TESIS – IF185401

PENGEMBANGAN MEKANISME *NETWORK BALANCING* PADA JARINGAN *SOFTWARE DEFINED NETWORK DATA PLANE* DENGAN MENGGUNAKAN *LEAST LOADED PATH*

MHD. FATTAHILAH RANGKUTY
05111850010015

Dosen Pembimbing
Royyana Muslim I., S. Kom., M. Kom., Ph.D.
Tohari Ahmad, S. Kom, MIT., Ph.D.

Departemen Teknik Informatika
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember

2020

[Halaman ini sengaja dikosongkan]

TESIS – IF185401

PENGEMBANGAN MEKANISME *NETWORK BALANCING* PADA *JARINGAN SOFTWARE DEFINED NETWORK DATA PLANE* DENGAN MENGGUNAKAN *LEAST LOADED PATH*

MHD. FATTAHILAH RANGKUTY
05111850010015

Dosen Pembimbing
Royyana Muslim I., S. Kom., M. Kom., Ph.D.
Tohari Ahmad, S. Kom, MIT., Ph.D.

Departemen Teknik Informatika
Fakultas Teknologi Elektro dan Informatika Cerdas
Institut Teknologi Sepuluh Nopember
2020

[Halaman ini sengaja dikosongkan]

LEMBAR PENGESAHAN TESIS

Tesis disusun untuk memenuhi salah satu syarat memperoleh gelar
Magister Komputer (M. Kom.)
di
Institut Teknologi Sepuluh Nopember Surabaya
oleh:
MHD. FATTAHILAH RANGKUTY
NRP: 05111850010015

Tanggal Ujian: 22 Juli 2020
Periode Wisuda: September 2020

Disetujui Oleh:
Pembimbing:

1. Royyana Muslim I., S. Kom., M .Kom.,Ph.D.
   NIP. 19770824 200604 1 001

2. Tohari Ahmad, S. Kom, MIT., Ph.D.
   NIP. 19750525 200312 1 002

Penguji:

1. Prof. Ir. Supeno Djanali, M.Sc., Ph.D.
   NIP. 19480619 197301 1 001

2. Dr. Eng. Radityo Anggoro, S. Kom., M.Sc.
   NIP. 19841016 200812 1 002

3. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D.
   NIP. 19810620 200501 1 003

Kepala Departemen Teknik Informatika
Fakultas Teknologi Elektro dan Informatika Cerdas

Dr. Eng. Chastine Fatichah, S.Kom., M.Kom
NIP. 19751220 200112 2002

[Halaman ini sengaja dikosongkan]

Pengembangan Mekanisme *Network Balancing* Pada Jaringan *Software Defined Network Data Plane* Dengan Menggunakan *Least Loaded Path*

Nama Mahasiswa : Mhd. Fattahilah Rangkuty
NRP : 05111850010015
Pembimbing : Royyana Muslim I., S. Kom., M. Kom., Ph.D.
Tohari Ahmad, S. Kom, MIT., Ph.D.

ABSTRAK

Software Defined Network (SDN) merupakan paradigma baru dalam teknologi jaringan, SDN memisahkan *control plane* dan *data plane* jaringan. SDN adalah pendekatan jaringan komputer yang memungkinkan administrator jaringan untuk mengelola layanan jaringan melalui abstraksi fungsionalitas pada tingkat yang lebih tinggi, dengan memisahkan sistem yang membuat keputusan tentang di mana *traffic* dikirim (*control plane*), kemudian meneruskan *traffic* ke tujuan yang dipilih (*data plane*). Untuk peningkatan performa dari suatu jaringan SDN, salah satu metode yang digunakan adalah *load balancing*. Teknik ini membagi seluruh beban secara merata pada setiap komponen jaringan seperti *data plane* atau pada jalur atau *path* yang menghubungkan *data plane* dan S-D (*Source Destination*) host, sehingga dapat menghindari kemacetan jaringan dan peningkatan *transfer rate* data dan penurunan *latency*.

Pada penelitian ini, peneliti mengembangkan mekanisme *path selection* dengan menggunakan metode *least loaded path* (LLP) untuk mencapai *load balancing*, yang disimulasikan pada jaringan *data plane* SDN. Algoritma tersebut digunakan pada *load balancing* untuk mengatur alur mekanisme dalam memperoleh pembagian beban yang maksimal, sehingga dapat dicapai peningkatan performa jaringan dalam hal ini *transfer rate data* dan *latency*. Konsep LLP yang merupakan pengembangan *Dijkstra*, melakukan pemilihan *best path* dengan mencari jalur terpendek dan beban *traffic* terkecil, beban *traffic* terkecil (*minimum cost*) didapatkan dari penjumlahan *tx* dan *rx* data pada *switchport data plane* yang terlibat dalam pengujian, hasil ini yang kemudian akan ditetapkan sebagai best path dalam proses load balancing.

Dari uji coba yang telah dilakukan, setelah proses mekanisme LLP pada S-D h1-h4 dan S-D h5-h8, terjadi penurunan rata-rata maksimum *latency* sebesar 5-10%. Sementara hasil *transfer rate* data meningkat mulai 7% hingga 96% setelah dilakukan mekanisme seleksi *path* dengan LLP pada kedua pasangan S-D *host*. Selain itu, peningkatan *throughput* juga terjadi setelah dilakukan mekanisme seleksi *path* dengan LLP. Peningkatan-peningkatan tersebut terjadi karena mekanisme seleksi *path* dengan LLP telah berhasil mengirimkan paket data melalui *best path* yang sudah di proses sebelumnya, sehingga proses *load balancing* dapat dilakukan. Hal ini tidak dilakukan pada *load balancing default* yang sudah

terintegrasi pada kontroler *floodlight*. Pada mekanisme LLP, selain memilih *shortest path*, penghitungan *cost* pada *path* yang memungkinkan untuk dilewati juga dilakukan, sementara pada *default* LB hanya menghitung *shortest* path saja. Hasil ini juga menunjukkan mekanisme ini dapat diterapkan pada topologi *fat tree* atau jaringan *real* skala kecil dan besar, sehingga dapat memenuhi fleksibilitas dan skalabilitas pada jaringan SDN *data plane*.

Kata Kunci*: Data plane, Floodlight, Load balancing, OpenFlow, Software defined networking.*

Development Of Network Balancing Mechanisms In Software Defined Network Data Plane Using Least Loaded Path

By : Mhd. Fattahilah Rangkuty
Student ID Number : 05111850010015
Supervisor : Royyana Muslim I., S. Kom., M. Kom., Ph.D.
Tohari Ahmad, S. Kom, MIT., Ph.D.

ABSTRACT

Software Defined Networks (SDN) is a new paradigm in network technology, SDN separating the *control* plane and the data plane of the network. SDN is an approach to computer networking that allows network administrators to manage network services through abstraction of higher-level functionality. This is done by decoupling the system that makes decisions about where traffic is sent (the *control* plane) from the underlying systems that forward traffic to the selected *destination* (the data plane). To improve the performance of an SDN network, one of the methods used is load balancing. This technique divides all loads evenly on each network component such as a data plane or on a path or path connecting a data plane and S-D (Source Destination) host, so as to avoid network congestion and increase data transfer rates and decrease latency.

In this study, researchers developed a path selection mechanism using the least loaded path (LLP) method to achieve load balancing, which is simulated on the SDN data plane network. The algorithm is used in load balancing to set the flow mechanism to obtain maximum load sharing, so that network performance can be achieved in this case the data transfer rate and latency. The LLP concept, which is a Dijkstra development, selects the best path by finding the shortest path and the smallest traffic load, the smallest traffic load (minimum cost) obtained from the addition of tx and rx data on the switchport data plane involved in the test, these results which will then be determined as best path in the load balancing process.

From the trials that have been carried out, after the LLP mechanism process in S-D h1-h4 and S-D h5-h8, an average reduction in maximum latency of 5-10% has occurred. While the results of the data transfer rate increased from 7% to 96% after the mechanism of path selection with LLP in both S-D host pairs. In addition, an increase in throughput also occurs after a path selection mechanism with LLP. These improvements occur because the path selection mechanism with LLP has successfully sent data packets through the best path that has been previously processed, so that the load balancing process can be carried out. This is not done in the default load balancing that has been integrated in the floodlight controller. In the LLP mechanism, in addition to choosing the shortest path, the calculation of the cost of the path that allows it to be passed is also done, while in default LB only counts the shortest path. This result also shows that this mechanism can be applied to *fat tree* topology or small and large scale real networks, so that it can meet the flexibility and scalability of SDN data plane networks..

Keywords: *Data plane, Floodlight, Load balancing, OpenFlow, Software defined networking.*

KATA PENGANTAR

Alhamdulillahirobbil'alamin, Segala puji hanya milik Allah ﷻ yang telah memberikan ilmu dan pengetahuan kepada hamba-hamba-Nya. Shalawat dan salam semoga senantiasa tercurah kepada utusan yang termulia, Rasulullah ﷺ, manusia terbaik dan termulia yang pernah ada di alam semesta ini, sehingga penulis dapat menyelesaikan penelitian tesis dengan judul **"Pengembangan Mekanisme *Network Balancing* Pada Jaringan *Software Defined Network Data Plane* Dengan Menggunakan *Least Loaded Path*"** dengan baik.

Dengan segala kerendahan hati penulis ingin mengucapkan terima kasih dan penghargaan setingi-tingginya atas bantuan, dukungan, motivasi serta bimbingan kepada semua pihak, sehingga penulis dapat menyelesaikan penelitian tesis ini dengan baik, antara lain kepada :

1. Allah ﷻ atas segala limpahan rahmat dan nikmat iman, islam, kesehatan, rezeki, kesempatan, serta seluruh karunia yang diberikan kepada penulis sehingga dapat menyelesaikan penelitian tesis ini dengan baik.
2. Rasulullah Muhammad ﷺ yang telah menjadi suri tauladan dan membawa seluruh umat manusia menuju jalan yang di-ridhoi Allah ﷻ.
3. Kedua orang tua, Ayahanda M. Syafii Rangkuti dan Ibunda Hawa Ningsih serta kedua mertua, Bapak Tulus Abdullah dan Ibu Kamini, yang selalu memberikan dorongan semangat dan do'a kepada penulis untuk mencapai kesuksesan.
4. Isteri tercinta Isti Ariyah yang selalu memberikan dukungan dan selalu menemani penulis selama menempuh pendidikan magister ini.
5. Bapak Royyana Muslim I., S. Kom., M. Kom., Ph.D. selaku dosen pembimbing yang telah banyak memberikan arahan dan membimbing penulis dengan sabar dalam menyelesaikan penelitian tesis ini.
6. Bapak Tohari Ahmad, S. Kom, MIT., Ph.D., selaku pembimbing kedua yang telah memberikan arahan kepada penulis, Bapak Prof. Ir. Supeno Djanali, M.Sc., Ph.D., Bapak Dr. Eng. Radityo Anggoro, S. Kom., M.Sc. dan Bapak Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D. selaku

penguji yang telah memberikan banyak saran dan masukan yang membuat penelitian tesis ini menjadi lebih baik lagi.

7. Kementerian Riset, Teknologi, dan Pendidikan Tinggi yang telah memberikan kesempatan dan beasiswa tugas belajar (Beasiswa PasTi T.A. 2018) kepada penulis untuk melanjutkan pendidikan magister ini.
8. Direktur Politeknik Negeri Batam, yang telah memberikan izin dan rekomendasi kepada penulis untuk memperoleh kesempatan mendapatkan beasiswa tugas belajar pada pendidikan magister ini.
9. Pembantu Direktur I dan II Politeknik Negeri Batam, yang telah memberikan rekomendasi kepada penulis untuk memproleh kesempatan mendpatkan beasiswa tugas belajar pada pendidikan magister ini,
10. Seluruh dosen, staf laboratorium, staf tata usaha, dan karyawan departemen informatika ITS Program Pascasarjana Informatika Institut Teknologi Sepuluh Nopember.
11. Teman-teman seperjuangan tim *Software Defined Network*.
12. Teman-teman mahasiswa magister informatika angkatan 2018.
13. Teman-teman seperjuangan karyasiswa PasTi angkatan 2018.
14. Seluruh pihak yang tidak dapat disebutkan satu-persatu atas dukungan baik moral dan materil.

Semoga Allah ﷻ senantiasa memberikan limpahan karunia-Nya dan memberikan balasan pahala yang berlipat ganda kepada semuanya atas semua kebaikan yang telah diberikan kepada penulis.

Penulis menyadari bahwa laporan penelitian ini masih jauh dari kesempurnaan. Oleh sebab itu, saran dan masukan yang membangun sangat diharapkan untuk perbaikan dimasa yang akan datang. Semoga penelitian ini dapat memberikan manfaat bagi perkembangan ilmu
pengetahuan dan dapat memberi kontribusi bagi penelitian selanjutnya.

Surabaya. 23 Juni 2020

Mhd. Fattahilah Rangkuty

DAFTAR ISI

[Halaman ini sengaja dikosongkan]

DAFTAR GAMBAR

DAFTAR TABEL

[Halaman ini sengaja dikosongkan]

DAFTAR LAMPIRAN

[Halaman ini sengaja dikosongkan]

BAB 1
PENDAHULUAN

1.1 Latar Belakang

Software Defined Network (SDN) merupakan paradigma baru pada jaringan komputer yang memiliki kemampuan untuk mengatasi permasalahan yang muncul pada jaringan tradisional. SDN memisahkan logika jaringan *control plane* dan elemen dari *forwarding* data yaitu *data plane (router atau switch),* dengan menghadirkan fungsi kontrol jaringan berdasarkan representasi jaringan yang abstrak. SDN menggunakan kontroler terpusat, yang dapat menyediakan peluang besar dalam peningkatan performa jaringan. Fungsi jaringan diimplementasikan dengan menghapus keputusan kontrol, seperti perutean, dari perangkat perangkat keras dan mengaktifkan *flow* table yang dapat diprogram dalam perangkat keras menggunakan protokol standar, seperti *OpenFlow* (OF) (Joshi and Gupta, 2019). *Software Defined Networking* (SDN) menawarkan kemampuan dalam mengatur jaringan menjadi fleksibel dengan merealisasikan fungsi jaringan secara dinamis berdasarkan pandangan jaringan secara global (Guo *et al.*, 2018).

Kontroler merupakan salah satu komponen SDN yang menjadi pusat pengendali, penentu kebijakan jaringan, dan pengaturan *flow table* pada *switch* data plane, dengan menggunakan protokol *OpenFlow.* Kontroler juga berisikan aplikasi atau perangkat lunak yang mengendalikan trafik pada *switch data plane*. Melalui kontroler, *administrator* jaringan menguraikan aturan yang mendefinisikan perutean paket pada perangkat jaringan. *Switch OpenFlow*, misalnya, dapat berperilaku sebagai *load balancer* atau *firewall*, tergantung pada aturan yang didefinisikan pada kontroler oleh *administrator* jaringan. Semua komunikasi antara perangkat jaringan dan pengontrol dilakukan melalui protokol yang memiliki fitur keamanan, *OpenFlow.*

OpenFlow mengambil fitur dari sebagian besar *switch* dan *router ethernet* yang modern berisi *flow table* untuk fungsi-fungsi jaringan yang penting, seperti *routing, subnetting*, proteksi *firewall*, dan analisis statistik aliran data. Dalam *switch OpenFlow*, setiap entri dalam tabel aliran memiliki tiga bagian, "*header*" untuk

mencocokkan paket yang diterima, "*Action*" untuk menentukan apa yang harus dilakukan untuk paket yang cocok, dan "statistik" dari arus lalu lintas yang cocok (Xia *et al.*, 2015).

Menurut (Wang *et al.*, 2017) Salah satu model topologi yang digunakan pada jaringan SDN adalah *fat tree*, *fat tree* merupakan topologi jaringan tipikal *data center* yang banyak digunakan untuk *high availability* dan *scalability*. *fat tree* dapat mendukung jaringan *High Performance Computing* dengan model *cluster* dan pada jaringan *data center*, salah satu kelebihan *fat tree* adalah skalabilitas dan jalur yang bervariasi atau beragam, hal ini memberikan *alternative path* dalam *load balancing* antara *source* dan *destination*, untuk menghindari *congestion* dan peningkatan penggunaan *bandwidth.* Penelitian yang dilakukan oleh (Al-Fares, Loukissas and Vahdat, 2008) menggunakan arsitektur jaringan *fat tree* dengan teknik *multi path routing* untuk meningkatkan *bandwidth* pada suatu *cluster data center* dengan puluhan ribu komponen *switch* atau *router*. Sementara (Li and Pan, 2013) menyebutkan *fat tree* telah digunakan pada jaringan *data center* berbasis *OpenFlow* SDN untuk menerapkan *load balancing* dengan dukungan *multipath*, yang akan meningkatkan performa dan menurunkan *latency*. Setelah mempelajari penelitian diatas, dengan menggunakan topologi *fat tree* yang memberikan kelebihan dalam menyediakan alternatif *path* dalam melakukan transmisi data, penulis ingin melakukan penelitian menggunakan topologi *fat tree* menggunakan mekanisme seleksi *path* yang diharapkan dapat mendukung dalam peningkatan kinerja jaringan *data center*.

Menurut (Cui and Xu, 2016) Untuk mencapai suatu *load balancing*, diperlukan pendistribusian *load* secara merata pada beberapa *path*. *Best path* disini dapat diartikan sebagai *path* yang memiliki *load* paling kecil dari semua *path* yang ada, atau bisa juga disebut *path* terpendek antara *source* dan *destination host*. *Path* dengan *load* paling kecil atau *least loaded path* dapat dipilih menjadi jalur yang akan dilewati data sehingga akan meningkatkan *transfer rate data* dan mengurangi *latency*.

Algoritma *round robin* digunakan pada kontroler untuk mencegah *congestion* dan menyeimbangkan jaringan dengan membagi beban pada *data plane* secara merata dengan *time sharing*. Pada tahun 2018, (Neghabi *et al.*, 2018)

menjelaskan, untuk mencapai persyaratan *Quality of Service* (QoS), *Load Balancing* (LB) pada SDN harus dipertimbangkan untuk mengatasi sumber daya jaringan yang terbatas. Beberapa mekanisme LB telah ditinjau, mereka memberikan manfaat dan kelemahan mekanisme LB pada SDN secara sistematis berdasarkan pada dua faksi, deterministik dan non-deterministik. Salah satunya LB dengan menggunakan *round robin* yang mendistribusikan *load* secara merata sesuai dengan *time sharing*. Namun algoritma *round robin* memiliki kekurangan dalam pemilihan *path* atau *link* yang akan digunakan untuk mengirim data, sehingga dapat menimbulkan kemacetan jaringan atau *link overloading*. Seperti yang telah disebutkan oleh (Tiwari *et al.*, 2019) *round robin* dapat mengarahkan *traffic* ke destinasi yang lebih jauh dan tidak memperhatikan status *link* apakah sedang terjadi *overload*.

Penelitian yang dilakukan oleh (Liu *et al.*, 2015) meneliti penyeimbangan beban berbasis SDN untuk *data center* yang disebabkan oleh *elephant flow*. Mekanisme ini mengumpulkan seluruh status jaringan, kemudian membagi *elephant flow* melalui beberapa jalur, penelitian ini berfokus pada penyelesaian ketidakseimbangan jaringan dan masalah kemacetan jaringan di *data center* berbasis SDN. Namun, penelitian ini tidak mempertimbangkan kondisi *link* yang akan dipilih untuk dilalui oleh *elephant flow*, sehingga memungkinkan terjadinya *link overload*.

Pemilihan jalur dibutuhkan untuk mencapai *load balancing* pada jaringan *data plane*, oleh karena itu dibutuhkan suatu mekanisme dalam pemilihan jalur dalam pengiriman paket data untuk menghindari *link overloading*. Usulan yang kami ajukan adalah dengan metode pemilihan jalur atau *data path selection* untuk mencegah kemacetan atau *congestion*, pemilihan jalur yang kemudian disebut *best path*, *best path* dapat di kategorikan sebagai *path* yang memiliki beban paling rendah dalam hal ini menggunakan algoritma *least loaded path*. Pemilihan *Least Loaded Path* dapat dilakukan pada topology *tree* dan *fat tree*. Topologi *fat tree* memiliki kelebihan dalam skala jaringan besar yang dapat diterapkan pada *cloud data center* maupun *big enterprise*, *fat tree topology* juga dapat mendukung konektivitas antara banyak segmen jaringan yang berbeda, sehingga dapat meningkatkan skalabilitas dan kemanan jaringan. Dalam merancang *load*

balancing yang baik, penentuan *best path* atau *shortest path* dibutuhkan, untuk menetapkan jalur mana yang akan dilewati oleh trafik data. Dengan menggunakan *best path* dapat meningkatkan *transfer rate data* pada jaringan dan menghindari *high latency*. Penelitian terkait *load balancing* dengan secara dinamis telah dilakukan dengan memindahkan *load* ke *shortest path* ketika *load* tersebut lebih besar dari *bandwidth* yang tersedia. Hal ini perlu dilakukan karena, dengan memindahkan *load* tersebut pada *shortest path* dapat mempercepat proses *transfer data*, dimana jika tetap menggunakan *path* sebelumnya akan terjadi kemacetan jaringan karena ketidakseimbangan antara *bandwidth* dan *load* data (Bhandarkar and Khan, 2015).

Penelitian *load balancing* dengan menggunakan algoritma *Dijkstra* telah dilakukan, dengan menemukan *shortest path* dari satu titik ke titik yang lain, pada penelitian ini hanya mempertimbangkan jalur terpendek dalam melakukan *transfer data*, tanpa melihat banyaknya trafik yang sedang melewati jalur terpendek tersebut (Tiwari *et al.*, 2019).

Penelitian ini melakukan pengembangan mekanisme *path selection* untuk mencapai *load balancing* dengan algoritma *least loaded path* untuk mendapatkan *load balancing* pada jaringan SDN yang disimulasikan pada jaringan *fat tree topology*. Metode yang dirancang akan meningkatkan performa jaringan SDN, dalam hal peningkatan *transfer rate data*, penggunaan *bandwidth* yang efisien, dan mengurangi *latency*. *Least loaded path* akan bertindak reaktif pada *data plane*, dengan tidak menghiraukan *flow* ketika terdeteksi kelebihan beban pada peralatan jaringan. Proposal ini mengusulkan metode *selection path* untuk *load balancing* dengan *least loaded path* untuk memberikan alternatif dalam pemilihan mekanisme *load balancing*, terutama pada jaringan SDN dengan *fat tree topology*. Untuk pengaturan eksperimental, *controller Floodlight*, *emulator* jaringan *Mininet*, dan analisis paket *Wireshark* digunakan.

1.2 Perumusan Masalah

Rumusan masalah yang diangkat dalam penelitian ini adalah sebagai berikut :

1. Bagaimana pengembangan dan penerapan mekanisme *path selection* dengan *least loaded path* pada jaringan SDN *data plane* dengan topologi *fat tree*?
2. Bagaimana melakukan pemilihan *best path* dengan metode *least loaded path*?
3. Bagaimana pengaruh *network response time, link utilization, transfer rate* dan *bandwidth data* pada *path data plane* setelah dilakukan proses *load balancing* antara dua *host source* dan *destination* ?
4. Bagaimana pengaruh performa jaringan SDN *data plane* topologi *fat tree,* setelah dilakukan mekanisme seleksi *path* dengan LLP?

1.3 Tujuan Penelitian

Tujuan yang ingin dicapai dalam pembuatan penelitian ini adalah, untuk mengembangkan metode *load balancing* pada jaringan SDN *data plane* topologi *fat tree*, dengan menggunakan algoritma *least loaded path*, dalam menentukan *best path* dan *load balancing,* sehingga dapat memberikan peningkatan performa pada jaringan SDN *data plane*, dalam hal *latency, transfer data* dan *bandwidth*, mereduksi kemacetan pada jaringan, beban berlebih pada *data plane* dan efisiensi pada kontroler yang terpusat.

1.4 Manfaat Penelitian

Manfaat dari penelitian ini adalah memberikan pengembangan metode *load balancing*, ditunjukkan dengan performa jaringan SDN *data plane* yang lebih baik dengan ditandai meningkatnya pengukuran *transfer rate data* dan *bandwidth* serta penurunan *latency*.

1.5 Kontribusi Penelitian

Penelitian terkini tentang *load balancing*, memiliki hasil dan metode yang berbeda-beda, ada yang menggunakan algoritma umum seperti: *Round Robin*, *Random*, dan *Weighted Round Robin*. Metode *load balancing* sebelumnya masih

belum maksimal dalam memanfaatkan algoritma *round robin*, setiap *data plane* atau *switch* pada metode *round robin*, akan diberikan *load* yang seimbang, namun hal ini akan mengakibatkan *delay* dan *latency* yang tinggi ketika terdapat paket data yang dikirim dari *host* yang tidak ditentukan *best path* nya ke tujuan *host* dengan kata lain antara *host source* dan *destination* memiliki *hop count* yang tinggi. Penelitian ini mengembangkan metode untuk mengurangi permasalahan tersebut.

Kontribusi dari penelitian ini adalah pengembangan metode untuk mekanisme seleksi *path* pada jaringan SDN data plane dengan kontroler terpusat, dengan meng-ekstrak informasi topologi dan menghitung transmisi *rate* dan *receiving rate* data pada *switchport data plane* untuk menentukan *best path* untuk pengiriman paket data pada *source destination pair* sehingga akan mendapatkan penurunan *latency*, peningkatan *transfer rate* data dan *throughput* (*bandwidth)* sehingga dapat mencapai *load balancing*.

1.6 Batasan Masalah

Batasan masalah pada penelitian ini adalah :

1. *Source destination pair* yang digunakan dan diukur berjumlah 2 pasang S-D *host*.
2. Topologi yang digunakan adalah *fat tree topology*, kontroller yang digunakan merupakan kontroller *floodlight* terpusat.
3. Pengukuran pada *source destination* (S-D) *pair* menggunakan *ping*, untuk mengukur *latency*, kemudian menggunakan *iperf* untuk mendapatkan hasil *transfer rate* data, *bandwidth*.
4. Simulasi dilakukan dengan membandingkan hasil pengukuran *default* LB dengan mekanisme LLP.
5. Simulasi menggunakan parameter *traffic environment* antara lain, *link occupancy* (*low* 5% dan 25%, *medium* 50%, dan high 85% dan 95%). Kemudian menggunakan variasi *bandwidth* 10 Mbps dan 100 Mbps.
6. Metode pengembangan seleksi *best path* algoritma *least loaded path* menggunakan REST API dan *script python* yang akan berinteraksi dengan kontroller *floodlight*.

BAB 2
KAJIAN PUSTAKA

2.1 Dasar Teori

2.1.1 Software Defined Network

SDN atau *Software Defined Network* merupakan paradigma baru pada jaringan komputer. SDN dapat diartikan sebagai sebuah arsitektur jaringan komputer dimana pemisahan fungsi *control* dan *forwarding* pada perangkat jaringan seperti *switch* dan *router*. Fungsi *forwarding data* pada *data plane* dikelola oleh kontroler. Konsep SDN sebagai sebuah arsitektur jaringan komputer didefinisikan menjadi empat pilar (Kreutz *et al.*, 2015) :

1. *Control plane* dan *data plane* dipisah, fungsi kontrol pada perangkat jaringan dihilangkan, sehingga perangkat jaringan hanya berfungsi sebagai *forwarding data*.
2. Keputusan *forwarding* berdasarkan *flow*, bukan berdasarkan *destination*. *Flow* didefinisikan sebagai aliran paket-paket dalam jaringan dimana *header-header* paketnya memiliki nilai-nilai yang sama.
3. Kontrol logis dipindahkan ke entitas eksternal yang disebut SDN kontroler atau NOS (*Network Operatying System*). NOS merupakan *platform software* yang berjalan pada server dan menyediakan sumberdaya penting dan abstraksi untuk memfasilitasi pemrograman perangkat *forwarding* berdasarkan tampilan jaringan yang abstrak dan terpusat.
4. Jaringan dapat diprogram melalui aplikasi *software* yang berjalan diatas NOS, yang berinteraksi dengan perangkat *data plane*.

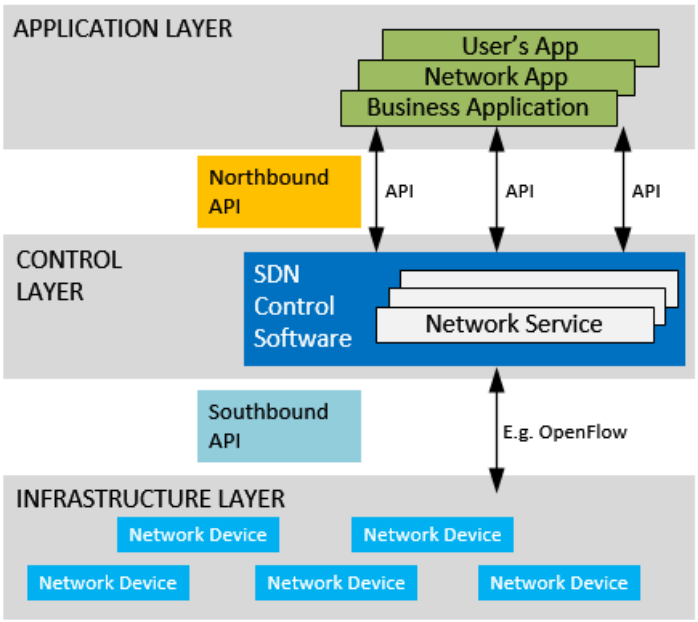
2.1.2 Arsitektur SDN

Arsitektur SDN seperti yang ditunjukkan pada Gambar. 2.1 secara umum terdiri dari tiga komponen:

1. *Application Layer*: Lapisan aplikasi terdiri dari aplikasi SDN, ini adalah program dengan perilaku komunikasi dan sumber daya yang diperlukan oleh pengontrol SDN melalui *Application Programming*

Interface (API). Antara *application layer* dan *control layer* berkomunikasi dengan *northbound* API. Ada beberapa jenis aplikasi yang dapat dikembangkan sesuai kebutuhan, seperti terkait dengan kebijakan dan keamanan jaringan, pemecahan masalah jaringan, konfigurasi dan manajemen jaringan, pemantauan jaringan, dan otomatisasi jaringan.

2. *Control Layer*: adalah area kontrol di mana logika cerdas yang dimiliki oleh kontroler SDN berfungsi untuk mengendalikan infrastruktur jaringan. Ini adalah area di mana setiap *vendor* perangkat jaringan bekerja untuk menghasilkan produk mereka sendiri untuk pengontrol dan *framework* SDN. Di lapisan ini, banyak *business logic* ditulis dalam pengontrol untuk mengambil dan memelihara berbagai jenis informasi jaringan, perincian status, perincian topologi, perincian statistik, dan banyak lagi.
3. *Infrastructure Layer*: terdiri dari berbagai peralatan jaringan yang membentuk lapisan jaringan bawah untuk meneruskan atau mem-*forward* trafik jaringan. Ini bisa berupa seperangkat *switch* dan *router* jaringan di *data center*. Lapisan ini akan menjadi fisik di mana virtualisasi jaringan akan diletakkan melalui lapisan kontrol (di mana kontroler SDN akan mengelola jaringan fisik dibawahnya).



Gambar 2.1 Arsitektur SDN ((Jiang *et al.*, 2014)

2.1.3 Control Plane

Control plane atau kontroler merupakan otak jaringan SDN yang berisi logika dasar dan mem-program *forwarding plane* dengan *southbound* API. Untuk melakukan komunikasi dengan *application Layer*, *control plane* berinteraksi dengan *southbound* API salah satunya menggunakan REST.

2.1.4 Forwarding Plane

Forwarding Plane, yang biasa disebut sebagai "*data path*", bertanggung jawab untuk menangani dan meneruskan paket. *Forwarding Plane* menyediakan fungsi *switching*, *routing*, dan *filtering*. Sumber daya dari *forwarding plane* termasuk tetapi tidak terbatas pada *filter*, pengukur, *markers* dan pengklasifikasi.

Perangkat Jaringan adalah entitas yang menerima paket pada *port*-nya dan melakukan satu atau lebih fungsi jaringan pada *port* tersebut. Misalnya, perangkat jaringan dapat meneruskan paket yang diterima, melakukan *drop*, mengubah *header* paket (atau muatan) dan meneruskan paket, dan seterusnya. Perangkat Jaringan adalah kumpulan beberapa sumber daya seperti *port*, cpu, memori dan antrian. Terdiri dari sumber daya sederhana atau dapat dikumpulkan untuk

membentuk sumber daya kompleks yang dapat dilihat sebagai satu sumber daya. Perangkat Jaringan itu sendiri merupakan sumber daya yang kompleks.

Perangkat jaringan dapat diimplementasikan dalam perangkat keras atau perangkat lunak dan dapat berupa elemen jaringan fisik atau *virtual*. Setiap perangkat jaringan memiliki *forwarding plane* dan *operational plane*.

2.1.5 Northbound API

Northbound API merupakan kumpulan fungsi-fungsi yang dapat melakukan komunkasi dua arah antara aplikasi dan *control plane,* hal ini memungkinkan dukungan *control plane* untuk melakukan *switching*, *routing*, *firewall*, dll.

2.1.6 Southbound API

Sama halnya dengan *Northbound* API, *Southbound* API dapat menangani komunikasi antara *control pane* dan *router dummy* menggunakan protokol standar *seperti OpenFlow.* Interaksi tingkat rendah antara *control plane* dan *forwarding plane* ini didefinisikan oleh *Southbound* API.

2.1.7 Management Plane

Management plane merupakan *layer* yang berisi aplikasi *software*. *Layer* ini menggunakan fungsi *Northboud* API dan dapat menyediakan berbagai fungsi jaringan, misalnya, *load balancers*, *monitoring* dan *routing*.

2.2 Studi Literatur

2.2.1 Protokol *OpenFlow*

Pada standar SDN, protokol *OpenFlow* didefinisikan sebagai salah satu protokol komunikasi yang memungkinkan kontroler SDN untuk berinteraksi dengan infrastruktur jaringan atau *switch* dari berbagai macam *vendor*. Dengan menggunakan protokol *OpenFlow*, jaringan dapat diadaptasikan menjadi lebih baik dengan melakukan pengaturan jaringan sesuai kebutuhan penggunanya. Kontroler SDN bekerja sebagai inisiator untuk melakukan instalasi *flow* pada *switch* atau

router untuk mengimplementasikan fungsi jaringan, seperti *forwarding* data, kontrol *flow*, dll.

OpenFlow adalah salah satu protokol dari standar kontroler SDN yang paling popular. *OpenFlow* dikembangkan oleh insinyur-insinyur dari universitas *Stanford*, yang memungkinkan bagi mereka untuk melakukan eksperimen pada jaringan kampus tanpa mengganggu konfigurasi jaringan yang sedang berjalan. *OpenFlow* memungkinkan *switch* membuat VLAN (*Virtual Local Area Networks*) pada konfigurasi jaringan yang sudah ada, dan memisahkan trafik yang bersifat uji coba (Mckeown *et al.*, 2008).

OpenFlow memungkinkan *switch* untuk memulai proses inisiasi dengan menghubungi kontroler melalui koneksi TCP. Untuk melakukan kontrol rute paket, juga dilakukan oleh *OpenFlow* dengan melakukan *traffic decisions* berdasarkan *flow* yang dapat menambah dan menghapus *flow* dalam bentuk *flow table* pada perangkat *switch*.

2.2.2 Flow Table

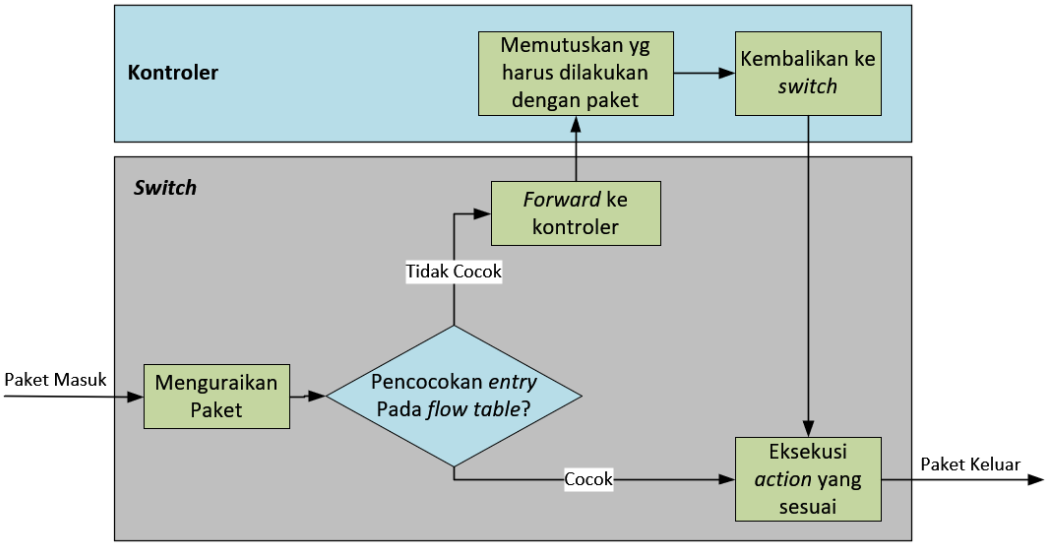
Flow table merupakan database yang berisikan *flow entries* yang berkaitan dengan *actions* untuk memerintahkan switch untuk menetapkan beberapa action pada flow tertentu (Marcial P. Fernandez, 2013).

Rule-rule ini akan menciptakan kondisi *match* dan *Action*. Sebuah *rule* yang memiliki semua nilai pada *field* nya disebut *exact rule* dan ketika tidak ada kondisi pada satu atau lebih *field* nya disebut *wildcard rule*. Kondisi *match* memutuskan *rule* mana yang akan diaplikasikan pada paket dan *Action* akan mendefinisikan bagaimana paket akan di tangani. Ketika paket sampai, *switch* akan mencocokkan *rule* pada paket dan yang memiliki prioritas paling tinggi *action* nya akan dipilih dan yang *match* akan di proses sesuai jalurnya. Jika tidak ada kondisi yang *match*, maka paket akan dikirim ke kontroler dalam bentuk pesan *OpenFlow*. Notifikasi ini disebut *event packet-in*, termasuk sebuah *prefix* dari isi paket yang sampai dengan beberapa tambahan Informasi (cukup dengan memasukkan *header* paket *Layer* 2, 3 dan 4), seperti *port* fisik dimana paket diterima. Dengan menerima notifikasi ini, kontroler akan memutuskan untuk menambahkan *rule* baru ke *flow*

table untuk menangani jenis paket seperti ini. *Match field* dari paket *OpenFlow* ditunjukkan pada gambar 2.2 berikut.

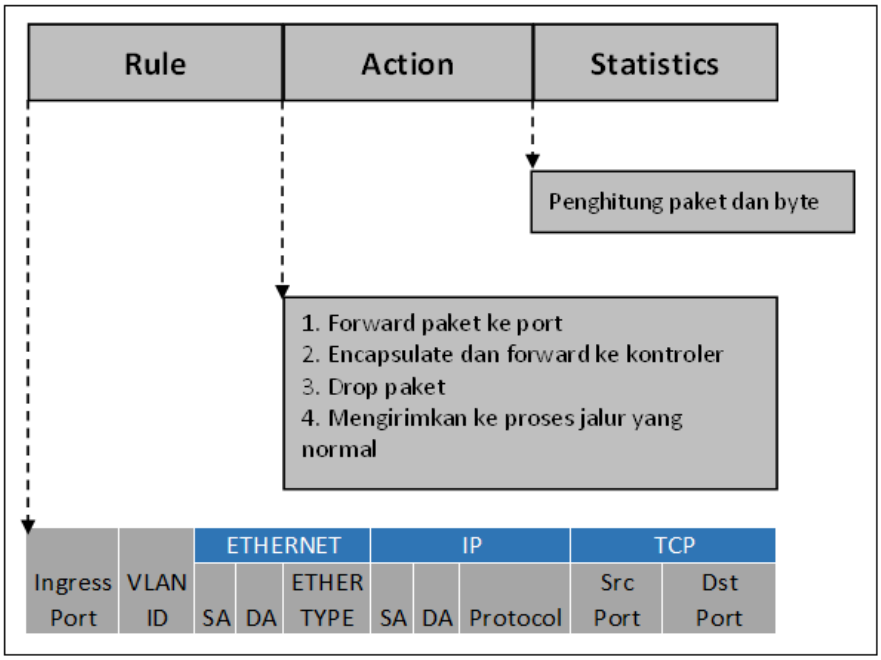
	ETHERNET			IP LAYER					TCP		UDP	
Ingress Port	SA	DA	ETHER TYPE	IP4 SA	IP4 DA	IPV6 SA	IPV6 DA	Protocol	Source Port	Destination Port	Source Port	Destination Port

Gambar 2.2 *Match field* paket *OpenFlow*

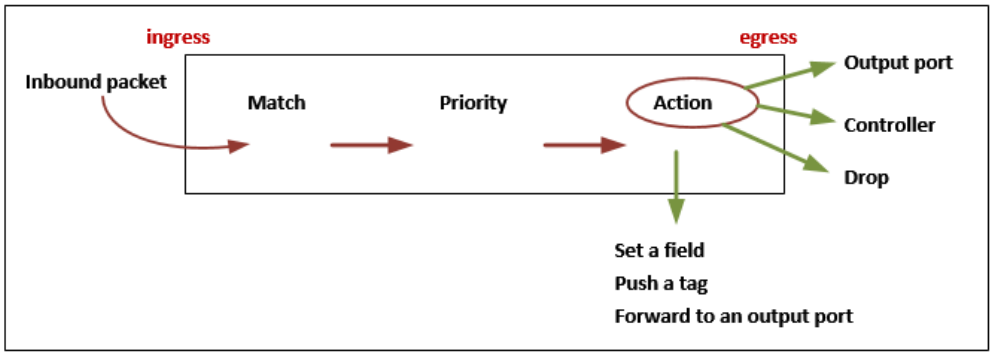


Gambar 2.3 Alur diagram *OpenFlow switch* dan kontroler

Terdapat tiga komponen dasar pada *entry flow OpenFlow*. *Rule* merupakan sebuah *field* yang digunakan untuk mencocokan *header paket*. Paket yang masuk pertama dicocokan dengan nomor *in-port* nya, VLAN ID hingga tujuan *port* TCP. User dapat mendefinisikan *flow entry* melalui kontroler dimana kemudian dimasukkan pada TCAM (*Ternary Content Addressable Memory*). Paket ini meng-enkapsulasi *Ehernet*, IP, TCP, dan protokol lainnya. Setiap kontroler, sebagai contoh *Ryu* dan *Floodlight* menangani enkapsulasi ini dengan cara yang berbeda tergantung bahasa pemrograman yang digunakan pada masing-masing kontroler. *Action*s didefinisikan oleh kontroler SDN yang harus diikuti oleh *switch*, ketika paket cocok atau sesuai dengan *rule*. *OpenFlow* juga memlihara perhitungan untuk *statistic*. Perhitungan ini akan terus meningkat pada *switch* yang kompatibel dan terus melakukan penghitungan jumlah dan ukuran paket yang melewati jaringan.



Gambar 2.4 Entri tabel *flow OpenFlow*



Gambar 2.5 Proses paket *OpenFlow* pada *switch*

Ketika sebuah paket memasuki dari *port* mana saja dari *switch*, paket tersebut akan melewati beberapa langkah tergantung dari *action* yang ditetapkan pada *flow table*. Packet *inbound* pertama dicocokkan ke *field* pada TCAM dan kemudian sesuai prioritas yang paling tinggi, *action* terkait akan diambil oleh logika *switch*. *Action* bisa berupa penetapan sebuah *field* pada paket atau membuat penanda pada *header*. *Switch* juga bisa melakukan *drop* atau *forward* ke *port* yang lain. Jika paket tidak didefinisikan pada *flow table*, maka paket tersebut dapat diarahkan ke kontroler untuk proses selanjutnya.

2.2.3 Tipe Pesan *OpenFlow*

Spesifikasi pada *OpenFlow* 1.3, pada dasarnya mendefinisikan tiga jenis pesan: *symmetric*, *asynchronous,* dan *controller-to-switch*. Pesan *symmetric* dibagi menjadi beberapa jenis yaitu:

1. *Hello*: bertukar pada saat dimulainya koneksi
2. *Echo*: menentukan *latency* dan *bandwidth* dari koneksi kontroler dan *switch*
3. *Experimenter*: menyediakan sebuah *standard* untuk menawarkan fungsionalitas tambahan atau sebagai dasar untuk revisi kedepannya

Untuk pesan *asynchronous,* dikirimkan oleh *switch* dengan jenis sebagai berikut:

1. *Packet-in*: pesan ini digunakan ketika paket yang sampai tidak memiliki kecocokan dengan *entry* manapun.
2. *Flow-removed*: pesan ini digunakan oleh kontroler untuk menghapus *routing flow* dari *switch*
3. *Port-status*: digunakan pada saat konfigurasi *status port*
4. *Error*: kontroler akan di berikan notifikasi oleh *switch* ketika terdapat *error*.

Kontroler dapat secara langsung mengirimkan pesan ke *switch* untuk mengatur kondisinya, tetapi bisa saja tidak membutuhkan respon dari *switch*, seperti pesan yang disebut *controller-to-switch*, berikut merupakan sebagian rangkumannya:

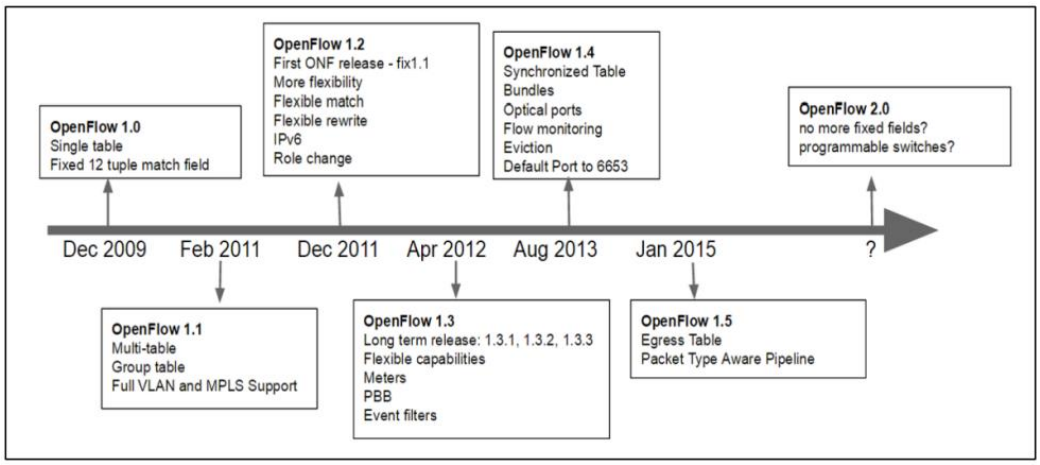
1. *Features*: kontroler dapat memahami kapabilitas dari *switch* dan respon dari *switch* dengan fitur *reply*
2. *Configuration*: tipe ini digunakan oleh kontroler untuk mengganti parameter konfigurasi pada *switch*.
3. *Packet-out*: tipe ini digunakan untuk mengirimkan paket keluar dari salah satu *port* pada *switch*. Pesannya harus berisikan beberapa baris *Action* yang harus diterapkan sesuai dengan urutan yang telah ditentukan.

2.2.4 OpenFlow Connections

Sebuah kontroler *OpenFlow* mengatur banyak *channel OpenFlow* secara simultan ke beberapa *switch*. *Beberapa channel* juga bisa diterapkan pada satu *switch* untuk tujuan reliabilitas. *Switch* terhubung ke kontroler dengan IP yang *fixed* atau statis dan menggunakan koneksi standard TCP atau TLS pada *port* TCP 6633. Terdapat pertukaran sertifikat antara dua perangkat, ketika memulai koneksi. *Switch* membedakan pessan yang masuk pada channel *OpenFlow* dengan trafik *internet*. Sebuah pesan *OFPT_HELLO* berisikan versi dari *OpenFlow* dipertukarkan antara kontroler dan *switch* ketika koneksi di jalankan. Jika versi nya tidak cocok yang lebih rendah dari keduanya yang mana dinegosiasikan sebagai versi yang utama untuk komunikasi selanjutnya, selain itu koneksi akan diputuskan jika tidak ada versi yang umum didukung oleh kedua perangkat. Jika terjadi interupsi pada koneksi antara *switch* dan kontroler, *switch* harus segera memasuki salah satu mode berikut yaitu, melakukan *drop* paket yang ditujukan untuk kontroler atau bertindak sebagai *switch* biasa.

2.2.5 OpenFlow Versions

OpenFlow memiliki siklus pengembangan yang aktif, dan berlanjut untuk ditingkatkan ke versi yang terbaru. Fitur seperti MPLS, monitoring *flow* dan egress *table* telah diperkenalkan pada versi utama.



Gambar 2.6 *Timeline* perkembangan *OpenFlow* (Ching-Hao, Chang and Dr. Ying-Dar Lin, 2015)

2.2.6 SDN Switch

Switch OpenFlow juga memelihara berbagai macam statistik trafik, termasuk setiap *byte port* dan penghitung paket dan setiap *byte flow* pada *entry table*. Setiap entry *flow table* dapat dikonfigurasikan dengan *timeout* yang *hard* dan *soft*. Karena keterbatasan kapasitas TCAM, sangat penting bagi kontroler jaringan untuk menggunakan kapasitas *flow table* secara efisien. Jika kontroler membutuhkan *rule* yang lebih banyak untuk *forwarding policy* nya dibandingkan kapasitas yang tersedia, maka kontroler tersebut harus melakukan proses pada paket untuk beberapa *flow* pada kontroler.

Switch pertama yang memiliki fungsi *OpenFlow* adalah *Open vSwitch*(OVS), yang diimplementasikan secara *software based* yang dapat menggunakan protokol *OpenFlow* untuk mentransmisikan data *flow table* dari kontroler ke perangkat keras *switch*ing. *OpenFlow* memungkinkan kontrol terprogram dan manajemen *interface* yang bebas dari vendor. *Open* v*Switch* dapat digunakan untuk mengatur jaringan untuk *virtual machine* pada *host* fisik yang sama atau meneruskan trafik ke *virtual machine* pada *host* fisik yang berbeda. *Open* v*Switch* dapat berjalan pada berbagai platform *virtual* berbasis UNIX seperti KVM, *Virtualbox* dan *Xen*.

2.2.7 Open vSwitch

Pengembang *OpenFlow* terlibat dalam menciptakan OVS, oleh karena itu OVS bersifat *open source*, virtual *switch* yang mendukung *multi-platform*. OVS merupakan alternatif dari *brdge native linux* dan antarmuka VLAN. OVS dirancang untuk bekerja pada lingkungan *virtual*. Untuk mencegah *looping* pada *Ethernet* LAN, OVS mengimplementasikan *caching flow* untuk mengurangi konsumsi dari sumberdaya *hypervisor*. Terdapat dua komponen pada *Open* v*Switch*, salah satunya bertugas sebagai manajer *layer* yang disebut *ovs-vswitchd* yang berada di posisi paling atas, yang kedua disebut OVS *user daemon* yang berada dibawah *ovs-vswitchd* dan berfungsi sebagai *forwarding plane*. *Ovs-vswitchd* berada pada *user space* dan berinteraksi dengan *user* dan mengumpulkan Informasi paket dan rute *flow*, sementara *user daemon* berkomunikasi dengan NIC atau *virtual* NIC pada mesin *virtual* dalam pengumpulan paket dan berada pada *kernel space*.

2.2.8 SDN Controller

Kontroler atau sistem operasi jaringan merupakan pusat dari pengendali yang menggunakan protokol *OpenFlow* untuk berinteraksi dengan *forwarding plane* yang ditangani oleh *switch*. Kontroler dapat melakukan pengaturan, pengendalian dan menyusun *flow table*. Kontroler-kontroler ini ditulis dalam perangkat lunak, sehingga menyediakan fleksibilitas dan dinamisme, namun memiliki kekurangan pada kecepatan dibandingkan kontroler perangkat keras. *Flow table* dapat diisi dengan *rule* sebelum aktifitas trafik data/sebelum paket dikirimkan (pro-aktif) atau setelah paket diterima (reaktiv). Terdapat beberapa pengembangan kontroler yang cukup bagus yang pada dasarnya sebuah perangkat lunak yang memerintahkan dan mengendalikan *switch OpenFlow*. Diantara kontroler open *source* yang populer antara lain *Pox*, *Nox*, *OpenDayLight*, *Floodlight*, Ryu dan beberapa kontroler populer yang berbayar antara lain Cisco APIC, HP VAN, VMware NSX. Semua kontroler tersebut memiliki GUI dan antarmuka REST API untuk terhubung ke *user space* dan menyediakan perintah-perintah untuk mengizinkan *routing* dan pengaturan *routing* melalui protokol *OpenFlow*.

Tabel 2.1 Perbandingan kontroler SDN

Name	Language	Developer	Learning Curve	Industrial Usage
OpenDaylight	Java	ONF	High	Yes
FloodLight	Java	Big Switch Network	Medium	Yes
Ryu	Python	Ryu Community/NTT	Medium	Yes
NOX	C++	Nicira	Low	Yes
POX	Python	Nicira	Low	Yes

2.2.9 Floodlight

Floodlight merupakan salah satu kontroler SDN yang bersifat open *source* dan berkinerja tinggi. *Floodlight* dibangun dengan bahasa Java. *Floodlight*

dikembangkan berdasarkan kontroler sebelumnya yaitu *Beacon*, sebuah kontroler eksperimental dari universitas *Standford*, dan saat ini didukung oleh banyak komunitas pengembang. Saat ini, *Floodlight* mendukung *OpenFlow* versi 1.4 (Spesifikasi 2013) dan dapat bekerja pada berbagai macam perangkat *switch* fisik dan *switch virtual* yang sudah mendukung *OpenFlow*. *Floodlight* sudah dilengkapi dengan aplikasi built-in *load balancer* untuk *flow ping*, TCP dan UDP. Pada dasarnya, aplikasi ini dibuat untuk pengembang aplikasi agar memahami cara pengembangan modul *Floodlight*, dan penggunaan dari API. Modul *load balance* dapat dikonfigurasi melalui sebuah REST API (Zhou *et al.*, 2014) yang mendukung pembuatan dasar VIP dan *Pools* dan menambahkan anggota pada pool tersebut. Namun, modul ini memiliki dua keterbatasan. Yang pertama menggunakan modul *Pusher Static Flow* (Ivancic *et al.*, 2014) yang menetapkan batas waktu *flow* adalah 0, ini berarti *flow* entry tidak dihapus setelah digunakan, dan menyebabkan *flow table* menjadi over*load* pada waktu tertentu. Kemudian keterbatasan yang kedua adalah modul ini menggunakan kebijakan *load balancing* statis sederhana seperti *Round Robin* dan Skema *Random*. Selain itu, fitur *monitoring* kinerja dan kondisi kontroler belum

2.2.10 Topologi Fat Tree

Dalam merancang jaringan *data center* yang dapat menyediakan *bandwidth* yang tinggi, dibutuhkan suatu desain jaringan yang efisien dalam interkoneksi antara banyak *server* pada *data center*, hal ini dapat dicapai dengan menggunakan *fat tree topology*, yang memiliki kelebihan dalam alternatif *path* untuk proses *routing* yang lebih bervariasi. Sehingga dapat memaksimalkan penggunaan *bandwidth* dan mendukung skalabilitas (Jo *et al.*, 2014).

Data center dibentuk dengan arsitektur jaringan hirarki dengan karakteristik *multi path* dikenal dengan topologi *fat tree*. Keberadaan rute *multi path* memfasilitasi dengan menyediakan rute yang berbeda untuk destination yang sama dan ini akan membantu dalam memberikan pilihan *load balancing* yang lebih baik (Askar, 2016).

Topologi *fat tree* pada umumnya digunakan pada jaringan *data center*. *Fat tree* digunakan karena link jaringan pada *level* atas topologi lebih banyak

dibandingkan dengan link pada *level* terbawah, sehingga dapat memberikan efisiensi pada pemilihan *routing* dan penggunaan teknologi yang lebih spesifik (Saisagar *et al.*, 2017).

2.2.11 SDN Emulator Mininet

Mininet adalah emulator jaringan yang menciptakan jaringan *host virtual*, *switch*, pengontrol, dan *link*. *Host Mininet* menjalankan perangkat lunak jaringan *Linux* standar, dan *switch-switch* yang mendukung *OpenFlow* untuk perutean kustom yang sangat fleksibel dan SDN.

Mininet mendukung penelitian, pengembangan, pembelajaran, pembuatan *prototype*, pengujian, *debugging*, dan tugas-tugas lain yang bisa mendapatkan manfaat dari memiliki jaringan eksperimental lengkap pada laptop atau PC lainnya.

2.2.12 Tools Benchmarking Jaringan

Pada penelitian ini, akan menggunakan beberapa aplikasi atau *tool benchmark* jaringan yang sudah sangat populer digunakan, terutama pada penelitian SDN, *Tool* yang digunakan merupakan aplikasi yang bersifat *open source* atau sudah termasuk dalam *pre installed* pada *mininet emulator*. Aplikasi *benchmark* tersebut antara lain:

1. *Iperf*

   Iperf adalah salah satu tool untuk mengukur *troughput bandwidth* dalam sebuah *link network*, agar bisa dilakukan pengukuran diperlukan *Iperf* yang terinstall point to point, baik disisi server maupun client. *Iperf* sendiri bisa digunakan untuk mengukur *performance link* dari sisi TCP maupun UDP.

2. *Wireshark*

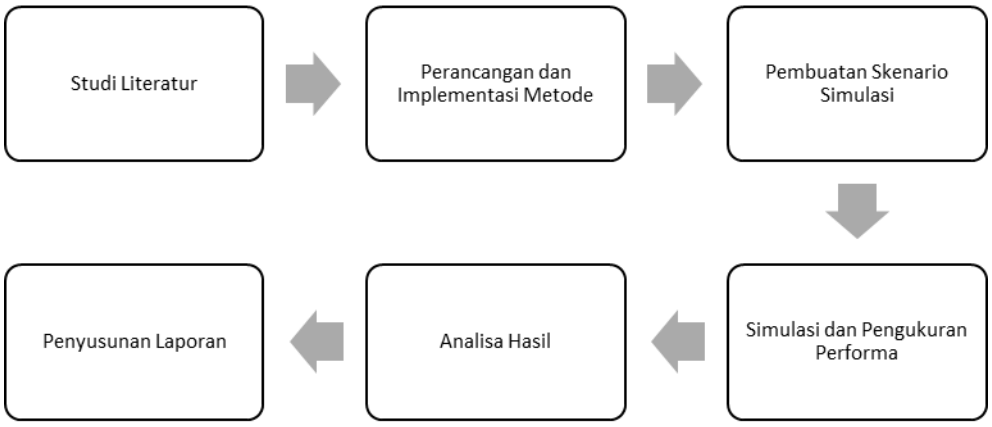
   Wireshark adalah salah satu dari sekian banyak tool *Network Analyzer* yang banyak digunakan oleh *Network Administrator* untuk menganalisa kinerja jaringannya dan mengontrol lalu lintas data di jaringan yang Anda kelola. *Wireshark* menggunakan *interface* yang menggunakan *Graphical User Interface* (GUI). *Wireshark* telah

menjadi *Network Protocol Analyzer* yang sangat terkenal dan telah menjadi standar di berbagai industri, dan merupakan sebuah proyek lanjutan yang dimulai tahun 1998.

BAB 3
METODE PENELITIAN

3.1 Tahapan Penelitian

Tahapan penelitian dibutuhkan agar tujuan yang diharapkan dari penelitian ini dapat tercapai. Pada Bab ini akan dijelaskan alur metodologi yang digunakan pada penelitian. Metodologi yang dilakukan terdiri dari studi literatur, perancangan metode yang diajukan, implementasi, pengukuran performa, analisa hasil, dan terakhir merupakan penyusunan laporan. Gambar 3.1. merupakan ilustrasi alur yang dilakukan.



Gambar 3.1 Diagram Alur Penelitian

3.2 Hasil Kajian Tentang Least Loaded Path dan Load Balancing

Tahap pertama yang dilakukan pada penelitian ini merupakan pengkajian literatur yang sesuai dengan topik penelitian. Literatur yang digunakan yaitu tentang studi pada algoritma *load balancing* SDN, teknik LB (*Load Balancing*) pada *data plane*, topologi *fat tree*, penentuan *best path*, dan metode untuk melakukan evaluasi pada metode yang diusulkan. Dari hasil studi literatur, kami mendapatkan informasi sebagai berikut:

A. Teknik *Load Balancing*

Pada SDN, terdapat beberapa teknik *load balancing* yang digunakan untuk penelitian, teknik *load balancing* menggunakan algoritma populer, antara lain:

1. Algoritma *Round Robin*

   Algoritma ini banyak dipakai pada *load balancing*, sering diimplementasikan pada *data center cluster based*, algoritma ini juga dapat digunakan pada jaringan SDN. Algoritma ini akan membagi *load* secara merata diantara semua *server* pada *server cluster*, ketika ada *request* baru pada *server*, algoritma ini akan mendistribusikan paket tersebut pada *server* dalam satu antrian secara berurutan. Algoritma ini memiliki kelebihan mudah dalam melakukan implementasi, dapat diprediksi, memiliki keseimbangan yang baik, bekerja sangat baik pada sumber daya *server-server* dengan spesifikasi yang sama. Namun memiliki kekurangan, dimana tidak memiliki prioritas, ketika ada *request* yang lebih penting, maka tidak akan diberikan prioritas khusus, selain itu algoritma ini juga dapat menimbulkan kemacetan jaringan ketika banyak request bersamaan pada *hop* count yang tinggi dari *source* ke *destination*. Algoritma ini juga tidak melakukan pemilihan *best path* untuk melakukan transmisi data.

2. Algoritma *Least Loaded Path*

   Algoritma ini merupakan pengembangan dari *Dijkstra* dalam menentukan *shortest path*, algoritma ini mencari jalur yang memungkinkan untuk dilewati berdasarkan *hop* yang sama, koneksi dengan *load* paling kecil akan dipilih untuk melewatkan paket. *Least loaded path* merupakan variasi dari algoritma *least connection*, berbeda dengan algoritma *least connection*, *least loaded path* mencari load terkecil tanpa melihat jumlah koneksi yang terjadi, namun menghitung cost minimum dari beban pada jalur tertentu. Sementara *least connection*, ketika terjadi *request* paket, *load balancer* akan memilih *data plane* dengan koneksi

paling sedikit meskipun jumlah paket atau *bandwidth* yang terpakai sudah cukup besar, hal ini berarti *least connection* tidak mempertimbangkan beban yang sedang berlangsung. Algoritma *least loaded path* merupakan algoritma yang bersifat dinamis, karena harus menghitung banyaknya *load* atau koneksi dari pada setiap jalur/*path*, ketika akan dilakukan proses *load balancing*.

B. Topologi *Fat Tree*

Untuk mendapatkan jaringan *data center* yang efisien, dapat menyediakan *bandwidth* yang tinggi, topologi *fat tree* dapat digunakan sebagai alternatif, dimana topologi ini memiliki kelebihan dalam penyediaan *path* atau jalur *routing* yang lebih bervariasi dan dapat memberikan skalabilitas pada jaringan.

C. Best Path

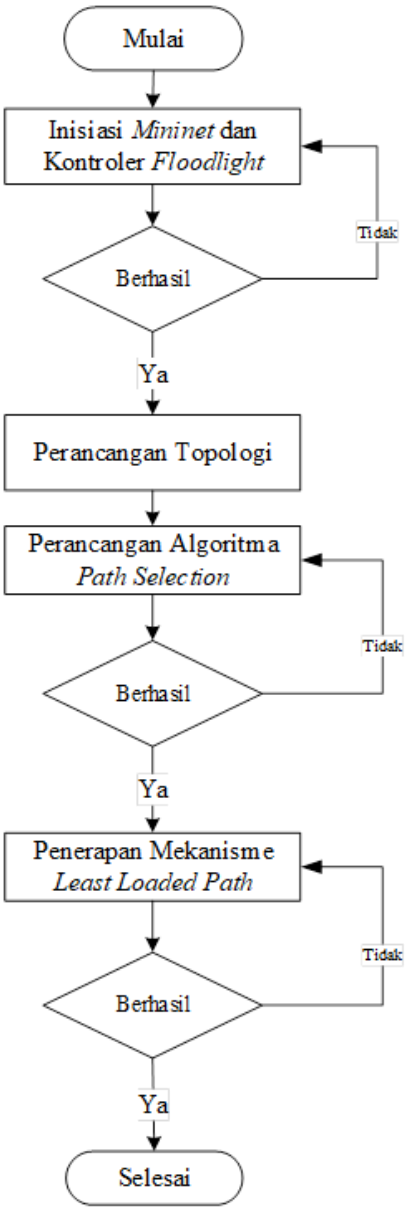
Best path merupakan salah satu bagian prosedur pada teknik *load balancing*. Pada metode *least loaded path*, *best path* akan ditentukan dengan mendapatkan *path* yang memiliki jalur terpendek dan *cost load* paling minimum.

D. Metode Evaluasi

Quality of Service (QoS) merupakan salah satu parameter untuk mengukur performa suatu jaringan, dengan melakukan pengukuran QoS, seperti l*atency*, *transfer rate data*, *throughput* atau *bandwidth* dapat diketahui sejauh mana kinerja atau peningkatan suatu jaringan.

3.3 Perancangan Sistem dan Implementasi Metode

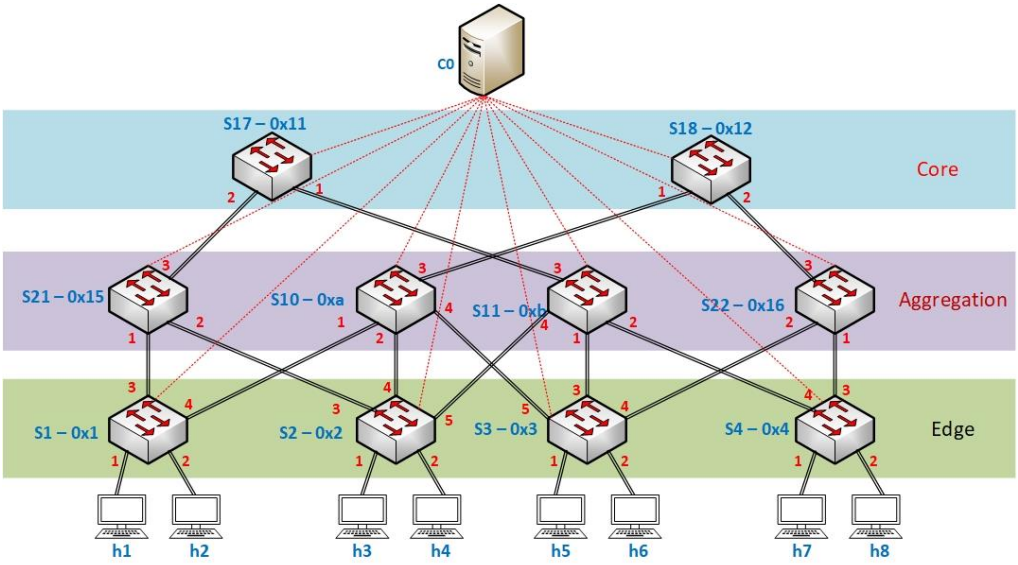
Perancangan sistem merupakan tahapan penelitian yang dilakukan untuk membangun sistem penelitian setelah melakukan studi literatur. Tujuan dilakukan perancangan sistem agar implementasi metode berjalan sistematis dan terstruktur. Perancangan sistem dapat dilihat pada Gambar 3.2.



Gambar 3.2 Diagram Alur Perancangan Sistem

3.3.1 Perancangan Topologi

Pada penelitian ini, dua skenario akan diuji menggunakan topologi *fat tree* dengan masing-masing dua host S-D. Topologi yang digunakan seperti ditunjukkan pada gambar 3.3.



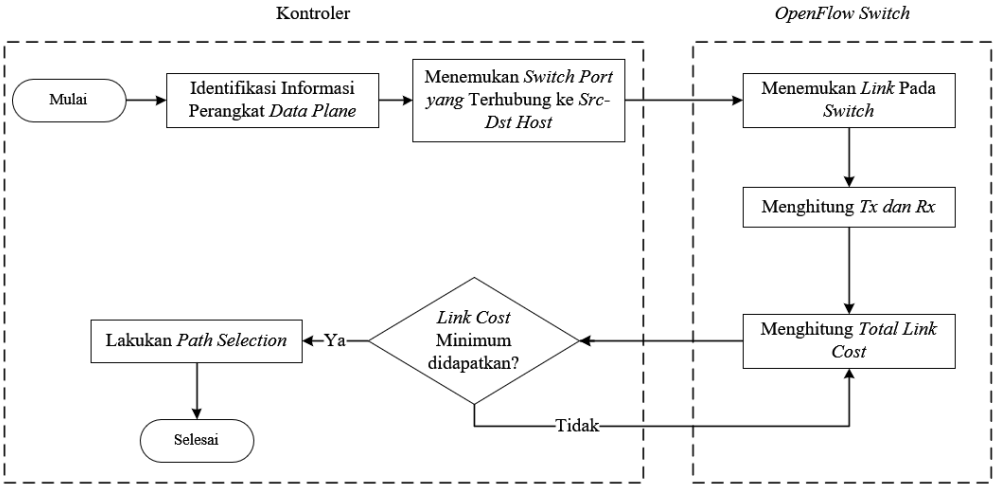
Gambar 3.3 Topologi *fat tree*

Topologi dibuat menggunakan *python script*, membentuk topologi *fat tree*, yang terdiri dari 10 o*penflow* (OF) *switch* dengan 3 *layer*, *core, aggregation*, dan *edge*. Pada *layer edge* OF *switch*, masing-masing *switch* terhubung dengan dua *host*. Selanjutnya topologi ini akan disimulasikan dengan menggunakan *emulator mininet*, dimana sebelumnya sudah dijalankan kontroler *floodlight*, untuk menghubungkan semua perangkat. Kontroler akan mengirimkan *flow* pada *data plane* setelah dilakukan mekanisme *data path selection* dengan metode *least loaded path*.

3.3.2 Perancangan Algoritma

Alur perancangan algoritma seleksi *path* dengan *Least Loaded Path* (LLP) secara keseluruhan dapat dilihat pada Gambar 3.4. Alur tersebut akan menjelaskan modifikasi *Dijkstra* dalam penetuan *best path* sebagai dasar untuk *data path selection*. Algoritma LLP merupakan pengembangan algoritma *Dijkstra*, selain menentukan *shortest path*, algoritma ini juga memiliki mekanisme menentukan *path* dengan *load* terkecil, yang didapat dari penjumlahan Tx (*Transmission Rate*) dan Rx (*Receiving Rate*) data atau yag disebut *cost*. *Cost* ini didapatkan dari *switchport* yang sudah terhubung dengan S-D host yang akan dilakukan pengujiannya. Untuk mendapatkan *cost* pada *switchport*, *traffic* data harus

dikirimkan dari *source host* ke *destiation host*, sehingga algoritma LLP dapat menghitung *path* mana yang meimiliki *cost* paling terkecil. Untuk *flowchart* penentuan *minimum cost* dapat dilihat pada Gambar 3.4.

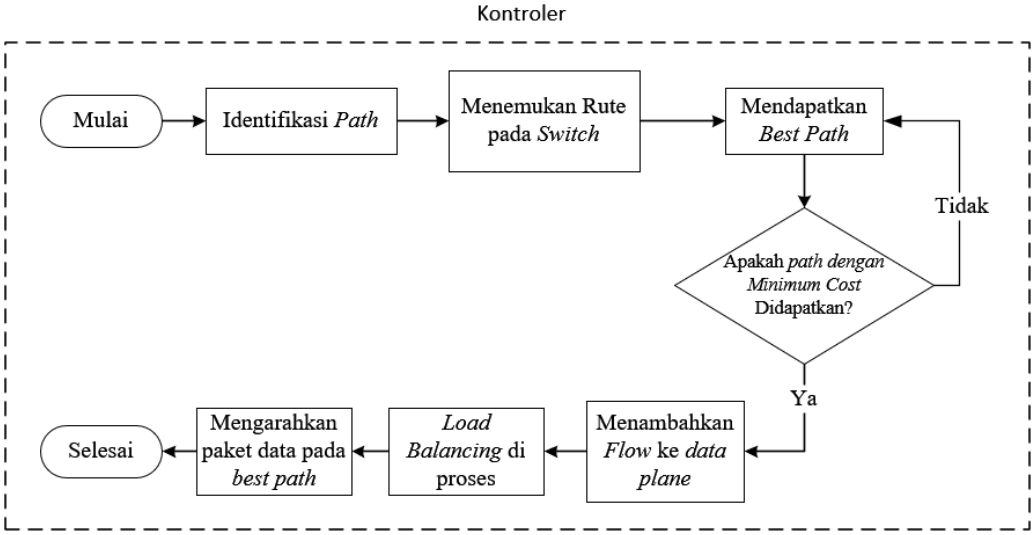


Gambar 3.4 Flowchart *Path Selection*

Setelah didapatkan *minimum link cost* pada *path*, kemudian dilakukan *path selection*, berupa *route* yang akan digunakan untuk melakukan transmisi paket data. kemudian kontroler akan menambahkan *flow* pada *switch* yang telah terpilih *path* nya, selanjutnya paket data dapat di transmisikan dan akan di diukur apakah *path* tersebut dapat meningkatkan transmisi data dari jaringan *data plane*.

3.3.3 Perancangan Algoritma Seleksi *Path* dengan LLP

Pada proses penentuan *best path*, menggunakan algoritma *Least Loaded Path*, metode ini mencari beberapa *path* atau beberapa jalur yang memungkinkan untuk mencapai *destination* pada *hop* yang sama. Jalur atau *path* dengan *port switch OpenFlow* yang memiliki *cost* paling minimum akan dipilih untuk melakukan transmisi paket. Algoritma ini dapat menjadi dinamis, karena harus menghitung *cost* dari *link* setiap akan menentukan *path* yang akan digunakan dalam mentransmisikan data. Alur diagram penentuan *path selection* dengan *least loaded path* dapat dilihat pada Gambar 3.5.



Gambar 3.5 *Flowchart load balancing* dengan *best path*

Pada proses awal, kontroler akan melakukan identifikasi *path*, hasil inisiasi topologi dengan emulator mininet. Kontroler akan mencari informasi terkait *switch*, *MAC Address*, *switchport*, dan *host port* yang sudah terhubung. Kemudian kontroler akan mencari setiap *route* yang memungkinkan untuk dilewati oleh S-D *host*. Untuk mendapatkan *best path* dengan *least loaded path*, perlu dilakukan pengiriman paket data dari S ke D *host*, agar kontroler mendapatkan nilai *Tx* dan *Rx*, yang kemudian akan dijumlahkan, *switchport* atau *path* dengan minimum *cost* akan dipilih untuk mentransmisikan data. Setelah *path* dengan minimum *cost* didapatkan, kontroler akan mengirimkan *flow* ke *flow table data plane* atau *switch*, Informasi seperti *In-Port*, *Out-Port*, *source* IP, *Destination* IP, *source* MAC, *Destination* MAC diberikan/diinput pada *flow*. Flow ini akan dijadikan sebagai *rule* untuk setiap paket data yang akan dikirim dari S ke D.

Setelah dilakukan *load balancing*, akan dilakukan capture packet pada path yang akan dilalui oleh paket data untuk memastikan penggunaan *path* menjadi *balance*, kemudian pengukuran dilakukan antara S-D *host* untuk mendapatkan hasil performa jaringan SDN topologi *fat tree* dan melihat apakah mekanisme LLP sudah bekerja dengan baik. Untuk melakukan pengukuran *transfer rate* data dan *bandwidth* menggunakan *tool iperf*. Pada saat pengukuran, *host* A sebagai *iperf* client dan *host* B sebagai *iperf* server. Untuk mengukur *latency* menggunakan

perintah *ping*, yang dieksekusi dari S ke *D*. Pengukuran ini dilakukan secara bergantian dan dilakukan beberapa kali, untuk mendapatkan hasil yang aktual.

3.4 Perancangan Skenario

Pada penelitian ini akan dilakukan 4 skenario pengujian, yang pertama, skenario topologi *fat tree* dengan S-D h1 dan h4, kedua, topologi *fat tree* dengan S-D h5 dan h8, ketiga skenario topologi *fat tree* dengan S-D h1 dan h4 tanpa *link occupancy*, keempat, skenario pengujian *path balance* antara S-D h1-h4, S-D h1-h3 dengan menggunakan *bandwidth* 10 Mbps dan 100 Mbps. Untuk skenario satu dan dua, menggunakan *network link environment* yang hampir sama, S-D h1 dan h4 menggunakan *link occupancy* atau *network link load status*, dengan parameter *low occupancy* 25% dari total *bandwidth*, *medium occupancy* 50% dari total *bandwidth*, dan high *occupancy* 85% dari total *bandwidth* 10 Mbps dan 100 Mbps. Sementara untuk h5 dan h8 hanya menggunakan dua parameter *link occupancy*, yaitu *low occupancy* 5% dari total *bandwidth* dan *high occupancy* 95% dari total *bandwidth* 10 Mbps. Penggunaan *link occupancy* ini dilakukan untuk mensimulasikan kondisi *path congestion* seperti kondisi *real*. *Penggunaan link occupancy* ini diharapkan dapat melihat kapabilitas mekanisme LLP terhadap *link condition* yang bervariasi, mulai kondisi *low traffic* hingga *high traffic* (*link overloading*). Parameter yang digunakan selanjutnya adalah dengan menggunakan alokasi *bandwidth* yang berbeda, yaitu 10 Mbps dan 100 Mbps. Hal ini diharapkan dapat mensimulasikan kondisi *bandwidth* pada jaringan *real*. Penambahan alokasi *bandwidth* yang berbeda diharapkan dapat memberikan hasil evaluasi terhadap kemampuan mekanisme LLP dalam mentransmisikan data dengan ketersediaan *bandwidth* yang bervariasi. Untuk skenario ketiga, akan dilakukan pengujian pada S-D h1-h4 tanpa menggunakan *link occupancy* dan alokasi *bandwidth*, untuk melihat kinerja mekanisme LLP pada lingkungan *link* secara *homogen*. Sementara skenario keempat, sebagai pengujian terakhir melakukan pengujian pada S-D h1, h3 dan h4 untuk melihat apakah *path balance* dapat dicapai, dengan menggunakan *bandwidth* 10 Mbps dan 100 Mbps. Penggunaan empat skenario ini dianggap dapat memberikan hasil pengujian apakah penggunaan algoritma LLP mampu mencapai

tujuan penelitian yang akan dilakukan yaitu *network balancing*, dan dapat melihat perbandingan dengan menggunakan parameter-parameter yang berbeda.
Untuk penjelasan keempat skenario sebagai berikut:

3.4.1 Skenario S-D h1-h4

Perancangan skenario 1 atau S-D h1-h4 menggunakan topologi *fat tree*, langkah pertama adalah menjalankan kontroler *floodlight* sebelum melakukan inisiasi topologi *fat tree* dengan *mininet*. Setelah topologi *fat tree* dijalankan, lakukan pengujian konektivitas antar *device* apakah sudah saling terhubung dengan perintah "*pingall*". Lakukan pengukuran latensi, *transfer data* dan *throughput* sebagai bahan evaluasi hasil dari penggunaan *default* LB pada kontroler *floodlight* dengan menggunakan parameter *link load status* dan *alokasi bandwidth* yang berbeda. Kemudian lakukan pengiriman paket data dari h1 ke h3 dan h4 yang dianggap sebagai kemacetan jaringan dan sebagai statistik dari *switchport* untuk perhitungan *Tx* dan *Rx*. Untuk memastikan paket data sudah diterima pada *switchport*, lakukan *capture paket* dengan *wireshark*. Kemudian jalankan mekanisme pemilihan *path* dengan *least loaded path*. Setelah *path* terpilih, lakukan pengiriman paket data dan lakukan pengukuran latensi, *transfer data* dan *throughput* sebagai bahan evaluasi. Hasil kedua pengukuran tersebut akan dibandingkan untuk melihat apakah performa LLP lebih baik dari *default* LB pada *floodlight*. Hasil evaluasi ini akan menentukan apakah mekanisme ini dapat melakukan network balancing sesuai tujuan penelitian.

3.4.2 Skenario S-D h5-h8

Perancangan skenario 2 atau S-D h5-h8 menggunakan topologi *fat tree* sebagai simulasi nya, setelah menjalankan skenario 1, seluruh komponen simulasi harus dilakukan *reboot*, kemudian diawali dengan menjalankan kontroler kembali, inisiasi topologi *fat tree* dengan *mininet*, pengujian konektivitas semua *device* dan melakukan pengukuran latensi, *transfer data* dan *throughput* sebagai bahan evaluasi hasil dari penggunaan *default* LB pada kontroler *floodlight* dengan menggunakan parameter *link load* status dan alokasi *bandwidth* yang berbeda.

Kemudian lakukan pengiriman paket data dari h5 ke h7 dan h8 yang dianggap sebagai kemacetan jaringan dan sebagai statistik dari *switcport* untuk perhitungan *Tx* dan *Rx*. Untuk memastikan paket data sudah diterima pada *switchport*, lakukan *capture paket* dengan *wireshark*. Kemudian jalankan mekanisme pemilihan *path* dengan *least loaded path*. Setelah *path* terpilih, lakukan pengiriman paket data dan lakukan pengukuran latensi, *transfer data* dan *throughput* sebagai bahan evaluasi. Hasil kedua pengukuran tersebut akan dibandingkan untuk melihat apakah performa LLP lebih baik dari *default* LB pada *floodlight*. Hasil evaluasi ini akan menentukan apakah mekanisme ini dapat melakukan *network balancing* sesuai tujuan penelitian.

3.4.3 Skenario S-D h1-h4 tanpa *link load* dan alokasi *bandwidth*

Perancangan skenario 3 atau S-D h1-h4 tanpa menggunakan *link occupancy* atau *link load* dan tanpa alokasi *bandwidth* juga menggunakan topologi *fat tree*, setelah menjalankan skenario 2, semua proses simulasi harus dilakukan *reboot*. kemudian diawali dengan menjalankan kontroler kembali, inisiasi topologi *fat tree* dengan *mininet*, pengujian konektivitas semua *device* dan melakukan pengukuran latensi, *transfer data* dan *throughput* sebagai bahan evaluasi hasil dari perbandingan *default* LB dan mekanisme LLP pada kondisi jaringan tanpa *link load* dan tanpa alokasi *bandwidth* atau lingkungan jaringan *homogen*.

3.4.4 Skenario S-D h1-h3-h4 *path balance*

Untuk skenario 4 atau S-D h1-h3 dan h1-h4 dilakukan untuk melihat apakah *path* menjadi *balance* setelah mekanisme LLP. Skenario ini tidak menggunakan *link occupancy*, namun diberikan batasan *bandwidth* 10 Mbps dan 100 Mbps. Skenario ini masih menggunakan topologi *fat tree*. setelah menjalankan skenario 2, semua proses simulasi harus dilakukan *reboot*. kemudian diawali dengan menjalankan kontroler kembali, inisiasi topologi *fat tree* dengan *mininet*, pengujian konektivitas semua *device* dan melakukan pengukuran *transfer rate* data secara bersamaan antara h1-h3 dan h1-h4, dan melihat bagaimana hasil *transfer rate* data pada kedua path setelah dilakukan mekanisme LLP.

Dari keempat skenario diatas, hasilnya akan dilakukan perbandingan untuk dilakukan analisa dan evaluasi, bagaimana kinerja mekanisme yang diusulkan dengan LB *default* pada kontroler *floodlight.*

3.5 Pengujian

Tahap pengujian pada penelitian ini dilakukan untuk menunjukkan bahwa mekanisme *network balancing* dengan LLP dapat berjalan dengan baik sesuai dengan perancangan dan skenario pengujian. Penelitian ini mengusulkan metode *least loaded path* (LLP) dengan mencari *best path* untuk mencapai *network balancing* atau peningkatan performa pada jaringan SDN *data plane*.

3.5.1 Lingkungan Simulasi

Dalam melakukan uji coba performa jaringan SDN topologi *tree* dan *fat tree* dengan penerapan *load balancing*, menggunakan *Laptop Intel core* i5 2.60GHz, RAM 8GB, dan OS Ubuntu 16.04.6 LTS. Spesifikasi detail perangkat keras dapat dilihat pada Tabel 3.1 Lingkungan Simulasi.

Tabel 3.1 Lingkungan Simulasi

Komponen	Spesifikasi
Sistem Operasi	Ubuntu 16.04.6 LTS
CPU	Intel core i5-2410M 2.30GHz
RAM	2x4 GB DDR3
Penyimpanan	80GB

Sedangkan perangkat lunak yang digunakan pada penelitian ini adalah:

1. *Mininet* versi 2.2.2, aplikasi untuk menjalankan simulasi topologi dan *host* yang akan diuji.
2. *Open vSwitch* versi 2.0.2, berfungsi sebagai *switch virtual* pada topologi yang diuji.
3. *Floodlight* versi 1.2, berfungsi sebagai kontroler yang me-*manage switch* atau *data plane* pada saat uji coba.

4. *Python* versi 2.7.6, bahasa pemrograman yang digunakan untuk membuat *script* topologi dan *load balancing*.
5. *Wireshark* versi 2.6.10, untuk memeriksa paket yang lewat pada *path* atau *switchport* pada saat simulasi berjalan.
6. *Iperf* versi 2.0.5, aplikasi atau perintah untuk mendapatkan data *transfer rate* atau *bandwidth* antara dua *host* yang diuji.
7. *Jperf* versi 2.0.2, aplikasi yang digunakan untuk melihat *path balance* pada jalur S-D *host*.
8. *Ping*, perintah untuk mendapatkan latensi antara dua *host* yang diuji.
9. *Hping3* versi 3.0.0-alpha-2, aplikasi atau perintah untuk mensimulasikan kemacetan jaringan pada *host* yang diuji.
10. *Vim* versi 7.4.1689, sebagai editor kode dalam proses perancangan topologi, load balancing dan pengumpulan data.
11. *Gnuplot* versi 5.0, sebagai renderer grafik atau chart dari data yang sudah dikumpulkan.

3.5.2 Parameter Evaluasi

Parameter evaluasi dalam pengujian seleksi *path* dengan metode *least loaded path*, sudah ditentukan dengan mempertimbangkan batasan masalah, parameter evaluasi antara lain:

1. *Round Trip Time Latency*

   Round Trip Time (RTT) Latency dihitung untuk mengetahui perubahan dari kecepatan transfer paket secara *round trip time* (RTT), penjumlahan waktu paket data diterima pada *destination* dengan waktu pengiriman paket kembali ke *source*. Pengukuran RTT *latency* menggunakan *tool ping*. Pengukuran *latency* digunakan untuk analisa hasil *network response time* pada kondisi *link* yang berbeda. Persamaan (3.1) merupakan penghitungan RTT *latency*.

$$Latency\ RTT\ (ms) = T_{Rcvd}Dst + T_{Rcvd}Src \quad (3.1)$$

Dimana :

$Latency\ RTT$ = Jumlah waktu transmisi paket dalam *millisecond* (ms)

$T_{Rcvd}Dst$ = Waktu paket diterima pada *host destination*

$T_{Rcvd}Src$ = Waktu paket diterima kembali pada *host source*

2. *Transfer Rate Data*

   Transfer rate data merupakan jumlah paket data yang terkirim dari *source* ke *destination* pada satu waktu. *Transfer rate* data menggunakan satuan *Megabytes* (MB) atau *Gigabytes* (GB). Pengukuran *transfer rate data*, dilakukan untuk menilai performa *data link* atau *path* dengan menggunakan *tool iperf* pada saat proses simulasi.

3. *Throughput*

   Throughput atau *bandwidth utilization* , pengukuran *bandwidth utilization* diperlukan agar diketahui performa *data link* atau *path* pada saat penggunaan *default* LB dan pada saat penggunaan mekanisme LLP. Tool *iperf* dapat digunakan untuk mengukur *bandwidth utilization* pada saat proses simulasi dijalankan. Untuk menghitung nilai *throughput* dapat menggunakan persamaan (3.2) sebagai berikut:

$$Throughput\ (mbps) = \frac{Jumlah\ data\ yang\ dikirim}{Waktu\ pengiriman\ data} \tag{3.2}$$

4. Jumlah *packet loss* pada saat *transfer data* setelah dilakukan *load balancing*, hal ini akan menentukan *buffer utilization*. Untuk menghitung nilai *packet loss* yang terjadi antara S-D *pair* adalah sebagai berikut:

$$PLoss\ (\%) = \frac{Packet_T - Packet_R}{Packet_T} \times 100\% \tag{3.3}$$

Dengan:

$Packet_T$ = Packet yang dikirimkan

$Packet_R$ = Paket yang diterima

3.6 Analisa dan Hasil Pengujian

Dalam melakukan analisa hasil pengujian, dilakukan dengan menganalisa parameter seperti *latency*, *transfer rate* data dan *throughput*, ditambah hasil akhir dari *network balancing* setelah proses seleksi *path* dengan LLP. Analisa dilakukan dengan membandingkan dua pasangan S-D *host* yaitu h1-h4 dan h5-h8, menggunakan LB *default floodlight* yang menggunakan algoritma *round robin* dan mekanisme seleksi path dengan metode *Least Loaded Path* (LLP).

3.7 Dokumentasi dan Jadwal Penelitian

Proses dokumentasi digunakan untuk penulisan laporan hasil penelitian yang dilakukan. Setiap tahapan dalam proses penelitian juga disertakan dalam laporan yang ditulis. Kegiatan dokumentasi ini akan dicantumkan pada jadwal penelitian yang direncanakan mulai bulan Januari 2020 hingga Mei 2020 yang secara rinci terlihat pada Tabel 3.2.

Tabel 3.2 Jadwal Kegiatan Penelitian

No.	Kegiatan	Bulan																			
		Januari				Februari				Maret				April				Mei			
1.	Studi Literatur																				
2.	Analisa dan Perancangan																				
3.	Implementasi																				
4.	Pengujian dan Evaluasi																				
5.	Penyusunan Buku Tesis																				

BAB 4
HASIL DAN PEMBAHASAN

Dari hasil uji coba, dengan menggunakan 2 skenario pasangan S-D h1-h4 dan h5-h8, terdapat kenaikan yg signifikan pada *latency*, *transfer rate data* dan *throughput*. Hal ini menunjukkan proses *seleksi path* telah menunjukkan peningkatan performa pada jaringan SDN *data plane*. Selain itu, hasil ini menunjukkan ketika *path* yang dipilih menjadi *best path*, maka kontroler akan mengirimkan *flow* ke *flow table* kepada *switch* atau *data plane* yang terlibat dan sudah terpilih menjadi *best path*, sehingga setiap *host* yang akan mengirimkan paket data, akan mengikuti rule yang sudah tercatat pada flow table *data plane*. Hal ini menjadikan mekanisme ini menjadi reaktif, dimana ketika terdapat path yang baru, kontroler baru mengirimkan *flow* ke *data plane*.

4.1 Analisa Hasil

Analisa hasil penelitian ini dibagi menjadi dua, yaitu hasil simulasi skenario S-D h1-h4 dan hasil simulasi skenario S-D h5-h8 dengan menggunakan topologi yang sama yaitu *fat tree*. Setiap hasil simulasi mencakup pembahasan tentang analisa hasil dari penggunaan *default* LB pada *floodlight*, dan implementasi *least loaded path* serta metrik QoS yang dibandingkan antara lain *transfer rate* data (*Kilobytes, Megabytes*), *bandwidth* (Kbps, Mbps) dan *latency* (*Millisecond*) pada kondisi jaringan *low, medium*, dan *high traffic* ditambah pembatasan alokasi *bandwidth* 10 Mbps dan 100 Mbps. Untuk S-D h5-h8 kondisi jaringan atau *network load*, kondisi *low* diturunkan hingga 5%, dan kondisi *high* menjadi 95%. Serta penggunaan *bandwidth* yang sama yaitu 10 Mbps dan 100 Mbps.

4.1.1 Hasil uji coba skenario S-D h1-h4 dengan *bandwidth* 10 Mbps

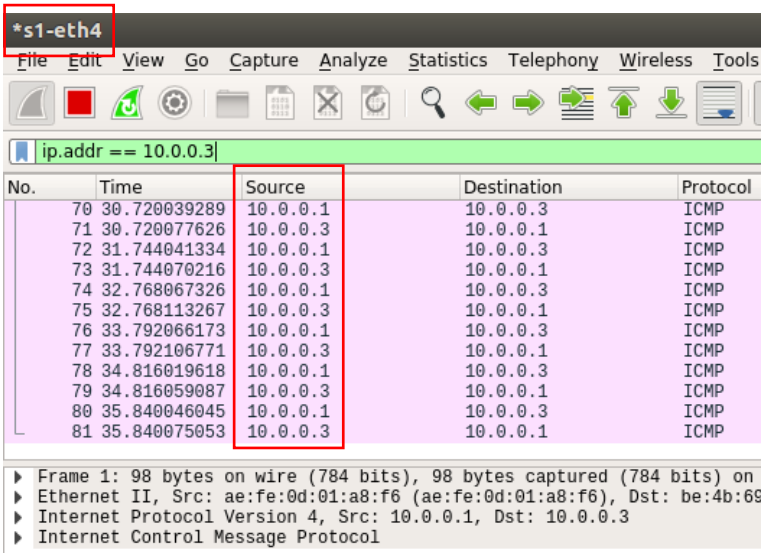
Skenario S-D h1-h4 dengan bandwidth 10 Mbps menggunakan topologi jaringan *fat tree* seperti yang ditunjukkan pada Gambar 3.3. Uji coba ini dilakukan untuk melihat apakah mekanisme LLP dapat bekerja dengan baik dan membandingkan hasil kinerja nya dengan algoritma LB *default* pada *floodlight*

untuk memastikan semua *host* terhubung, kami menggunakan perintah "*pingall*" pada *mininet*. Pada saat dilakukan pengujian dengan kondisi *load network; low* (25%), *medium* (50%) dan *high* (85%) dari total *bandwidth* 10 Mbps, didapatkan hasil *latency* dengan mekanisme LLP lebih rendah dan peningkatan *transfer rate* data, dibandingkan menggunakan *path* dengan algoritma *default*. Ketika menggunakan *default* LB *floodlight, best path* atau *route* yang digunakan oleh h1 untuk melakukan transfer paket ke h3 dan h4 adalah [S1:4-S10:2-S2:4], paket data yang dikirim ke dua *host* yang berbeda menggunakan *path* yang sama, sehingga menyebabkan *latency* menjadi tinggi. Untuk membuktikannya dengan menggunakan *packet capture wireshark*, yang dapat dilihat pada Gambar 4.1 dan Gambar 4.2.

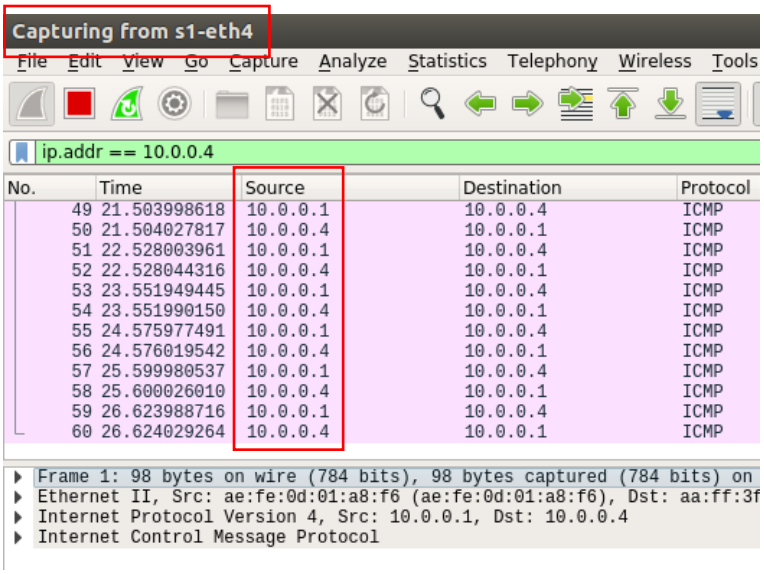
Setelah menjalankan topologi *fat tree* dengan alokasi *bandwidth* 10 Mbps menggunakan perintah berikut, dilakukan *capture* paket data untuk memastikan paket melewati *default path* (tanpa mekanisme LLP).

```
$ sudo mn --custom tree_topology.py --topo mytopo --
controller=remote,ip=127.0.0.1,port=6653 -link tc,bw=10
```

Capture paket data dilakukan pada S1 *port* 4, S10 *port* 2 dan S2 *port* 4, untuk S-D h1-h3 dan h1-h4, apakah paket ICMP dapat dideteksi pada *port* tersebut dengan melakukan *ping* dari *host* h1 ke h3 dan h4.



Gambar 4.1 *Capture* paket data h1-h3 pada S1 *port* 4 tanpa seleksi *path*



Gambar 4.2 *Capture* paket data h1-h4 pada S1 *port* 4 tanpa seleksi *path*

Dari Gambar 4.1 dan 4.2, dapat dilihat pengiriman paket data baik dari S-D h1-h3 dan h1-h4, sama-sama melalui S1 *port* 4. Hal ini menunjukkan *default best path* nya adalah melalui [S1:4-S10:2-S2:4]. Sebelum dilakukan mekanisme seleksi *path* dengan LLP, dilakukan pengukuran dari h1 ke h4, dengan menambahkan parameter-parameter yang telah dijelaskan sebelumnya.

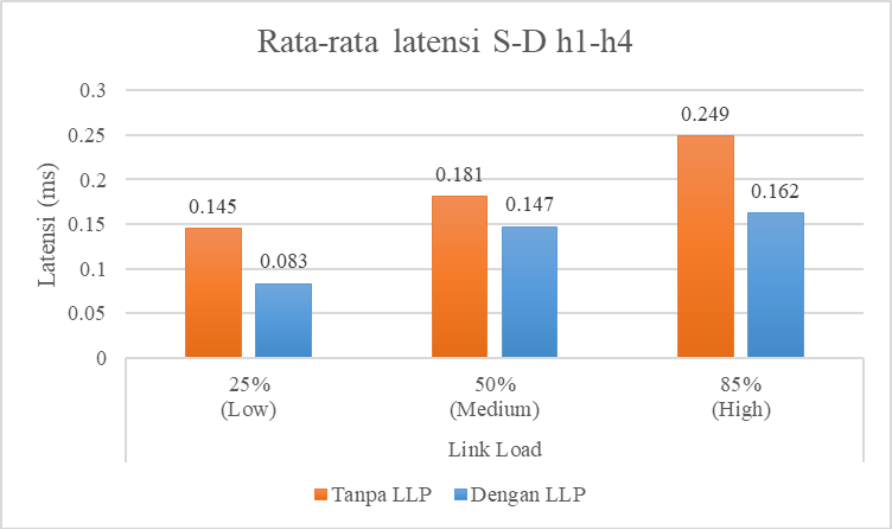
Untuk perbandingan hasil *latency* pada kondisi *network load low, medium*, dan *high* dapat dilihat pada Tabel 4.1 dan Tabel 4.2.

Tabel 4.1 Rata-rata latensi S-D h1-h4 tanpa LLP

Rata-rata Latensi tanpa seleksi *path* LLP (ms)		
25% (Low)	50% (Medium)	85% (High)
0.145	0.181	0.249

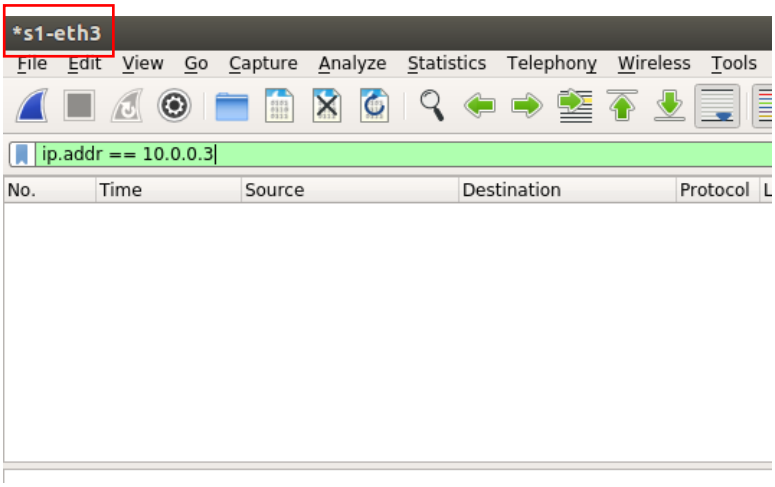
Tabel 4.2 Rata-rata latensi S-D h1-h4 dengan LLP

Rata-rata Latensi dengan seleksi *path* LLP (ms)		
25% (Low)	50% (Medium)	85% (High)
0.083	0.147	0.162



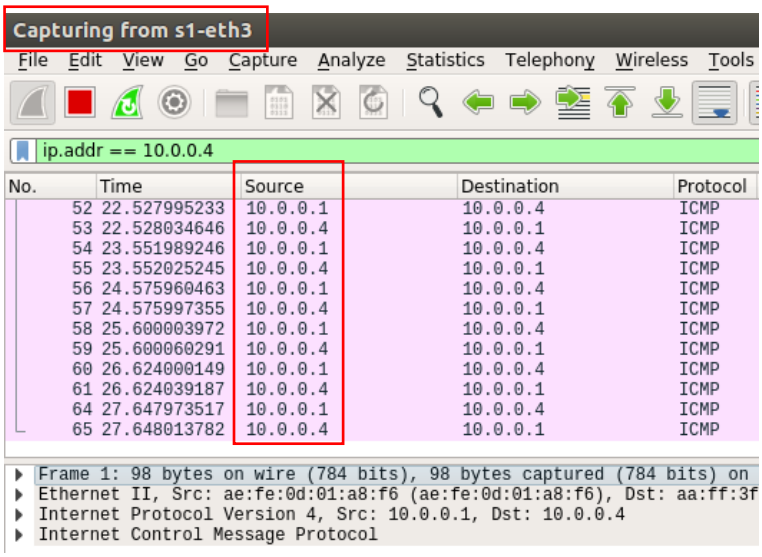
Gambar 4.3 Grafik skenario h1-h4 10 Mbps

Berdasarkan Gambar 4.3 dapat dilihat bahwa hasil latensi dengan mekanisme LLP mengalami penurunan dibandingkan dengan *default path*, penurunan latensi juga terjadi pada *link load* yang lebih tinggi, hal ini disebabkan setelah menggunakan mekanisme LLP, maka *path* yang baru akan dipilih untuk mengirimkan paket data dari h1 ke h4 yaitu, melalui *path* [S1:3-S21:2-S2:3], sehingga waktu yang dibutuhkan untuk mengirimkan paket data lebih cepat. Sementara pengiriman paket data dari h1 ke h3 tetap menggunakan *path* [S1:4-S10:2-S2:4], ini dapat dibuktikan dengan melakukan *capture* paket data dari h1 ke h3 pada S1 *port* 3, serta *capture* paket data dari h1 ke h4 pada S1 *port* 3.



Gambar 4.4 *Capture* paket data h1-h3 pada S1 *port* 3 dengan seleksi *path*

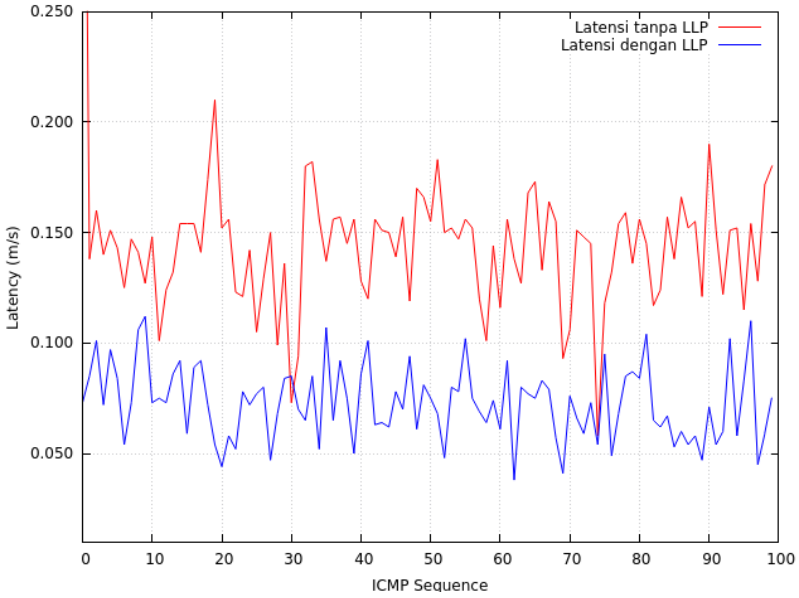
Setelah dilakukan mekanisme seleksi *path*, Gambar 4.4 menunjukkan pengiriman paket data dari h1-h3 tidak melalui S1 *port* 3, namun tetap melewati *path* sebelumnya, yaitu [S1:4-S10:2-S2:4].



Gambar 4.5 *Capture* paket data h1-h4 pada S1 *port* 3 dengan seleksi *path*

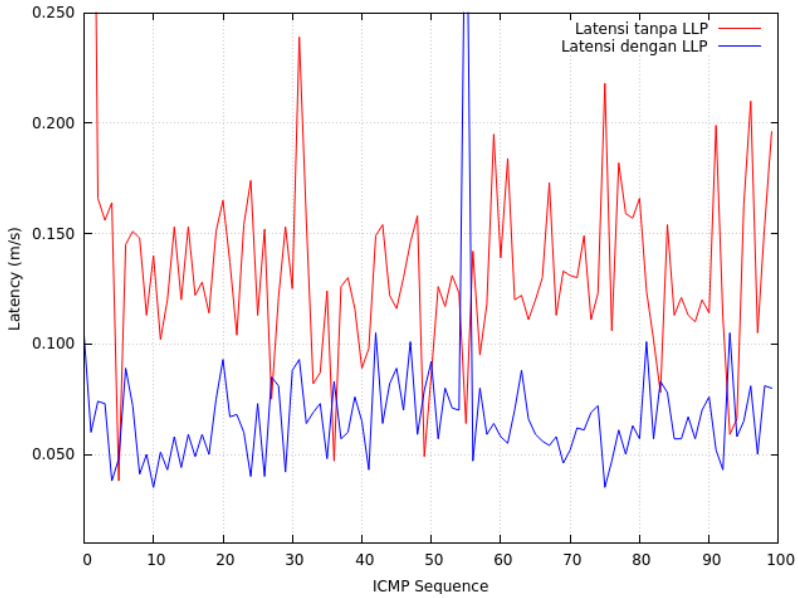
Sementara Gambar 4.5 menunjukkan, setelah dilakukan mekanisme seleksi *path* dengan LLP, maka pengiriman paket data dari h1 ke h4 menggunakan *path* yang baru [S1:3-S21:2-S2:3], dimana *path* tersebut dinilai memiliki *cost* terendah sehingga dipilih sebagai *best path*. Proses *capture* paket dan pengukuran latensi pada skenario h1-h4 ini menunjukkan bahwa mekanisme LLP telah berhasil dalam

melakukan *load balancing* pada *path* yang telah ditentukan.



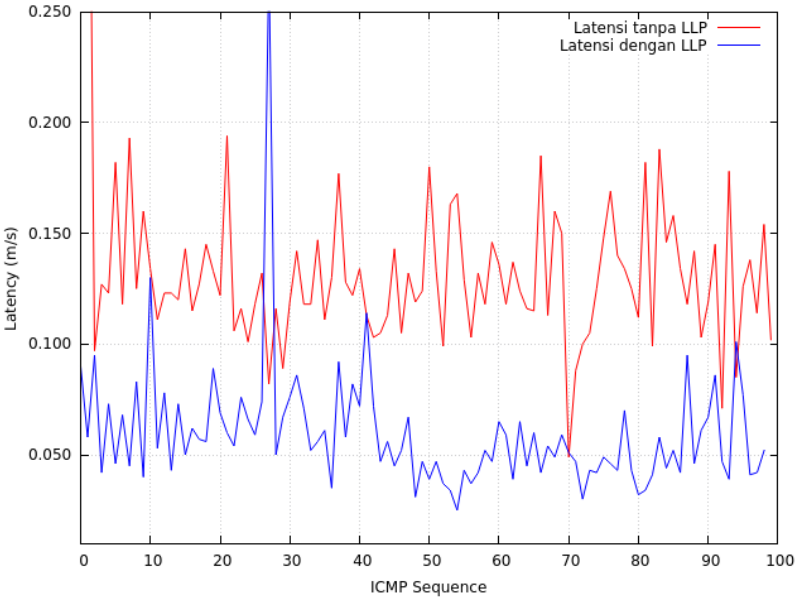
Gambar 4.6 Perbandingan maksimum latensi h1-h4 10 Mbps *link load* (25%)

Pada Gambar 4.6 juga ditunjukkan grafik perbandingan maksimum latensi antara penggunaan *default* LB dan mekanisme LLP, menggunakan parameter *link load* yang berbeda, pengujian dilakukan sebanyak 100 kali pengiriman paket data dengan protokol ICMP. Rata-rata latensi menggunakan *default* LB (tanpa LLP) pada kondisi *link load* 25% adalah 0,145 ms, sementara menggunakan LLP latensinya turun menjadi 0,072 ms, hal ini disebabkan karena *best path* baru yang hanya digunakan untuk mengirimkan paket data dari h1 ke h4, sementara paket data dari h1 ke h3 tetap menggunakan jalur yang lama, atau dapat dikatakan pendistribusian paket pada *dual path* telah dilakukan dengan menggunakan mekanisme seleksi *path*.



Gambar 4.7 Perbandingan maksimum latensi h1-h4 10 Mbps link load (50%)

Hasil penurunan latensi juga ditunjukkan pada Gambar 4.7, dimana sebelum menggunakan LLP rata-rata latensinya adalah 0,181 ms, sementara setelah LLP menjadi 0,068 ms. Sama seperti pengukuran pada *link load* 25%, pada *link load* 50%, penurunan latensi terjadi karena pendistribusian paket menggunakan *dual path* dari h1 ke h3 dan h4 telah berhasil dilakukan, sehingga pengiriman paket data dapat dilakukan lebih cepat.



Gambar 4.8 Perbandingan maksimum latensi h1-h4 10 Mbps *link load* (85%)

Sementara pada Gambar 4.8 menunjukkan hasil penurunan latensi setelah dilakukan LLP dengan *link load* 85%, dimana pada saat menggunakan *default* LB rata-rata latensinya adalah 0,249 ms, sedangkan setelah menggunakan LLP latensi menjadi 0,059 ms. Dari ketiga pengukuran latensi dengan *link load* yang berbeda, dapat dilihat penurunan latensi selalu terjadi, hal ini membuktikan mekanisme seleksi *path* dengan LLP telah berjalan dengan baik dan mampu meningkatkan waktu pengiriman paket data.

Untuk hasil *transfer rate* data dan *throughput*, ditunjukkan pada Tabel 4.3 dengan menggunakan *link load* yang berbeda, yaitu 25%, 50% dan 85% dari total 10 Mbps *bandwidth*.

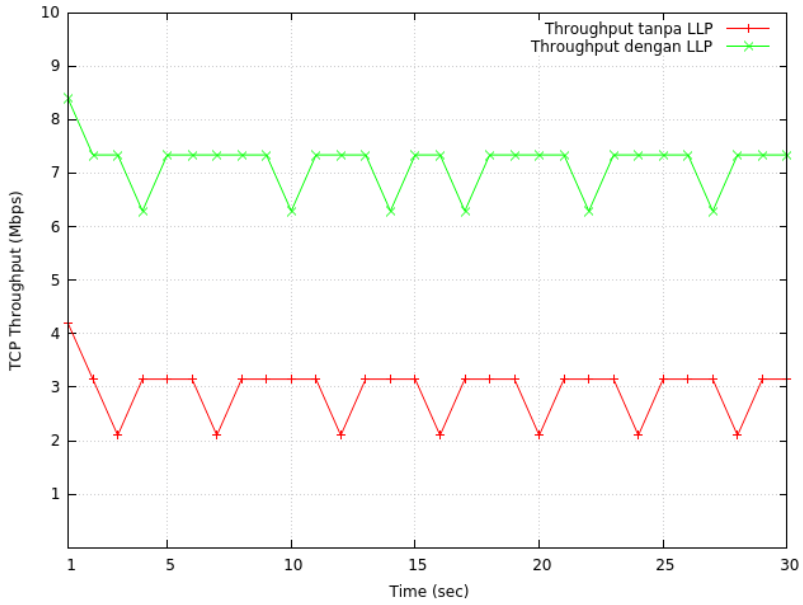
Tabel 4.3 *transfer rate* dan *throughput* h1-h4 10 Mbps

	Default LB						**Dengan LLP**					
	Transfer Rate Data (Kbytes)			**Throughput (Kbps-Mbps)**			**Transfer Rate Data (Kbytes)**			**Throughput (Kbps-Mbps)**		
Interval (sec)	**25% Load**	**50% Load**	**85% Load**	**25% Load**	**50% Load**	**85% Load**	**25% Load**	**50% Load**	**85% Load**	**25% Load**	**50% Load**	**85% Load**
0-1	512	384	256	4,19	3,15	2,1	1 MB	768	384	8,39	6,29	3,15
1-2	384	384	0 B	3,15	3,15	0	896	512	128	7,34	4,19	1,05
2-3	256	256	128	2,1	2,1	1,05	896	640	128	7,34	5,24	1,05
3-4	384	256	128	3,15	2,1	1,05	768	512	128	6,29	4,19	1,05
4-5	384	256	0 B	3,15	2,1	0	896	640	128	7,34	5,24	1,05
5-6	384	384	128	3,15	3,15	1,05	896	512	256	7,34	4,19	2,1
6-7	256	256	0 B	2,1	2,1	0	896	512	128	7,34	4,19	1,05
7-8	384	256	128	3,15	2,1	1,05	896	640	128	7,34	5,24	1,05
8-9	384	256	128	3,15	2,1	1,05	896	512	128	7,34	4,19	1,05
9-10	384	256	0 B	3,15	2,1	0	768	640	128	6,29	5,24	1,05
10-11	384	384	128	3,15	3,15	1,05	896	512	128	7,34	4,19	1,05
11-12	256	256	0 B	2,1	2,1	0	896	640	256	7,34	5,24	2,1
12-13	384	256	128	3,15	2,1	1,05	896	512	128	7,34	4,19	1,05
13-14	384	256	0 B	3,15	2,1	0	768	512	128	6,29	4,19	1,05
14-15	384	384	128	3,15	3,15	1,05	896	640	128	7,34	5,24	1,05
15-16	256	256	128	2,1	2,1	1,05	896	512	128	7,34	4,19	1,05
16-17	384	256	0 B	3,15	2,1	0	768	640	128	6,29	5,24	1,05
17-18	384	256	128	3,15	2,1	1,05	896	512	128	7,34	4,19	1,05
18-19	384	384	0 B	3,15	3,15	0	896	640	256	7,34	5,24	2,1
19-20	256	256	128	2,1	2,1	1,05	896	512	128	7,34	4,19	1,05
20-21	384	256	0 B	3,15	2,1	0	896	512	128	7,34	4,19	1,05
21-22	384	256	128	3,15	2,1	1,05	768	640	128	6,29	5,24	1,05
22-23	384	256	128	3,15	2,1	1,05	896	512	128	7,34	4,19	1,05
23-24	256	384	0 B	2,1	3,15	0	896	640	128	7,34	5,24	1,05

24-25	384	128	128	3,15	1,05	1,05	896	512	256	7,34	4,19	2,1
25-26	384	384	0 B	3,15	3,15	0	896	512	128	7,34	4,19	1,05
26-27	384	256	128	3,15	2,1	1,05	768	640	128	6,29	5,24	1,05
27-28	256	256	128	2,1	2,1	1,05	896	512	128	7,34	4,19	1,05
28-29	384	256	0 B	3,15	2,1	0	896	640	128	7,34	5,24	1,05
29-30	384	384	128	3,15	3,15	1,05	896	512	128	7,34	4,19	1,05
0-30	0,6 MB	8,6 MB	2,5 MB	2,9 Mbps	2,35 Mbps	633 Kbps	25.,8 MB	16,9 MB	4,6 MB	7,14 Mbps	4,67 Mbps	1,26 Mbps

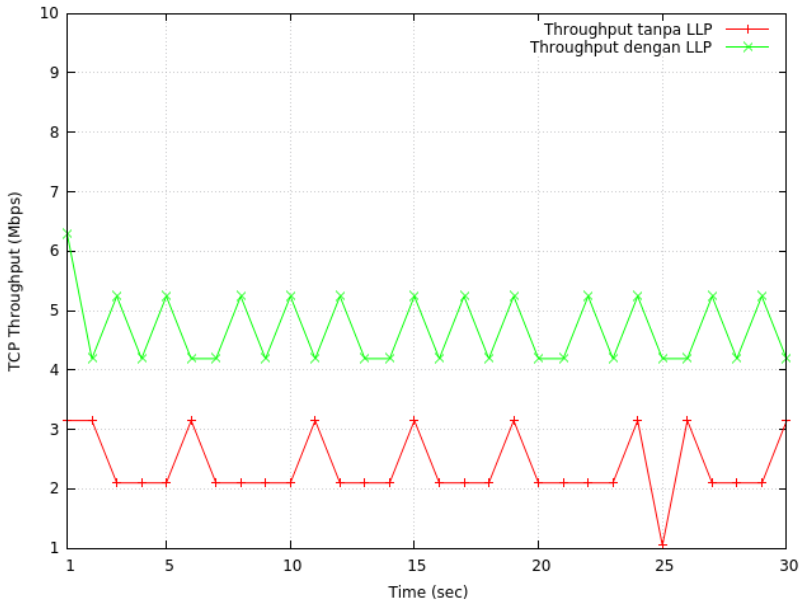
Hasil *transfer rate* data dan t*hroughput* (*bandwidth*) pada Tabel 4.3 menunjukkan, peningkatan selalu terjadi setelah menggunakan mekanisme seleksi *path* dengan LLP, baik pada kondisi *link load low, medium* dan *high*. Hal ini dapat terjadi karena, setelah dilakukan mekanisme seleksi *path* dengan LLP, maka pengiriman paket data tidak hanya menggunakan satu *path* saja, namun menggunakan *dual path*, sehingga pendistribusian pengiriman data lebih merata dan seimbang. Namun ketika kondisi *load* semakin bertambah, *transfer rate* data dan *throughput* juga menjadi turun, ini menunjukkan kapasitas *bandwidth* sudah sesuai dengan skenario pengujian dan pemanfaatan *bandwidth* menjadi maksimal.

Untuk hasil perbandingan *throughput* S-D h1-h4 dengan menggunakan *link load* yang berbeda pada alokasi *bandwidth* 10 Mbps dapat dilihat pada Gambar 4.9, 4.10 dan 4.11.



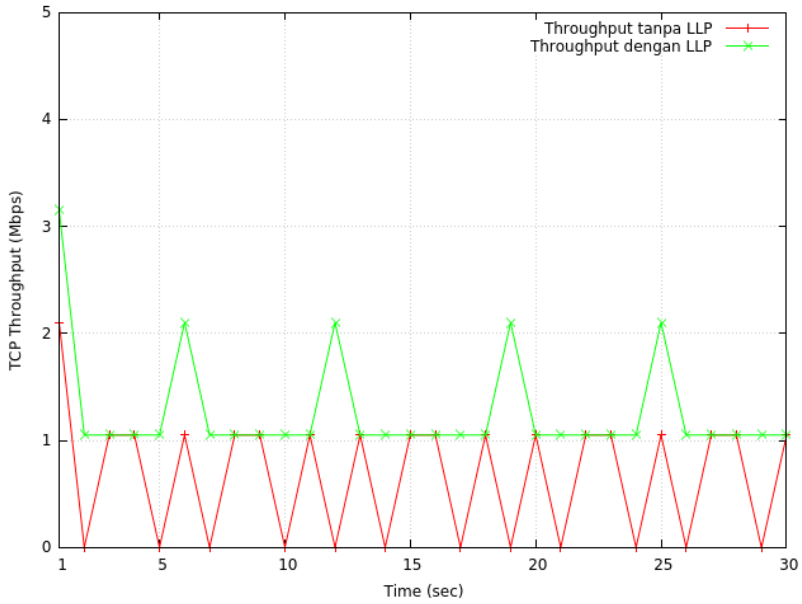
Gambar 4.9 Perbandingan *throughput* h1-h4 10 Mbps *link load* (25%)

Gambar 4.9 menunjukkan graafik perbandingan *throughput* yang menggunakan *default* LB dan menggunakan LLP dengan kondisi *link load* 25% dari total *bandwidth* 10 Mbps atau 2,5 Mbps. Grafik tersebut menunjukkan terjadi kenaikan *throughput* setelah menggunakan LLP, dimana terjadi kenaikan rata-rata 3 Mbps menjadi 7 Mbps setelah diuji selama 30 detik dengan *default* TCP *window size default* 85,3 *KByte*. Hal ini menunjukkan mekanisme LLP dapat meningkatkan *throughput*, karena pada saat pengiriman paket data dari h1 ke h3 dan ke h4 menggunakan *dual path*, sedangkan pada *default* LB hanya menggunakan satu *path* saja.



Gambar 4.10 Perbandingan *throughput* h1-h4 10 Mbps *link load* (50%)

Pada Gambar 4.10 menunjukaan hasil peningkatan dengan kondisi *link load* 50%, rata-rata peningkatan *throughput* nya adalah dari 2-3 Mbps menjadi 4-5 Mbps, peningkatan yang terjadi tidak sebesar pada kondisi *link load* 50%, hal ini disebabkan kondisi *load path* yang semakin bertambah, sehingga pengiriman paket hanya memanfaatkan 50% sisa *bandwidth*. Namun hal ini sudah menunjukkan peningkatan dalam pengiriman paket data menggunakan mekanisme LLP.



Gambar 4.11 Perbandingan *throughput* h1-h4 10 Mbps *link load* (85%)

Pada Gambar 4.11 juga menunjukkan hasil peningkatan *throughput* pada kondisi *link load* 85%, yaitu rata-rata peningkatan dari 0-1 Mbps menjadi 1-2 Mbps. Karena load path yang semakin bertambah, maka pengiriman paket data dari h1 ke h3 dan h4 juga semakin kecil. Hal ini ditandai dengan *throughput* yang semakin menurun dari pengujian menggunakan *link load* 25% dan 50%. Namun peningkatan *throughput* tetap terjadi setelah dilakukan mekanisme seleksi *path* dengan LLP.

4.1.2 Hasil uji coba skenario S-D h1-h4 dengan *bandwidth* 100 Mbps

Pengujian selanjutnya menggunakan *bandwidth* sebesar 100 Mbps, namun masih menggunakan kondisi *network load* yang berbeda untuk menilai kinerja jaringan yang menggunakan *default* LB dan menggunakan mekanisme LLP. Pengujian dilakukan mengikuti langkah-langkah yang sama pada pengujian dengan *bandwidth* 10 Mbps.

Setelah menjalankan topologi *fat tree* dengan alokasi *bandwidth* 100 Mbps menggunakan perintah berikut, dilakukan *capture* paket data untuk memastikan paket melewati *default path* (tanpa mekanisme LLP).

```
$ sudo mn --custom tree_topology.py --topo mytopo --
controller=remote,ip=127.0.0.1,port=6653 -link tc,bw=100
```

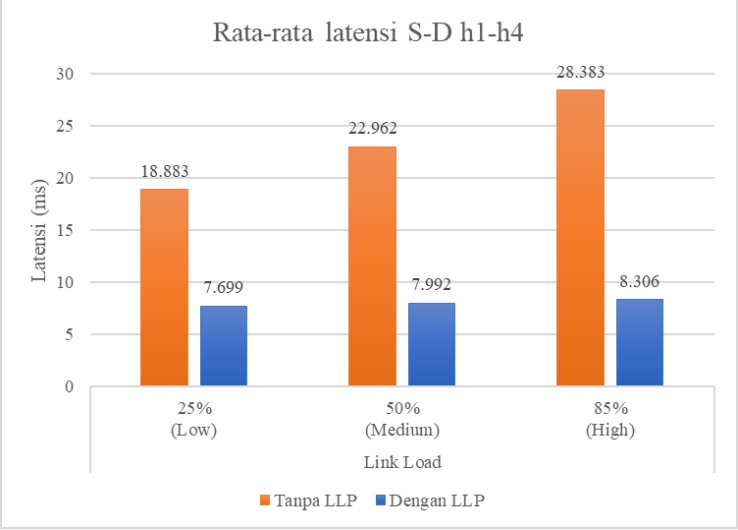
Pengujian dengan *bandwidth* 100 Mbps, menunjukkan hasil yang sama seperti pengujian pada *bandwidth* 10 Mbps. Terjadi penurunan *latency*, peningkatan *transfer rate* data dan peningkatan *throughput* setelah menggunakan mekanisme LLP dibandingkan menggunakan *default* LB. Untuk perbandingan hasil *latency* dapat dilihat pada Tabel 4.4 dan Tabel 4.5 pada kondisi *network load low*, *medium*, dan *high*.

Tabel 4.4 Rata-rata latensi S-D h1-h4 tanpa LLP 100 Mbps

Rata-rata Latensi tanpa seleksi *path* LLP (ms)		
25% (Low)	50% (Medium)	85% (High)
18.883	22,962	28.383

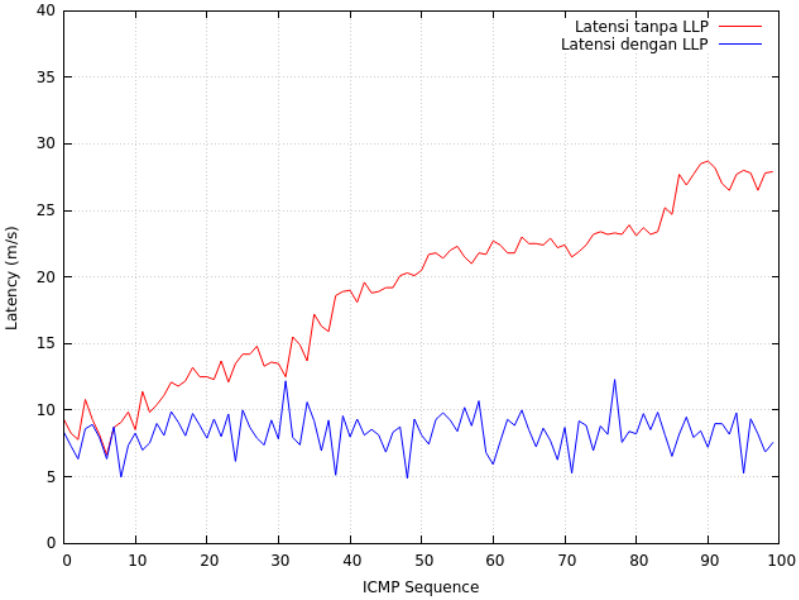
Tabel 4.5 Rata-rata latensi S-D h1-h4 dengan LLP 100 Mbps

Rata-rata Latensi dengan seleksi *path* LLP (ms)		
25% (Low)	50% (Medium)	85% (High)
7,699	7,992	8,306



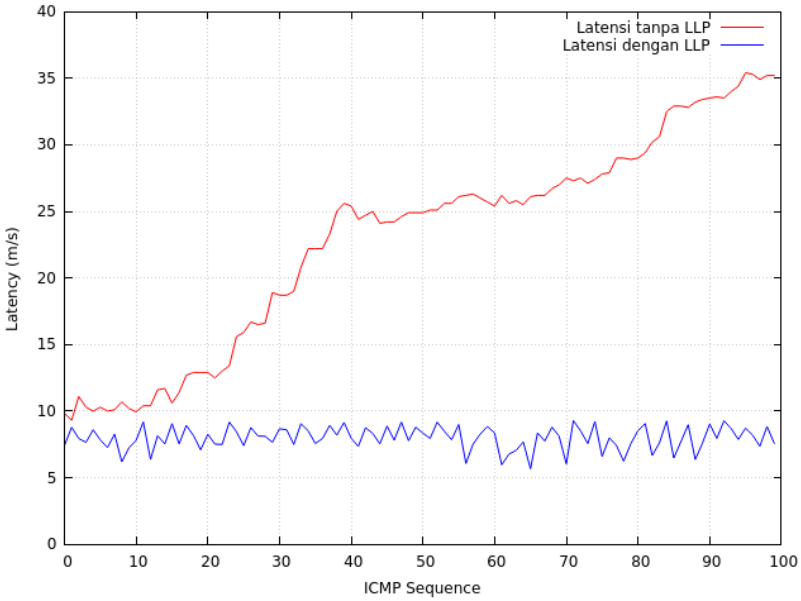
Gambar 4.12 Grafik skenario h1-h4 100 Mbps

Berdasarkan Gambar 4.12 dapat dilihat bahwa hasil latensi dengan mekanisme LLP mengalami penurunan dibandingkan dengan *default path*, penurunan latensi juga terjadi pada *link load* yang lebih tinggi, hal ini disebabkan setelah menggunakan mekanisme LLP, maka *path* yang baru akan dipilih untuk mengirimkan paket data dari h1 ke h4 yaitu, melalui *path* [S1:3-S21:2-S2:3], sehingga waktu yang dibutuhkan untuk mengirimkan paket data lebih cepat. Sementara pengiriman paket data dari h1 ke h3 tetap menggunakan *path* [S1:4-S10:2-S2:4], ini dapat dibuktikan dengan melakukan *capture* paket data dari h1 ke h3 pada S1 *port* 3, serta *capture* paket data dari h1 ke h4 pada S1 *port* 3.



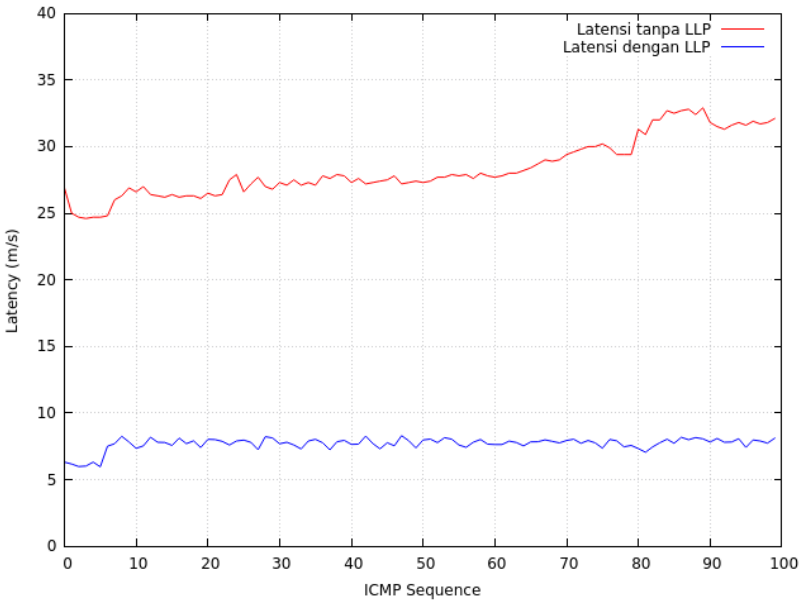
Gambar 4.13 Perbandingan maksimum latensi h1-h4 100 Mbps *link load* (25%)

Pada Gambar 4.13 menunjukkan hasil penurunan latensi setelah menggunakan LLP. Rata-rata latensi menggunakan *default* LB pada kondisi *link load low* (25%) adalah 18,883 ms, turun menjadi 8,306 ms setelah dilakukan pengukuran dengan mengirimkan paket data dari h1 ke h3 dan h4 dengan protokol ICMP sebanyak 100 kali.



Gambar 4. 14 Perbandingan maksimum latensi h1-h4 100 Mbps *link load* (50%)

Pada Gambar 4.14 juga menunjukkan hasil penurunan latensi setelah menggunakan LLP. Rata-rata latensi menggunakan *default* LB pada kondisi *link load medium* (50%) adalah 22,962 ms, turun menjadi 7,992 ms setelah dilakukan pengukuran dengan menirimkan paket data dari h1 ke h3 dan h4 dengan protokol ICMP sebanyak 100 kali.



Gambar 4. 15 Perbandingan maksimum latensi h1-h4 100 Mbps *link load* (85%)

Pada Gambar 4.14 menunjukkan hasil penurunan latensi setelah menggunakan LLP. Rata-rata latensi menggunakan *default* LB pada kondisi *link load high* (85%) adalah 28,383 ms, turun menjadi 7,699 ms setelah dilakukan pengukuran dengan mengirimkan paket data dari h1 ke h3 dan h4 dengan protokol ICMP sebanyak 100 kali. Ketiga pengujiian latensi ini menunjukkan mekanisme seleksi *path* dengan LLP telah bekerja dengan baik, dengan melakukan pendistribusian paket pada *dual path*, seperti yang telah dilakukan pada pengujian sebelumnya dengan menggunakan parameter *bandwidth* 10 Mbps.

Untuk hasil *transfer rate* data dan *throughput*, ditunjukkan pada Tabel 4.6 dengan menggunakan *link load* yang berbeda, yaitu 25%, 50% dan 85% dari total 100 Mbps *bandwidth*.

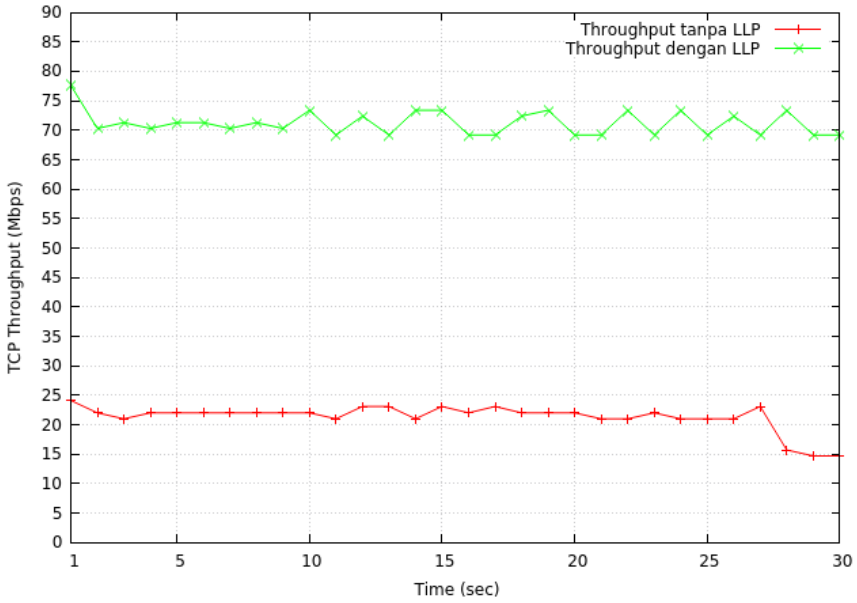
Tabel 4.6 *Transfer rate* dan *throughput* h1-h4 *bandwidth* 100 Mbps

	Default LB						**Dengan LLP**					
	Transfer Rate Data (Megabytes)			**Throughput (Mbps)**			**Transfer Rate Data (Megabytes)**			**Throughput (Mbps)**		
Interval (sec)	**25% Load**	**50% Load**	**85% Load**	**25% Load**	**50% Load**	**85% Load**	**25% Load**	**50% Load**	**85% Load**	**25% Load**	**50% Load**	**85% Load**
0-1	2,88	2,62	1	24,1	22	8,39	9,25	5,88	1,75	77,6	49,3	14,7
1-2	2,62	2,38	768 KB	22	19,9	6,29	8,38	5,5	1,38	70,3	46,1	11,5
2-3	2,5	2,5	640 KB	21	21	5,24	8,5	5,62	1,5	71,3	47,2	12,6
3-4	2,62	2,38	768 KB	22	19,9	6,29	8,38	5,5	1,38	70,3	46,1	11,5
4-5	2,62	2,38	768 KB	22	19,9	6,29	8,5	5,62	1,38	71,3	47,2	11,5
5-6	2,62	2,25	640 KB	22	18,9	5,24	8,5	5,38	1,5	71,3	45,1	12,6
6-7	2,62	2,12	768 KB	22	17,8	6,29	8,38	5,62	1,25	70,3	47,2	10,5
7-8	2,62	2,12	768 KB	22	17,8	6,29	8,5	5,75	1,38	71,3	48,2	11,5
8-9	2,62	2,25	640 KB	22	18,9	5,24	8,38	5,75	1,5	70,3	48,2	12,6
9-10	2,62	2,38	640 KB	22	19,9	5,24	8,75	5,25	1,38	73,4	44	11,5
10-11	2,5	2,38	768 KB	21	19,9	6,29	8,25	5,62	1,38	69,2	47,2	11,5
11-12	2,75	2,38	768 KB	23,1	19,9	6,29	8,62	5,62	1,5	72,4	47,2	12,6
12-13	2,75	2,5	640 KB	23,1	21	5,24	8,25	5,62	1,38	69,2	47,2	11,5
13-14	2,5	2,38	768 KB	21	19,9	6,29	8,75	5,25	1,38	73,4	44	11,5
14-15	2,75	2,25	768 KB	23,1	18,9	6,29	8,75	5,62	1,5	73,4	47,2	12,6
15-16	2,62	2,5	640 KB	22	21	5,24	8,25	5,5	1,38	69,2	46,1	11,5
16-17	2,75	2,5	768 KB	23,1	21	6,29	8,25	5,75	1,38	69,2	48,2	11,5
17-18	2,62	2,5	640 KB	22	21	5,24	8,62	5,62	1,38	72,4	47,2	11,5
18-19	2,62	2,38	768 KB	22	19,9	6,29	8,75	5,62	1,5	73,4	47,2	12,6
19-20	2,62	2,5	768 KB	22	21	6,29	8,25	5,25	1,38	69,2	44	11,5
20-21	2,5	2,62	640 KB	21	22	5,24	8,25	5,62	1,38	69,2	47,2	11,5
21-22	2,5	2,75	768 KB	21	23,1	6,29	8,75	5,62	1,5	73,4	47,2	12,6

22-23	2,62	2,25	768 KB	22	18,9	6,29	8,25	5,75	1,25	69,2	48,2	10,5
23-24	2,5	2,12	640 KB	21	17,8	5,24	8,75	5,38	1,38	73,4	45,1	11,5
24-25	2,5	2,25	896 KB	21	18,9	7,34	8,25	5,5	1,5	69,2	46,1	12,6
25-26	2,5	2,25	768 KB	21	18,9	6,29	8,62	5,5	1,38	72,4	46,1	11,5
26-27	2,75	2,25	768 KB	23,1	18,9	6,29	8,25	5,5	1,38	69,2	46,1	11,5
27-28	1,88	2,25	768 KB	15,7	18,9	6,29	8,75	6,5	1,5	73,4	54,5	12,6
28-29	1,75	2,25	768 KB	14,7	18,9	6,29	8,25	5,38	1,38	69,2	45,1	11,5
29-30	1,75	2,25	768 KB	14,7	18,9	6,29	8,25	5,12	1,38	69,2	43	11,5
0-30	76,2	71	1,9	21,3	19,8	6,06	255	167	42,6	71,2	46,8	11,9

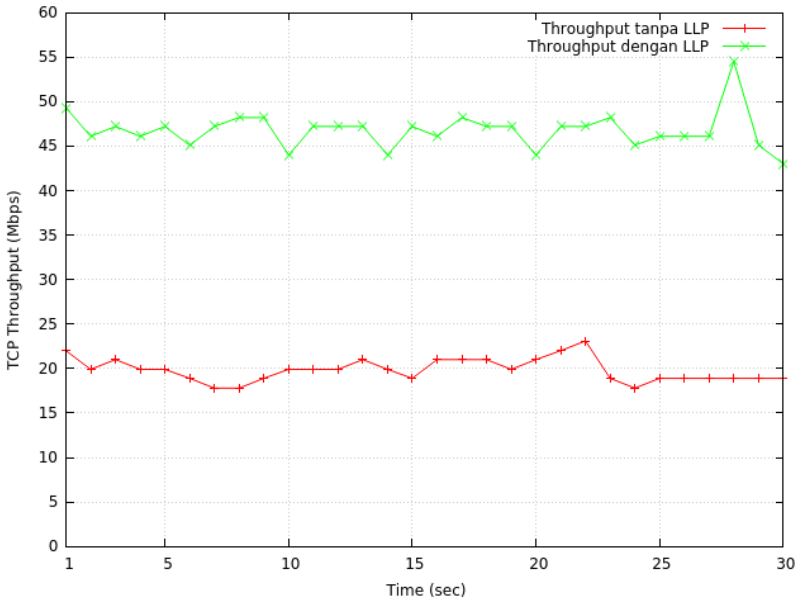
Hasil *transfer rate* data dan *throughput* (*bandwidth*) pada Tabel 4.6 menunjukkan, peningkatan terjadi setelah menggunakan mekanisme seleksi *path* dengan LLP, baik pada kondisi *link load low, medium* dan *high*. Hal ini dapat terjadi karena, setelah dilakukan mekanisme seleksi *path* dengan LLP, maka pengiriman paket data tidak hanya menggunakan satu *path* saja, namun *menggunakan dual path*, sehingga pendistribusian pengiriman data lebih merata dan seimbang. Namun ketika kondisi *load* semakin bertambah, *transfer rate* data dan *throughput* juga menjadi turun, ini menunjukkan kapasitas *bandwidth* sudah sesuai dengan skenario pengujian dan pemanfaatan *bandwidth* menjadi maksimal.

Untuk hasil perbandingan *throughput* S-D h1-h4 dengan menggunakan *link load* yang berbeda pada alokasi *bandwidth* 100 Mbps dapat dilihat pada Gambar 4.16, 4.17 dan 4.18.



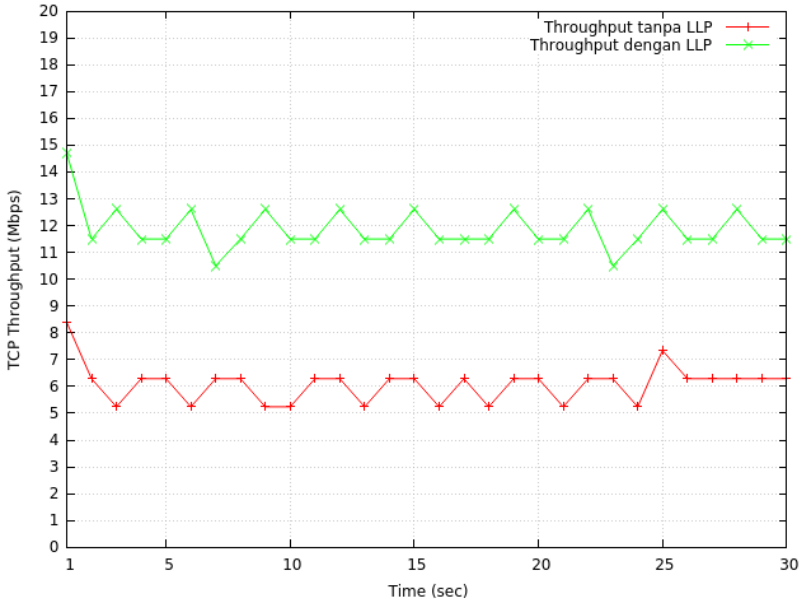
Gambar 4.16 Perbandingan *throughput* h1-h4 100 Mbps *link load* (25%)

Gambar 4.6 menunjukkan perbandingan *throuhgput* menggunakan *default* LB dan menggunakan mekanisme seleksi *path* dengan LLP. Pengukuran dilakukan dengan *Iperf* menggunakan TCP *windows size* 85, 3 *KByte* selama 30 detik pada kondisi *link load* 25% dari total *bandwidth* 100 Mbps atau 25 Mbps. Grafik tersebut menunjukkan peningkatan *throughput* setelah menggunakan mekanisme LLP dari 21-24 Mbps menggunakan *default* LB menjadi 69-73 Mbps. Peningkatan ini menunjukkan dengan menggunakan mekanisme LLP, pengiriman paket data dapat dilakukan menggunakan *dual path*, sehingga dapat mendistribusikan paket yang dikirim dari h1 ke h3 dan h4 pada dua *path* yang berbeda, sehingga dapat memaksimalkan penggunaan *bandwidth*.



Gambar 4. 17 Perbandingan *throughput* h1-h4 100 Mbps *link load* (50%)

Pada Gambar 4.17 juga menunjukkan peningkatan *throughput* setelah dilakukan mekanisme LLP pada kondisi *link load* 50% atau 50 Mbps. Peningkatan dari 18-22 Mbps menggunakan *default* LB menjadi 43-49 Mbps. Sama seperti pengujian dengan *link load* 25%, mekanisme LLP telah bekerja dengan baik dalam mendistribusikan paket pada *dual path*, sehingga penggunaan *bandwidth* menjadi maksimal.



Gambar 4. 18 Perbandingan t*hroughput* h1-h4 100 Mbps *link load* (85%)

Pada Gambar 4.18 juga menunjukkan peningkatan *throughput* dari 5-8 Mbps menjadi 10-14 Mbps. Peningkatan *throughput* tidak sebesar sebelumnya, hal ini dapat terjadi karena kondisi *link load* yang bertambah, yaitu 85% dari total *bandwidth* atau 85 Mbps, sehingga menyebabkan paket data yang dikirim hanya menggunakan sisa *bandwidht* yang tersedia, namun setelah mekanisme LLP, peningkatan dapat terjadi karena paket yang dikirim dari h1 ke h3 dan h4 dapat melalui dua *path*, dimana hal ini tidak terjadi pada *default* LB.

4.1.3 Hasil uji coba skenario S-D h5-h8 dengan *bandwidth* 10 Mbps

Berbeda dengan skenario S-D h1-h4, pada pengujian skenario S-D h5-h8, hanya menggunakan dua kondisi *network load*, yaitu *low* (5%) dan *high* (95%) dari total *bandwidth* 10 Mbps, dimana kondisi *low* mensimulasikan kondisi trafik jaringan yang rendah, sedangkan kondisi *high* mensimulasikan kondisi trafik jaringan yang tinggi dan menggunakan hampir seluruh *bandwidth* yang tersedia. Langkah yang sama dengan skenario S-D h1-h4 telah dilakukan dalam melakukan pengujian skenario ini, dengan menjalankan topologi *fat tree* dan melakukan *capture* paket dengan *wireshark* untuk memastikan *default path route* dan *best path route* Dari hasil uji coba yang dilakukan sebanyak 100 kali untuk setiap pengukuran latensi, dan waktu 30 detik untuk setiap pengukuran *transfer rate* dan

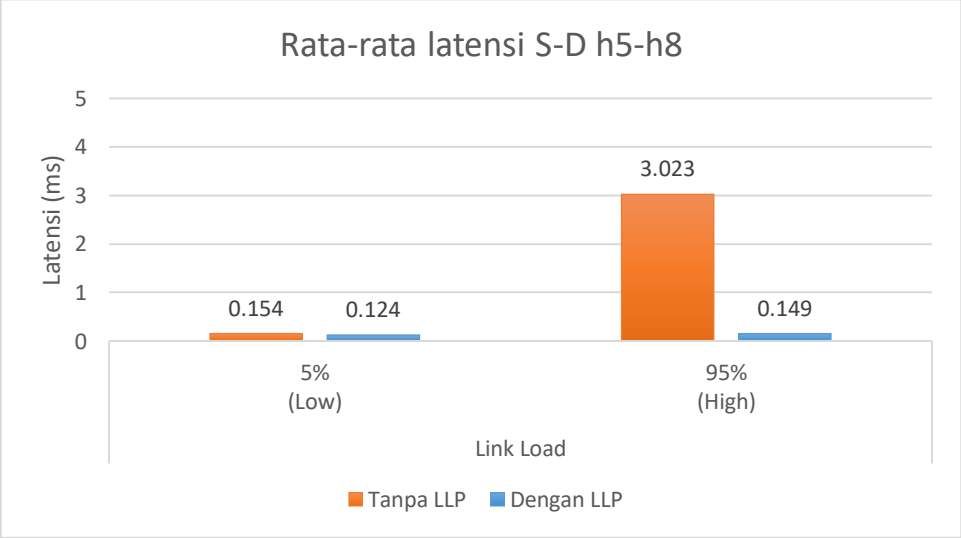
throughput (*bandwidth*) didapatkan perbedaan latensi yang ditunjukkan pada Tabel 4.7 dan Tabel 4.8.

Tabel 4.7 Rata-rata latensi S-D h5-h8 tanpa LLP 10 Mbps

Rata-rata Latensi tanpa seleksi *path* LLP (ms)	
5% (Low)	95% (High)
0,154	3,023

Tabel 4.8 Rata-rata latensi S-D h5-h8 dengan LLP 10 Mbps

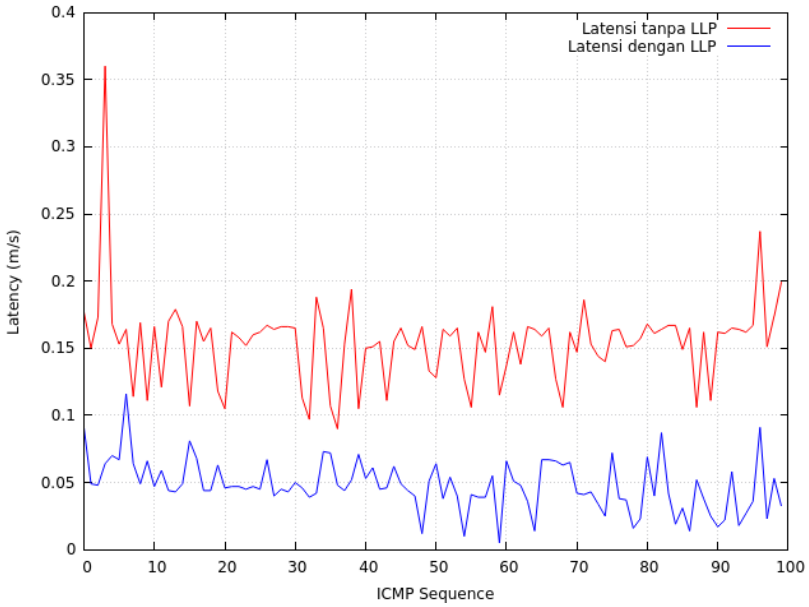
Rata-rata Latensi tanpa seleksi *path* LLP (ms)	
5% (Low)	95% (High)
0,124	0,149



Gambar 4.19 Grafik skenario h5-h8 10 Mbps

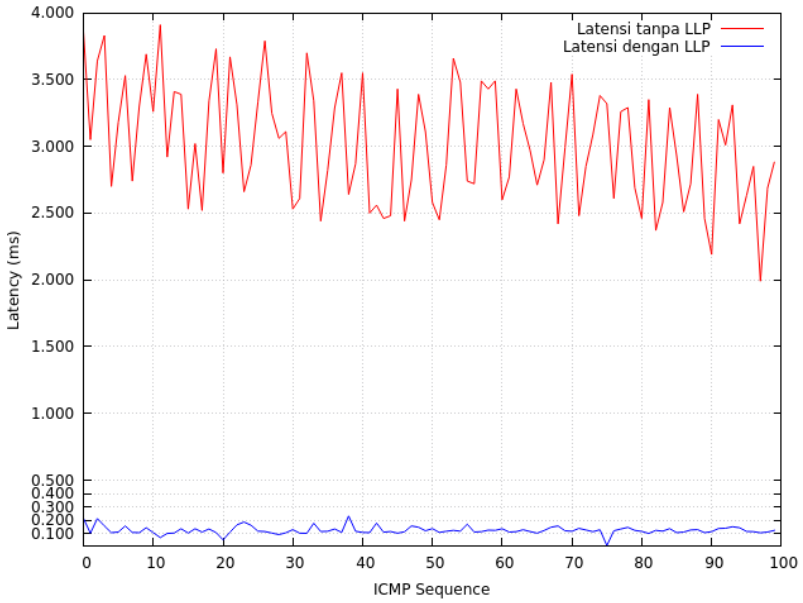
Berdasarkan Gambar 4.19 dapat dilihat bahwa hasil latensi dengan mekanisme LLP mengalami penurunan dibandingkan dengan *default path*, penurunan latensi juga terjadi pada *link load* yang lebih tinggi, hal ini disebabkan setelah menggunakan mekanisme LLP, maka *path* yang baru akan dipilih untuk mengirimkan paket data dari h5 ke h8 yaitu, melalui *path* [S3:4-S22:1-S4:3], sehingga waktu yang dibutuhkan untuk mengirimkan paket data lebih cepat.

Sementara pengiriman paket data dari h5 ke h7 tetap menggunakan *path* [S3:3-S11:2-S4:4], ini dapat dibuktikan dengan melakukan *capture packet* data dari h5 ke h7 pada S3 *port* 3, serta *capture* paket data dari h5 ke h8 pada S3 *port* 4. Setelah dilakukan mekanisme LLP, paket data dari h5 ke h7 tidak akan terlihat pada S3 *port* 4. Hal ini membuktikan pengiriman paket data sudah menggunakan *best path* yang baru.



Gambar 4.20 Perbandingan maksimum latensi h5-h8 10 Mbps *link load* (5%)

Pada Gambar 4.20 menunjukkan hasil penurunan latensi setelah dilakukan mekanisme LLP. Rata-rata latensi menggunakan *default* LB adalah 0,154 ms, sedangkan rata-rata latensi dengan mekanisme LLP adalah 0,124. Penurunan latensi yang terjadi tidak cukup signifikan dikarenkan sisa *bandwidth* yang tersedia masih cukup banyak yaitu sebesar 95% atau 95 Mbps, sehingga waktu yang dibutuhkan untuk pengiriman paket data hampir sama.



Gambar 4.21 Perbandingan maksimum latensi h5-h8 10 Mbps *link load* (95%)

Penurunan latensi juga dapat dilihat pada Gambar 4.21. Rata-rata latensi menggunakan *default* LB pada kondisi *link load* 95% adalah 3,023 ms, sedangkan menggunakan mekanisme LLP turun menjadi 0,149 ms.

Untuk hasil *transfer rate* data dan *throughput*, ditunjukkan pada Tabel 4.9 dengan menggunakan *link load* yang berbeda, yaitu 5% dan 95% dari total 10 Mbps *bandwidth*.

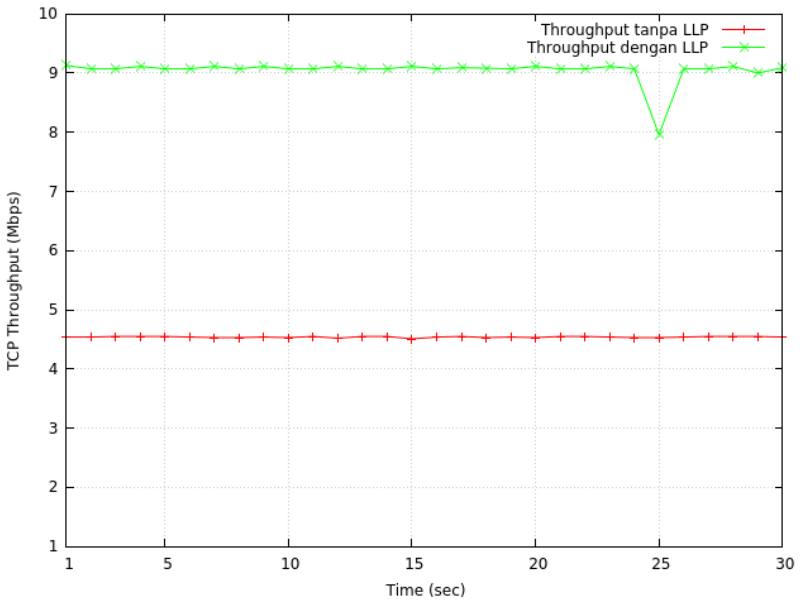
Tabel 4.9 *Transfer rate* dan *throughput* h5-h8 *bandwidth* 10 Mbps

Interval (sec)	**Default LB**				**Dengan LLP**			
	Transfer Rate Data (Megabytes)		**Throughput (Mbps)**		**Transfer Rate Data (Megabytes)**		**Throughput (Mbps)**	
	5% Load	**95% Load**	**5% Load**	**95% Load**	**5% Load**	**95% Load**	**5% Load**	**95% Load**
0-1	554 KB	62 KB	4,54	510 Kbps	1,09	130 KB	9,13	521 Kbps
1-2	554 KB	59 KB	4,54	487 Kbps	1,08	59 KB	9,07	487 Kbps
2-3	556 KB	61 KB	4,55	498 Kbps	1,08	72 KB	9,07	591 Kbps
3-4	556 KB	57 KB	4,55	463 Kbps	1,09	55 KB	9,11	452 Kbps
4-5	556 KB	64 KB	4,55	521 Kbps	1,08	69 KB	9,07	568 Kbps
5-6	554 KB	64 KB	4,54	521 Kbps	1,08	58 KB	9,07	475 Kbps
6-7	553 KB	58 KB	4,53	475 Kbps	1,09	69 KB	9,11	568 Kbps
7-8	553 KB	58 KB	4,53	475 Kbps	1,08	64 KB	9,07	521 Kbps
8-9	554 KB	69 KB	4,54	568 Kbps	1,09	64 KB	9,11	521 Kbps
9-10	553 KB	58 KB	4,53	475 Kbps	1,08	64 KB	9,07	521 Kbps

10-11	556 KB	58 KB	4,55	475 Kbps	1,08	64 KB	9,07	521 Kbps
11-12	551 KB	64 KB	4,52	521 Kbps	1,09	64 KB	9,11	521 Kbps
12-13	556 KB	64 KB	4,55	521 Kbps	1,08	61 KB	9,07	498 Kbps
13-14	556 KB	58 KB	4,55	475 Kbps	1,08	67 KB	9,07	544 Kbps
14-15	550 KB	61 KB	4,51	498 Kbps	1,09	58 KB	9,11	475 Kbps
15-16	554 KB	67 KB	4,54	544 Kbps	1,08	69 KB	9,07	568 Kbps
16-17	556 KB	58 KB	4,55	475 Kbps	1,08	58 KB	9,09	475 Kbps
17-18	553 KB	61 KB	4,53	498 Kbps	1,08	69 KB	9,08	568 Kbps
18-19	554 KB	61 KB	4,54	498 Kbps	1,08	58 KB	9,07	475 Kbps
19-20	553 KB	58 KB	4,53	475 Kbps	1,09	69 KB	9,11	568 Kbps
20-21	556 KB	58 KB	4,55	475 Kbps	1,08	58 KB	9,07	475 Kbps
21-22	556 KB	69 KB	4,55	568 Kbps	1,08	69 KB	9,07	568 Kbps
22-23	554 KB	58 KB	4,54	475 Kbps	1,09	58 KB	9,11	475 Kbps
23-24	553 KB	58 KB	4,53	475 Kbps	1,08	69 KB	9,07	568 Kbps
24-25	553 KB	64 KB	4,53	521 Kbps	971	61 KB	7,96	498 Kbps
25-26	554 KB	64 KB	4,54	521 Kbps	1,08	67 KB	9,07	544 Kbps
26-27	556 KB	58 KB	4,55	475 Kbps	1,08	64 KB	9,07	521 Kbps
27-28	556 KB	64 KB	4,55	521 Kbps	1,09	61 KB	9,11	498 Kbps
28-29	556 KB	64 KB	4,55	521 Kbps	1,07	64 KB	9,00	521 Kbps
29-30	554 KB	57 KB	4,54	463 Kbps	1,08	64 KB	9,09	521 Kbps
0-30	6.5 MB	1.8 MB	4,54	500 Kbps	32,8	2.0 MB	9,04	536 Kbps

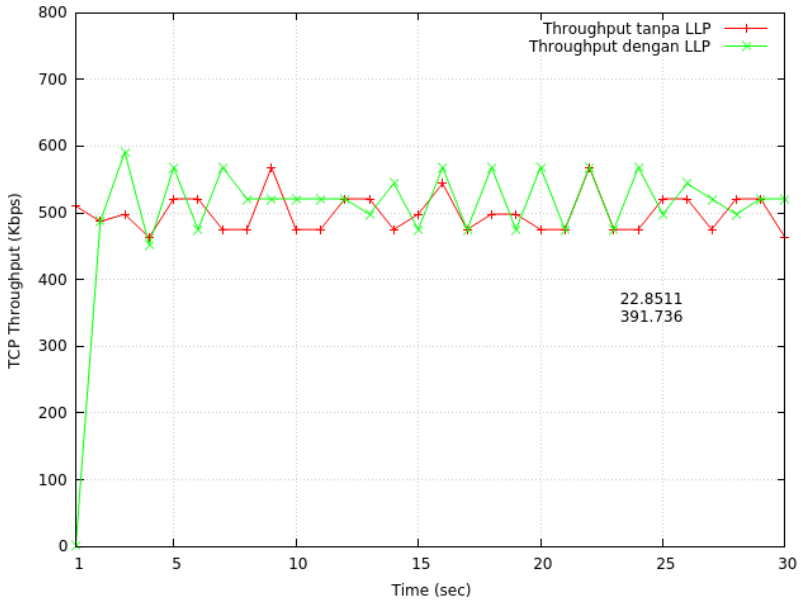
Pada Tabel 4.9 dapat dilihat hasil perbandingan *transfer rate* data dan *throughput* pada kondisi *link load* 5% dan 95%. Setelah dilakukan pengukuran dengan *Iperf*, mengunakan TCP *windows size default* 85,3 *KByte*, terjadi peningkatan *transfer rate* data dan *throughput* setelah menggunakan mekansime LLP. Penggunaan *dual path* pada saat pengiriman paket data dari h5 ke h7 dan h8 menjadikan pemanfaatan *bandwidth* yang tersedia menjadi lebih maksimal. Penggunaan mekanisme LLP juga menunjukkan kondisi *load balancing* dapat dicapai.

Untuk hasil perbandingan *throughput* S-D h5-h8 dengan menggunakan *link load* yang berbeda pada alokasi *bandwidth* 10 Mbps dapat dilihat pada Gambar 4.22 dan Gambar 4.23.



Gambar 4.22 Perbandingan *throughput* h5-h8 10 Mbps *link load* (5%)

Pada Gambar 4.22 menunjukkan penignkatan *throughput* yang cukup stabil, setelah dilakukan pengukuran selama 30 detik. Pada saat penggunaan *default* LB *throughput* nya adalah 4 Mbps, setelah menggunakan mekanisme LLP, *throughput* meningkat menjadi 9 Mbps. Hal ini dapat terjadi karena pada saat dilakukan pengiriman paket data dari h5 ke h7 dan h8 menggunakan dua *path* yang berbeda, sehingga penggunaan *bandwidth* lebih maksimal. Sementara ketika menggunakan *default* LB atau sebelum mekanisme LLP, path yang digunakan dalam pengiriman paket data hanya satu.



Gambar 4.23 Perbandingan t*hroughput* h5-h8 10 Mbps *link load* (95%)

Hasil yang cukup berbeda ditunjukkan pada kondisi *link load* 95%. Setelah dilakukan pengukuran *Iperf*, tidak terjadi peningkatan *throughput* yang cukup signifikan, hal ini terjadi karena *bandwidth* yang terpakai hampir mencapai 100%, sehingga menyebabkan pengiriman paket data yang tidak maksimal, meskipun sudah dilakukan mekanisme LLP.

Dari pengujian skenario h5-h8 dengan *bandwidth* 10 mbps, dapat disimpulkan penurunan latensi dapat terjadi setelah mekanisme LLP baik menggunakan *link load* 5% dan 95%. Peningkatan *transfer rate* data dan *throughput* juga dapat terjadi setelah mekanisme LLP pada kondisi *link load* 5%, namun pada kondisi *link load* 95% hasil *throughput* menunjukkan nilai yang hampir sama.

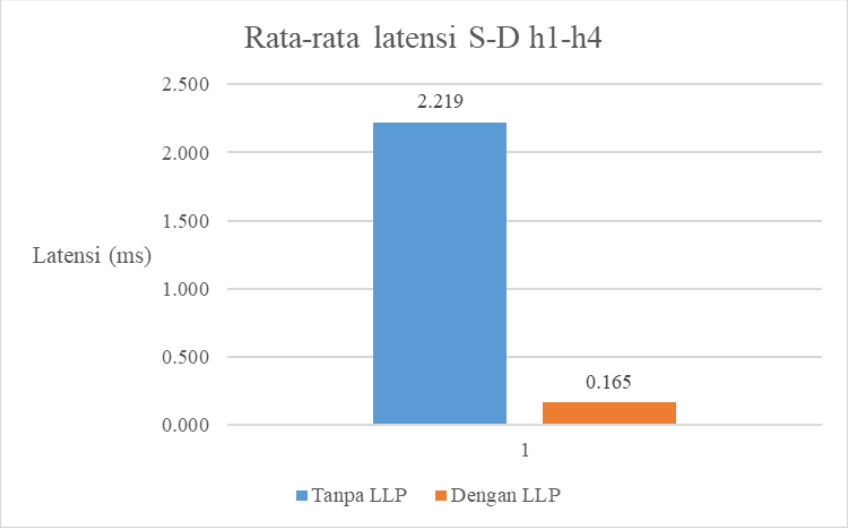
4.1.4 Hasil uji coba skenario S-D h1-h4 tanpa *link load* dan *bandwidth*

Pada pengujian skenario selanjutnya, yaitu menggunakan S-D h1-h4, tanpa menggunakan *link load* dan tidak menggunakan alokasi *bandwidth*. Langkah yang sama dengan skenario S-D h1-h4 telah dilakukan dalam melakukan pengujian skenario ini, dengan menjalankan topologi *fat tree* dan melakukan *capture* paket

dengan *wireshark* untuk memastikan *default path route* dan *best path route* Dari hasil uji coba yang dilakukan sebanyak 100 kali untuk setiap pengukuran latensi, didapatkan perbedaan latensi yang ditunjukkan pada Tabel 4.10. dan Gambar 4.24 dan menggunakan waktu 30 detik untuk setiap pengukuran *transfer rate* dan *throughput* (*bandwidth*) ditunjukkan pada Tabel 4.11 dan perbandingan *throughput* pada Gambar 4.25.

Tabel 4.10 Rata-rata latensi S-D h1-h4 tanpa LLP dan dengan LLP

Rata-rata Latensi (ms)		
Tanpa LLP	Dengan LLP	Selisih
2,219	0,165	2,054



Gambar 4.24 Grafik skenario h1-h4 tanpa *link load*

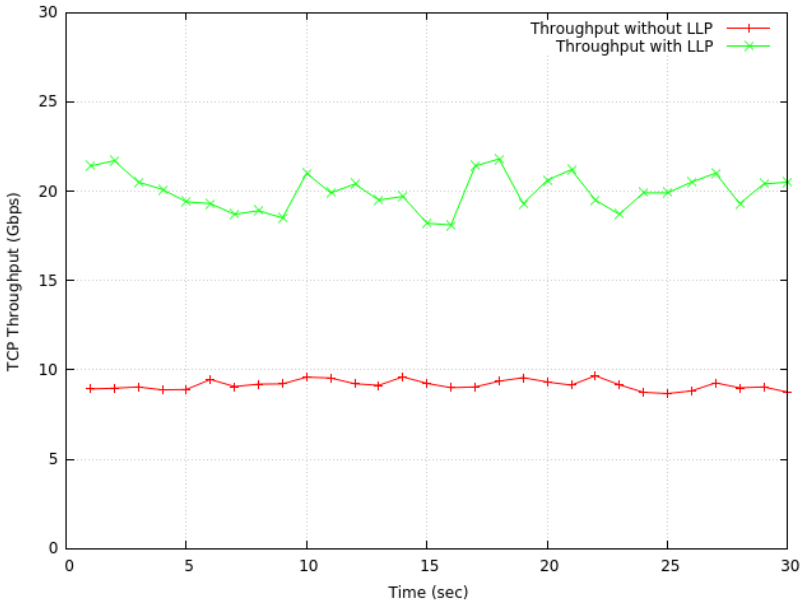
Berdasarkan Gambar 4.24 dapat dilihat bahwa hasil latensi dengan mekanisme LLP mengalami penurunan dibandingkan dengan *default path*, penurunan latensi juga terjadi pada *link load* yang lebih tinggi, hal ini disebabkan setelah menggunakan mekanisme LLP, maka *path* yang baru akan dipilih untuk mengirimkan paket data dari h1 ke h4 yaitu, melalui *path* [S1:3-S21:2-S2:3], sehingga waktu yang dibutuhkan untuk mengirimkan paket data lebih cepat. Sementara pengiriman paket data dari h1 ke h3 tetap menggunakan *path* [S1:4-

S10:2-S2:4], ini dapat dibuktikan dengan melakukan *capture* paket data dari h1 ke h3 pada S1 *port* 3, serta capture paket data dari h1 ke h4 pada S1 *port* 3.

Tabel 4.11 *Transfer rate* dan *throughput* h1-h4 tanpa *link load*

Interval (Sec)	Tanpa LLP		Dengan LLP	
	Transfer Data (Gigabyte)	Throughput (Gbps)	Transfer Data (Gigabyte)	Throughput (Gbps)
0-1	1,04	8,93	2,49	21,40
1-2	1,04	8,97	2,52	21,70
2-3	1,05	9,04	2,39	20,50
3-4	1,03	8,88	2,34	20,10
4-5	1,04	8,90	2,26	19,40
5-6	1,10	9,46	2,25	19,30
6-7	1,05	9,06	2,18	18,70
7-8	1,07	9,20	2,20	18,90
8-9	1,07	9,21	2,16	18,50
9-10	1,12	9,59	2,45	21,00
10-11	1,11	9,54	2,31	19,90
11-12	1,07	9,22	2,38	20,40
12-13	1,06	9,13	2,27	19,50
13-14	1,12	9,60	2,30	19,70
14-15	1,08	9,24	2,12	18,20
15-16	1,05	9,00	2,11	18,10
16-17	1,05	9,04	2,49	21,40
17-18	1,09	9,36	2,54	21,80
18-19	1,11	9,56	2,24	19,30
19-20	1,08	9,32	2,40	20,60
20-21	1,06	9,14	2,46	21,20
21.0-22	1,13	9,67	2,27	19,50
22.0-23	1,07	9,16	2,17	18,70
23.0-24	1,02	8,74	2,32	19,90
24.0-25	1,01	8,67	2,32	19,90
25.0-26	1,03	8,82	2,38	20,50
26.0-27	1,08	9,27	2,45	21,00
27.0-28	1,05	8,99	2,25	19,30
28.0-29	1,05	9,04	2,38	20,40
29.0-30	1,02	8,75	2,39	20,50
0.0-30	32,00	9,15	69,80	20,00

Pada Tabel 4.11 dapat dilihat hasil perbandingan *transfer rate* data dan *throughput* pada kondisi tanpa *link load* dan tanpa alokasi *bandwidth*. Setelah dilakukan pengukuran dengan *Iperf*, mengunakan *TCP window size default* 85,3 KByte, terjadi peningkatan *transfer rate* data dan *throughput* setelah menggunakan mekansime LLP. Penggunaan *dual path* pada saat pengiriman paket data dari h1 ke h3 dan h4 menjadikan pemanfaatan *bandwidth* yang tersedia menjadi lebih maksimal. Penggunaan mekanisme LLP juga menunjukkan kondisi *load balancing* dapat dicapai.

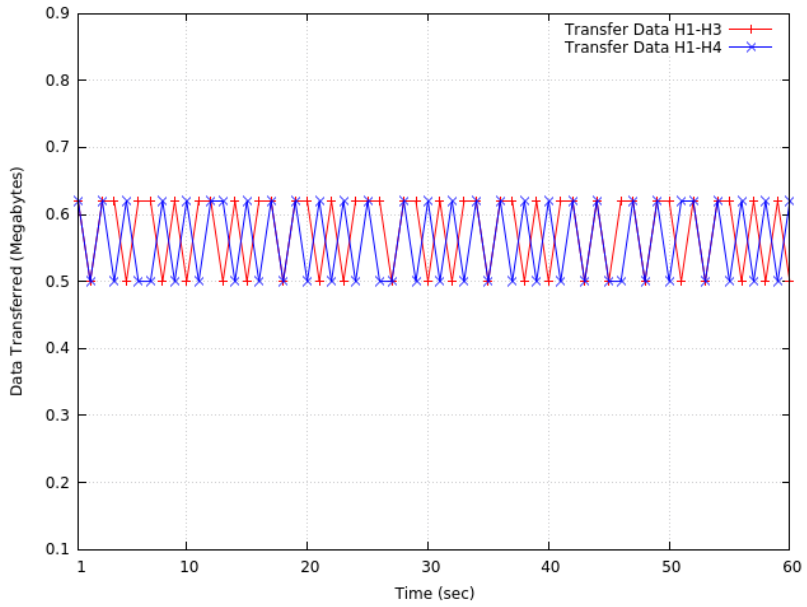


Gambar 4.25 Perbandingan *throughput* h1-h4 tanpa *link load*

Pada Gambar 4.25 menunjukkan peningkatan *throughput* yang cukup stabil, setelah dilakukan pengukuran selama 30 detik. Pada saat penggunaan *default* LB *throughput* nya adalah 9,15 Gbps, setelah menggunakan mekanisme LLP, *throughput* meningkat menjadi 20 Gbps. Hal ini dapat terjadi karena pada saat dilakukan pengiriman paket data dari h1 ke h3 dan h4 menggunakan dua *path* yang berbeda, sehingga penggunaan *bandwidth* lebih maksimal. Sementara ketika menggunakan *default* LB atau sebelum mekanisme LLP, hanya menggunakan satu *path* dalam pengiriman paket data.

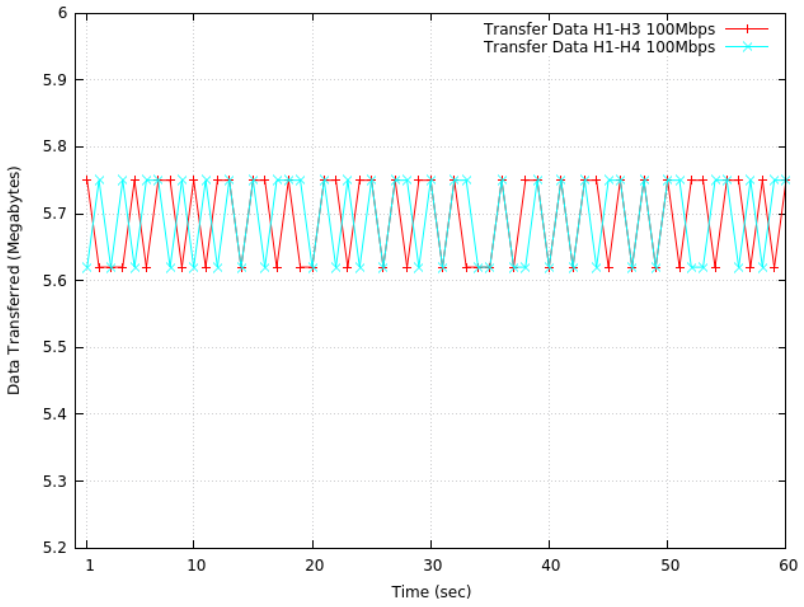
4.1.5 Hasil uji coba skenario S-D h1-h3-h4 *path balance*

Untuk pengujian terakhir dilakukan pada S-D h1-h3 dan h1-h4 dengan menggunakan aplikasi *jperf*, mengirimkan data secara bersamaan dari h1 ke h3 dan dari h1 ke h4, untuk melihat apakah *path* menjadi *balance* setelah dilakukan mekanisme LLP. Pengujian ini dilakukan pada *link bandwidth* 10 Mbps dan 100 Mbps. Hasil pengujian ditunjukkan pada Gambar 4.26 dan Gambar 4.27. Hasilnya menunjukkan baik *path* [S1:3-S21:2-S2:3] dan [S1:4-S10:2-S2:4] dapat mengirimkan paket data dengan jumlah yang hampir sama, hal ini dapat membuktikan bahwa kondisi jaringan sudah seimbang terutama pada *data plane*.



Gambar 4.26 Grafik *transfer data* H1-H3-H4 10 Mbps

Dari hasil pengujian dengan *JPerf*, selama 60 detik, TCP *Window Size* 56 *Kbyte* pada *bandwidth* 10 Mbps, menunjukkan keseimbangan pengiriman data dari H1 ke H3 dan dari H1 ke H4, transmisi data dilakukan secara bersamaan menunjukkan jumlah data yang ditransmisikan cukup stabil dan seimbang pada *range* 0,50 *Megabytes* hingga 0,62 *Megabytes* seperti yang dtampilkan pada Gambar 4.26.



Gambar 4.27 Grafik *transfer data* H1-H3-H4 100 Mbps

Sementara hasil pengiriman data yang stabil dan seimbang ditunjukkan pada Gambar 4.27. Hasil pengujian dengan *JPerf* , selama 60 detik, TCP *Window Size* 56 *Kbyte* menggunakan *bandwidth* 100 Mbps, untuk pengiriman data pada range 5,62 *Megabytes* hingga 5,75 *Megabytes*.

[Halaman ini sengaja dikosongkan]

BAB 5
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Pengujian dan analisis yang telah dilakukan menghasilkan beberapa kesimpulan, yaitu:

1. Mekanisme seleksi *path* dengan LLP (*Least Loaded Path*) adalah modifikasi dari algoritma *Dijkstra*, selain mencari *shortest path*, juga menambahkan seleksi *best path*, dengan kriteria *path* yang memiliki *cost* paling minimum, yang kemudian *path* tersebut digunakan sebagai *best path* yang baru, sehingga pengiriman paket data dapat dilakukan melalui dua *path*.
2. Berdasarkan uji coba dari beberapa skenario, pada skenario h1-h4 dengan *link load* 25%, 50% dan 85% *bandwidth* 10 Mbps dan 100 Mbps, mengalami penurunan rata-rata *latency* 5%-10%, dan peningkatan *transfer rate* data sebesar 7%-96% serta peningkatan *throughput* sebesar 49%-70%, setelah dilakukan proses mekanisme seleksi *path*.
3. Pada skenario h5-h8 dengan *link load* 5% dan 95% *bandwidth* 10 Mbps, peningkatan *throughput* yang tidak terlalu signifikan terjadi pada *link load* 95% atau nilainya hampir sama, meskipun terjadi penurunan rata-rata latensi sebesar 0.03 ms pada *link load* 5% dan sebesar 2.874 ms pada *link load* 95%. Peningkatan transfer rate data yang tidak terlalu signifikan juga terjadi pada *link load* 95% yaitu sebesar 0,2 *Megabytes* atau 10%, sedangkan peningkatan *transfer rate* data yang cukup signifikan terjadi pada *link load* 5% yaitu sebesar 26,3 *Megabytes* atau 80% setelah menggunakan mekanisme LLP.
4. Setelah dilakukan pengujian *path balance* dengan mekanisme LLP, pengiriman paket data dapat dilakukan dengan menggunakan dual *path* dari *source* ke *destination hosts* dengan memberikan hasil pengiriman data dengan jumlah yang seimbang pada setiap *path* yang digunakan. Pengujian ini dilakukan untuk melihat apakah kondisi jaringan menjadi seimbang

setelah dilakukan mekanisme LLP, terutama kondisi pada *path* yang terhubung ke *data plane*.

5. Dari beberapa pengujian tersebut, dapat disimpulkan bahwa mekanisme seleksi path dengan LLP telah berhasil menunjukkan kinerja yang baik dalam menangani distribusi pengiriman paket data pada dua *path* yang berbeda, pada kondisi *link load* yang rendah maupun tinggi. Hal ini menunnjukkan mekanisme LLP dapat mencapai load balancing pada jaringan SDN data plane.

5.2 Saran

Kontribusi penelitian ini adalah pengembangan mekanisme *path selection* pada SDN data plane menggunakan metode *least loaded path* untuk mencapai *load balancing*, dan kami dapat menyarankannya untuk diterapkan pada skala jaringan apa pun (skala kecil atau jaringan skala besar), meskipun pada jaringan skala besar performa nya dapat lebih baik karena memiliki lebih banyak alternatif rute pengiriman paket data, penerapan mekanisme ini akan membuat jaringan lebih efisien dan lebih terukur. Kami sadar bahwa penelitian ini memiliki potensi pengembangan lain, kami dapat mengusulkan topologi lain dengan ratusan *switch* dan ribuan *host* untuk menerapkan *load balancer* dengan mekanisme *path selection* sebagai penelitian kedepannya, kita akan mengetahui apakah *load balancing* dengan *path selection* cocok untuk jaringan dengan kepadatan yang tinggi. Oleh karena itu kami juga dapat mensimulasi *load balancing* di jaringan *traffic* yang tinggi, untuk mengamati kinerja dan kompatibilitas jaringan SDN.

DAFTAR PUSTAKA

Al-Fares, M., Loukissas, A. and Vahdat, A. (2008) "A Scalable, Commodity Data Center Network Architecture," *Computer Communication Review*, 38(4), hal. 63-74. doi: 10.1145/1402946.1402967.

Askar, S. (2016) "Adaptive Load Balancing Scheme for Data Center Networks Using Software Defined Network," *Science Journal of University of Zakho*, 4(2), hal. 275–286. doi: 10.25271/2016.4.2.118.

Bhandarkar, S. and Khan, K. A. (2015) "Load Balancing in Software-defined Network (SDN) Based on Traffic Volume," *Advances in Computer Science and Information Technology*, 2(7), hal. 72–76. Online ISSN : 2393-9915.

Cui, C. X. and Xu, Y. Bin (2016) "Research on Load Balance Method in SSN," *International Journal of Grid and Distributed Computing*, 9(1), hal. 25–36. doi: 10.14257/ijgdc.2016.9.1.03.

Guo, Z. *et al.* (2018) "Balancing Flow Table Occupancy And Link Utilization In Software-Defined Networks," *Future Generation Computer Systems*. Elsevier B.V., 89, hal. 213–223. doi: 10.1016/j.future.2018.06.011.

Ivancic, F. *et al.* (2014) "(12) Patent Application Publication (10) Pub . No .: US 2014 / 0332121 A1 Eiongation (%) Patent Application Publication," 1(19), hal. 33. Tersedia pada: https://patents.google.com/patent/US20140331942A1/en.

Jiang, J. R. *et al.* (2014) "Extending Dijkstra's shortest Path Algorithm for Software Defined Networking," *APNOMS 2014 - 16th Asia-Pacific Network Operations and Management Symposium*. doi: 10.1109/APNOMS.2014.6996609.

Jo, E. *et al.* (2014) "A Simulation And Emulation Study of SDN-Based Multipath Routing for Fat-Tree Data Center Networks," in *Proceedings of the 2014 Winter Simulation Conference*, hal. 1689–1699. doi: 10.1017/CBO9781107415324.004.

Joshi, N. and Gupta, D. (2019) "A Comparative Study on Load Balancing Algorithms in Software Defined Networking," *Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST*. Springer International Publishing, 276, hal. 142-150. doi: 10.1007/978-3-030-20615-4_11.

Kreutz, D. *et al.* (2015) "Software-Defined Networking: A Comprehensive

Survey", *Proceedings of the IEEE*. IEEE, 103(1), hal. 14–76. doi: 10.1109/JPROC.2014.2371999.

Li, Y. and Pan, D. (2013) "OpenFlow based Load Balancing for Fat-Tree Networks with Multipath Support," *Proc. 12th IEEE International Conference on Communications (ICC'13)*, hal. 1–5. doi: 10.1016/j.jcrs.2009.05.026.

Liu, J. *et al.* (2015) "SDN Based Load Balancing Mechanism for Elephant Flow in Data Center Networks," *International Symposium on Wireless Personal Multimedia Communications, WPMC*, 2015-January, hal. 486–490. doi: 10.1109/WPMC.2014.7014867.

Mckeown, N. *et al.* (2008) "OpenFlow: Enabling Innovation in Campus Networks," 38(2), hal. 69–74. doi: 10.1145/1355734.1355746.

Neghabi, A. *et al.* (2016) " Load Balancing Mechanisms in the Software Defined Networks: A Systematic and Comprehensive Review of the Literature," *IEEE Access*, 6(c), hal. 14159–14178. doi: 10.1109/ACCESS.2018.2805842

Saisagar, J. *et al.* (2017) "SDN Enabled Packet Based Load-Balancing (PLB) Technique in Data Center Networks," *ARPN Journal of Engineering and Applied Sciences*, 12(16), hal. 4762–4768. ISSN: 1819-6608

Tiwari, G. *et al.* (2019) "A Survey Paper on Dynamic Load Balancing in Software Defined Networking," *International Journal of Recent Technology and Engineering*, 7(6), hal. 1996–1998. ISSN: 2277-3878.

Wang, S. *et al.* (2017) "Randomized Load-balanced Routing for Fat-tree Networks," hal. 1–13. Tersedia pada: http://arxiv.org/abs/1708.09135.

Xia, W. *et al.* (2015) "A Survey on Software-Defined Networking," *IEEE Communications Surveys & Tutorials*. IEEE, 17(1), hal. 27–51. doi: 10.1109/COMST.2014.2330903.

Zhou, W. *et al.* (2014) "REST API Design Patterns for SDN Northbound API," *Proceedings - 2014 IEEE 28th International Conference on Advanced Information Networking and Applications Workshops, IEEE WAINA 2014*, hal. 358–365. doi: 10.1109/WAINA.2014.153.

LAMPIRAN

Lampiran 1. file topologi *tree_topology.py*

```
#!/usr/bin/python


from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch
from mininet.topo import Topo

class TreeTopo(Topo):

    "Tree Topology"

    def __init__(self):
        "Create simple tree Topology"

        Topo.__init__(self)

        #Add hosts
        h1 = self.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)
        h2 = self.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
        h4 = self.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)
        h3 = self.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)
        h5 = self.addHost('h5', cls=Host, ip='10.0.0.5', defaultRoute=None)
        h6 = self.addHost('h6', cls=Host, ip='10.0.0.6', defaultRoute=None)

        #Add switches
        s1 = self.addSwitch('s1', cls=OVSKernelSwitch)
        s2 = self.addSwitch('s2', cls=OVSKernelSwitch)
        s3 = self.addSwitch('s3', cls=OVSKernelSwitch)
        s4 = self.addSwitch('s4', cls=OVSKernelSwitch)
        s5 = self.addSwitch('s5', cls=OVSKernelSwitch)

        #Add links
        self.addLink(s1, s5)
        self.addLink(s1, s3)
        self.addLink(s5, s4)
        self.addLink(s5, s3)
        self.addLink(s1, s2)
        self.addLink(s2, s3)
        self.addLink(s3, s4)
        self.addLink(s1, s4)
        self.addLink(s2, h1)
        self.addLink(s2, h2)
        self.addLink(s3, h3)
        self.addLink(s3, h4)
        self.addLink(s4, h5)
        self.addLink(s4, h6)

topos = { 'mytopo': (lambda: TreeTopo() ) }
```

Lampiran 2. file topologi *fatTreeTopo.py*

```
#!/usr/bin/python


from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch
from mininet.topo import Topo

class fatTreeTopo(Topo):

    "Fat Tree Topology"

    def __init__(self):
        "Create Fat tree Topology"

        Topo.__init__(self)

        #Add hosts
        h7 = self.addHost('h7', cls=Host, ip='10.0.0.7', defaultRoute=None)
        h8 = self.addHost('h8', cls=Host, ip='10.0.0.8', defaultRoute=None)
        h1 = self.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)
        h2 = self.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
        h4 = self.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)
        h3 = self.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)
        h5 = self.addHost('h5', cls=Host, ip='10.0.0.5', defaultRoute=None)
        h6 = self.addHost('h6', cls=Host, ip='10.0.0.6', defaultRoute=None)

        #Add switches
        s10 = self.addSwitch('s10', cls=OVSKernelSwitch)
        s3 = self.addSwitch('s3', cls=OVSKernelSwitch)
        s17 = self.addSwitch('s17', cls=OVSKernelSwitch)
        s4 = self.addSwitch('s4', cls=OVSKernelSwitch)
        s18 = self.addSwitch('s18', cls=OVSKernelSwitch)
        s1 = self.addSwitch('s1', cls=OVSKernelSwitch)
        s11 = self.addSwitch('s11', cls=OVSKernelSwitch)
        s21 = self.addSwitch('s21', cls=OVSKernelSwitch)
        s22 = self.addSwitch('s22', cls=OVSKernelSwitch)
        s2 = self.addSwitch('s2', cls=OVSKernelSwitch)

        #Add links
        self.addLink(h1, s1)
        self.addLink(h2, s1)
        self.addLink(h3, s2)
        self.addLink(h4, s2)
        self.addLink(h5, s3)
        self.addLink(h6, s3)
        self.addLink(h7, s4)
        self.addLink(h8, s4)
        self.addLink(s1, s21)
        self.addLink(s21, s2)
        self.addLink(s1, s10)
        self.addLink(s2, s10)
        self.addLink(s3, s11)
        self.addLink(s4, s22)
        self.addLink(s11, s4)
        self.addLink(s3, s22)
        self.addLink(s21, s17)
        self.addLink(s11, s17)
```

Lampiran 3. File kontroler *floodlight.sh*

```
#!/bin/sh

# Set paths
FL_HOME=`dirname $0`
FL_JAR="${FL_HOME}/target/floodlight.jar"
FL_LOGBACK="${FL_HOME}/logback.xml"

# Set JVM options
JVM_OPTS=""
JVM_OPTS="$JVM_OPTS -server -d64"
JVM_OPTS="$JVM_OPTS -Xmx2g -Xms2g -Xmn800m"
JVM_OPTS="$JVM_OPTS -XX:+UseParallelGC -XX:+AggressiveOpts -XX:+UseFastAccess
orMethods"
JVM_OPTS="$JVM_OPTS -XX:MaxInlineSize=8192 -XX:FreqInlineSize=8192"
JVM_OPTS="$JVM_OPTS -XX:CompileThreshold=1500 -XX:PreBlockSpin=8"
JVM_OPTS="$JVM_OPTS -Dpython.security.respectJavaAccessibility=false"

# Create a logback file if required
[ -f ${FL_LOGBACK} ] || cat <<EOF_LOGBACK >${FL_LOGBACK}
<configuration scan="true">
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%level [%logger:%thread] %msg%n</pattern>
        </encoder>
    </appender>
    <root level="INFO">
        <appender-ref ref="STDOUT" />
    </root>
    <logger name="org" level="ALL"/>
    <logger name="LogService" level="DEBUG"/> <!-- Restlet access logging -->
    <logger name="net.floodlightcontroller" level="ALL"/>
    <logger name="net.floodlightcontroller.logging" level="ALL"/>
</configuration>
EOF_LOGBACK

echo "Starting floodlight server ..."
java ${JVM_OPTS} -Dlogback.configurationFile=${FL_LOGBACK} -jar ${FL_JAR}
```

Lampiran 4. File modul *load balancer loadbalancer.py*

```
#!/usr/bin/env python

import requests
import json
import unicodedata
from subprocess import Popen, PIPE
import time
import networkx as nx
from sys import exit

# Method To Get REST Data In JSON Format
def getResponse(url,choice):

        response = requests.get(url)

        if(response.ok):
                jData = json.loads(response.content)
                if(choice=="deviceInfo"):
                        deviceInformation(jData)
                elif(choice=="findSwitchLinks"):
                        findSwitchLinks(jData,switch[h2])
                elif(choice=="linkTX"):
                        linkTX(jData,portKey)

        else:
                response.raise_for_status()

# Parses JSON Data To Find Switch Connected To H4
def deviceInformation(data):
        global switch
        global deviceMAC
        global hostPorts
        switchDPID = ""
        for i in data:
                if(i['ipv4']):
                        ip = i['ipv4'][0].encode('ascii','ignore')
                        mac = i['mac'][0].encode('ascii','ignore')
                        deviceMAC[ip] = mac
                        for j in i['attachmentPoint']:
                                for key in j:
                                        temp = key.encode('ascii','ignore')
                                        if(temp=="switchDPID"):
                                                switchDPID = j[key].encode('a
scii','ignore')

                                                switch[ip] = switchDPID
                                        elif(temp=="port"):
                                                portNumber = j[key]
                                                switchShort = switchDPID.spli
t(":")[7]

                                                hostPorts[ip+ "::" + switchSh
ort] = str(portNumber)

# Finding Switch Links Of Common Switch Of H3, H4

def findSwitchLinks(data,s):
        global switchLinks
        global linkPorts
```

```
        global G

        links=[]
        for i in data:
                src = i['src-switch'].encode('ascii','ignore')
                dst = i['dst-switch'].encode('ascii','ignore')

                srcPort = str(i['src-port'])
                dstPort = str(i['dst-port'])

                srcTemp = src.split(":")[7]
                dstTemp = dst.split(":")[7]

                G.add_edge(int(srcTemp,16), int(dstTemp,16))

                tempSrcToDst = srcTemp + "::" + dstTemp
                tempDstToSrc = dstTemp + "::" + srcTemp

                portSrcToDst = str(srcPort) + "::" + str(dstPort)
                portDstToSrc = str(dstPort) + "::" + str(srcPort)

                linkPorts[tempSrcToDst] = portSrcToDst
                linkPorts[tempDstToSrc] = portDstToSrc

                if (src==s):
                        links.append(dst)
                elif (dst==s):
                        links.append(src)
                else:
                        continue

        switchID = s.split(":")[7]
        switchLinks[switchID]=links

# Finds The Path To A Switch

def findSwitchRoute():
        pathKey = ""
        nodeList = []
        src = int(switch[h2].split(":",7)[7],16)
        dst = int(switch[h1].split(":",7)[7],16)
        print src
        print dst
        for currentPath in nx.all_shortest_paths(G, source=src, target=dst, w
eight=None):
                for node in currentPath:

                        tmp = ""
                        if node < 17:
                                pathKey = pathKey + "0" + str(hex(node)).spli
t("x",1)[1] + "::"
                                tmp = "00:00:00:00:00:00:00:0" + str(hex(node
)).split("x",1)[1]
                        else:
                                pathKey = pathKey + str(hex(node)).split("x",
1)[1] + "::"
@
```

```
                                tmp = "00:00:00:00:00:00:00:" + str(hex(node)
).split("x",1)[1]
                        nodeList.append(tmp)

                pathKey=pathKey.strip("::")
                path[pathKey] = nodeList
                pathKey = ""
                nodeList = []

        print path

# Computes Link TX

def linkTX(data,key):
        global cost
        port = linkPorts[key]
        port = port.split("::")[0]
        for i in data:
                if i['port']==port:
                        cost = cost + (int)(i['bits-per-second-tx'])


# Method To Compute Link Cost

def getLinkCost():
        global portKey
        global cost

        for key in path:
                start = switch[h2]
                src = switch[h2]
                srcShortID = src.split(":")[7]
                mid = path[key][1].split(":")[7]
                for link in path[key]:
                        temp = link.split(":")[7]

                        if srcShortID==temp:
                                continue
                        else:
                                portKey = srcShortID + "::" + temp
                                stats = "http://localhost:8080/wm/statistics/
bandwidth/" + src + "/0/json"
                                getResponse(stats,"linkTX")
                                srcShortID = temp
                                src = link
                portKey = start.split(":")[7] + "::" + mid + "::" + switch[h1
].split(":")[7]
                finalLinkTX[portKey] = cost
                cost = 0
                portKey = ""

def systemCommand(cmd):
        terminalProcess = Popen(cmd, stdout=PIPE, stderr=PIPE, shell=True)
        terminalOutput, stderr = terminalProcess.communicate()
        print "\n***", terminalOutput, "\n"

def flowRule(currentNode, flowCount, inPort, outPort, staticFlowURL):
```

```
def flowRule(currentNode, flowCount, inPort, outPort, staticFlowURL):
        flow = {
                'switch':"00:00:00:00:00:00:00:" + currentNode,
            "name":"flow" + str(flowCount),
            "cookie":"0",
            "priority":"32768",
            "in_port":inPort,
                "eth_type": "0x0800",
                "ipv4_src": h2,
                "ipv4_dst": h1,
                "eth_src": deviceMAC[h2],
                "eth_dst": deviceMAC[h1],
            "active":"true",
            "actions":"output=" + outPort
        }

        jsonData = json.dumps(flow)

        cmd = "curl -X POST -d \'" + jsonData + "\' " + staticFlowURL

        systemCommand(cmd)

        flowCount = flowCount + 1

        flow = {
                'switch':"00:00:00:00:00:00:00:" + currentNode,
            "name":"flow" + str(flowCount),
            "cookie":"0",
            "priority":"32768",
            "in_port":outPort,
                "eth_type": "0x0800",
                "ipv4_src": h1,
                "ipv4_dst": h2,
                "eth_src": deviceMAC[h1],
                "eth_dst": deviceMAC[h2],
            "active":"true",
            "actions":"output=" + inPort
        }

        jsonData = json.dumps(flow)

        cmd = "curl -X POST -d \'" + jsonData + "\' " + staticFlowURL

        systemCommand(cmd)

def addFlow():
        print "Proses sedang dikerjakan"

        # Deleting Flow
        #cmd = "curl -X DELETE -d \'{\"name\":\"flow1\"}\' http://127.0.0.1:8
080/wm/staticflowpusher/json"
        #systemCommand(cmd)

        #cmd = "curl -X DELETE -d \'{\"name\":\"flow2\"}\' http://127.0.0.1:8
080/wm/staticflowpusher/json"
        #systemCommand(cmd)
```

```
        flowCount = 1
        staticFlowURL = "http://127.0.0.1:8080/wm/staticflowpusher/json"

        shortestPath = min(finalLinkTX, key=finalLinkTX.get)
        print "\n\nShortest Path: ",shortestPath


        currentNode = shortestPath.split("::",2)[0]
        nextNode = shortestPath.split("::")[1]

        # Port Computation

        port = linkPorts[currentNode+"::"+nextNode]
        outPort = port.split("::")[0]
        inPort = hostPorts[h2+"::"+switch[h2].split(":")[7]]

        flowRule(currentNode,flowCount,inPort,outPort,staticFlowURL)

        flowCount = flowCount + 2


        bestPath = path[shortestPath]
        previousNode = currentNode

        for currentNode in range(0,len(bestPath)):
                if previousNode == bestPath[currentNode].split(":")[7]:
                        continue
                else:
                        port = linkPorts[bestPath[currentNode].split(":")[7]+
"::"+previousNode]
                        inPort = port.split("::")[0]
                        outPort = ""
                        if(currentNode+1<len(bestPath) and bestPath[currentNo
de]==bestPath[currentNode+1]):
                                currentNode = currentNode + 1
                                continue
                        elif(currentNode+1<len(bestPath)):
                                port = linkPorts[bestPath[currentNode].split(
":")[7]+"::"+bestPath[currentNode+1].split(":")[7]]
                                outPort = port.split("::")[0]
                        elif(bestPath[currentNode]==bestPath[-1]):
                                outPort = str(hostPorts[h1+"::"+switch[h1].sp
lit(":")[7]])

                        flowRule(bestPath[currentNode].split(":")[7],flowCoun
t,str(inPort),str(outPort),staticFlowURL)
                        flowCount = flowCount + 2
                        previousNode = bestPath[currentNode].split(":")[7]

# Method To Perform Load Balancing
def loadbalance():
        linkURL = "http://localhost:8080/wm/topology/links/json"
        getResponse(linkURL,"findSwitchLinks")

        findSwitchRoute()
        getLinkCost()
```

```
        addFlow()

# Main

# Stores H1 and H2 from user
global h1,h2,h3

h1 = ""
h2 = ""

print "Enter Host 1"
h1 = int(input())
print "\nEnter Host 2"
h2 = int(input())
print "\nEnter Host 3 (H2's Neighbour)"
h3 = int(input())

h1 = "10.0.0." + str(h1)
h2 = "10.0.0." + str(h2)
h3 = "10.0.0." + str(h3)


while True:

        # Stores Info About H3 And H4's Switch
        switch = {}

        # Mac of H3 And H4
        deviceMAC = {}

        # Stores Host Switch Ports
        hostPorts = {}

        # Stores Switch To Switch Path
        path = {}

        # Switch Links

        switchLinks = {}

        # Stores Link Ports
        linkPorts = {}

        # Stores Final Link Rates
        finalLinkTX = {}

        # Store Port Key For Finding Link Rates
        portKey = ""

        # Stores Link Cost
        cost = 0
        # Graph
        G = nx.Graph()

        try:

                # Enables Statistics Like B/W, etc
```

```
                # Enables Statistics Like B/W, etc
                enableStats = "http://localhost:8080/wm/statistics/config/ena
ble/json"
                requests.put(enableStats)

                # Device Info (Switch To Which The Device Is Connected & The
MAC Address Of Each Device)
                deviceInfo = "http://localhost:8080/wm/device/"
                getResponse(deviceInfo,"deviceInfo")

                # Load Balancing

                loadbalance()

                # -------------- PRINT --------------

                print "\n\n############ RESULT ############\n\n"

                # Print Switch To Which H4 is Connected
                print "Switch H4: ",switch[h3], "\tSwitchH3: ", switch[h2]

                print "\n\nSwitch H1: ", switch[h1]

                # IP & MAC
                print "\nIP & MAC\n\n", deviceMAC

                # Host Switch Ports
                print "\nHost::Switch Ports\n\n", hostPorts

                # Link Ports
                print "\nLink Ports (SRC::DST - SRC PORT::DST PORT)\n\n", lin
kPorts

                # Alternate Paths
                print "\nPaths (SRC TO DST)\n\n",path

                # Final Link Cost
                print "\nFinal Link Cost (First To Second Switch)\n\n",finalL
inkTX

                print "\n\n########################################\n\n"

                time.sleep(60)

        except KeyboardInterrupt:
                break
                exit()
```

BIODATA PENULIS

Mhd. Fattahilah Rangkuty, lahir di Medan, 19 Juli 1985. Penulis adalah anak kedua dari enam bersaudara, Penulis menempuh pendidikan sekolah dasar di SDN 060927 Medan Johor lalu melanjutkan pendidikan Madrasah Tsanawiyah di MTs YPI Deli Tua dan penulis menempuh pendidikan menengah atas di SMK Negeri 2 Medan. Selanjutnya penulis melanjutkan pendidikan D4 di Politeknik Negeri Batam dan melanjutkan pendidikan pascasarjana di Departemen Teknik Informatika, Fakultas Teknologi Elektro dan Informatika Cerdas, Institut Teknologi Sepuluh Nopember Surabaya.

Sebelum melanjutkan pendidikan D4, tahun 2007 hingga 2009 penulis pernah bekerja sebagai process engineering technician di perusahaan internasional PT. AMC-Bintan, merupakan perusahaan PCBA yang memproduksi alat instrumen elektronik untuk pesawat udara. Saat pendidikan diploma 4, penulis mengambil bidang Teknik Informatika. Sebagai mahasiswa, penulis pernah mempublikasikan penelitiannya ke seminar internasional IEEE melalui program kolaborasi penelitian dengan dosen informatika politeknik negeri batam.

Saat ini penulis bekerja sebagai staf UPT-SI (Unit Pelaksana Teknis Sistem Informasi) di Politeknik Negeri Batam. Mulai bergabung dengan Polibatam sejak tahun 2009. Untuk menyelesaikan pendidikan pascasarjana, penulis merupakan awardee atau karyasiswa program beasiswa PasTi Kemenristekdikti Tahun Anggaran 2018. Dalam menyelesaikan pendidikan pascasarjana, penulis mengambil bidang minat Komputasi Berbasis Jaringan (KBJ). Penulis dapat dihubungi melalui nomor handphone: 081372829480 atau email: fattahilah.rangkuty@gmail.com.

[Halaman ini sengaja dikosongkan]