperlukan (Boero et al. 2016). Namun implementasi dari algoritma *load balancing* pada *load balancer* tradisional membutuhkan biaya yang mahal serta kompleksitas yang tinggi disamping keterbatasan *hardware* yang dapat menjadi *bottleneck* di dalam sistem (Zhong et al. 2017). Berbagai permasalahan tersebut membatasi teknologi *load balancing* untuk dapat diterapkan pada *load balancer* yang efisien, fleksibel, dan memiliki skalabilitas yang tinggi.

Arsitektur Software Defined Network (SDN) yang semakin berkembang akhir-akhir ini memberikan sebuah paradigma baru dalam pengembangan algoritma *load balancing.* Konsep dasar dari SDN yang memisahkan antara *control plane* yang bertugas untuk menentukan bagaimana sebuah paket akan diteruskan dengan *data plane* yang bertugas meneruskan paket (Al-Najjar et al. 2016) memungkinkan untuk melakukan sentralisasi terhadap *control plane*. Hal tersebut menyebabkan SDN memiliki kemampuan untuk dapat diprogram ulang, mudah untuk dikelola, serta terbuka dengan inovasi terbaru (Boero et al. 2016). Arsitektur SDN sendiri dapat terdiri dari dua atau lebih entitas yaitu perangkat yang mendukung SDN (*switch* atau *router*) dan *central controller* (SDN controller). SDN *switch* akan memproses dan meneruskan paket berdasarkan aturan yang tersimpan pada *flow table* (*forwarding table*), sedangkan SDN *controller* akan memproses dan mengatur *forwarding table* dari setiap SDN *switch* menggunakan protokol standar yang dinamakan OpenFlow (OF) (McKeown et al. 2008). SDN *controller* juga bertanggung jawab untuk membuat *virtual topology* yang merepresentasikan *physical topology*. *Virtual topology* tersebut kemudian akan digunakan oleh modul aplikasi yang berjalan pada SDN *controller* untuk menerapkan berbagai macam *control logic* dan *network functions* (*routing, traffic enginering, QoS, load balancing*, dll.) (Boero et al. 2016). Dengan berbagai keutungan yang ditawarkan tersebut, SDN diharap dapat menjadi sebuah paradigma baru untuk memecahkan permasalahan yang terdapat pada pengembangan algoritma *load balancing* dan implementasinya pada *load balancer*.

Banyak penelitian yang telah dilakukan untuk mengembangkan metode *load balancing* pada arsitektur SDN. Penelitian yang dilakukan S.Kaur et al., mencoba mengembangkan algoritma *load balancing* Round Robin pada arsitektur SDN. *Load balancer* kemudian diimplementasikan pada *simulator* Mininet. Hasilnya didapatkan *load balancer* dapat bekerja pada arsitektur SDN dan memiliki performa yang lebih baik daripada algoritma Random (Kaur et al. 2015). Penelitian yang dilakukan K. Kaur dan S. Kaur mencoba mengembangkan *load balancer* sederhana dengan mengambil jumlah *flow* terendah pada *flow entry* yang telah dibuat pada *switch* untuk menghubungkan *client* dengan *server* yang dituju. *Request* dari *client* kemudian akan diarahkan kepada *server* dengan jumlah *flow connection* terendah. Hasilnya ditemukan metode yang dilakukan menghasilkan *throughput*, *response time*, dan *availability* yang lebih baik dari metode Round Robin (Kaur & Kaur 2016). Penelitian yang dilakukan Zhong et al., mencoba menerapkan metode *load balancing* yang efisien pada SDN. SDN *controller* akan mengambil sampel *server response time* dari *server* dengan mengirimkan pesan *paket_out* kepada *server*. Hasilnya akan dibandingkan kemudian *flow table* akan dibuat dan *client* akan diarahkan pada *server* dengan *response time* paling kecil (Zhong et al. 2017). Dikarenakan metode tersebut hanya mempertimbangakan *server response time* dari *server*, hasil yang diperoleh dapat dipengaruhi faktor lain seperti *delay* yang terdapat antara *controller* dan *server*. Disamping itu metode tersebut juga tidak dapat diterapkan pada *server cluster* yang bersifat heterogen. Peneliti juga tidak mempertimbangan penggunaan interval untuk mengambil sampel dari masing-masing server.

Untuk menjawab berbagai permasalahan tersebut penulis mengusulkan sebuah metode *load balancing* yang diimplementasikan pada *load balancer* yang menggunakan arsitektur SDN. Penulis menambahkan parameter CPU dan *memory utilization* dari *server* bersamaan dengan *server response time* untuk menentukan

pemilihan terhadap *server* pada *load balancer*. CPU dan *memory* ditambahkan karena jumlah *client* dan lama waktu dari *client* mengakses sebuah *server* dapat mempengaruhi penggunaan CPU dan *memory* pada *server* (Gandhi et al. 2002). Ketiga parameter tersebut kemudian akan dimasukkan kedalam sebuah variabel untuk kemudian dibandingkan dengan hasil dari server lain. Penulis juga akan melakukan analisis terhadap performa dari metode yang diusulkan dengan mengunakan parameter yang telah disebutkan beserta dampaknya terhadap distribusi *request* pada masing-masing *server*. Hasilnya kemudian dibandingakan dengan metode Round Robin dan LBBSRT yang diusulkan peneliti sebelumnya. Metode yang dikembangan ini diharapkan mampu untuk mendistribusikan *load* secara berimbang berdasarkan *resource* yang tersedia pada setiap *server* yang terdapat pada *cluster*.

2. KAJIAN KEPUSTAKAAN

Pada penelitian yang telah dilakukan sebelumnya untuk mengembangan metode *load balancing* baik pada SDN maupun pada jaringan tradisional beberapa peneliti telah mengusulkan beberapa penelitian. Penelitian-penelitian tersebut kemudian menjadi dasar dalam penyusunan penelitian ini. Xu dan Wang pada penelitiannya yang berjudul "A modified round-robin load-balancing algorithm for cluster-based web servers" mengusulkan untuk melakukan modifikasi terhadap algoritma *load balancing* Round-Robin. Modified Round-Robin (MRR) melakukan pendistribusian terhadap permintaan *client* dengan mempertimbangkan *load* yang akan dibebankan kepada *web server*. Hasilnya algoritma tersebut dapat mendistribusikan *load* secara lebih merata kepada seluruh *server* yang berada di dalam *cluster* (Xu & Wang 2014). Namun implementasi dari penelitian tersebut sangat sulit dilakukan pada *dispacther-based load balancer*, disamping karena akan memberatkan sistem, peneliti juga tidak menjelaskan bagaimana cara sistematis untuk menentukan *load* pada *server*. Konsep penggunan parameter berupa *server load* merupakan ide yang menarik dan terbukti secara teori (dan simulasi) dapat membagi *load* secara lebih merata.

Penelitian awal yang mengembangkan *load balancer* pada arsitektur SDN telah dilakukan oleh beberapa peneliti. Penelitian yang dilakukan S.Kaur et al., dengan judul "Round-Robin Based Load Balancing in Software" mencoba mengembangkan algoritma *load balancing* Round Robin pada arsitektur SDN. *Load balancer* kemudian diimplementasikan pada *simulator* Mininet. Hasilnya didapatkan *load balancer* dapat bekerja pada arsitektur SDN dan memiliki performa yang lebih baik daripada algoritma Random (Kaur et al. 2015). Penelitian yang dilakukan K. Kaur dan S. Kaur dengan judul "Flow Statistics Based Load Balancing in OpenFlow" mencoba mengembangkan *load balancer* sederhana dengan mengambil jumlah *flow* terendah pada *flow entry* yang telah dibuat pada *switch* untuk menghubungkan client dengan *server* yang dituju. *Request* dari *client* kemudian akan diarahkan kepada *server* dengan jumlah *flow connection* terendah. Hasilnya ditemukan metode yang dilakukan menghasilkan *throughput*, *response time*, dan *availability* yang lebih baik dari metode Round Robin (Kaur & Kaur 2016). Kedua penelitian tersebut merupakan penelitian awal yang mengembangkan *load balancer* sederhana tanpa memperhatikan kondisi keseluruhan dari *server*.

Penelitian yang dilakukan Zhong et al., dengan judul "LBBSRT: An Efficient SDN load balancing scheme based on server response time" mencoba menerapkan metode *load balancing* yang efisien pada SDN dengan menggunakan parameter yang didapatkan dari *server*. SDN controller akan mengambil sampel *server response time* dari *server* dengan mengirimkan pesan *paket_out* kepada *server*. Hasilnya akan dibandingkan kemudian *flow table* akan dibuat dan *client* akan diarahkan pada *server* dengan *response time* paling kecil. Peneliti berasumsi jika *server* di dalam *cluster* memiliki performa dan spesifikasi yang sama serta menawarkan layanan yang sama maka semakin besar *load* dari *server*, semakin besar pula *response time* dari *server* tersebut (Zhong et al. 2017). Dikarenakan metode tersebut hanya mempertimbangakan *server response time* dari *server*, hasil yang diperoleh dapat dipengaruhi faktor lain seperti *delay* yang terdapat antara *controller* dan *server*. Disamping itu metode tersebut juga tidak dapat diterapkan pada *server cluster* yang bersifat heterogen. Peneliti juga tidak mempertimbangan pemilihan interval untuk melakukan pemilihan server. Hal ini menciptakan kemungkinan untuk mengembangkan metode yang akan digunakan oleh peneliti.

3. SOFTWARE DEFINED NETWORK

Software Defined Network (SDN) adalah sebuah paradigma baru dalam pengontrolan dan manajemen jaringan komputer. Konsep dasar dari SDN adalah pemisahan antara control plane yang bertugas untuk menentukan bagaimana sebuah paket akan diteruskan dengan data plane yang bertugas meneruskan paket (Al-Najjar et al. 2016). Menurut Nadeu & Gray dalam bukunya "SDN: Software Defined Network", SDN adalah pendekatan arsitektural yang memberikan optimasi serta menyederhanakan operasi di dalam jaringan dengan lebih mendekatkan interaksi antara aplikasi, network service, serta network device pada jaringan asli maupun virtual. Hal tersebut dapat dicapai dengan mengimplementasikan sebuah logical *server* yang tersentralisasi (dalam hal ini bisa disebut SDN controller) yang mengatur, memediasi, dan memfasilitasi aplikasi untuk berkomunikasi dengan perangkat jaringan, dan perangkat jaringan yang ingin mengirimkan informasi menuju aplikasi (Nadeau & Gray 2013). Komunikasi antar aplikasi dilakukan dengan menggunakan protokol standard yang disebut OpenFlow (OF) (McKeown et al. 2008). Arsitektur SDN tergolong baru sehingga memberikan ruang yang luas untuk penelitian dan pengembangan. Secara umum bagian dari SDN dapat dikategorikan menjadi 3 bagian, yaitu *control plane*, *data plane*, dan *application plane*, serta protokol yang mengatur komunikasi antara *controller* dengan *data plane* yaitu OpenFlow.

4. LOAD BALANCER

Teknologi *load balancing* pertama kali diterapkan pada DNS. *Load balancer* tersebut menggunakan satu nama yang dapat diakses oleh *client* kemudian meneruskan *request* dari *client* menuju beberapa IP dari *server*. Hal ini memungkinkan beberapa *client* yang berbeda untuk mengakses *server* yang berbeda agar dapat menyeimbangkan *load* dari *server* (Shang et al. 2013). Teknologi *load balancing* tersebut kemudian berkembang menggunakan berbagai macam metode. Load balancing dapat dibagi menjadi empat metode dasar, *DNS-based, client-based, server-based, dan dispatcher-based* (Cardellini et al. 1999).

Beberapa penelitan juga dilakukan untuk mendapatkan aturan yang lebih dinamis. Pao dan Chen mengusulkan *load balancing* berdasarkan *resource* yang tersisa dari *server*. *Load balancing* tersebut memungkinkan pembagian *load* pada *server* yang *heterogen* (Pao & Chen 2006). Tiwari & Kanugo mengusulkan algoritma *load balancing* berdasarkan pengklasifikasian *request* dari *client* ke dalam beberapa kategori berdasarkan beban dari setiap *request*. *Request* tersebut kemudian akan diteruskan kepada *server* dengan load terkecil (Tiwari & Kanungo 2010). Xu dan Wang mengusulkan untuk melakukan modifikasi terhadap algoritma *load balancing* Round-Robin. *Modified Round-Robin* (MRR) melakukan pendistribusian terhadap permintaan *client* dengan mempertimbangkan *load* yang akan dibebankan kepada *web server* (Xu & Wang 2014). Namun keterbatasan dari jaringan tradisional terutama pada *dedicated server* yang menjadi *reverse proxy* mengakibatkan berbagai solusi yang diusulkan tersebut susah atau bahkan tidak efisien untuk diterapkan. *Load balancer* tradisional juga membutuhkan biaya yang mahal serta kompleksitas yang tinggi disamping keterbatasan *hardware* yang dapat menjadi *bottleneck* di dalam sistem (Zhong et al. 2017).

Implementasi dari *load balancing* pada SDN memanfaatkan berbagai keunggulan yang ditawarkan oleh SDN. Pemisahan antara *controller* dan *data plane* pada arsitektur SDN memungkinkan untuk melakukan konfigurasi *software* pada *controller* untuk melakukan *load balancing* yang lebih efisien (Zhong et al. 2017). SDN controller yang dikombinasikan dengan protokol OpenFlow dapat berperan sebagai *load balancer* dan mengumpulkan berbagai macam data yang diperlukan untuk memilih server tujuan. *Server* kemudian akan mengganti aturan (*flow table entries*) pada *SDN switch* dengan menggunakan pesan *Flow-mod* berdasarkan hasil perhitungan sebelumnya. Aturan baru tersebut memberikan *SDN switch* kontrol maksimum terhadap bagaimana paket diteruskan ke jaringan (Al-Najjar et al. 2016).

5. PERANCANGAN DAN IMPLEMENTASI

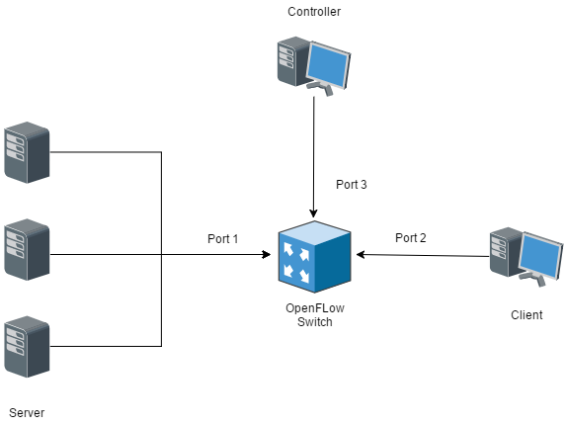
Penelitian ini mengimplementasikan tiga buah *web server* yang akan digunakan untuk melayani *request* dari *client*. Ketiga *web server* tersebut memiliki spesifikasi hardware yang sama (homogen) dan berjalan pada sistem operasi linux Ubuntu Server 14.04 LTS dengan aplikasi *server* Flask. Pada masing-masing *server* akan dipasang aplikasi berbasis web yang dibangun untuk menguji performa dari *server*

dengan pengalamatan yang berbeda untuk *client* dan *server*. Untuk melakukan perhitungan terhadap CPU dan *memory utilization* digunakan modul python psutil. Konfigurasi tambahan dilakukan untuk mengganti penggunaan *core* dan alokasi *memory* pada masing-masing *server* untuk menciptakan *server* yang memiliki spesifikasi berbeda (heterogen). Gambar 1 menjelaskan sistem pengalamatan dari ketiga *server* tersebut.

Gambar 1 Alokasi pengalamatan IP pada server

Penelitian ini akan mengimplementasikan arsitektur SDN dengan tiga buah *server* yang telah terkonfigurasi dan siap untuk melayani *client*, satu buah *SDN controller*, dan satu buah *SDN switch* yang akan mengarahkan *request* dari *client* menuju *server*.

SDN switch yang digunakan mengimplementasikan Open vSwitch pada komputer dengan sistem operasi linux Ubuntu Server 14.04 LTS. *Switch* tersebut kemudian akan dikonfigurasi untuk untuk meneruskan paket melalui tiga *port* yang berbeda. *Port* pertama dengan *interface* eth0 terhubung dengan *server cluster*, *port* kedua dengan *interface* eth1 terhubung dengan *client*, dan *port* ketiga dengan *interface* eth2 terhubung dengan *controller*.



Gambar 2 Topologi arsitektur SDN

SDN controller dalam penelitian ini akan menggunakan sebuah komputer dengan sistem operasi linux Ubuntu Desktop 14.04 LTS dan sistem operasi jaringan (NOS) Ryu 4.12 yang dapat diunduh dari github resmi Ryu. Protokol OpenFlow 1.3 digunakan sebagai protokol komunikasi antara *SDN controller* dengan *SDN switch* dan *server*. Gambar 2 menjelaskan topologi jaringan yang digunakan.

Load balancing bedasarkan *real time server status* akan menggunakan *SDN controller* dengan mode *proactive*. *Controller* melakukan perulangan sebanyak jumlah *server* untuk mengirimkan *request* menuju *switch* dengan *source address* berisi alamat IP dari *controller* dan *destination address* berisi alamat IP dari *server* ke-i untuk masing-masing *server*. *Controller* juga akan mencatat waktu perginya pesan. *Server* kemudian akan menjawab sehingga *controller* akan menerima *response* berisi data penggunaan *Memory* (M_i) dan penggunaan CPU (C_i) serta jumlah *request* dari *client* yang diterima *server*. Ketika pesan tiba di *controller*, *response time* dari masing-masing *server* akan dicatat (R_i). Variable SPM_i kemudian akan dihitung berdasarkan ketiga variable tersebut dengan rumus sebagai berikut.

$$SPMi = \frac{Ci+Mi+(Ri \times 100/n)}{3} \quad (1)$$

Nilai SPM_i dari masing server kemudian disimpan kedalam sebuah *variable global* untuk kemudian dicari nilai minimum (SPM_{min}) dari keseluruhan hasil. Setelah SPM_{min} didapatkan *controller* kemudian akan membuat *flow* table baru yang akan mengarahkan *request* dari *client* menuju *server i*. *Controller* kemudian akan meneruskan *flow table* tersebut menuju *switch*. Proses akan diulang berdasarkan interval waktu *n* yang telah ditetapkan sebelumnya.

Client digunakan untuk melakukan request terhadap *server*. *Client* akan menjalankan aplikasi httperf untuk masing-masing skenario pengujian.

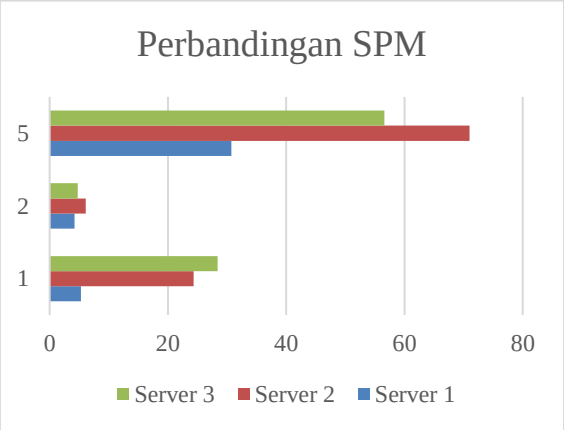
6. PENGUJIAN DAN HASIL

6.1. Pengujian Variasi Interval Pengiriman oleh Controller

Pengujian dilakukan dengan melakukan pengiriman *request* dari *client* dengan durasi yang telah ditentukan pada interval 1, 2, dan 5 detik. *Controller* kemudian akan mencatat *response* dari setiap *server*.

Pada ketiga skenario pengujian yang telah dijalankan ditemukan bahwa interval waktu

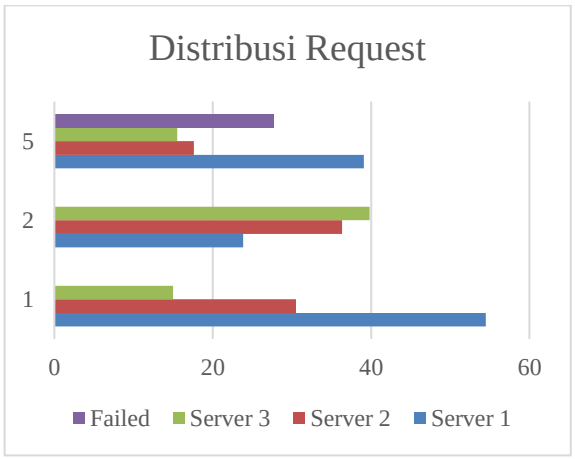
yang digunakan *controller* untuk mengirimkan *request* menuju *server* mempengaruhi kinerja dari *load balancer* itu sendiri.



Gambar 3 Perbandingan nilai SPM pengujian interval

Pada interval 1 detik, jeda waktu yang diberikan terlalu sempit sehingga dapat terjadi *timeout* pada pengecekan yang dilakukan oleh *controller*. Hal tersebut menyebabkan *controller* memberikan nilai maksimum pada penggunaan *CPU*, penggunaan *memory*, dan *response time* dari *server* yang bersangkutan. Hal tersebut secara komulatif berkontribusi terhadap besarnya nilai SPM dari *server* yang bersangkutan. Sedangkan pada interval 5 detik *server* kewalahan untuk melayani *request* yang dikirimkan oleh *client* dikarenakan jeda waktu penggantian *server* yang lebih lama sehingga kemungkinan terjadinya *overload* pada *server* menjadi besar. Terbukti rasio *timeout* dari pengujian dengan interval 5 detik cukup besar pada ketiga *server*. Penggunaan interval 2 detik terbukti lebih efektif dikarenakan tidak terdapat *timeout* dalam pengecekan yang dilakukan oleh *controller*. Gambar 3 menunjukkan perbandingan nilai SPM masing-masing server pada ketiga skenario.

Disamping itu penggunaan interval juga berpengaruh terhadap jumlah dan distibusi *request* pada setiap *server*. Pada pengujian dengan interval 1 detik jumlah *request* pada Server 1 lebih banyak dari *server* lain dikarenakan tingginya nilai SPM pada *server* tersebut. Sedangkan pada pengujian dengan interval 2 detik distribusi *request* dapat lebih merata dikarenakan *controller* dapat mengetahui status sebenarnya dari *server* dan mengarahkan *request* berdasarkan status tersebut. Pada pengujian dengan interval 5 detik terdapat request yang tidak berhasil dengan jumlah yang cukup besar. Hal tersebut terjadi karena *server* kewalahan melayani *request* yang diberikan sehingga *server* menjadi *crash* dan tidak bisa melayani *request* selanjutnya. Gambar 4 menujukan distribusi *request* dari ketiga interval. Bedasarkan pemaparan di atas, penggunaan interval 2 detik dipilih karena lebih efektif digunakan dalam *load balancer*.



Gambar 4 Perbandingan distribusi request pengujian interval

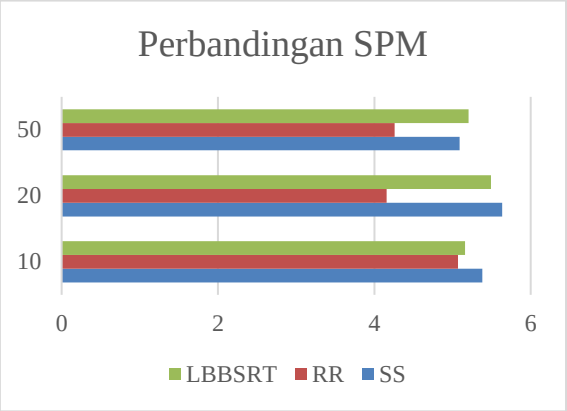
6.2. Pengujian Performa

Pada pengujian ping dari *controller* menuju *server* ditemukan bahwa Round Trip Time dari *controller* menuju masing-masing *server* memiliki perbedaan satu sama lain. Hal tersebut akan berpengaruh terutama pada algoritma yang mengandalkan hanya *Response* time dari *server* untuk menentukan pengalokasian *request* dari *client*. Oleh karena itu perlu diberikan tambahan parameter lain pada algoritma *load balancing* untuk meminimalisasi faktor ekternal yang tidak dibutuhkan.

Pada pengujian download ditemukan bahwa algoritma *load balancer* masih belum efektif untuk melakukan koneksi dalam waktu yang panjang serta jumlah paket TCP yang banyak. Terjadinya kegagalan dalam transfer *file* berukuran besar dapat disebabkan oleh perpindahan *server* yang terjadi sebelum koneksi TCP belum ditutup sehingga mengakibatkan terjadinya *error* di tengah proses transfer yang terjadi. Paket TCP yang seharusnya ditujukan pada *server* tertentu menjadi berpindah tujuan menuju server lain yang ditunjuk oleh load *balancer* sehingga paket menjadi tidak relevan dan diabaikan oleh *server* yang baru. Akibatnya transfer file menjadi terputus pada *sequence* tertentu.

6.3. Pengujian pada *Server* Homogen

Pengujian dilakukan dengan melakukan pengiriman *request* dari *client* dengan durasi yang telah ditentukan pada *server* sengan spesifikasi yang sama. Pengiriman dilakukan dengan kecepatan 10, 20, dan 50 *request* per detik.

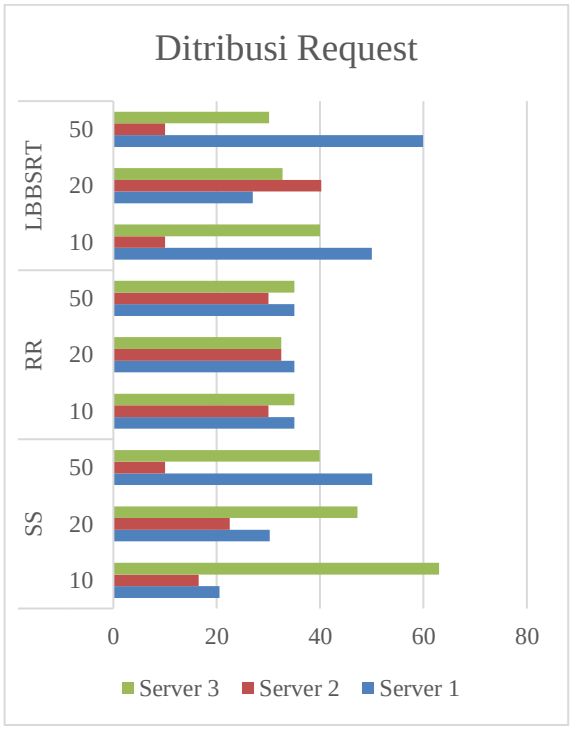


Gambar 5 Perbandingan nilai SPM pada server homogen

Pada ketiga skenario pengujian yang dijalankan ditemukan bahwa algoritma SS lebih efektif ketika digunakan pada pengujian dengan *request* per detik tinggi. Hal tersebut terjadi karena SS mendistribusikan *request* berdasarkan status dari *server* dan tidak dilakukan oleh kedua algoritma lainnya. SS juga lebih stabil dari LBBSRT dalam mendistribusikan *load* pada masing-masing *server*. Hal tersebut dibuktikan dengan kecilnya perubahan nilai SPM dalam setiap *sequence number* nya serta rendahnya standar deviasi dari nilai SPM tersebut. Gambar 5 menunjukkan perbandingan nilai SPM masing-masing server pada ketiga algoritma.

Distribusi *request* pada SS berbanding terbalik dengan nilai SPM dari *server* yang bersangkutan. Sedangkan pada RR *request* dibagi merata tanpa memperhatikan status dari *server*. Hal tersebut mengakibatkan terjadinya peningkatan *response time* dan penggunaan dari CPU pada *server* yang mengakibatkan tingginya nilai SPM. LBBSRT juga memiliki pola yang sama dengan SS namun dengan jumlah request yang berbeda pada masing-masing *server*. Hal tersebut dikarenakan LBBSRT lebih peka terhadap perubahan CPU dan keadaan dari jaringan sehingga tidak begitu menggambarkan keadaan sebenarnya dari *server*. Gambar 6 menunjukkan perbandingan distribusi request masing-masing server pada ketiga algoritma.

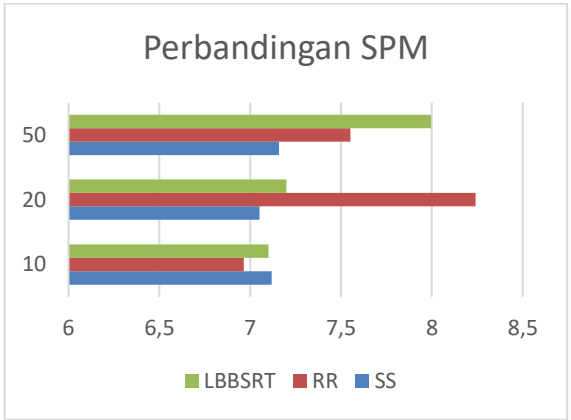
Pada pengujian dengan *request* per detik rendah perbedaan nilai SPM dari ketiga algoritma tidak terlalu signifikan. Hal ini kemungkinan besar disebabkan karena kecilnya *load* yang dibebankan pada *server* sehingga perubahan penggunaan *resource* menjadi tidak terlalu besar.



Gambar 6 Perbandingan distribusi request pada server homogen

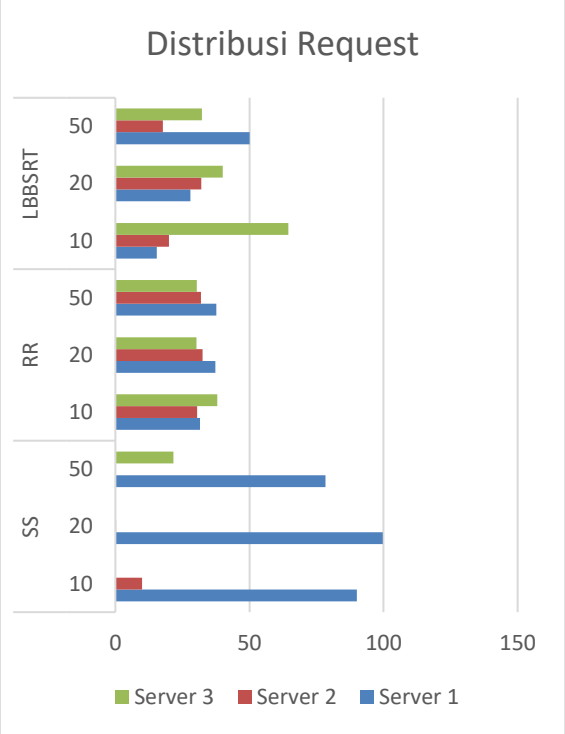
6.4. Pengujian pada *Server* Heterogen

Pengujian dilakukan dengan melakukan pengiriman *request* dari *client* dengan durasi yang telah ditentukan pada *server* sengan spesifikasi yang berbeda. Pengiriman dilakukan dengan kecepatan 10, 20, dan 50 *request* per detik.



Gambar 7 Perbandingan nilai SPM pada server heterogen

Pada ketiga skenario pengujian yang dilakukan ditemukan bahwa algoritma SS tetap memiliki efektifitas yang tinggi pada skenario pengujian dengan *request* per detik yang tinggi. Pada pengujian ini juga ditemukan bahwa penggunaaan *CPU* berpegaruh lebih dominan terhadap pemilihan *server* yang akan melayani *request* dari *client* disamping juga *response time* dari *server* menuju *controller*. Penggunaan *memory* tidak terlalu memberikan dampak signifikan dikarenakan fluktuasinya yang kecil meskipun dengan kapasitas yang berbeda. Namun penggunaan *memory* tetap dapat digunakan karena merepresentasikan status dari *server* secara keseluruhan. Hal tersebut berbeda pada algoritma RR dan LBBSRT yang tidak mempertimbangkan spesifikasi dari *server* yang digunakan. Hal tersebut dibuktikan dengan tingginya fluktuasi dari nilai SPM yang disebabkan oleh beban *load* yang diberikan pada *server* dengan kapasitas kecil pada saat *server* dengan kapasitas besar masih mampu untuk melayani *request* dari *client*. Gambar 7 menunjukkan perbandingan nilai SPM masing-masing server pada ketiga algoritma.



Gambar 8 Perbandingan distribusi request pada server heterogen

Pada distribusi *request* masing-masing algoritma ditemukan bahwa SS cenderung lebih banyak mengalokasikan *request* pada *server* yang memiliki kapasitas paling besar kemudian pada *server* yang memiliki kapasitas lebih rendah. Distribusi *request* dari SS dapat mereprestasikan variasi nilai SPM dari masing-masing *server*. Algoritma LBBSRT juga memiliki pola yang sama namun lebih dominan pada *response time* yang banyak dipengaruhi oleh penggunaan *CPU* dari *server*. Sedangkan algoritma RR cenderung membagi *request* secara merata tanpa mempertimbangkan spesifikasi dan kapasitas dari *server* yang dituju. Gambar 8 menujukan pebedaan distribusi *request* dari ketiga algoritma.

Pada pengujian dengan *request* per detik rendah dan sedang nilai SPM dan dari SS cenderung memiliki nilai yang lebih rendah dan lebih stabil dibanding kedua algoritma lainnya.

7. KESIMPULAN

Penulis berhasil mengimplementasikan load balancer berdasarkan real time server status pada arstitektur Sofware Defined Network (SDN). Dari keseluruhan pengujian yang dilakukan, didapatkan algoritma SS dapat lebih efektif digunakan pada kondisi dimana *request* per detik pada *server* cukup tinggi pada kondisi server yang memiliki spesifikasi berbeda.

Berdasarkan hasil pengujian tersebut penulis menyimpulkan bahwa karakteristik dari masing-masing algoritma memiki keutungan dan kekurangan masing-masing. Algoritma RR akan lebih efisien apabila diterapkan pada *server* dengan spesifikasi yang sama dan beban *load* yang juga sama. Algoritma LBBSRT cenderung lebih cocok digunakan pada *server* dengan spesifikasi sama namun memiliki beban *load* yang berbeda (layanan berbeda yang ditawarkan oleh *server*). Sedangkan algoritma RTSS akan lebih cocok untuk diterapkan pada *server* dengan spesifikasi dan *load* yang berbeda pada masing-masing *server*.

DAFTAR PUSTAKA

Al-Najjar, A., Layeghy, S. & Portmann, M., 2016. Pushing SDN to the end-host, network load balancing using OpenFlow. *2016 IEEE International Conference on Pervasive Computing and Communication Workshops, PerCom Workshops 2016*.

Boero, L. et al., 2016. BeaQoS: Load balancing and deadline management of queues in an OpenFlow SDN switch. *Computer Networks*, 106, pp.161–170.

Cardellini, V., Colajanni, M. & Yu, P.S., 1999.

Dynamic load balancing on web-server systems. *IEEE Internet Computing*, 3(3), pp.28–39.

Kaur, K. & Kaur, S., 2016. Flow Statistics Based Load Balancing in OpenFlow. *2016 Intl. Conference on Advances in Computing*, pp.378–381.

Kaur, S. et al., 2015. Round-Robin Based Load Balancing in Software. *Computing for Sustainable Global Development (INDIACom), 2015 2nd International Conference*, pp.2–5.

McKeown, N. et al., 2008. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM Computer Communication Review*, 38(2), p.69. Available at: http://portal.acm.org/citation.cfm?doid=1355734.1355746.

Nadeau, T. & Gray, K., 2013. *SDN: Software Defined Networks*, O'Reilly Media. Available at: http://oreilly.com/catalog/errata.csp?isbn=9781449342302.

Pao, T.L. & Chen, J.B., 2006. The scalability of heterogeneous dispatcher-based web server load balancing architecture. *Parallel and Distributed Computing, Applications and Technologies, PDCAT Proceedings*, pp.213–216.

Shang, Z. et al., 2013. Design and implementation of server cluster dynamic load balancing based on OpenFlow. *2013 International Joint Conference on Awareness Science and Technology and Ubi-Media Computing: Can We Realize Awareness via Ubi-Media?, iCAST 2013 and UMEDIA 2013*, pp.691–696.

Tiwari, A. & Kanungo, P., 2010. Dynamic load balancing algorithm for scalable heterogeneous web server cluster with content awareness. *Proceedings of the 2nd International Conference on Trendz in Information Sciences and Computing, TISC-2010*, pp.143–148.

Xu, Z. & Wang, X., 2014. A modified round-robin load-balancing algorithm for cluster-based web servers. *Proceedings of the 33rd Chinese Control Conference, CCC 2014*, pp.3580–3584.

Zha, J. et al., 2010. Research on load balance of Service Capability Interaction Management. In *2010 3rd IEEE International Conference on Broadband Network and Multimedia Technology (IC-BNMT)*. pp. 212–217.

Zhong, H., Fang, Y. & Cui, J., 2017. LBBSRT: An efficient SDN load balancing scheme based on server response time. *Future Generation Computer Systems*, 68, pp.183–190.