Pemerataan Utilisasi Jaringan *Multipath* dengan Mode *Controller Proactive-Reactive* pada *Software Defined Networking*

Akhmad Mukhtarom, Achmad Basuki, Muhammad Aswin

***Abstract*—Distributing traffic flows on fat-tree network topology used in data center networks is essential. Multipath routing is the common technique used to balance the traffic flows. In Software Defined Networking (SDN), the routing path is fully controlled by a controller to select the optimal path or the multiple paths in order to optimize flows throughput. This paper presents an implementation of a Hybrid proactive and reactive controller mode in SDN to load-balance traffic flows. The controller monitors the path utilization and will reactively install flow-tables to the appropriate switches whenever there are significant traffic spikes for a certain period. For the case of frequently high-utilized paths, the controller will proactively apply flow-tables to the appropriate switches. The evaluation results in a fat-tree topology show that the proposed Hybrid method outperforms LABERIO by 2.38 Mbps higher throughput and 18 seconds lower latency.**

***Index Terms*—SDN, fat-tree, data center, traffic, path utilization, multipath, load balance**

***Abstrak*---Mendistribusikan trafik pada topologi jaringan *fat-tree* yang sering digunakan di jaringan *data center* sangatlah penting. *Multipath routing* adalah teknik umum yang digunakan untuk menyeimbangkan trafik. Dalam *Software Defined Networking* (SDN), jalur *routing* sepenuhnya dikendallikan oleh *controller* untuk memilih jalur optimal atau beberapa jalur untuk mengoptimalkan *throughput* trafik. Penelitian ini mengimplementasikan mode hibridisasi *controller proactive-reactive* pada SDN untuk menyeimbangkan trafik. SDN *Controller* memonitor utilisasi jalur dan secara reaktif akan menginstal *flow-tables* ke *switch* yang sesuai setiap kali ada lonjakan trafik yang signifikan untuk periode tertentu. Untuk kasus jalur seringkali tinggi utilisasi trafiknya, SDN *Controller* akan secara proaktif menerapkan *flow-table* ke *switch* yang sesuai. Hasil evaluasi pada topologi *fat-tree* menunjukkan bahwa bahwa mode hibridisasi yang diusulkan berkinerja lebih baik daripada LABERIO dengan *throughput* 2,38 Mbps lebih tinggi dan latensi 18 detik lebih rendah.**

***Kata Kunci*—SDN, *fat-tree*, *data center*, trafik, utilisasi jalur, *multipath*, pemerataan beban**

Akhmad Mukhtarom, Achmad Basuki, Muhammad Aswin are with the Electrical Engineering Department of Brawijaya University, Malang, Indonesia (corresponding author provide phone 0341-554166; email dek.tiram@student.ub.ac.id)

I. PENDAHULUAN

FAT-TREE adalah topologi yang banyak digunakan dalam jaringan *data center*. Topologi *fat-tree* dapat memberikan kapasitas koneksi yang tinggi antar *server-node* [1]. Dengan beberapa *path* yang tersedia, sangat memungkinkan untuk melakukan *multipath* untuk komunikasi antar *server-node* [2]. Keadaan dimana terjadi *overload* pada salah satu *path* (*overload-path*), sedangkan *load* pada *path* lain masih rendah sangat tidak dikehendaki [3]. Tidak meratanya penyebaran trafik pada seluruh *path* mengakibatkan penurunan *throughput* trafik. Penurunan *throughput* juga mengakibatkan semakin besarnya latensi pengiriman data [4]. SDN adalah teknologi yang bekerja berdasarkan *flow* yang dikendalikan dari *controller*. Trafik pada SDN disebut juga sebagai *traffic-flow*. SDN memungkinkan untuk melakukan pemerataan *traffic-flow* kepada seluruh *path* yang tersedia dalam jaringan.

Dibutuhkan suatu metode *load-balance* dalam SDN untuk mendistribusikan *traffic-flow* ke seluruh *path* yang tersedia. Mengingat *traffic-flow* yang terjadi pada *data center* sangat dinamis, maka metode *load-balance* yang digunakan harus mampu mendeteksi keadaan *overload-path* dan membagi *traffic-flow* kepada *path* yang lain. Sistem deteksi *overload-path* dengan menggunakan ambang batas beban (*load-threshold*) 80% dari *bandwidth path* merupakan sistem deteksi *overload-path* yang ideal. Slavica *dkk.* dalam [5] menerapkan metode yang membedakan *traffic-flow* TCP dan UDP untuk membagi *traffic-flow* pada jaringan agar *traffic-flow* UDP tidak mempengaruhi performa *traffic-flow* TCP. Metode tersebut gagal menyeimbangkan *traffic-flow* apabila hanya terdapat *flow* TCP.

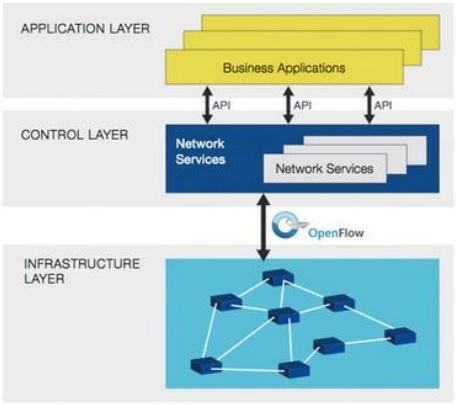
Metode LABERIO [6] mendeteksi *overload-path* dan perubahan terjadinya kenaikan *traffic-flow* pada jaringan. LABERIO melakukan perubahan *route-path* (rute *path* yang dilalui dari *node* sumber ke *node* tujuan) terhadap *traffic-flow* terbesar pada *overload-path*. LABERIO membedakan *traffic-flow* lebih detil daripada metode Slavica berdasarkan *node* IP sumber dan *node* IP tujuan tanpa membedakan protokol yang digunakan. Namun demikian, LABERIO tidak dapat meratakan *traffic-flow* apabila terdapat beberapa flow dengan node IP sumber dan tujuan yang sama, sekalipun berbeda *port*. Metode LABERIO disempurnakan dengan metode

DLPO [7] dimana perubahan *route-path* tidak lagi digunakan dan diganti dengan memberikan nilai prioritas lebih rendah pada *traffic-flow* terbesar yang terdapat pada *overload-path*. Akan tetapi metode DLPO dapat menyebabkan kondisi *overload-path* bila terdapat beberapa *traffic-flow* dengan *node* sumber dan tujuan yang sama. Mengingat pada *data center* sering kali terjadi perubahan *traffic-flow*, maka sangat memungkinkan sekali terjadinya lonjakan *traffic-flow* secara sesaat (*traffic-spike*). Karena pada LABERIO dan DLPO hanya menggunakan *load-threshold*, maka akan mengakibatkan proses pengalihan *route-path* berkali – kali saat terjadinya *traffic-spike*. Hal ini tentunya akan mempengaruhi kinerja *controller* dan menurunkan *throughput traffic-flow*.

Penelitian ini mengatasi masalah yang terjadi pada metode Slavica, LABERIO dan DLPO. Dengan menggunakan hibridisasi mode *controller proactive* dan *reactive* serta membedakan *traffic-flow* berdasarkan detil IP *address*, protokol dan *port number*. Metode tersebut akan lebih meratakan beban *traffic-flow* ke seluruh *path* dalam jaringan. Selain itu pada penelitian ini ditambahkan parameter *time-threshold* untuk mengatasi terjadinya *traffic-spike* dalam periode tertentu. Metode dalam penelitian ini disebut dengan metode Hydrid untuk mengatasi kondisi dinamis utilisasi jaringan untuk mengoptimalkan *throughput traffic-flow* dan latensi pada jaringan.

II. *SOFTWARE DEFINED NETWORKING* (SDN)

Software Defined Network (SDN) adalah sebuah konsep atau paradigma baru yang memberikan kemudahan kepada pengguna dalam mendesain, membangun dan mengelola jaringan komputer. SDN menyediakan pengelolaan distribusi jaringan terpusat (*centralized*) agar layanan jaringan menjadi lebih efisien, otomatis, dan cepat. *Controller* jaringan lebih bersifat *software* sehingga lebih fleksibel untuk dikonfigurasikan dan mekanisme jaringan (*forwarder*) menjadi lebih mudah dikontrol [8].



Gambar 1. Arsitektur SDN

Pada Gambar 1 nampak terdapat 3 *layer* dalam arsitektur SDN. *Layer* infratruktur merupakan *data-plane* yang berupa *switch* sebagai device yang mempunya fungsi untuk meneruskan *flow*. *Control layer* adalah *control-plane* yang mempunyai fungsi kontrol terhadap konfigurasi dan kinerja *data-plane*. Sedangkan *application layer* adalah *application-plane* yang mengatur sistem kerja jaringan secara keseluruhan. *Application plane* berisi tentang algoritma dan metode untuk mengatur cara kerja jaringan [9].

A. *Openflow*

Openflow adalah protokol standar yang digunakan untuk komunikasi antar *control-plane* dan *data-plane* [10]. Dengan protokol *Openflow* memungkinkan *control-plane* melakukan modifikasi kinerja *data-plane* dalam meneruskan *flow*. *Openflow* pada *switch* bekerja berdasarkan *flow-table* yang tersimpan pada *switch*. Setiap *flow* yang masuk diidentifikasi berdasarkan beberapa parameter. Parameter identifikasi *flow* ditunjukkan pada Gambar 2.

Switch Port	MAC src	MAC dst	Eth type	VLAN ID	IP Src	IP Dst	IP Prot	TCP sport	TCP dport

Gambar 2. *Parameter pada flow-table*

Pada Gambar 2 ditunjukkan bahwa *flow* dapat dibedakan berdasarkan MAC, IP, protokol dan *port number* [11].

B. *Algoritma Dijkstra*

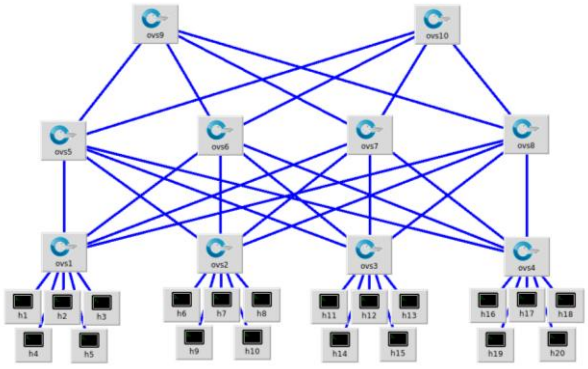
Dijkstra adalah suatu algoritma untuk menentukan jalur terpendek berdasarkan data *graph* dan nilai *cost*/*weight* pada edge-*graph* [12]. Nilai *cost* pada *path* jaringan diformulasikan seperti dalam Persamaan 1.

$$Cost = \frac{\text{Reference\_}Bandwidth}{Interface\_Bandwidth} \qquad (1)$$

Dimana *Reference_Bandwidth* adalah *path-load* dan *Interface Bandwidth* adalah kapasitas maksimum dari *path* tersebut.

III. METODOLOGI PENELITIAN

A. *Topologi Jaringan*



Gambar 3. Topologi fat-tree

Topologi jaringan yang digunakan adalah *fat-tree* dengan 4 *ToR switch*, 4 *aggregation switch* dan 2 *core switch*. Masing – masing *ToR switch* terkoneksi dengan 5 *host* seperti nampak pada Gambar 3.

B. Network Discovery

Proses *network discovery* adalah proses dimana sistem mengumpulkan data informasi tentang jaringan. Data jaringan yang dikumpulkan meliputi :

1) *Switch*
   Data *switch* meliputi dpid (*datapath identity*) yang merupakan indentitas unik dari *switch* dalam protokol *openflow* dan MAC *address* dari seluruh *network interface* yang terdapat pada *switch* tersebut.
2) *Host*
   Data *host* meliputi data MAC *address network interface* dan IP *address* yang digunakan.
3) *Koneksi*
   Data koneksi adalah daftar koneksi antar *network interface*, baik *network interface* pada *switch* maupun pada *host*.

Dengan data tersebut sistem akan mengetahui topologi jaringan, *switch* yang digunakan dan pengalamatan IP *address* seluruh *host* pada jaringan.

C. Segmentasi

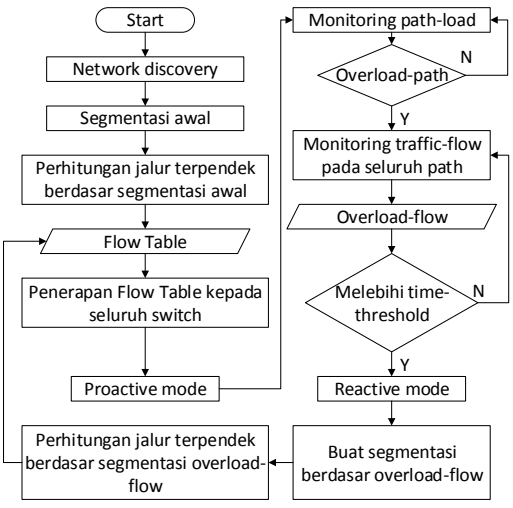
Segmentasi adalah proses pengelompokan *traffic-flow* menjadi bagian yang lebih kecil. Segmentasi pada penelitian ini menggunakan parameter IP *address* sumber dan tujuan, protokol serta *port number* yang digunakan oleh *traffic-flow*. Identitas segment *traffic-flow* dinotasikan seperti Gambar 2.

Source			***Destination***	
IP *address*	***Port number***	Protokol	***Port number***	**IP *address***
10.0.2.1	49152	TCP	80	10.0.2.10
"10.0.2.1/-1/6/80/10.0.2.10"				

Gambar 4. Contoh segmentasi traffic-flow

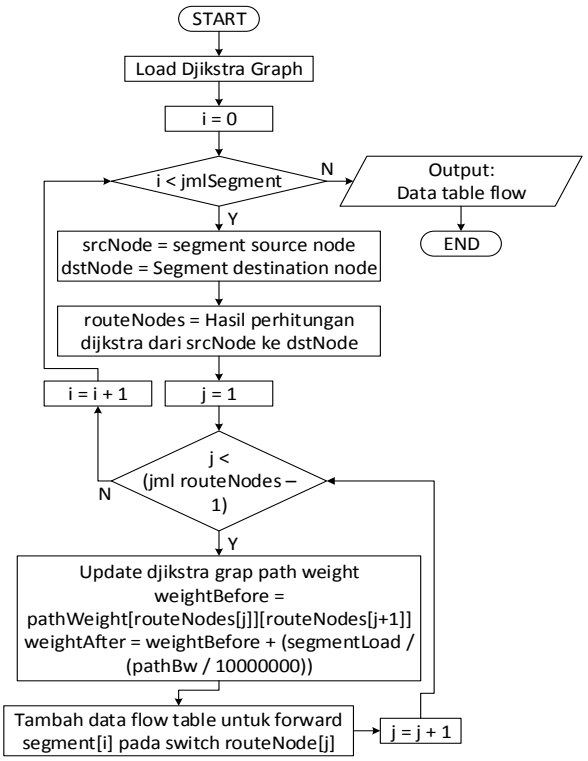
Gambar 4 adalah contoh untuk segmentasi *traffic-flow* dari 10.0.2.1 ke 10.0.2.10 menggunakan protokol TCP dengan *port number* 80. *Port number* 49152 adalah *dynamic port* yang diberikan oleh OS (Operating System) [13]. *Dynamic port* dinotasikan dengan ”-1”.

D. Perancangan Sistem



Gambar 5. Diagram alir sistem

Gambar 5 adalah diagram alir cara kerja sistem. Pada awal sistem dijalankan, sistem akan melakukan *network discovery* dengan tujuan mengunmpulkan data informasi tentang jaringan. Kemudian sistem akan melakukan segmentasi *traffic-flow* berdasarkan kombinasi *source* dan *destination* data IP *address host*. Dilanjutkan dengan proses menghitung *route-path* tiap segmen menggunakan *Dijkstra*. Data hasil perhitungan adalah data *flow-table* yang selanjutnya diterapkan pada seluruh *switch* pada jaringan. Dengan demikian *controller* akan bekerja dengan *proactive* mode dan komunikasi antar *server-node* dapat berlangsung tanpa peran *controller*. Pada fase ini *controller* melakukan *monitoring* terhadap *path-load* seluruh *path*. Apabila terdapat satu atau beberapa *overload-path*, sistem akan menunggu sampai batas *time-threshold*. Dalam rentang waktu sebelum *time-threshold*, sistem akan melakukan pembacaan seluruh *traffic-flow* yang terjadi pada jaringan (*overload-flow*). Apabila *overload-path* terjadi melebihi *time-threshold* maka sistem akan melakukan segmentasi dan perhitunga *route-path* berdasarkan data *overload-flow*. Pada saat sistem melakukan segmentasi dan perhitungan *route-path*, sistem akan merubah mode *controller* menjadi mode *reactive* untuk memproses *traffic-flow* pada jaringan sebelum *flow-table* diterapkan pada seluruh *switch*.



Gambar 6. Diagram alir perhitungan route-path

Gambar 6 adalah diagram alir untuk perhitungan untuk menentukan *route-path* dari setiap segmen *traffic-flow* menggunakan metode *Dijkstra*. Data *graph Dijkstra* dimuat berdasarkan data yang diperoleh pada proses *network discovery*. Proses perhitungan *Dijkstra* dilakukan pada setiap segmen *traffic-flow*. Tiap perhitungan satu segmen sistem akan menrubah nilai *weight*/*cost* edge *graph* pada *Dijkstra* sesuai dengan nilai kapasitas *traffic-flow*

untuk perhitungan segmen berikutnya. Nilai 10000000 adalah konstanta normalisasi *bandwidth*. Nilai ini diambil dari nilai *bandwidth* terbesar *path* pada jaringan.

E. Simulasi

Untuk membuktikan kinerja metode Hydrid dilakukan simulasi sistem menggunakan *emulator* jaringan *mininet*. Untuk *controller* yang digunakan menggunakan *Ryu* dan beberapa script perhitungan dibangun dengan menggunakan bahasa pemrograman *Python*. Pengujian yang dilakukan adalah memberikan beberapa *traffic-flow* pada jaringan.

TABEL I.
DAFTAR BEBAN TRAFFIC-FLOW PENGUJIAN

ID	Src Node	Dst Node	Proto	Port Num.	Load (Mbps)	Time (minutes)
FL1	h1	h16	UDP	3001	5	10
FL2	h2	h17	UDP	3002	5	10
FL3	h3	h18	UDP	3003	5	10
FL4	h4	h19	UDP	3004	5	10
FL5	h5	h20	TCP	2001	2.5	10
FL6	h5	h20	TCP	2002	2.5	10
FL7	h5	h20	TCP	2003	2.5	10
FL8	h5	h20	TCP	2004	2.5	10

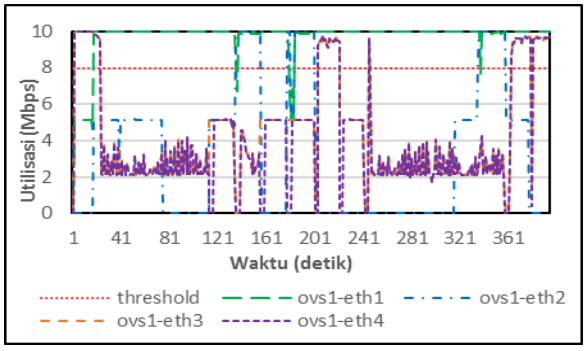
Tabel I adalah *traffic-flow* yang diberikan untuk menguji dan membandingkan performa metode Hydrid dengan metode lainnya (Slavica, LABERIO dan DLPO). *Traffic-flow* tersebut diberikan secara bersamaan. Parameter yang diukur dan dibandingkan adalah kemampuan untuk meratakan *traffic-flow* kepada seluruh *path* pada jaringan dan nilai *throughput traffic-flow* secara keseluruhan.

Selain pengujian menggunakan *traffic-flow* pada Tabel 1, diberikan juga pengujian kecepatan dalam melakukan *copy* file. Pengujian dilakukan dengan menambahkan proses *copy* file sebesar 50MB dari *node* h6 ke h11.

IV. HASIL DAN PEMBAHASAN

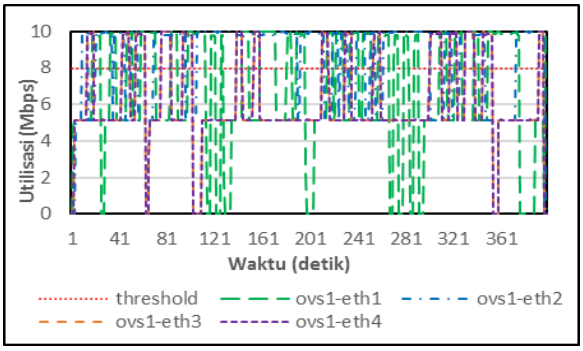
A. Utilisasi Path

Pengukuran dilakukan pada *path egress* dari *traffic-flow*. Semua *traffic-flow* pada pengujian melewati ovs1, maka *interface* dalam grafik utilisasi diambil dari *throughput interface* dari ovs1 yaitu ovs1-eth1, ovs1-eth2, ovs1-eth3 dan ovs1-eth4.



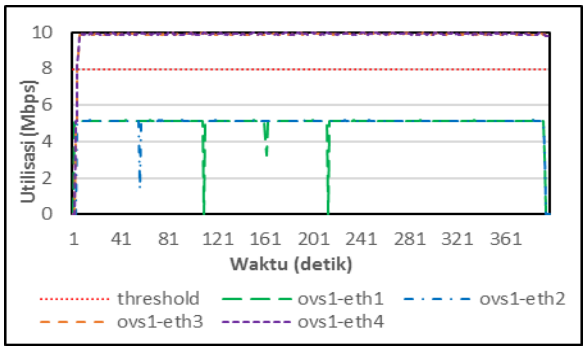
Gambar 7. Grafik utilisasi path pada metode Slavica

Nampak pada gafik Gambar 7, metode Slavica tidak dapat meratakan *traffic-flow* secara optimal kepada seluruh *path* yang tersedia. Beban pada ovs1-eth1 hampir selalu melebihi batas *load-threshold*, sedangkan *path* yang lain masih jauh di bawah batas *load-threshold*.



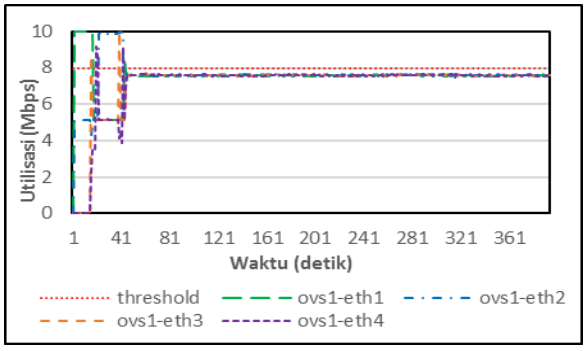
Gambar 8. Grafik utilisasi path pada metode LABERIO

Sedangkan grafik utilisasi *path* untuk metode LABRIO seperti yang ditunjukkan dalam Gambar 8 memperlihatkan utilisasi *path* yang sangat dinamis. Beberapa *path* masih berada di atas batas *load-threshold* sepanjang pengujian. Pemerataan *traffic-flow* pada LABERIO belum bisa dikatakan optimal.



Gambar 9. Grafik utilisasi path pada metode DLPO

DLPO tidak melakukan pengalihan *route-path* melainkan hanya merubah nilai prioritas pada *flow-table*. Terlihat pada grafik Gambar 9 ovs1-eth3 dan ovs1-eth4 berada konstan melebihi batas *load-threshold*. Metode DLPO juga belum dapat meratakan *traffic-flow* secara optimal.



Gambar 10. Grafik utilisasi path pada metode Hydrid

Grafik hasil pengujian utilisasi *path* pada metode Hydrid ditunjukkan pada Gambar 10. Hydrid dapat meratakan *traffic-flow* kepada pat yang tersedia

dalam jaringan. Utilisasi *path* pada Hydrid lebih optimal dan berada di bawah batas *load-threshold*. Pada bagian awal grafik nampak Hydrid melakukan perubahan *route-path traffic-flow* dan menerapkan *flow-table* baru terhadap seluruh *switch* pada jaringan. Dengan *flow-table* baru semua *traffic-flow* berhasil diratakan pada *path* yang tersedia.

B. Throughput

TABEL II.
TABEL THROUGHPUT TRAFFIC-FLOW

ID	Bandwidth (Mbps)			
	Slavica	**LABERIO**	**DLPO**	**Hydrid**
FL1	3.53	4.84	4.97	4.95
FL2	4.31	5	5	4.84
FL3	4.5	4.79	5	4.8
FL4	3.74	5	5	4.94
FL5	1.17	1.55	1.33	2.28
FL6	2.29	1.65	1.12	2.23
FL7	0.352	1.76	1.38	2.18
FL8	1.03	1.51	1.24	2.26
Total	**20.922**	**26.1**	**25.04**	**28.48**

Tabel II menunjukkan perolehan *throughput* setiap *traffic-flow*. *Throughput traffic-flow* UDP pada LABERIO dan DLPO lebih tinggi daripada metode lainnya, tapi tidak untuk *traffic-flow* TCP. Perolehan *throughput* pada Slavica lebih rendah daripada metode lainnya. Untuk perolehan *throughput traffic-flow* secara keseluruhan, metode Hydrid lebih besar 7.558 Mbps dari Slavica, 2.38 Mbps dari LABERIO dan 3.44 Mbps dari DLPO.

C. Latensi

TABEL III.
TABEL HASIL KECEPATAN PROSES COPY FILE

Metode	Avg. Bw (Kbps)	Durasi (detik)
Slavica	642.5	76
LABERIO	775.1	63
DLPO	739.8	66
Hydrid	1100	45

Pada pengujian *copy* file nampak pada Tabel III metode Hydrid memperoleh *bandwidth* paling tinggi yaitu 1.1 Mbps. Dengan *bandwidth* rata – rata tertinggi Hydrid dapat meyelesaikan proses *copy* file selama 45 detik. Sedangkan Slavica dengan *bandwidth* rata – rata 642.5 Kbps selisih 31 detik lebih lama dari Hydrid. Sedangkan untuk LABERIO selisih 18 detik dan DLPO 21 detik lebih lama daripada Hydrid.

V. KESIMPULAN

Hibridasi mode *proactive* dan *reactive controller* SDN serta segmentasi *traffic-flow* yang lebih detil dapat meratakan penyebaran *traffic-flow* ke beberapa *path* yang tersedia, sehingga dapat meningkatkan *throughput traffic-flow* secara keseluruhan dan menurunkan latensi pengiriman file. Berdasarkan hasil pengujian, metode Hydrid mampu meningkatkan *throughput traffic-flow* secara keseluruhan 2.38 Mbps lebih tinggi dari metode LABERIO. Untuk latensi pengiriman file, metode Hydrid lebih cepat 18 detik dari metode LABERIO. Dengan demikian metode Hydrid mampu mengoptimalkan performa jaringan lebih baik dari metode Slavica, LABERIO dan DLPO.

VI. DAFTAR PUSTAKA

[1] F. Yao, J. Wu, G. Venkataramani, and S. Subramaniam, "A comparative analysis of data center network architectures," *Proc. IEEE Int. Conf. Commun. - Next Gener. Netw.*, pp. 3106–3111, 2014.

[2] S. Mahapatra, X. Yuan, and W. Nienaber, "Limited multi-path routing on extended generalized fat-trees," *Proc. 2012 IEEE 26th Int. Parallel Distrib. Process. Symp. Work. IPDPSW 2012*, pp. 938–945, 2012.

[3] L. Y. Chen, W. Denzel, and R. Luijten, "Optimization of link bandwidth for parallel communication performance," *2009 IEEE 28th Int. Perform. Comput. Commun. Conf. IPCCC 2009*, pp. 137–144, 2009.

[4] F. Hassen, "Congestion-Aware Multistage Packet-Switch Architecture for Data Center Networks," 2016.

[5] S. Tomovic, N. Lekic, and I. Radusinovic, "A new approach to dynamic routing in SDN networks," vol. 2012, no. 315970, pp. 18–20, 2016.

[6] H. Long, Y. Shen, M. Guo, and F. Tang, "LABERIO: Dynamic load-balanced routing in OpenFlow-enabled networks," *Proc. - Int. Conf. Adv. Inf. Netw. Appl. AINA*, pp. 290–297, 2013.

[7] Y. L. Lan, K. Wang, and Y. H. Hsu, "Dynamic load-balanced path optimization in SDN-based data center networks," *2016 10th Int. Symp. Commun. Syst. Networks Digit. Signal Process. CSNDSP 2016*, pp. 0–5, 2016.

[8] E. P. Aprilianingsih, R. Primananda, and A. Suharsono, "Analisis Fail Path Pada Arsitektur Software Defined Network Menggunakan Dijkstra Algorithm," vol. 1, no. 3, pp. 174–183, 2017.

[9] B. A. A. Nunes, M. Mendonca, X. N. Nguyen, K. Obraczka, and T. Turletti, "A survey of software-defined networking: Past, present, and future of programmable networks," *IEEE Commun. Surv. Tutorials*, vol. 16, no. 3, pp. 1617–1634, 2014.

[10] A. Mahesh and A. H. Firewalls, "Cloud based firewall on OpenFlow SDN network."

[11] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, p. 69, 2008.

[12] G. A. Nugroho, A. S. Lamaida, and Y. Ahsana, "Analisis Algoritma Pencarian Rute Terpendek Dengan Algoritma Dijkstra dan Bellman - Ford," *Tek. Inform. ITB*, 2006.

[13] M. Muggeridge, "Programming with TCP / IP – Best Practices," pp. 1–31.