OPTIMALISASI ROUTING MENGGUNAKAN MODIFIKASI ALGORITMA *DEPTH FIRST SEARCH* PADA SOFTWARE DEFINED NETWORK

TUGAS AKHIR

Diajukan Untuk Memenuhi Salah Satu Syarat Kelulusan
Sarjana Strata 1

Oleh
Haris Abdul Afif 19201062

PROGRAM STUDI TEKNIK INFORMATIKA
FAKULTAS TEKNOLOGI DAN DESAIN
INSTITUT TEKNOLOGI DAN BISNIS ASIA MALANG
2023

PERSETUJUAN TUGAS AKHIR

Judul	:	Optimalisasi Routing Menggunakan Modifikasi Algoritma Depth First Search pada Software Defined Network
Oleh	:	Haris Abdul Afif
NIM	:	19201062
Program Studi	:	Teknik Informatika

Malang, [……..]

Menyetujui
Dosen Pembimbing

Fransiska Sisilia Mukti S.T., M.T.

Ketua Prodi Informatika

Jaenal Arifin, S.Kom., M.M., M.Kom

KETERANGAN LULUS UJIAN

Yang bertanda tangan dibawah ini menerapkan bahwa:

Nama : Haris Abdul Afif
NIM : 19201062
Program Studi : Teknik Informatika

Telah lulus ujian Tugas Akhir pada tanggal [……] di Insitut Teknologi dan Bisnis Asia Malang.

Malang, [……..]

Tim Penguji
Ketua Tim Penguji

[……]

Penguji 1

[……]

Penguji 2

[……]

PERNYATAAN KEASLIAN

Yang bertanda tangan dibawah ini adalah:

Nama	: Haris Abdul Afif
NIM	: 19201062
Tempat/Tgl Lahir	: Surabaya, 14 Desember 2000
Program Studi	: Teknik Informatika
Alamat	: Jl. Candi Panggung no 17, Lowokwaru

Menyatakan bahwa Karya Ilmiah/Tugas Akhir yang berjudul:

"Optimalisasi Routing Menggunakan Modifikasi Algoritma Depth First Search pada Software Defined Network"

Adalah bukan merupakan karya tulis orang lain, baik sebagian maupun keseluruhan, kecuali dalam bentuk kutipan yang disebutkan sumbernya.

Demikian surat pernyataan ini saya buat dengan sebenar-benarnya dan apabila pernyataan ini tidak benar, saya bersedia menerima sanksi akademik.

Malang […….]

Mengetahui,
Dosen Pembimbing

Fransiska Sisilia Mukti S.T., M.T

Yang menyatakan

Haris Abdul Afif

PERNYATAAN PERSETUJUAN PUBLIKASI

Sebagai Civitas Akademik Institut Teknologi dan Bisnis Asia Malang, saya yang bertanda tangan di bawah ini:

Nama : Haris Abdul Afif
NIM : 19201062
Program Studi : Teknik Informatika
Jenis Karya : Tugas Akhir

Demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Institut Teknologi dan Bisnis Asia Malang Hak Bebas Royalti atas tugas akhir yang berjudul:

"Optimalisasi Routing Menggunakan Modifikasi Algoritma Depth First pada Software Defined Network"

Beserta perangkat yang ada (jika diperlukan)

Dengan Hak Bebas Royalti ini Institut Teknologi dan Bisnis Asia Malang berhak untuk menyimpan, mengalih media/formatkan, mengelola dalam bentuk pangkalan data (database), merawat dan mempublikasikan Tugas Akhir sata tanpa meminta ijin dari saya selama tahap mencantumkan nama saya sebagai penulis/pencipta dan pemilik hak cipta.

Demikian pernyataan ini saya buat dengan sebenarnya.

Malang, [……..]

Yang menyatakan

Materai
Rp. 6000

Haris Abdul Afif

ABSTRAKSI

Haris Abdul Afif. 19201062

OPTIMALISASI ROUTING MENGGUNAKAN MODIFIKASI ALGORITMA DEPTH FIRST SEARCH PADA SOFTWARE DEFINED NETWORK

Teknik Informatika, Institut Teknologi dan Bisnis ASIA Malang, 2023
Kata Kunci : Optimalisasi Routing, Modifikasi Depth First Search, Software Defined Network (SDN)
(xiv + 50 + Lampiran)

Routing, adalah proses menentukan rute atau jalur yang diambil oleh paket dimana mereka mengalir dari pengirim ke penerima. Algoritma yang menghitung jalur ini disebut algoritma *routing*. Sebagian besar protokol *routing* yang digunakan saat ini menggunakan algoritma yang menghasilkan satu jalur tunggal saja. Perhitungan ini dilakukan secara otomatis dengan menggunakan nilai satuan beban (metrik) yang dihitung. Selain itu nilai metrik ini ditetapkan tanpa mempertimbangkan beban trafik. Sehingga menyebabkan peningkatan rasio kemacetan dalam jaringan.

Multipath Routing atau (MP) merupakan pendekatan alternatif yang dapat digunakan untuk menyelesaikan permasalahan yang ditimbulkan dari penggunaan *single-path routing* (SP). Peningkatan performa jaringan dicapai melalui penggunaan beberapa jalur secara Bersama. Penggabungan antara *Multipath routing* dan algoritma pencarian jalur yang popular seperti *Depth First Search* (DFS) mampu menemukan lebih dari satu jalur, namun masih terdapat persamaan pada jalurnya, sehingga perlu adanya modifikasi pada algoritma yang digunakan agar menemukan jalur yang independen.

Skenario pengujian menggunakan 10 skenario dengan beban *bandwidth* 100 Mbit – 1000 Mbit selama 10 detik, dengan parameter pengujian pencarian jalur, *Quality of Service* (QoS) (*delay, throughput, jitter,* dan *packet loss*). Pada hasil pengujian pencarian jalur, algoritma Modifikasi DFS mampu menemukan jalur – jalur yang tidak memiliki kesamaan. Pada pengujian QoS, *Delay, Throughput, Jitter, dan Packet Loss* memiliki nilai yang cukup stabil dan mengalami peningkatan.

Daftar Pustaka (2017 – 2023)

ABSTRACT

Haris Abdul Afif. 19201062
ROUTING OPTIMIZATION USING DEPTH FIRST SEARCH ALGORITHMS IN DEFINED NETWORK SOFTWARE
Informatics Engineering, ASIA Institute of Technology and Business Malang, 2023
Keywords : Routing Optimization, Depth First Search Modification, Software Defined Network (SDN)
(xiv + 50 + Attachments)

Routing is the process of determining the route or path taken by packets as they flow from the sender to the receiver. The algorithms that calculate these paths are referred to as routing algorithms. Most of the routing protocols used today employ algorithms that produce only a single path. This calculation is automated using a unit value of load (metric) that is calculated. Additionally, this metric value is assigned without considering traffic load, leading to an increase in congestion ratio within the network.

Multipath Routing, or (MP), is an alternative approach that can be used to solve the issues arising from the use of single-path routing (SP). Enhanced network performance is achieved by utilizing multiple paths simultaneously. The combination of Multipath Routing and well-known path-finding algorithms like DFS can discover more than one path, but similarities still exist among these paths. Therefore, there is a need for modifications to the utilized algorithm to find independent paths.

The testing scenario involves 10 scenarios with bandwidth loads ranging from 100 Mbps to 1000 Mbps for 10 seconds, with path-finding testing parameters and Quality of Service (QoS) evaluations including delay, throughput, jitter, and packet loss. In the path-finding test results, the Modified DFS algorithm managed to discover paths that are distinct. In QoS testing, Delay, Throughput, Jitter, and Packet Loss have values that are quite stable and have increased.

Bibliography (2017 - 2023)

KATA PENGANTAR

Puji syukur kehadirat Tuhan Yang Maha Esa atas segala rahmat dan hidayahnya sehingga laporan tugas akhir yang berjudul ”Optimalisasi Routing Menggunakan Modifikasi Algoritma Depth First Search pada Software Defined Network” ini dapat tersusun hingga selesai.

Tidak lupa penulis juga mengucapkan banyak terimakasih atas bantuan dari pihak yang telah berkontribusi dengan memberikan sumbangan baik materi maupun pikirannya, yaitu:

1. Ibu Risa Santoso., B.A., M. Ed., selaku Rektor Institut Teknologi dan Bisnis Asia Malang.
2. Ibu Rina Dewi Indah Sari., S.Kom., M.Kom, selaku dekan fakultas Teknologi dan Desain Teknik Informatika Institut Teknolgi dan Bisnis Asia Malang.
3. Bapak Jaenal Arifin., S.Kom., M.M, M.Kom, selaku ketua program studi Teknik Informatika.
4. Ibu Fransiska Sisilia Mukti S.T., M.T, selaku dosen pembimbing.
5. Bapak Lukman Hakim S.Si., M.Si. selaku dosen wali.
6. Bapak, Ibu, dan Saudara yang selalu memberikan dukungan dan semangat untuk menyelesaikan Tugas Akhir.

Dan harapan penulis semoga laporan tugas akhir ini dapat menambah pengetahuan dan pengalaman bagi para pembaca, untuk kedepannya dapat memperbaiki bentuk maupun menambah isi laporan tugas akhir agar menjadi lebih baik lagi.

Karena keterbatasan pengetahuan maupun pengalaman penulis, penulis yakin masih banyak kekurangan dalam laporan ini, dapat membangun dari pembaca demi kesempurnaan laporan tugas akhir.

Malang, [……]

Penulis

DAFTAR ISI

Halaman

DAFTAR GAMBAR

Gambar **Halaman**

DAFTAR TABEL

Tabel **Halaman**

DAFTAR PERSAMAAN

Persamaan **Halaman**

DAFTAR LAMPIRAN

BAB I
PENDAHULUAN

1.1 Latar Belakang Masalah

Routing, adalah proses menentukan rute atau jalur yang diambil oleh paket dimana mereka mengalir dari pengirim ke penerima. Algoritma yang menghitung jalur ini disebut algoritma routing **(Wibowo et al., 2018).** Mengingat pentingnya perutean untuk jaringan komunikasi, beragam perutean algoritma telah dirancang. Memberikan kualitas layanan (QoS) merupakan salah satu persyaratan penting untuk berbagai pengaturan jaringan komunikasi dan aplikasi. Penggunaan *Routing* yang lazim digunakan adalah *single-path* karena cukup mudah dalam hal perutean dan pengaturan. Sehingga *single-path Routing* menjadi pilihan awal daripada digunakannya *Multipath Routing* pada jaringan.

Single-path Routing atau satu jalur konvensional, seperti algoritma *Dijkstra* pada OSPF (*Open Shortest Path First),* pengirman *packet* hanya satu jalur saja, walaupun disediakan dua atau lebih jalur menuju *destination host.* **(Sutawijaya et al., 2020)** Hal ini mengakibatkan distribusi trafik menjadi tidak seimbang. Oleh karena itu sebagai alternative lain dalam pengiriman pengiriman data dengan tujuan mendistribusikan beban dan mengurangi *congestion* maka digunakan skema baru pengiriman data yaitu *multipath* routing **(Suryo Wicaksono & Hari Trisnawan, 2021)**.

Multipath Routing merupakan suatu metode *Routing* dengan menggunakan beberapa jalur yang yang tersedia. Berbeda dengan *single-path Routing* yang hanya menggunakan satu jalur ke sumber tujuan. Pada algoritma *single-path Routing*, keseluruhan jalur yang ada pada topologi jaringan yang disebut dengan kemampuan *Multipath* tidak dapat sepenuhnya dimanfaatkan. Seperti pada algoritma spanning tree, topologi jaringan selalu dipotong dan dikurangi menjadi bentuk tree (pohon) sehingga kemampuan *Multipath* pada topologi yang memiliki cabang redundan (berlebihan) tidak dapat dimanfaatkan dan menyebabkan pemborosan sumber daya jaringan **(Yudha et al., 2018).** Dengan *Multipath Routing* topologi jaringan yang memiliki cabang redundan dapat dimanfaatkan sepenuhnya. Selain itu, *Multipath Routing* dapat mengurangi kemacetan (congestion) pada jaringan **(Yudha et al., 2018).**

Implementasi *Multipath Routing* pada arsitektur jaringan saat ini sangat sulit karena tidak ada antarmuka untuk melakukan eksperimen

dan pengambangan. Solusi dari permasalahan tersebut dapat diselesaikan dengan menggunakan konsep jaringan atau paradigma yang disebut *Software-Defined Networking* (SDN). SDN adalah konsep atau paradigma jaringan yang memiliki sebuah perangkat lunak utama yang disebut *controller* sebagai penentu perilaku keseluruhan jaringan. Pada SDN, control plane yaitu kecerdasan yang diterapkan di *controller* dipisahkan dengan data plane yaitu perangkat yang melakukan penerusan paket **(Yudha et al., 2018)**.

Pada perkembangan *multipath* routing saat ini, sudah ada beberapa penelitian yang sudah menggunakan beberapa algoritma yang sudah dilakukan pengujian menggunakan *multipath* routing, antara lain adalah algoritma Dijkstra, dan algoritma DFS. Dalam hal ini algoritma DFS merupakan algoritma yang cukup populer dalam pencarian simpul secara mendalam dalam sebuah topologi *graph tree*. Penelitian yang dilakukan W. Maulana et al., tentang *'Multipath Routing dengan Load Balancing pada Openflow Software Defined Network"* menerapkan *routing* menggunakan algoritma pencarian jalur yang masih menggunakan DFS konvensional sehingga jalur yang ditemukan merupakan jalur yang tidak independen dan memiliki kesamaan sehingga bisa menyebabkan *congestion*.

Berdasar dari permasalahan tersebut , maka fokus dari penelitian ini adalah mengimplementasikan *Multipath Routing* menggunakan algoritma *Depth First Search* (DFS) yang dimodifikasi untuk mengatasi permasalahan pemerataan utilisasi dan kekurangan dari singlepath *Routing*. Algoritma DFS yang dimodifikasi digunakan untuk mencari beberapa jalur terpendek yang independen atau kesamaan jalur.

1.2 Rumusan Masalah

a. Bagaimana menemukan jalur *Multipath Routing* pada jaringan openflow menggunakan algoritma DFS yang dimodifikasi ?
b. Bagaimana pemerataan utilisasi jaringan pada *Multipath Routing* dengan *Software defined network* ?
c. Bagaimana kinerja dari *Multipath Routing* berbasis algoritma DFS yang dimodifikasi ?

1.3 Batasan Masalah

a. Protokol SDN yang digunakan adalah Openflow menggunakan Ryu controller sebagai controllernya.

b. Penelitian menggunakan emulator Mininet 2.2.2 pada linux Lubuntu 20.04.
c. Topologi yang digunakan adalah fat-tree.
d. Parameter pengujian meliputi pencarian jalur, delay, throughput, jitter, dan packet loss.

1.4 Tujuan Dan Manfaat Penelitian

1.4.1 Tujuan

a. Mengimplementasikan algoritma DFS yang dimodifikasi sehingga dapat menemukan jalur dalam *Multipath Routing* jaringan Openflow.
b. Membantu Network Administrator dalam mengatur dan memposisikan trafik jaringan multi-path dalam *software defined network*.
c. Menganalisis kinerja dari *Multipath Routing* berbasis algoritma DFS yang dimodifikasi pada jaringan Openflow.

1.4.2 Manfaat Bagi Penulis

a. Mengaplikasikan disiplin ilmu yang telah diperoleh selama belajar di Institut Asia Malang Program Studi Informatika
b. Dapat meningkatkan pengetahuan teoritis dan aplikatif bagi penulis.
c. Meningkatkan kemampuan coding dalam di Bahasa pemrograman python.

1.4.3 Manfaat Bagi Institut Teknologi dan Bisnis Asia Malang

a. Dapat mengukur sejauh mana keberhasilan proses belajar mengajar di dalam kelas, dan capaian materinya.
b. Menjadi bahan kajian yang dapat dikembangkan dikemudian hari.

1.5 Metodologi Penelitian

Untuk mendukung penyelesaian penelitian ini digunakan beberapa metodologi, yaitu :

a. Studi Literatur

Dengan mempelajari buku – buku referensi dan jurnal yang berkaitan dengan permasalahan yang diangkat serta

mencari solusi yang terbaik dengan tujuan memperoleh dasar teoritis gambaran dari apa yang dilakukan. Teori yang dipelajari yaitu : *Software defined network* (SDN), Manajemen trafik, dan *Multipath Routing*.

b. Analisa

   Melakukan uji coba secara teoritis terhadap masalah yang diangkat guna menganalisa apakah rancangan algoritma yang digunakan dapat menghasilkan solusi yang sesuai dengan tujuan penelitian.

c. Perancangan

   Berdasarkan hasil analisa yang telah dilakukan selanjutnya dilakukan perancangan skema *Multipath Routing* pada jaringan berbasis *software defined network*. Proses perancangan menunjukkan hubungan antara masalah dan penanganan system yang ditunjukkan dalam bentuk diagram alur.

d. Implementasi

   Membuat program dari hasil rancangan algoritma yang telah dibuat untuk mengimplementasikan serta membuktikan bahwa hasil analisa secara teoritis yang telah dilakukan benar – benar sesuai yang diharapkan.

e. Pengujian

   Pengujian dilakukan untuk melihat apakah masalah trafik jaringan akan di proses sesuai dengan penanganan yang diharapkan. Hal ini juga dilakukan untuk mengevaluasi apakah metode yang diusulkan mampu menjawab tujuan yang telah diusulkan.

f. Dokumentasi

   Merupakan langkah akhir, penyusunan laporan mulai dari latar belakang permasalahan sampai dengan pengambilan kesimpulan.

1.6 Sistematika Penulisan

Sistematika penulisan bertujuan memudahkan dalam pemahaman permasalahan secara detail dari laporan tugas akhir. Sistematika penulisan laporan tugas akhir antara lain terdiri dari :

BAB I Pendahuluan

Dalam bab ini menjelaskan secara umum penyusunan penelitian ini yang meliputi latar belakang masalah, rumusan masalah, batasan masalah, tujuan dan manfaat, metodologi penulisan dan sistematika penulisan laporan.

BAB II Landasan Teori

Bab ini menjelaskan tentang teori-teori yang berkaitan dengan penelitian yaitu teori sistem SDN, Topologi, *Routing* dan algoritma yang akan digunakan dalam pengujiannya.

BAB III Pembahasan

Bab ini membahas tentang analisa kebutuhan dan perancangan sistem optimalisasi routing menggunakan dfs yang yang dimodifikasi. Serta perancangangan controller dan topologi yang akan digunakan.

BAB IV Implementasi dan Pengujian

Bab ini membahas tentang implementasi dan sistem yang di buat secara keseluruhan, serta tahapan pengujian terhadap sistem tersebut dapat menyelesaikan permasalahan berdasarkan parameter QOS (*Quality of Services*).

BAB V Penutup

Bab ini terdiri dari dua bagian yaitu kesimpulan dan saran dari uraian pada bab sebelumnya. Kesimpulan berisi rangkuman secara singkat dari hasil pembahasan masalah. Sedangkan saran berisi harapan dan kemungkinan lebih lanjut dari hasil pembahasan masalah.

BAB II
LANDASAN TEORI

2.1 Penelitian Terkait

Pada penelitian yang sebelumnya dilakukan **Chiang, et al., (2017)**. yang berjudul "*A Mutipath Transmission Scheme for the Improvement of Throughput over SDN*" yang mengimplementasikan multipath routing dengan load balancing pada jaringan Openflow menggunakan algoritma Dijkstra dengan melakukan iterasi hingga tidak ditemukan rute lagi. Setiap rute yang ditemukan *link* dan *node* dihapus dari pencarian kecuali *node* sumber dan *node* tujuan.

Penelitian yang dilakukan **W.Maulana. et al (2017)**. yang berjudul "*Multipath routing dengan Load-Balancing pada Openflow Software-Defined Network*" mengimplementasikan multipath routing pada jaringan Openflow dengan menggunakan algoritma DFS untuk menemukan jalurnya.

Penelitian yang dilakukan **Carlo Sembiring et al. (2018)** yang berjudul "*Mencari Jalur K Terpendek Menggunakan Yen Algoritm Unutk Multipath Routing pada Openflow Software-Defined Network*" yang mana penelitian ini membandingkan antara algoritma Yens dengan Dijkstra dan DFS. Dengan hasil algoritma Yens throughput lebih unggul pada topoplogi Abilens atau Network2.

Penelitian yang dilakukan oleh **Suryo Wicaksono et al (2021).** yang berjudul "*Implementasi Multipath Routing menggunakan Algoritma Iterative Deepening Depth First Search pada Openflow Software Defined Network*" menggunakan algoritma *Iterative Deepening Depth First Search* (IDDFS) untuk menemukan jalur pada topologi kompleks Fat – Tree dengan membandingkan algoritma DFS dan BFS (*Breadth First Search*) dan IDDFS unggul pada topologi fat-tree.

2.2 Software Defined Network

Software-Defined Networking (SDN) adalah sebuah paradigma arsitektur baru dalam bidang jaringan komputer, yang memiliki karakteristik dinamis, *manageable*, *cost-effective*, dan *adaptable*, sehingga sangat ideal untuk kebutuhan aplikasi saat ini yang bersifat dinamis dan *high-bandwidth*. Arsitektur ini memisahkan antara *network control* dan fungsi *forwarding*, sehingga *network control* tersebut menjadi *directly programmable* (dapat diprogram secara langsung), sedangkan infrastruktur yang mendasarinya dapat diabstraksikan untuk layer aplikasi dan *network services*.

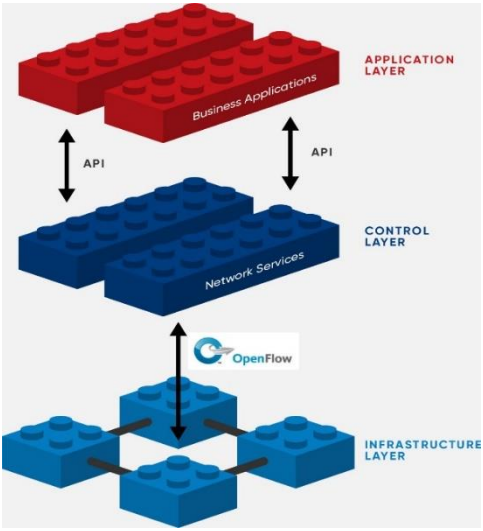
(Mulyana & Arif, 2023) menjelaskan, *Software Defined Network*, adalah istilah yang merujuk pada konsep/paradigma baru dalam mendesain, mengelola dan mengimplementasikan jaringan, terutama untuk mendukung kebutuhan dan inovasi di bidang ini yg semakin lama semakin kompleks. Konsep dasar SDN adalah dengan melakukan pemisahan eksplisit antara *control plane* dan *forwarding plane*, serta kemudian melakukan abstraksi sistem dan meng-isolasi kompleksitas yg ada pada komponen atau sub-sistem dengan mendefinisikan antar-muka (*interface*) yg standard.

Menurut **(Mulyana & Arif, 2023)** beberapa aspek penting dari SDN adalah :

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

2.3 Arsitektur Software Defined Network

Dalam SDN, terdapat 3 lapisan utama yang menyusun arsitektur dari jaringan SDN, yaitu *Application Layer, Control Layer, dan Infrastructure Layer.*



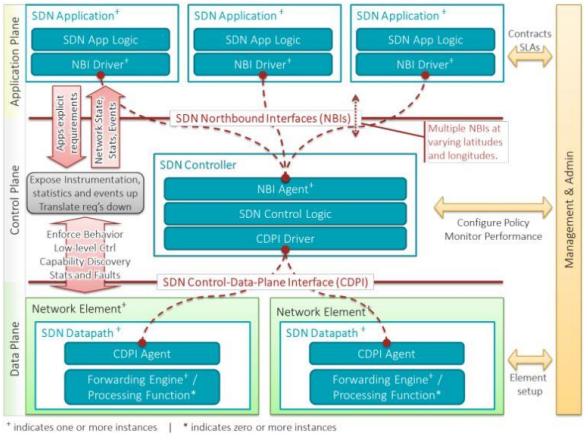
Gambar 2.1 Arsitektur *Software defined network*.
(Sumber Gambar : https://opennetworking.org/sdn-definition/)

1. *Application Layer*
   Lapisan yang berada paling atas, memiliki fungsi untuk menyediakan *interface* untuk pembuatan program aplikasi yang mengatur *network requirement* dan *network behavior* sesuai yang diinginkan. Sehingga dapat mengoptimalkan jaraingan secara baik dan fleksibel **(Saputra & Subardono, 2020).**
2. *Control Layer*
   Entitas control (SDN *Controller*) yang mentranslasikan kebutuhan aplikasi dengan infrastruktur dengan memberikan intrusksi yang sesuai untuk SDN *Datapath* serta memberikan informasi yang relevan dan dibutuhkan oleh SDN *Aplication* **(Saputra & Subardono, 2020).**

3. *Infrastructure Layer*
   Terdiri dari elemen jaringan yang dapat menerima instruksi dari *Control plane. Interface* antara *Control Plane* dan *Data Plane* disebut *South Bound Interface* (SBI), atau *Control-To-Data-Plane Interface* (CDPI) **(Saputra & Subardono, 2020)**.

2.4 Komunikasi Komponen Software Defined Network

Pada arsitektur ini jaringan dilihat sebagai data, kontrol, dan aplikasi, dari gambar 2.2 dibagi menjadi 3 area utama yaitu bagian atas, tengah, dan bawah. Dibagian bawah tersebut terdapat *data plane* yang terdiri dari elemen jaringan, yang mana SDN datapath dapat berkomunikasi melalui SDN *Control-Data-Plane Interface (CDPI) Agent.* Dibagian atas, aplikasi SDN / antar muka berada pada bagian *Application Plane*, dan untuk dapat saling berkomunikasi melalui *NorthBound Interface (NBI) Drivers*, dan pada bagian tengah SDN *Controller* menterjemahkan keperluan bagian bawah dan memberikan kontrol tingkat atas SDN *datapath*, *controller* juga bertugas memberikan informasi mengenai bagian bawah ke aplikasi SDN / Antarmuka **(Mulyana & Arif, 2023)**



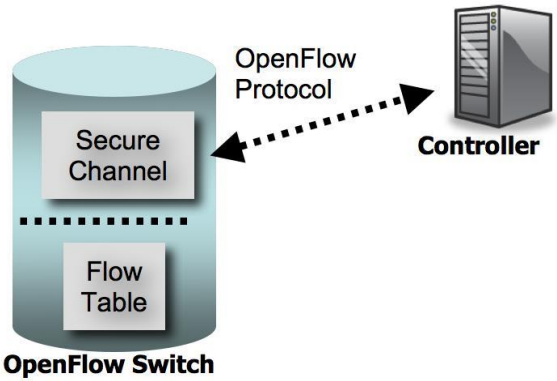
Gambar 2.2 Komponen Arsitektur dan Interaksi Komponen SDN **(Mulyana & Arif, 2023)**

Management dan Admin bertanggung jawab untuk menyiapkan elemen jaringan, menetapkan datapath terhadap *controller*. Bagian ini

juga bertanggung jawab untuk menyiapkan elemen jaringan, menetapkan datapath terhadap *controller*. Bagian ini juga bertanggung jawab untuk melakukan konfigurasi terhadap perangkat SDN **(Mulyana & Arif, 2023)**.

2.5 OpenFlow

Openflow (OF) adalah salah satu standar SDN pertama. Semula OF menetapkan protocol komunikasi pada lingkungan SDN yang memungkinkan *controller* SDN untuk dapat langsung berinteraksi dengan *forwarding plane* dari perangkat jaringan seperti *switch* dan *router,* baik secara fisik maupun *virtual (hypervisor – based),* menjadikan jaringan dapat beradaptasi pada kebutuhan bisnis **(Saputra & Subardono, 2020)**.



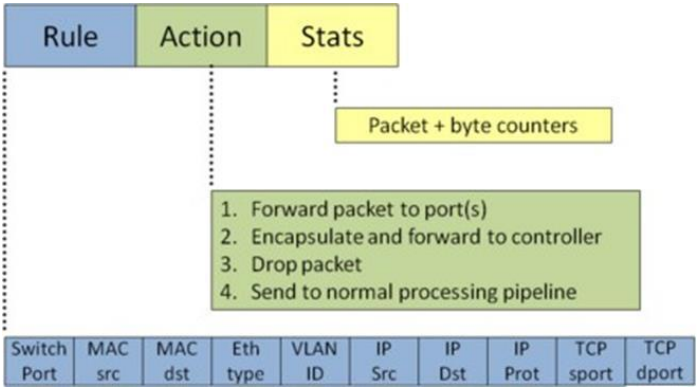
Gambar 2.3 Arsitektur Openflow

Dengan *OpenFlow* di tempatkan pada sebuah switch atau router maka dapat dilakukan *flow forwarding* berbasis *network layer* dan juga dapat melakukan pengaturan pergerakan paket secara terpusat mulai dari layer 2 sampai layer 7 *forwading (flow granuality),* sehingga aliran paket di jaringan dapat di program secara independent. Hal ini dapat dilakukan dengan dengan membuat algoritma forwading rules di *controller* kemudian aturan tersebut didistribusikan ke *switch* dan *router* yang telah menggunakan protokol ini. **(Saputra & Subardono, 2020).**

Secara dasar fungsi switch berubah dari awalnya semua pengaturan atau *controller* dilakukan pada switch itu sendiri namun pada switch *OpenFlow*, switch hanya digunakan untuk meneruskan

paket – paket data sesuai dengan flow yang telah diberikan oleh *controller*. Dengan *openflow* atau SDN adalah cara untuk memisahkan antara fisik jaringan dengan logika jaringan atau dengan kata lain jaringan bukan hanya sekedar sebuah perangkat yang dapat dikonfigurasi namun jaringan yang dapat diprogram dan juga adanya teknoloagi ini sangat mendukung teknologi virtualisasi yang sangat populer sekarang ini. Pada switch *OpenFlow* terdapat table yang berisi dari 3 bagian yaitu : *Rule, Action, dan Statistic.*

Untuk arsitektur pada paket data dapat dilihat pada gambar 2.4. Rule merupakan sekumpulan kondisi yang akan di bandingkan dengan paket yang akan masuk ke switch, yang akan dibaca adalah *header – header* dari setiap lapisan seperti *mac address*, *ip address*, *port number*, protocol dan lain sebagainya. *Action* merupakan tindakan yang akan dilakukan jika terdapat paket yang masuk ke swich dan sesuai dengan rule, dapat berupa perintah untuk meneruskan paket keluar ke port sekian atau men-drop paket dan lain sebagainya. Pada flow tabel juga terdapat statistic dari masing-masing *flow* berupa jumlah paket dan jumlah *bytes*.



Gambar 2.4 Arsitektur Paket Data

Salah satu bagian penting yang lainnya adalah *controller*. Fungsinya adalah untuk membuat *flow* dan juga mendistribusikannya ke switch yang terkoneksi. Jadi *controller* ini melakukan penambahan dan penghapusan *flow* dari *flow table* yang terdapat pada switch secara dinamis **(Saputra & Subardono, 2020)**.

2.6 Controller

Sistem operasi tradisional menyediakan abstraksi (*high level API)* untuk mengakses perangkat-perangkat dengan level lebih bawah seperti *hard drive,* CPU, dan memori. Sedangkan pada perangkat jaringan saat ini masih dikelola dan dikonfigurasi sesuai dengan instruksi masing-masing vendor (Cisco, Juniper) dan kebanyakan tidak menyediakan antarmuka untuk melakukan pengembangan. *Controller* pada SDN adalah sebuah aplikasi yang merupakan otak dari jaringan dan memberikan pandangan tersentralisasi untuk memungkinkan jaringan cerdas **(Yudha et al., 2018)**.

Controller SDN berfungsi sebagai system operasi jaringan untuk mengelola *flow-control* pada switch melalui *southbound* API seperti *OpenFlow*. *Controller* SDN menyediakan *northbound* API yang digunakan untuk mengembangkan aplikasi jaringan. Aplikasi control jaringan yang pertama dibuat adalah NOX. *Controller* ini di program menggunakan Bahasa C. Selain itu, juga terdapat *controller* yang lain seperti POX (python), Ryu (Python), FloodLight (Java), OpenDayLight (Java), ONOS (Java) **(Yudha et al., 2018)**.

2.7 Ryu Controller

Ryu dalam bahasa jepang “flow/aliran”. Ryu adalah komponen berdasarkan SDN Framework. Ryu menyediakan komponen perangkat lunak dengan API (*Application Programming Interface*) didefinisikan dengan baik yang membuatnya mudah untuk pengembang dalam membuat membuat manajemen jaringan baru dan control aplikasi. Pengembangan aplikasi untuk ryu dapat dilakukan dengan menggunakan Bahasa Python atau dengan mengirimkan pesan JSON melalui API yang tersedia. Ryu mendukung berbagai protocol untuk mengelola perangkat jaringan seperti Openflow, Netconf, OF-config dan lain – lain. Ryu mendukung penuh Openflow versi 1.0, 1.2, 1.3, 1.4, 1.5 dan Nicara Extension. Semua kode tersedia secara bebas di bawah lisensi Apache 2.0 **(Wibowo et al., 2018)**.

2.8 Routing

2.8.1 Definisi Umum

Routing adalah suatu Teknik pemilihan jalur dalam sebuah jaringan. *Routing protocol* merupakan sekumpulan aturan dalam menentukan jalur di sebuah jaringan. Ada banyak jenis *routing*

protocol. Routing terbagi menjadi dua jenis yaitu *static routing* dan *dynamic routing.* **(Firdausi & Wardani, 2020)**

Algoritma yang menghitung jalur *routing* ini disebut algoritma *Routing.* Sebuah algoritma *Routing* akan menentukan. Algoritma *Routing* yang digunakan adalah algoritma *Dijkstra* dan algoritma DFS. Algortime *Link-State* atau yang biasa disebut dengan algoritma *Dijkstra.* Dalam algoritma *Dijkstra* / *link-state* . Topologi jaringan dan bobot semua *link* diketahui oleh setiap *node*. Sedangkan algoritma DFS yaitu metode pencarian jalur yang dilakukan dengan menggunakan struktur data *stack* yang dapat menyimpan rute kemudian melakukan expansi lagi meskipun lagi sudah ditemukan satu jalur ke tujuan. **(Firdausi & Wardani, 2020)**

2.8.2 Single Path Routing

Algoritma *routing* 1 jalur (*single-path*) atau jalur konvensional, seperti algoritma Dijkstra pada *Open Shortest Path First (OSPF),* merupakan algoritma yang hanya memanfaatkan satu jalur yang sudah terbaca. Sehingga belum bisa memanfaatkan semua jalur pada topology jaringan. Pada algoritma *tree* seperti algoritma *Dijkstra,* topology jaringan akan terus dipotong dengan ini mengakibatkan seperti *tree.* Ini yang menyebabkan menghapus *multipath* pada topologi yang memiliki banyak jalur (cabang) sehingga menyebabkan kemacetan pada jaringan **(Sembiring et al., 2018).**

2.8.3 Multipath Routing

Multipath Routing terdiri atas dua kata yaitu *multipath* dan *routing. Multipath* bisa diartikan sebagai jalur – jalur. Dan *routing* adalah proses pemilihan jalur bagi sebuah trafik yang bergerak dari sumber (*source*) menuju tujuan (*destination*) di dalam sebuah jaringan atau diantara kumpulan jaringan. Sehingga *multipath routing* bisa diartikan sebagai protokol untuk menentukan jalur – jalur alternatif yang berbeda yang terdapat di dalam jaringan dalam pengiriman paket data dari *source* ke *destination* **(Suryo Wicaksono & Hari Trisnawan, 2021).**

2.9 Depth-First Search

Depth-First Search (DFS) adalah algoritma yang digunakan untuk pencarian jalur. Pencarian dilakukan pada satu *node* dalam setiap level dari yang paling kiri, jika pada level yang paling dalam, solusi belum ditemukan, maka pencarian dilanjutkan pada *node*

sebelah kanan. *Node* yang kiri dapat dihapus dari memori. Jika pada level yang paling dalam ditemukan solusi, maka pencarian dilanjutkan pada level sebelumnya **(Wibowo et al., 2018).**

Tabel 2.1 Kode Sumber DFS

```
DFS ( G, v ) ( v is the vertex where the
search starts )
    Stack S := {}; (start with an empty stack
    )
    for each vertex u, set visited[u] :=
    false;
    push S, v;
    while (S is not empty) do
      u := pop S;
          if (not visited(u]) then
            visited[u] := true;
      for each unvisited neighbour w of u
          push S, w;
      end if
    end while
END DFS()
```

2.10 Quality of Service

Quality of Service (QoS) didefinsikasn sebagai suatu pengukuran tentang seberapa baik jaringan dan merupakan suatu usaha untuk mendefinisikan karakteristik dan sifat dari suatu layanan. *QOS* mengacu pada kemampuan jaringan untuk menyediakan layanan yang lebih baik pada jaringan trafik jaringan tertentu melalui teknologi keseluruhan. Tujuan dari *QOS* adalah untuk memenuhi kebutuhan layanan yang berbeda, yang menggunakan infrastruktur yang sama **(Turmudi & Abdul Majid, 2019).**

Parameter dari *Quality of Service* (QoS) adalah *delay*, *throughput, jitter,* dan *packet loss.* Berikut adalah penjelasan tentang parameter-parameter tersebut :

2.10.1 Latency / Delay

Delay (Latency) didefinisikan sebagai total waktu tunda suatu paket yang diakibatkan oleh proses transmisi dari satu titik lain yang menjadi tujuannya. *Delay* di dalam jaringan dapat digolongkan sebagai berikut *delay processing, delay packetization, delay*

serialization, delay jitter buffer dan delay network **(Turmudi & Abdul Majid, 2019)**

Berikut adalah standarisasi nilai delay versi Tiphon (1999):

$$Delay = \frac{Packet\ Length}{Link\ Bandwidth}$$

Tabel 2.2 Standarisasi Nilai Delay Versi TIPHON (1999)

Kategori Latensi	Besar Delay (ms)	Indeks
Sangat Bagus	< 150 ms	4
Bagus	150 ms s/d 300 ms	3
Sedang	300 ms s/d 450 ms	2
Jelek	> 450 ms	1

2.10.2 Jitter

Jitter atau variasi kedatangan paket, hal ini diakibatkan oleh variasi – variasi dalam Panjang antrian, dalam waktu pengolahan data, dan juga dalam waktu penghimpunan ulang paket – paket di akhir perjalanan *jitter. Jitter* lazimnya disebut variasi *delay* berhubungan erat dengan *latency,* yang menunjukkan banyaknya variasi *delay* pada tranmisi data di jaringan. *Delay* antrian pada *router* dan *switch* dapat menyebabkan *jitter.* **(Turmudi & Abdul Majid, 2019).**

Berikut adalah standarisasi nilai *jitter* versi Tiphon (1999):

$$Jitter = \frac{Total\ Variasi\ Delay}{Total\ Packet\ yang\ Diterima}$$

Tabel 2.3 Standarisasi Nilai *Jitter* Versi TIPHON (1999)

Kategori Jitter	Jitter (ms)	Indeks
Sangat Bagus	0 ms	4
Bagus	0 ms s/d 75 ms	3
Sedang	75 ms s/d 125 ms	2
Jelek	125 ms s/d 225 ms	1

2.10.3 Packet Loss

Packet Loss adalah merupakan suatu parameter yang menggambarkan suatu kondisi yang menunjukkan jumlah total paket yang hilang. Salah satu penyebab *packet loss* adalah antrian yang melebihi kapasitas *buffer* pada setiap *node*. Beberapa penyebab terjadinya *packet loss* yaitu :

1. *Congestion,* disebabkan terjadinya antrian yang berlebihan dalam jaringan.
2. *Node* yang bekerja melebihi kapasitas *buffer*.
3. *Memory* yang terbatas pada *node*.
4. *Policing atau control* terhadap jaringan untuk memastikan bahwa jumlah trafik yang mengalir sesuai dengan besarnya bandwidth. Jika besarnya trafik yang mengalir di dalam jaringan melebihi dari kapasitas *bandwidth* yang ada maka *policing control* akan membuang kelebihan trafik yang ada **(Turmudi & Abdul Majid, 2019).**

Standarisasi perhitungan Packet Loss :

$$Packet\ Loss = \frac{(Packet\ data\ dikirim - Paket\ data\ diterima)\ x\ 100\%}{Paket\ data\ yang\ dikirim}$$

Tabel 2.4 Standarisasi Nilai *Packet Loss* Versi TIPHON (1999)

Kategori Degredasi	Packet Loss (%)	Indeks
Sangat Bagus	0	4
Bagus	3	3
Sedang	15	2
Jelek	24	1

2.10.4 Throughput

Throughput yaitu kecepatan (*rate*) transfer data efektif, yang diukur dalam *bit per second (bps). Throughput* merupakan jumlah total kedatangan paket yang sukses yang diamati pada *destination* selama *interval* waktu tertentu dibagi oleh durasi *interval* waktu tersebut (sama dengan jumlah pengiriman paket IP sukses per *service second*). Dalam standar TIPHON *throughput* dihitung dalam persen, untuk mendapatkan nilai *throughput* dalam persen hasil perhitungan *throughput* kemudian dibagi dengan besarnya nilai *bandwidth* dan dikalikan 100 % untuk mengetahui besarnya persentase nilai *throughput* yang sebenarnya **(Turmudi & Abdul Majid, 2019).**

Standarisasi perhitungan Throughput :

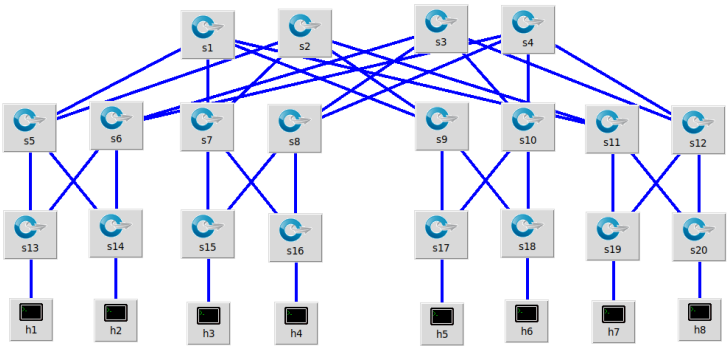
$$Throughput = \frac{Packet\ Data\ Diterima}{Lama\ Pengamatan} \text{ X } 100\%$$

Tabel 2.5 Standarisasi Nilai *Throughput* Versi TIPHON (1999)

Kategori	Throughput	Indeks
Sangat Bagus	100 %	4
Bagus	75 %	3
Sedang	50 %	2
Jelek	< 25 %	1

2.11 Topologi Fat – Tree

Topologi *Fat – Tree* diperkenalkan pertama kali oleh Al-Fares *et al* untuk membangun jaringan pusat data. Kemudian banyak penelitian yang menggunakan topologi ini dalam penelitian jaringan pusat data. Topologi *Fat Tree* memiliki beberapa jalur dari satu *server* ke *server* sehingga *Multipath Routing* dapat diterapkan pada topologi tersebut. **(Rangkuty et al., 2020).**



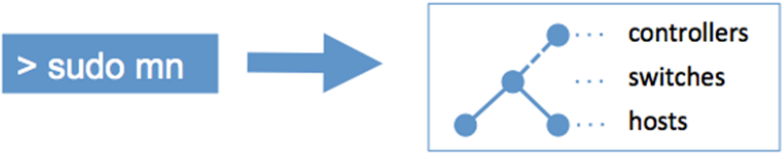
Gambar 2.5 Topologi Fat - Tree

2.12 Emulator Mininet

Mininet adalah emulator jaringan yang realistis, menjalankan kernel asli, *switch* dan aplikasi *code* pada satu mesin (VM, cloud, atau *native*), dalam hitungan detik dengan satu **(Wibowo et al., 2018).**

Pada mininet bisa dilakukan perancangan dengan topologi jaringan yang diinginkan, secara sederhana mininet berfungsi sebagai emulator pada bagian *datapath* untuk melakukan percobaan pada jaringan SDN.

Sedangkan untuk melakukan percobaan mininet dapat dilakukan dengan perintah “sudo mn”, dengan command ini mininet akan secara default mengemulasikan konfigurasi jaringan yang terdiri dari dari 1 buah *controller*, 1 *switch* dan 2 *host* **(Contributors, 2023).**

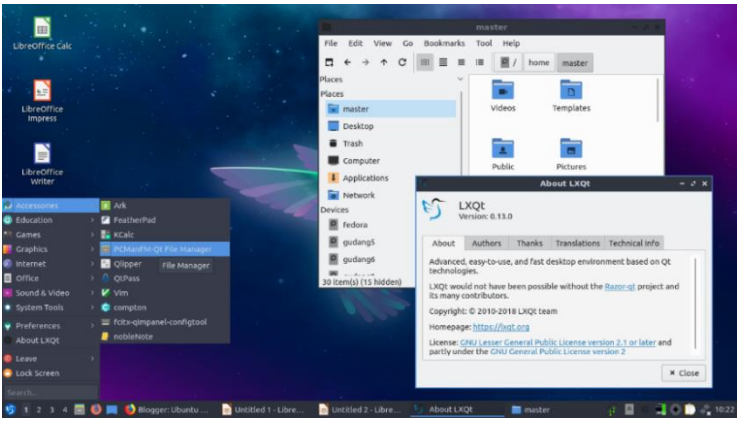


Gambar 2.6 Mininet

Aplikasi jaringan baru bisa dikembangkan dan diuji terlebih dahulu pada emulasi jaringan mininet. Hal ini kemudian dapat dipindahkan ke infrastruktur jaringan yang sebenarnya. Secara default, mininet mendukung OpenFlow v1.0. Namun, bisa dimodifikasi untuk mendukung sebuah switch virtual yang telah diimplementasikan sebelum digunakan **(Contributors, 2023).**

2.13 Linux Ubuntu

Sejarah ubuntu dimulai pada tahun 2004, yang diperkenalkan oleh perusahaan inggris bernama Canonical. OS ini didasarkan pada Debian, yang merupakan salah satu distro populer. Ubuntu banyak digunakan di berbagai kebutuhan seperti desain, keamanan, dan kebutuhan sehari – hari atau *daily driver*. Ubuntu berbasis pada *system* operasi linux. Ubuntu saat ini dapat digunakan pada basis *cloud server*. Untuk versi ubuntu terbaru saat ini adalah versi Ubuntu 23.04 dengan kode nama Lunar Lobster **(Ubuntu. (n.d.)., 2023)**



Gambar 2.7 Sistem Operasi Ubuntu

2.14 Iperf

Iperf merupakan open source tools yang digunakan untuk mengukur kecepatan, kecepatan, *throughput* dan kualitas *link* jaringan. Tools ini menggunakan protocol TCP dan UDP. TCP digunakan untuk mengukur kecepatan dan throughput link, UDP digunakan untuk mengukur jitter (variasi dari paket ke paket) dan Packet Loss. Keuntungan tambahan menggunakan iperf untuk pengujian kinerja jaringan antara dua server, keduanya berbeda secara geografis lokasi, dan anda ingin mengukur kinerja jaringan antara keduanya.. **(Gueant, 2023).**

```
sdn@linux:~$ iperf -h
Usage: iperf [-s|-c host] [options]
       iperf [-h|--help] [-v|--version]

Client/Server:
  -b, --bandwidth #[kmgKMG | pps]  bandwidth to send at in bits/sec or packets per second
  -e, --enhancedreports    use enhanced reporting giving more tcp/udp and traffic information
  -f, --format    [kmgKMG]   format to report: Kbits, Mbits, KBytes, MBytes
  -i, --interval  #        seconds between periodic bandwidth reports
  -l, --len       #[kmKM]    length of buffer in bytes to read or write (Defaults: TCP=128K, v4 UDP=1470, v6
DP=1450)
  -m, --print_mss          print TCP maximum segment size (MTU - TCP/IP header)
  -o, --output    <filename> output the report or error message to this specified file
  -p, --port      #        server port to listen on/connect to
  -u, --udp                use UDP rather than TCP
      --udp-counters-64bit use 64 bit sequence numbers with UDP
  -w, --window    #[KM]    TCP window size (socket buffer size)
  -z, --realtime           request realtime scheduler
  -B, --bind <host>[:<port>][%<dev>] bind to <host>, ip addr (including multicast address) and optional port
nd device
  -C, --compatibility      for use with older versions does not sent extra msgs
  -M, --mss       #        set TCP maximum segment size (MTU - 40 bytes)
  -N, --nodelay            set TCP no delay, disabling Nagle's Algorithm
  -S, --tos       #        set the socket's IP_TOS (byte) field
```

Gambar 2.8 Iperf

BAB III
PEMBAHASAN

3.1 Analisis Masalah

Berdasarkan latar belakang pada penelitian yang telah dibahas pada bab pertama, maka penulis akan dilakukan penelitian mengenai Optimalisasi routing menggunakan algoritma DFS yang dimodifikasi. Optimalisasi jaringan merupakan permasalahan yang signifikan pada jaringan tradisional, cenderung menjadi lebih besar pada jaringan SDN karena fungsi sentralisasi pada *controller* SDN yang menjadi target dari lalu lintas data

"Pada saat ini router/switch yang berada pada jaringan backbone / utama memiliki akses jaringan. Router/switch tersebut masih menggunakan konfigurasi tradisional dan belum menggunakan SDN, yang mana pada jaringan tradisional memiliki kekurangan dalam hal konfigurasi yang harus dilakukan pada tiap – tiap perangkat router/switch. Dari router/switch tersebut tidak menutup kemungkinan untuk bertambahnya router/switch yang baru, dengan bertambahnya router baru yang menjadikan jaringan semakin komplek, hal ini akan menjadi kendala bagi administrator jaringan karena harus mengkonfigurasi tiap – tiap router/switch yang baru **(Sembiring et al., 2018)**.

Pada penelitian sebelumnya tentang (*Multipath Routing dengan load balancing pada openflow software-defined network)* yang sudah dilakukan oleh Syahidillah, W., M. Dari Universitas Brawijaya pada tahun 2017. Penelitian yang dilakukan adalah implementasi dengan menggunakan algoritma DFS *(Depth First Search).* Pada penelitiannya parameter yang digunakan adalah cost sebagai penentu rutenya pada openflow. Beberapa parameter yang diuji adalah yaitu *throughput, bandwidth* **(Sembiring et al., 2018).**

Terdapat juga masalah dimana pada perubahan pemodelan topologi yang sewaktu waktu dapat berubah. Dengan diterapkannya SDN pada infrastruktur jaringan dapat mengurangi permasalahan jaringan dan memberikan kemudahan dalam melakukan konfigurasi, karena pada jaringan SDN konfigurasi dilakukan pada bagian kontroler tanpa harus melakukan pada setiap switch. Pada jaringan SDN terdapat sebuah protokol yang menghubungkan kontroler dengan perangkat jaringan yaitu *OpenFlow.* Untuk itu pihak Administrator perlu memaksimalkan trafik data pada perangkat-perangkat tersebut **(Sembiring et al., 2018)**.

Pada permasalahan yang dijelaskan diatas, akan dilakukan optimalisasi *routing* menggunakan algoritma *Depth First Search* yang dimodifikasi pada SDN *(Software Defined Network)* untuk mengatasi kelemahan dari *single-path* routing. Pada penelitian ini dilakukan untuk mencari jalur terpendek dan menggunakan cost sebagai penentuan pemilihan jalurnya pada pengiriman datanya. Serta mengoptimalkan throughput, jitter, dan packet loss dengan memodifikasi algoritma DFS yang sudah ada. Ryu digunakan sebagai controllernya. Bahasa pemrograman python digunakan sebagai basenya. Dengan itu dapat mengembangkan *routing* yang bisa melakukan *multipath* dengan jalur yang ada pada topologi.

3.2 Gambaran Umum Sistem

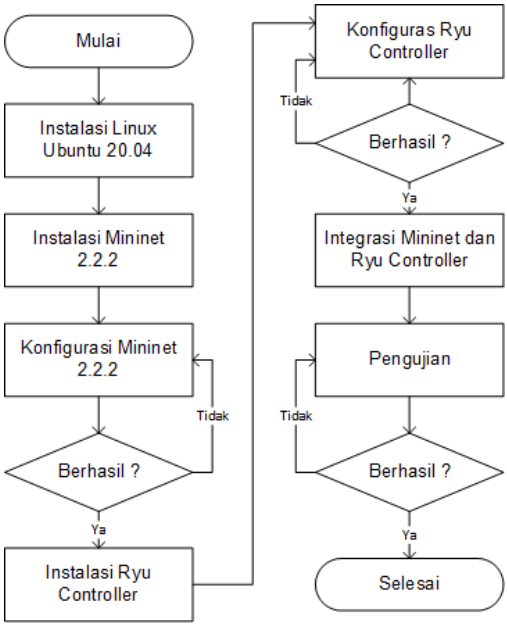
Implementasi *Software defined network* dilakukan pada 20 buah switch yang ada pada jaringan simulator Mininet, switch tersebut menggunakan OpenvSwitch dengan protokol OpenFlow menjadikan switch OpenFlow software-base. Agar infrastruktur SDN dapat berjalan perlu ditambahkan satu *node* yang berfungsi sebagai kontroller. Kontroller ini menggunakan Ryu *Controller*. Dengan diterapkannya SDN pada jaringan simulator tersebut dapat mengubah aliran data dan pengambilan keputusan yang awalnya dilakukan oleh router/switch menjadi diambil alih oleh *control plane*.

Selanjutnya dilakukan analisis perfomansi terhadap infrastruktur *Software defined network* (SDN) dengan mengukur perbandingan *Quality of Service* (QoS) dengan jaringan menggunakan SDN dan pengujian pada topologi *Fat - Tree*.

Pada perancangan sistem diperlukan adanya emulator jaringan Mininet dan Ryu *Controller* sebagai *control plane* SDN. Semua alur proses pembentukan jaringan SDN akan ditampilkan dengan *flowchart* pada gambar 3.1.

Pada gambar 3.1 dijelaskan bahwa sistem yang telah didesain diperlukan beberapa langkah mulai dari beberapa kebutuhan package yang harus terinstal mulai dari perancangan desain jaringan, desain *controller*, serta sistem operasi yang digunakan yaitu Linux 20.04. Installasi Mininet dan melakukan konfigurasi, hal ini juga berlaku terhadap Ryu *Controller* yang digunakan sebagai kontrol terhadap sistem. Setelah semua terinstall maka dilakukan konfigurasi agar sistem saling terintegrasi satu sama lain sehingga dapat di lakukan pengujian terhadap sistem. Proses pengujian dilihat dari proses yang

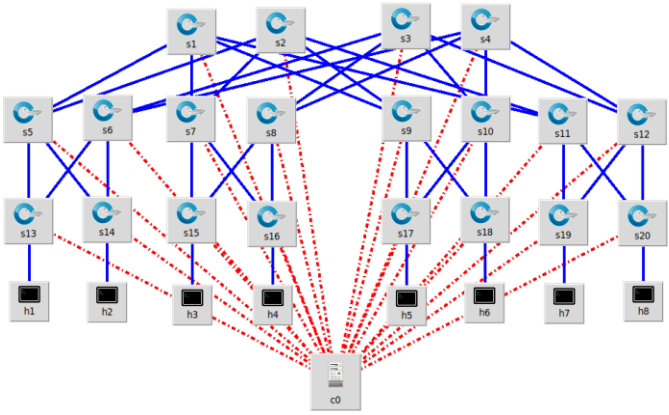
terjadi seperti efektifnya konfigurasi optimalisasi routing yang telah dibuat dan lalu lintas komunikasi jaringan yang terjadi.



Gambar 3.1 Alur Desain Sistem

3.2.1 Desain Topologi

Dalam penelitian ini akan menggunakan jenis topologi Fat - Tree didalam Mininet. Desain topologi akan dibuat menggunakan fitur miniedit yang sudah disediakan oleh aplikasi Mininet. Tampilan desain topologi menggunakan miniedit pada gambar 3.2. Pada gambar 3.2 terlihat desain yang dibangun adalah topologi menggunakan 20 buah switch. Semua switch akan tersambung juga dengan Ryu *Controller* sebagai *control plane* dalam topologi tersebut.



Gambar 3.2 Perancangan Topologi Fat – Tree

Setelah dilakukan persiapan topologi maka akan dilakukan juga persiapan konfigurasi antara *node* dalam topologi tersebut. Persiapan dimulai dari pencatatan alamat *host* kemudian identitas *switch* yang terdapat pada tabel 3.1.

Tabel 3.1 Tabel pengalamatan host

ID Host	IP	Netmask
h1	10.0.0.1	255.0.0.0
h2	10.0.0.2	255.0.0.0
h3	10.0.0.3	255.0.0.0
h4	10.0.0.4	255.0.0.0
h5	10.0.0.5	255.0.0.0
h6	10.0.0.6	255.0.0.0
h7	10.0.0.7	255.0.0.0
h8	10.0.0.8	255.0.0.0

Setelah dilakukan perancangan maka akan dilanjutkan konfigurasi topologi tersebut pada bab selanjutnya.

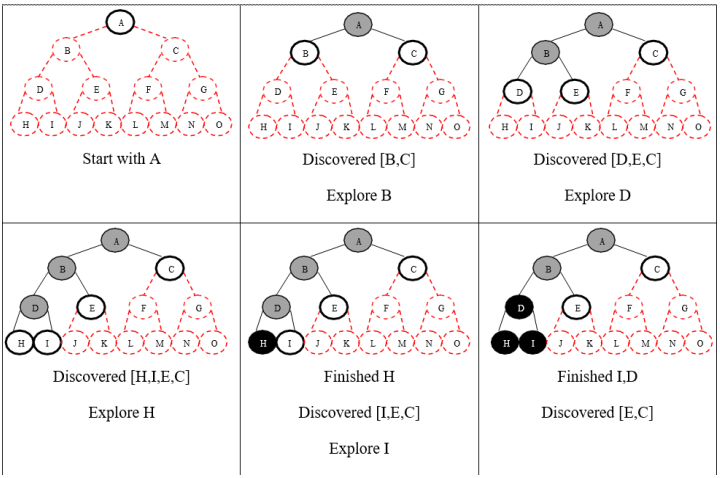
3.2.2 Perancangan Algoritma DFS Konvensional

Berikut ini merupakan algoritma perancangan jalur menggunakan metode Depth First Search (DFS) :

1. **Pilih Node Awal**: Mulailah dengan memilih node awal (sumber) dari mana pencarian rute akan dimulai.

2. **Tandai Node**: Tandai node awal sebagai "dikunjungi" atau "dalam proses". Ini membantu menghindari loop yang tak terbatas dengan menghindari mengunjungi kembali node yang sama.
3. **Periksa Tetangga**: Periksa semua tetangga yang terhubung dengan node saat ini. Ini bisa dilakukan dengan mengikuti tautan atau koneksi yang ada dari node saat ini ke node lain.
4. **Rekursi**: Untuk setiap tetangga yang belum dikunjungi, rekursif pindah ke tetangga tersebut dan ulangi langkah-langkah 2 hingga 4 untuk tetangga tersebut.
5. **Mundur**: Jika tidak ada tetangga lagi yang dapat dikunjungi, mundurlah ke node sebelumnya (node yang lebih tinggi dalam hierarki).
6. **Periksa Tujuan**: Saat menjelajahi node-node dan tautan-tautan, periksa apakah node tujuan yang diinginkan telah ditemukan. Jika ya, rute atau jalur telah ditemukan.
7. **Backtracking**: Jika tautan yang ditemui tidak mengarah ke node tujuan atau tidak ada rute yang ditemukan, maka algoritma akan "backtrack" atau mundur ke node sebelumnya yang memiliki tetangga lain yang belum dikunjungi.
8. **Tandai Selesai**: Setelah semua tetangga dari suatu node telah dikunjungi atau jika tujuan telah ditemukan, tandai node sebagai "selesai" atau "dikunjungi sepenuhnya".
9. **Selesai**: Algoritma DFS selesai ketika telah mencoba semua kemungkinan jalur atau ketika telah menemukan jalur yang mengarah ke node tujuan.

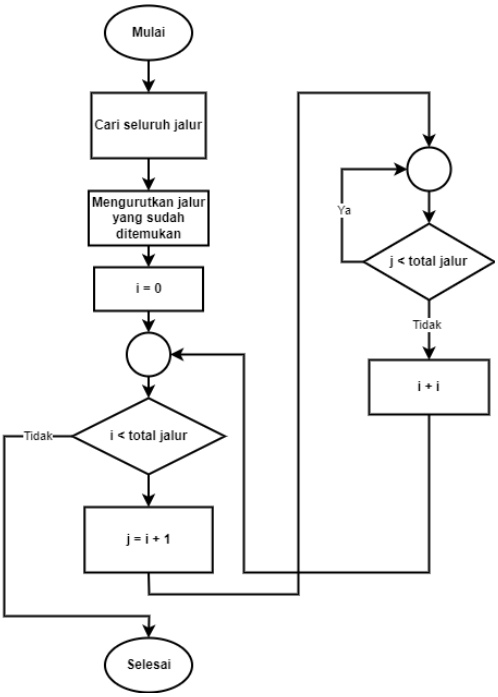
Metode DFS akan ditampilkan sebagai berikut:



Gambar 3.3 Algoritma Depth First Search

Didapatkan gambar 3.3 algoritma Depth First Search maka jalur dilewati untuk mencari solusi yang diinginkan, pada kasus ini solusi yang dipergunakan adalah jumlah jalur dan tujuannya. Berikut adalah flowchart rangkaian proses pada algoritma DFS konvensional pada gambar 3.4.

Dengan metode DFS konvensional maka akan dilakukan pencatatan semua jalur dengan dilakukannya pencarian solusi pada tujuan dengan berulang kali sampai tidak ditemukannya kembali jalur pencariansolusi. Setiap jalur pencarian solusi yang ditemukan akan selaludisimpan dan dihitung jarak pada setiap solusi tersebut yang kemudian dilanjutkan dengan pencarian selanjutnya serta hitungannya. Pada konsep ini akan dilakukan penumpukan jalur pencarian solusi menggunakan DFS konvensional sehingga pada tumpukan jalur tidak akan ada pencarian solusi kembali dengan jalur yang sama.



Gambar 3.4 Flowchart DFS Konvensional

Setelah dilakukan penumpukan dan hitungan jalur yang menggunakan DFS konvensional tersebut, maka akan dilakukan penyortiran atau pemilihan jalur terpendek dengan hitungan terkecil. Jumlah jalur yang dipilih akan disesuaikan dengan keinginan atau kebutuhan pengelola jaringan, dikarenakan dengan jumlah yang tepat pada penggunaan jalur dapat diperoleh hasil yang lebih maksimal pada implementasi *Multipath*.

Tabel 3.2 Kode Sumber Algoritma DFS

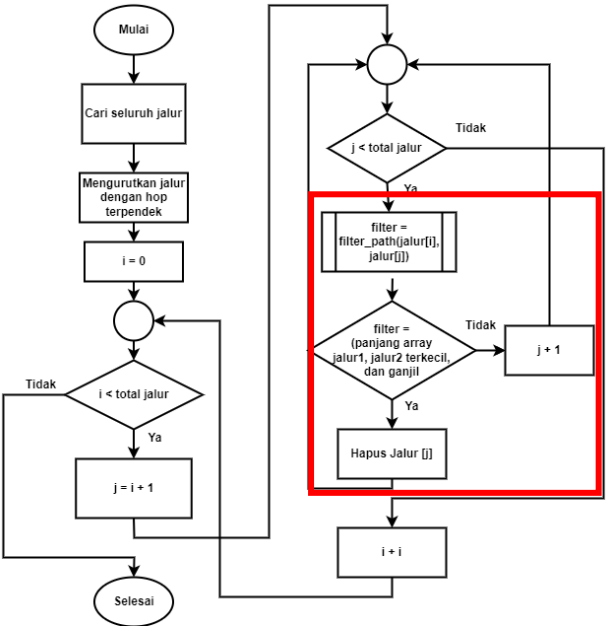
```
DFS (G, s, d)
    p array to store paths
    stack initialized as (s, [s])
    while stack is not empty do
        pop stack, save to node, p
        for next in (G[node] - p)
           if next is d
             append p + [next]
to P
```

Tabel 3.2 Lanjutan

```
            else
              push  (next, p + [next]) to stack
            end for
        end while
    sort elemen in p by length of path
        i = 0
        pd array to store deleted paths
        while i < length of p
            j = i + 1
            while j < length of p
                if filter_path (p[i]), p[j])
                  append p[j] to pd
                  remove p[j] from p
                else
                  increment j
              increment i
        return p
end
```

3.2.3 Perancangan Algoritma DFS Modifikasi

Berdasar penelitian (Maulana, 2017) yang berjudul "*Multipath* Routing dengan *Load-Balancing* pada *Openflow Software-Defined Network*" telah menerapkan *multipath routing* menggunakan algoritma DFS konvensional sehingga dapat menemukan keseluruhan jalur yang ada. Akan tetapi algoritma DFS konvensional mengekspansi node terjauh, sehingga hasil pencarian jalur yang ditemukan pertama kali adalah jalur terjauh dari sumber ke tujuan. Oleh sebab itu perlu pengembangan algoritma untuk menemukan jalur terpendek dan menghapus jalur lain yang memiliki *shared edge* (*link* antara dua *node* yang ada pada jalur lain.



Gambar 3.5 Flowchart Modifikasi DFS

Sehingga menghasilkan jalur – jalur independen terdekat. Modifikasi algoritma DFS yang dilakukan adalah dengan mencari seluruh jalur dari suatu sumber ke tujuan kemudian mengurutkan jalur dengan *hop* atau jarak terpendek. Setelah itu dilakukan seleksi terhadap jalur sehingga didapatkan jalur – jalur yang independent atau jalur yang tidak memiliki kesamaan yang menyebabkan kemacetan trafik. Gambar 3.5 menunjukkan diagram alir algoritma DFS yang dimodifikasi dan tabel 3.3 merupakan *psedocoude* dari algoritma DFS yang dimodifikasi.

Pada modifikasi DFS nilai array yang memiliki nilai genap diganti dengan nilai ganjil yaitu 1 dan 3 karena ketika nilai genap dimasukkan maka akan menyebabkan adanya kesamaan jalur pada hasil pencarian jalurnya seperti pada gambar 3.6 merupakan hasil menggunakan nilai genap, dan pada gambar 3.7 menggunakan nilai ganjil

```
=== Running Multipath DFS Modified algorithm ===
Iterasi :  11335
Path Execution Time :  0.017030000686645508
Chosen paths from: 20  to  13  =  [20, 11, 2, 5, 13] cost =  4.0
Chosen paths from: 20  to  13  =  [20, 11, 1, 5, 13] cost =  4.0
Path installation finished in  0.04499053955078125
============================
```

Gambar 3.6 Menggunakan Nilai Genap

```
=== Running Multipath DFS Modified algorithm ===
Iterasi :  11335
Path Execution Time :  0.02875828742980957
Chosen paths from: 20  to  13  =  [20, 12, 3, 6, 13] cost =  4.0
Chosen paths from: 20  to  13  =  [20, 11, 2, 5, 13] cost =  4.0
Path installation finished in  0.049593448638916016
```

Gambar 3.7 Menggunakan Nilai Ganjil

Tabel 3.3 Kode Sumber Modifikasi DFS

```
DFS (G, s, d)
    p array to store paths
    stack initialized as (s, [s])
    while stack is not empty do
        pop stack, save to node, p
        for next in (G[node] - p)
          if next is d
           append p + [next]
to P
          else
           push  (next, p + [next]) to stack
          end for
        end while
    sort elemen in p by length of path
        i = 0
        pd array to store deleted paths
        while i < length of p
            j = i + 1
            while j < length of p
                if filter_path (p[i]), p[j])
                  append p[j] to pd
                  remove p[j] from p
                else
                  increment j
              increment i
```

Tabel 3.3 Lanjutan

```
        return p
end
filter_path(path1, path2)
    for i in range (length of path1- 1)
       for j in range (length of path2 – 1)
          if path1[i] == path2[j] and path1[i+1]
          == path2 [j+1]
            return True
    return False
end
```

3.3 Perancangan Metrik Penilaian Jalur

Dalam konteks ini, jalur yang terbentuk tidak memiliki nilai bobot. Oleh karena itu, diperlukan suatu nilai bobot untuk menggambarkan biaya setiap perjalanan. Dalam penelitian ini, bobot link dari jalur dihitung menggunakan rumus yang dinyatakan dalam persamaan (3.1) berikut ini :

$$ew = \frac{BR}{BL} \tag{3.1}$$

Berdasarkan rumus 3.1. Dimana *ew* adalah *edge weight (link cost),* B_R adalah *reference bandwitdth* yang besarnya 100 Mbps sesuai pada protocol OSPF dari Cisco. dan B_L adalah *link bandwidth* dari sepasang router yang berhubungan **(Suryo Wicaksono & Hari Trisnawan, 2021)**

3.4 Perancangan Pemilihan Jalur

Tidak semua jalur yang ditemukan akan digunakan dalam skema routing *multipath. Bobot dari suatu jalur p(path weight)* dinyatakan oleh *pw(p)* dimana *p(V, E)* adalah jalur yang terdiri atas sekumpulan *edge* yang dinyatakan oleh *E*. Bobot tersebut didapatkan dengan menjumlahkan seluruh *node weight* dan *edge weight* pada suatu jalur. Dari itu dapat dinyatakan bahwa jalur terbaik merupakan jalur dengan *path weight* terendah. Nilai pw(p) ditentukan menggunakan persamaan (2) **(Suryo Wicaksono & Hari Trisnawan, 2021)** berikut ini :

$$pw(p) = \sum e \in E \; ew[e] \tag{3.2}$$

Banyaknya jalur yang hendak digunakan pun dibatasi tergantung dari nilai batasan yang diberikan dalam skema *multipath,* bisa 2 jalur, 3 jalur dan seterusnya. *Bobot kualitatif jalur (w(p))* dihitung dengan menentukan persentase *pw(p)* dari masing – masing jalur melalui persamaan **(Suryo Wicaksono & Hari Trisnawan, 2021)** (3) dibawah ini :

$$w(p) = \left(1 - \frac{pw(p)}{\sum_{i<n}^{i=0} pw(i)}\right) \; x\, 10 \tag{3.3}$$

3.5 Skenario Pengujian Sistem

Dalam penelitian ini, untuk mengukur kinerja dari setiap algoritma dalam *single-path* dan *multipath* routing, digunakan jenis topologi jaringan. Dua jenis topologi jaringan tersebut adalah topologi kompleks (topologi fat-tree). Pada setiap topologi, beberapa parameter pengujian akan diuji, termasuk pencarian jalur, *delay, throughput, jitter,* dan *packet loss.*

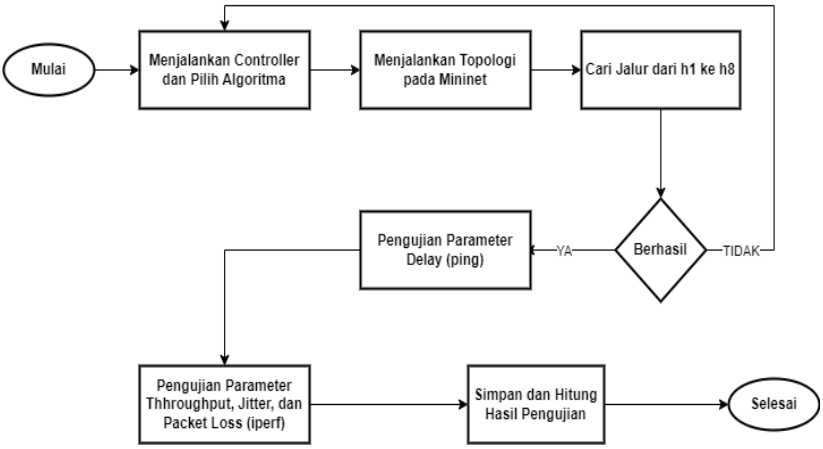
Pada topologi *fat – tree,* parameter *delay* akan diuji menggunakan *ping* dengan melakukan ping dari h1 ke h8 dengan 10 skenario, mulai dari 100 packet sampai 1000 packet selama 10 detik. Parameter *throughput, jitter,* dan *packet loss* akan diuji secara bersamaan dengan melakukan *iperf* dan host h1 sebagai *client* dan host h8 sebagai *server.* Selama 10 detik iperf dijalankan. Dalam parameter *throughput, jitter dan packet loss, iperf* diuji dengan 10 kondisi: pertama, dengan bandwidth 100 Mbits dalam waktu 10 detik, kedua 200 Mbits dalam waktu 10 detik, ketiga 300 Mbits dalam waktu 10 detik, keempat 400 Mbits dalam waktu 10 detik, kelima 400 Mbits dalam waktu 10 detik, kelima 500 Mbits dalam waktu 10 detik, keenam 600 Mbits dalam waktu 10 detik, ketujuh 700 Mbits dalam waktu 10 detik, kedelapn 800 Mbits dalam waktu 10 detik, kesembilan 900 Mbits dalam waktu 10 detik, kesepuluh 1000 Mbits dalam waktu 10 detik.

3.6 Simulasi Pengujian

Implementasi system untuk implementasi optimalisasi *Routing* menggunakan modifikasi algoritma depth first search pada *Software*

Defined Network berbasis text atau cli (*command line interface*). Bahasa pemrograman yang digunakan adalah python 3.8 untuk ryu controller dan python 2.7 untuk mininet. Di Dalam linux lebih banyak menggunakan terminal dan berbasis text sehingga cukup memudahkan bagi administrator jaringan dalam mengkonfigurasi.

Proses ini menggunakan terminal dan dijalankan melalui terminal langsung pada ubuntu linux 20.04 yang menggunakan aplikasi virtualbox versi 6.1. Namun juga bisa diakses dari luar atau remote melalui aplikasi mobaxterm v23.0 denga mamasukkan alamat ip address serta username dan password yang sudah dibuat saat instalasinya. sehingga lebih memudahkan Ketika mengkonfigurasi dari luar (remote).



Gambar 3.8 Simulasi Pengujian

Dalam implementasi ini menggunakan virtual machine yang didalamnya sudah terinstall system operasi linux lubuntu 20.04. untuk text editor menggunakan visual studio code 1.77.0 untuk membangun controller yang akan digunakan.

Berdasarkan perancangan system, maka system akan menjalankan beberapa aplikasi yang dimulai dari mengaktifkan ryu controller kemudian diikuti dengan menjalankan emulator mininet 2.2.2.

BAB IV
IMPLEMENTASI DAN PENGUJIAN

4.1 Spesifikasi Implementasi

Hasil dari aplikasi yang dibuat yaitu dalam bentuk file *report* yang didapatkan dari pemasangan pada *virtual machine* dengan system operasi *linux.* Proses tersebut dilakukan dengan bantuan aplikasi bawaan yang sudah terdapat di system operasi yang terinstall di *virtual machine.* Dibutuhkan spesifikasi dari segi hardware maupun software.

Untuk melakukan implementasi system, perangkat keras yang digunakan adalah sebuah Laptop dengan spesifikasi perangkat keras sebagai berikut :

Processor	: Intel® Core™ I5-4300U @ 1.90 GHz
Memory	: DDR3L 8192 MB
HDD	: 500 GB
OS	: Linux Ubuntu 20.04

Dalam menjalankan dan implementasi membutuhkan berbagai macam perangkat lunak tambahan yang dapat menunjang proses implementasinya, antara lain sebagai berikut :

Bahasa Pemrograman	: Python 3.8.10 dan 2.7.18
Controller SDN	: Ryu Controller 4.30
Emulator Jaringan	: Mininet 2.2.2
Text Editor	: Visual Studio Code 1.77.0
Virtual Machine	: Oracle VM VirtualBox 6.1.30
Remote Server	: MobaXterm Personal v23.0

4.2 Instalasi Software

Untuk membangun simulasi Optimalisasi *Routing* Menggunakan Modifikasi Algoritma Depth First Search Pada Software Defined Network perlu melakukan instalasi software yang dibutuhkan pada system operasi Linux Ubuntu.

4.2.1 Instalasi Aplikasi Mininet

Mininet adalah sebuah aplikasi atau software yang berfungsi sebagai emulator jaringan digunakan untuk simulasi *Routing*. Berikut Langkah – Langkah dalam melakukan instalasi Mininet adalah sebagai berikut :

1. Mengunduh *source code* Mininet dari *repository* github :

```
$ git clone
git://github.com/mininet/mininet.git
```

2. Memilih versi Mininet :

```
$ cd mininet; git tag
$ git checkout -b 2.2.2; cd..
```

3. Proses instalasi Mininet

```
$ sudo mininet/util/install.sh -nfv
```

4. Proses tes hasil instalasi Mininet

```
$ sudo mn –switch ovsbr –test pingall
```

4.2.2 Instalasi Controller Software Defined Network

Untuk instalasi controller software defined network menggunakan Ryu controller, berikut cara installasinya :

1. Mengunduh *source code* Ryu dari *repository* github :

```
$ git clone
https://github.com/faucetsdn/ryu.git
```

2. Proses instalasi Ryu controller :

```
$ cd ryu; pip install .
```

4.2.3 Installasi Iperf

Iperf3 digunakan sebagai alat dalam pengujian throughput. Berikut cara installasinya :

1. Memperbarui repository Lubuntu :

```
$ sudo apt update
```

2. Proses instalasi Iperf :

```
$ sudo apt install iperf
```

4.3 Konfigurasi Software

Simulasi akan dilakukan dengan menggunakan beberapa variable dan konstanta dalam metode Modifikasi Depth First Search sebagai berikut :

1. Bandwidth path = 1000 Mbps
2. MAX_PATH = 2 jalur

4.3.1 Konfigurasi Topologi

Dalam penelitian ini, konfigurasi topologi dibuat menggunakan topologi fat – tree, dimana semua simpul, baik host maupun switch, diatur seperti membentuk sebuah pohon jaringan. Penggunaan topologi fat – tree memberikan keuntungan untuk penelitian jaringan,

yang mana membutuhkan banyak simpul dalam penelitian. Oleh karena itu, penggunaan emulator Mininet sangat penting untuk penelitian ini. Berikut adalah desain topologi fat – tree pada gambar terlihat bahwa dalam desain topologi tersebut terdapat 28 titik berikut ini :

1. Host, dalam desain topologi ini, ada h1, h2, h3, h4, h5, h6, h7, dan h8 yang masing – masing terhubung dengan switch dan berfungsi sebagai host. Pada simpul ini akan dilakukan pengaturan IP yang disesuaikan.
2. Switch, dalam desain topologi ini, terdapat s1, s2, s3, s4, s5, s6, s7, s8, s9, s10, s11, s12, s13, s14, s15, s16, s17, s18, s19, s20. Titik – titk tersebut berfungsi sebagai switch yang menggunakan metode ovs-kernel (ovsk), yang mengimplementasikan protocol openflow.Pada protokol tersebut switch digunakan dalam menjalankan software defined network (SDN).
3. Controller, pada gambar , semua switch terhubung ke titik c0, dimana semua aliran data pada masing – masing switch dikendalikan. Dalam kasus ini, penulisan menggunakan controller ryu untuk mengendalikan semua protokol openflow dalam jaringan yang didefinisikan oleh perangkat lunak SDN.

Dalam script topologi fat – tree, berikut adalah penulisannya :

```
info( '*** Add hosts\n')
h1 = net.addHost('h1', cls=Host, ip='10.0.0.1/8', defaultRoute=None)
h2 = net.addHost('h2', cls=Host, ip='10.0.0.2/8', defaultRoute=None)
h3 = net.addHost('h3', cls=Host, ip='10.0.0.3/8', defaultRoute=None)
h4 = net.addHost('h4', cls=Host, ip='10.0.0.4/8', defaultRoute=None)
h5 = net.addHost('h5', cls=Host, ip='10.0.0.5/8', defaultRoute=None)
h6 = net.addHost('h6', cls=Host, ip='10.0.0.6/8', defaultRoute=None)
h7 = net.addHost('h7', cls=Host, ip='10.0.0.7/8', defaultRoute=None)
h8 = net.addHost('h8', cls=Host, ip='10.0.0.8/8', defaultRoute=None)
```

Gambar 4.1 Konfigurasi IP pada Host

Gambar 4.1 menunjukkan bagaimana penambahan parameter akan mewakili penggunaan IP pada setiap host dalam simulasi. Setelah mendeklarasikan host, penambahan switch dilakukan. Penambahan link akan dilakukan setelah semua simpul dideklarasikan.

```
info( '*** Add switches\n')
s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
s3 = net.addSwitch('s3', cls=OVSKernelSwitch)
s4 = net.addSwitch('s4', cls=OVSKernelSwitch)
s5 = net.addSwitch('s5', cls=OVSKernelSwitch)
s6 = net.addSwitch('s6', cls=OVSKernelSwitch)
s7 = net.addSwitch('s7', cls=OVSKernelSwitch)
s8 = net.addSwitch('s8', cls=OVSKernelSwitch)
```

Gambar 4.2 Konfigurasi Penambahan Switch

Gambar 4.2 menunjukkan penambahan switch dan id switch yang sudah dirancang sebelumnya yaitu dari id switch s1 sampai s20 dengan menggunakan protocol OVSKernelSwitch.

```
info( '*** Add links\n')
net.addLink(s5, s1)
net.addLink(s7, s1)
net.addLink(s9, s1)
net.addLink(s11, s1)
net.addLink(s5, s2)
net.addLink(s7, s2)
```

Gambar 4.3 Konfigurasi Penambahan Simpul

Setelah melakukan penambahan link seperti yang terlihat pada gambar 4.4 dapat diamati adanya koneksi antar simpul yang digunakan dalam topologi ini. Dalam pembuatan topologi fat – tree, perlu menambahkan parameter saat menggunakan emulator. Hal ini dilakukan untuk mendukung eksekusi script topologi fat tree yang meniru kondisi fisik perangkat jaringan. Berikut adalah parameter yang digunakan :

1. Topo
   Parameter ini digunakan untuk mengidentifikasi nama topologi yang telah kita buat atau nama *class* yang sudah dideklarasikan dalam skrip topologi fat – tree.
2. Mac

Penggunaan parameter ini bertujuan untuk mengatur urutan Mac address pada setiap simpul yang sebelumnya dihasilkan secara acak oleh sistem atau emulator Mininet.

3. Link
   Parameter ini mencakup berbagai variabel pengaturan dalam hubungan antar *node – node* atau titik – titik dalam topologi.
4. Switch parameter ini digunakan untuk memilih jenis switch yang akan digunakan dalam topologi yang akan digunakan. dalam kasus ini, penggunaan variabel OVSKernelSwitch mengindikasikan penggunaan protokol openflow.
5. Controller
   Parameter ini digunakan untuk menentukan alamat akses kontrol ke *controller* dalam jaringan. Disini, penulis menggunakan variabel remote yang secara default mengarah ke alamat IP localhost.

Setelah eksekusi script fat – tree topologi beserta konfigurasinya maka, maka Mininet akan berjalan dan mendapatkan hasil pada gambar 4.4.

```
*** Adding controller
Unable to contact the remote controller at 127.0.0.1:6633
*** Add switches
*** Add hosts
*** Add links
*** Starting network
*** Configuring hosts
h1 h2 h3 h4 h5 h6 h7 h8
*** Starting controllers
*** Starting switches
*** Post configure switches and hosts
*** Starting CLI:
mininet>
```

Gambar 4.4 Eksekusi Mininet

4.3.2 Konfigurasi file launcher

Untuk mempermudah dalam menjalankan controller di ryu dan topologi di mininet, penulis membuat 2 jenis file launcher antara lain sebagai berikut :

1. Controller .sh
   Pada file launcher ini berfungsi untuk menampilkan daftar algoritma yang digunakan. Terdapat 4 jenis algoritma yang

ditampilkan yaitu : *singlepath* DFS konvensional, *Multipath* DFS konvensioanal, dan *Multipath* DFS modifikasi.

Eksekusi file controller.sh :

```
$ sudo ./controller.sh
```

2. Topo .sh

   Pada file ini berisi script yang berfungsi untuk menjalankan topologi fat – tree tanpa perlu menulis panjang pada terminal linux.

   Eksekusi file topo.sh :

```
$ sudo ./topo.sh
```

```
sdn@linux:~/Desktop/Proyek$ sudo ./controller.sh
 == Pilih Algoritma ==
1. Singlepath DFS
2. Multipath DFS
3. Multipath DFS Modification
Masukan pilihan (1-3):
```

Gambar 4.5 Eksekusi File Controller.Sh

Pada gambar 4.5 merupakan tampilan dari file launcher yang terdapat 4 pilihan yaitu singlepath DFS konvensional, *Multipath* DFS konvensional, dan *Multipath* DFS modifikasi.

```
#!/bin/bash

echo " == Pilih Algoritma =="
echo "1. Singlepath DFS"
echo "2. Multipath DFS"
echo "3. Singlepath DFS Modification"
echo "4. Multipath DFS Modification"
echo "5. Rate Monitor"
echo "6. Custom"
```

Gambar 4.6 Konfigurasi File Controller.Sh

Gambar 4.7 merupakan script file launcher yang mana terdapat variabel controller1 sampai controller4 digunakan untuk memanggil file controller ryu yang sudah disiapkan.

```
sdn@linux:~/Desktop/Proyek$ sudo ./topo.sh
 == Pilih Topologi ==
1. Topologi Fat Tree
Masukan nilai :
```

Gambar 4.7 Eksekusi File Topo.Sh

Pada gambar 4.8 merupakan tampilan dari file launcher topo.sh yang ada pilihan topologi yang digunakan yaitu topologi fat – tree.

```
#!/bin/bash

echo " == Pilih Topologi =="
echo "1. Topologi Fat Tree"
echo "2. Topologi Bipartite"
echo "3. Topologi Internet2"
```

Gambar 4.8 Konfigurasi File Topo.sh

Tujuan dari 2 file launcher ini untuk memudahkan dalam menjalankan program, baik itu controller maupun topologi tanpa perlu mengetikkan perintah yang panjang ke terminal linux.

4.3.3 Konfigurasi Program *Controller*

Pada konfigurasi ryu *controller* diperlukan untuk merubah atau menulis ulang script yang sudah disediakan ryu *controller* sebagai dasar software defined network dengan menggunakan bahasa pemrograman python 3.8.10.

Konfigurasi *Multipath* dan *single-path* pada software defined network Pada penelitian ini, penggunaan algoritma Depth First Search (DFS) diterapkan dalam pengaplikasian *Multipath* (multi jalur) dan *singlepath* (satu jalur) pada software defined network. Metode DFS digunakan karena dapat mengeksplorasi kemungkinan simpul dalam graf dengan menemukan simpul terdalam terlebih dahulu sebelum melakukan backtrack untuk menemukan simpul lain yang mungkin menggunakan tumpukan. Penggunaan tumpukan dalam algoritma ini memberikan fitur yang berguna terutama untuk perutean *Multipath*.

Karena dapat mengubah algoritma untuk menemukan jalur yang memungkinkan antara dua simpul atau lebih.

```
def get_paths(self, src, dst):
    computation_start = time.time()
    if src == dst:
        return [[src]]
    paths = []
    stack = [(src, [src])]
    i = 0
    while stack:
        (node, path) = stack.pop()
        for next in set(self.adjacency[node].keys()) - set(path):
            if next is dst:
                paths.append(path + [next])
            else:
                stack.append((next, path + [next]))
        i = i + 1
    paths.sort(key=len)
    #print(paths)
    print("=== Running Multipath DFS Modified algorithm ===")
    print("Iterasi : ", i)
    print("Path Execution Time : ", time.time() - computation_start)
    return paths
```

Gambar 4.9 Script Pencarian Jalur

Pada gambar 4.9, terdapat fungsi yang digunakan untuk mencari semua jalur yang dapat digunakan untuk berkomunikasi. Fungsi ini menerima alamat awal dan alamat tujuan yang dikirimkan dari tabel openflow. Setelah semua jalur ditemukan, dilakukan perhitungan jalur terpendek. Batas temuan jalur tersingkat digunakan untuk menentukan jalur yang akan dilalui atau yang sesuai kebutuhan. Jalur terpilih akan disimpan dan digunakan sebagai jalur utama. Digambar 4.10 terdapat fungsi perhitungan dan pemilihan jalur terpendek berdasarkan simpul yang dilalui serta penyimpanannya. Terdapat tiga fungsi dalam gambar 4.10, yaitu :

1. get_link_cost adalah fungsi yang membandingkan antar 2 jalur yang akan digunakan.
2. Get_path_cost adalah fungsi untuk menghitung jalur yang ditemukan.
3. Get_optimal_path adalah fungsi menentukan jalur terbaik untuk digunakan.

```
    def get_link_cost(self, s1, s2):
        e1 = self.adjacency[s1][s2]
        e2 = self.adjacency[s2][s1]
        bl = min(self.bandwidths[s1][e1], self.bandwidths[s2][e2])
        ew = REFERENCE_BW/bl
        return ew

# calculate link cost
    def get_link_cost(self, s1, s2):
        e1 = self.adjacency[s1][s2]
        e2 = self.adjacency[s2][s1]
        bl = min(self.bandwidths[s1][e1], self.bandwidths[s2][e2])
        ew = REFERENCE_BW/bl
        return ew
```

Gambar 4.10 Script Pencarian Jalur Optimal, Perhitungan, dan Penyimpanannya

Dari semua fungsi tersebut maka pada gambar 4.12 dapat dilakukannya pemilihan jalur optimal, perhitungan, dan penyimpanan jalur utama komunikasi data. Ada penamaan grup dan port yang akan didaftarkan pada flowtable. Berikut penulisan penamaan penambahan grup dan penambahan port pada gambar 4.11 :

```
def add_ports_to_paths(self, paths, first_port, last_port):
    paths_p = []
    for path in paths:
        p = {}
        in_port = first_port
        for s1, s2 in zip(path[:-1], path[1:]):
            out_port = self.adjacency[s1][s2]
            p[s1] = (in_port, out_port)
            in_port = self.adjacency[s2][s1]
        p[path[-1]] = (in_port, last_port)
        paths_p.append(p)
    return paths_p
```

Gambar 4.11 Script Penambahan Port Input Dan Output Switch Pada Jalur

```
def install_paths(self, src, first_port, dst, last_port, ip_src, ip_dst):
    computation_start = time.time()
    paths = self.get_optimal_paths(src, dst)
    pw = []
    for path in paths:
        pw.append(self.get_path_cost(path))
        print("Chosen paths from:", src, " to ",dst, " = ", path, "cost = ", pw[len(pw) -1 ])
```

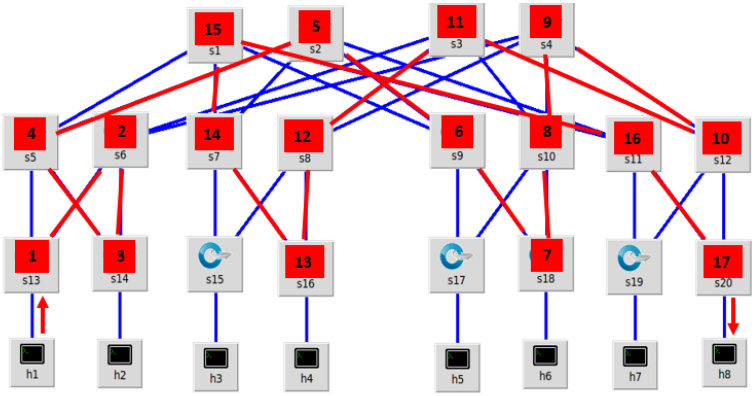
Gambar 4.12 Script Instalasi Jalur

4.4 Pengujian

Pada sub bab ini akan dibahas mengenai pengujian untuk memperoleh data *Quality of Service* dengan poin pengujian sebagai berikut : Pencarian jalur, *Delay, Throughput, Jitter, Packet Loss.*

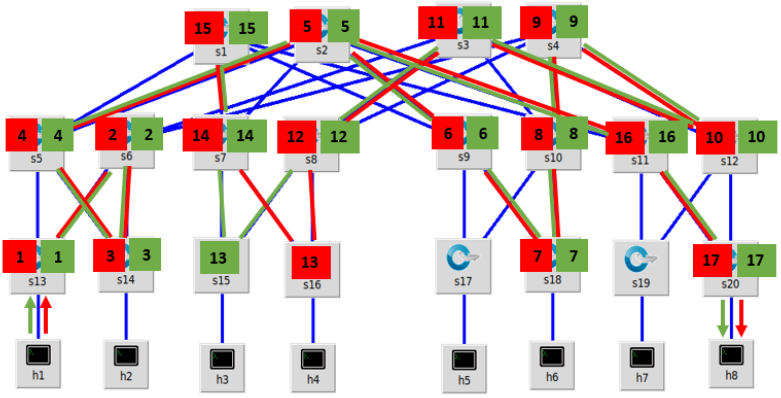
4.4.1 Pengujian Pencarian Jalur

Pengujian dilkukan untuk mencari jalur dalam topologi fat-tree berdasarkan desainnya. Pada topologi tersebut, pengamatan dilakukan terhadap jalur yang ditemukan. Link bandwidth yang digunakan pada topologi tersebut adalah 1000 Mbps. Beberapa scenario pengujian dilakukan, yaitu pencarian jalur *single-path* dan *multipath* menggunakan algoritma DFS konvensional dan modifikasi DFS. Untuk melakukan pencarian jalur, aplikasi ping digunakan dari sumber h1 ke tujuan h8.



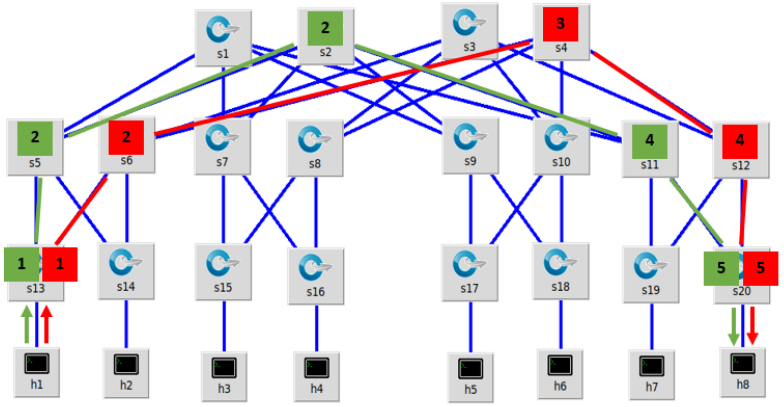
Gambar 4.13 Singlepath Menggunakan DFS Konvensional

Gambar 4.13 ditemukannya jalur – jalur dari *single path* DFS konvensional, dari host 1 ke host 8 dengan melewati cukup banyak switch.



Gambar 4.14 Multipath Menggunakan DFS Konvensional

Gambar 4.14 multipath menggunakan algoritma DFS yang tidak dimodifikasi menghasilkan dua jalur dan tidak independent dan masih memiliki kesamaan.



Gambar 4.15 Multipath Menggunakan DFS Modifikasi

Gambar 4.15 multipath menggunakan algoritma DFS yang dimodifikasi menghasilkan dua jalur yang independent dan tidak adanya persamaan jalur yang dilalui.

Tabel 4.1 Hasil Pengujian Pencarian Jalur

	DFS	
Single – path	[13,6,14,5,2,9,18,10, 4,12,3,8,16,7,1,11,20]	
	DFS	**M.DFS**
Multipath	[13,6,14,5,2,9,18,10, 4,12,3,8,16,7,1,11,20] [13,6,14,5,2,9,18,10,4, 12,3,8,15,7,1,11,20]	[13,6,4,12,20] [13,5,2,11,20]

Berdasarkan hasil pengujian pencarian jalur yang telah dilakukan dari h1 ke h8, dapat dilihat pada tabel 4.1 bahwa pencarian *single-path* menggunakan algoritma DFS yang tidak dimodifikasi menghasilkan satu jalur terjauh yang sudah ditemukan yaitu (13,6,14,5,2,9,18,10,4,12,3,8,16,7,1,11,20). Untuk pencarian *multipath* menggunakan DFS yang tidak dimodifikasi, dihasilkan jalur – jalur yang tidak independen (13,6,14,5,2,9,18,10,4,12,3,8,16,7,1,11,20)dan(13,6,14,5,2,9,18,10,4 ,12,3,8,15,7,1,11,20) yang keduanya memiliki beberapa kesamaan pada jalurnya. Pada pencarian *multipath* menggunakan algoritma DFS yang dimodifikasi, dapat ditemukan jalur – jalur yang independen (13,6,4,12,20) dan (13,5,2,11,20).

4.4.2 Pengujian Quality of Service

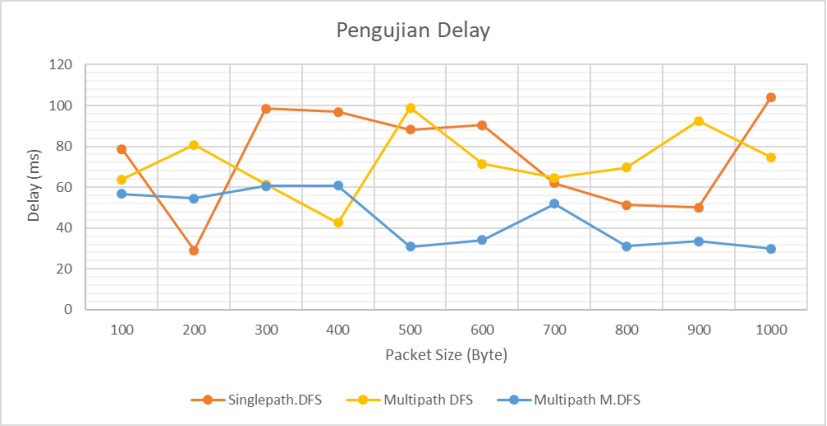
Salah satu cara untuk mengukur kinerja layanan adalah dengan menggunakan parameter Qos yang terdiri dari *Delay, Throughput, Jitter dan Packet Loss*.

Pengujian yang pertama adalah untuk menemukan nilai delay / latency dari jaringan antar node / simpul. Pada pengujian ini aplikasi menggunakan ping dengan tujuan ke host paling jauh (h8) serta variabel -c sebagai jumlah hitungan : 10s dan variabel -s sebagai beban paket : 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000. Pengujian akan mengambil nilai rata – rata yang didapatkan dari penggunaan ping yang sudah ditentukan.

Tabel 4.2 Tabel Hasil Pengujian Delay

No	Beban Paket (Byte)	Single-path (ms)	Multipath (ms)	
		DFS	DFS	M.DFS
1	100	78,632	63,763	56,687
2	200	29,016	80,733	54,511
3	300	98,293	61,395	60,597
4	400	96,905	42,653	60,721
5	500	88,227	98,923	30,864
6	600	90,385	71,425	34,090
7	700	62,015	64,628	51,968
8	800	51,319	69,673	31,038
9	900	50,163	92,389	33,449
10	1000	103,977	74,627	29,885
Rata - Rata		74,9032	72,0209	44,3855

Didapatkan poin dari dari uji delay / latency pada tabel 4.2 dan gambar 4.16 Delay / Latency : Rata – rata pengujian delay pada *single-path* DFS konvensional memiliki nilai lebih besar yaitu 74,903 ms. Rata – rata pengujian delay pada *Multipath* modifikasi DFS memiliki nilai lebih kecil yaitu 44,385 ms. Pada *multipath* modifikasi DFS memiliki nilai yang cukup stabil dibandingkan dengan *singlepath* DFS konvensional yang mengalami perbedaan yang cukup signifikan terutama pada 200 Mbit dan 1000 Mbit. Dengan poin tersebut dapat disimpulkan bahwa delay dari *Multipath* modifikasi DFS memiliki nilai lebih rendah dibandingkan dengan *single-path* terutama *Multipath* DFS konvensional. Yang mana modifikasi DFS cukup stabil dalam parameter *delay.*



Gambar 4.16 Grafik Delay / Latency

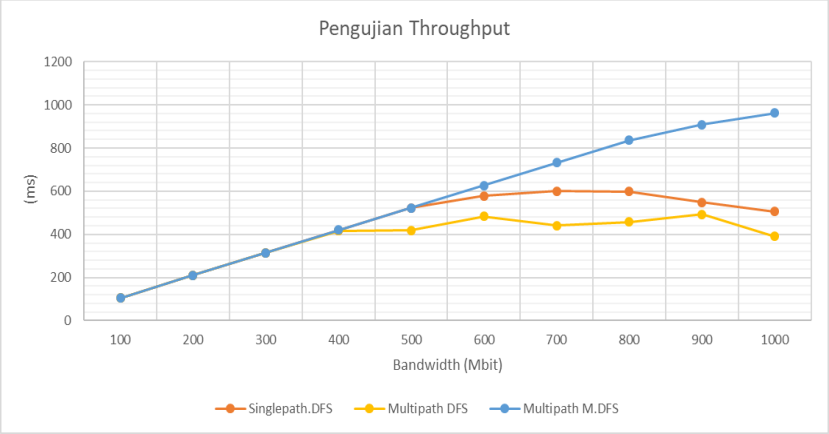
Pada pengujian selanjutnya digunakan untuk memperoleh *throughput, jitter,* serta *packet loss* dari jaringan antar simpul / node. Pada pengujian ini menggunakan aplikasi iperf dengan host tujuan (h8) sebagai server dan host client (h1), beserta variabel waktu -t yaitu 10s, variabel -u untuk paket UDP, dan variabel -b bandwidth paket: 100Mb, 200Mb, 300Mb, 400Mb, 500Mb, 600Mb, 700Mb, 800Mb, 900Mb, 1000Mb. Pengujian akan mengambil rata-rata yang didapatkan dari penggunaan iperf yang sudah ditentukan.

Tabel 4.3 Tabel Hasil Pengujian Throughput

No	Beban Paket (Mbit)	Single-path (Mbps)	Multipath (Mbps)	
		DFS	DFS	M.DFS
1	100	105	105	105
2	200	210	210	210
3	300	314	314	315
4	400	419	417	420
5	500	522	419	523
6	600	578	483	627
7	700	601	441	732

Tabel 4.3 Lanjutan

No	Beban Paket (Mbit)	Single-path (Mbps)	Multipath (Mbps)	
		DFS	DFS	M.DFS
8	800	598	458	836
9	900	548	493	908
10	1000	506	391	962
Rata - Rata		440,1	373	563,8



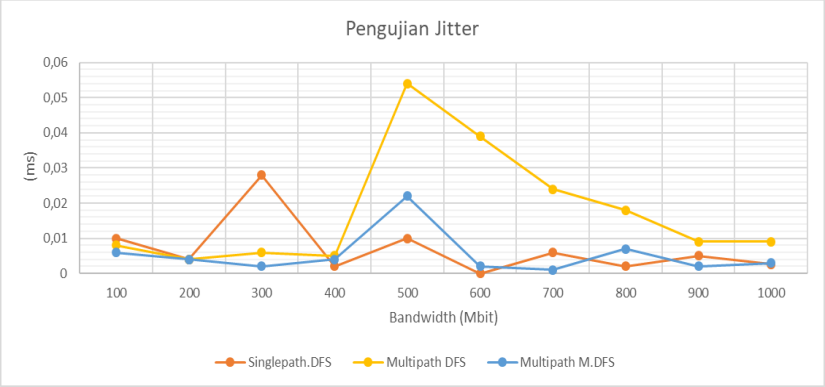
Gambar 4.17 Grafik Throughput

Pada hasil pengujian *Throughput* di tabel 4.3 dan gambar 4.17 Rata - rata *throughput* pada *Multipath* modifikasi DFS memiliki nilai yang lebih tinggi yaitu 563,8 Mbps. *Multipath* DFS konvensional memiliki nilai rata – rata *throughput* paling kecil yaitu 373,1 Mbps.

Throughput pada *multipath* modifikasi DFS memiliki kenaikan yang cukup konstan serta stabil mulai dari parameter 100 Mbit sampai 1000 Mbit dibandingkan *multipath* DFS konvensional yang mana nilai *throughputnya* tidak stabil sedangkan *single-path* DFS konvensional mulai pada beban *bandwidth* 700 Mbit mengalami penurunan nilai *throughput*. Dengan poin tersebut dapat disimpulkan bahwa *throughput* dari *Multipath* modifikasi DFS mengalami peningkatan dan stabil dibandingkan DFS konvensional pada dua skema routing.

Tabel 4.4 Tabel Hasil Pengujian Jitter

No	Beban Paket (Mbit)	Single-path (ms)	Multipath (ms)	
		DFS	DFS	M.DFS
1	100	0,010	0,008	0,006
2	200	0,004	0,004	0,004
3	300	0,028	0,006	0,002
4	400	0,002	0,005	0,004
5	500	0,010	0,054	0,022
6	600	0	0,039	0,002
7	700	0,006	0,024	0,001
8	800	0,002	0,018	0,007
9	900	0,005	0,009	0,002
10	1000	0,002	0,009	0,003
Rata - Rata		0,0069	0,0176	0,0053



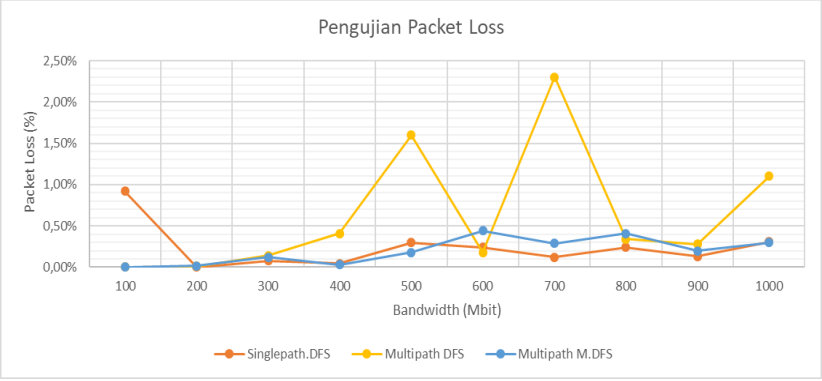
Gambar 4.18 Grafik Jitter

Pada hasil pengujian *Jitter* di tabel 4.4 dan gambar 4.18 sebagai berikut : *Multipath* Modifikasi DFS memiliki nilai rata – rata jitter paling sedikit yaitu 0,005 ms. Rata - rata hasil pengujian packet loss multipath DFS konvensional memiliki nilai tertinggi dibandingkan modifikasi DFS yaitu 0,17 ms.Pada multipath *multipath* DFS konvensional hasil jitter memiliki perbedaan yang signifikan yang mana pada beban bandwidth mulai dari 500 Mbit mengalami kenaikan

serta tidak stabil dibandingkan dengan *multipath* DFS modifikasi yang memiliki nilai yang cukup stabil walaupun pada 500 Mbit mengalami kenaikan tapi tidak terlalu signifikan. Dengan poin tersebut dapat disimpulkan bahwa *jitter* dari *multipath* modifikasi DFS dapat memberikan nilai yang cukup stabil dibandingkan *multipath* DFS konvensional.

Tabel 4.5 Tabel Hasil Pengujian Packet Loss

No	Beban Paket (Mbit)	Single-path (%)	Multipath (%)	
		DFS	DFS	M.DFS
1	100	0,92	0	0
2	200	0	0,01	0,02
3	300	0,08	0,14	0,12
4	400	0,04	0,41	0,03
5	500	0,30	1,60	0,18
6	600	0,24	0,18	0,44
7	700	0,12	2,30	0,29
8	800	0,24	0,34	0,41
9	900	0,13	0,28	0,20
10	1000	0,31	1,10	0,30
Rata - Rata		0,24	1	0,20



Gambar 4.19 Grafik Packet Loss

Pada pengujian packet loss pada 100 Mbits memiliki packet loss yang cukup tinggi seperti yang terlihat pada tabel 4.5 dan gambar 4.19. Pada algoritma *single-path* DFS konvensional terutama pada skenario 100 Mbits, sedangkan *Multipath* modifikasi DFS memiliki nilai packet loss 0,2 %.

4.5 Analisa Hasil Pengujian

Sub bab ini menganalisa dari hasil pengujian sebelumnya yaitu *delay*, *throughput*, *jitter*, dan *packet loss* dengan nilai rata – rata dan beban terbesar, Metode ini digunakan untuk mendapatkan data sebagai berikut :

Tabel 4.6 Analisa Pengujian

Parameter	***Single-path***	***Multipath***	
	DFS	**DFS**	**M.DFS**
Delay	74,903 ms	72,020 ms	44,385 ms
Throughput	440,1 Mbps	373,1 Mbps	563,8 Mbps
Jitter	0,006 ms	0,017 ms	0,005 ms
Packet Loss	0,31%	1,10 %	0,20%

Dari tabel 4.6 dapat diambil beberapa kesimpulan yaitu : *multipath* DFS memiliki *delay* lebih rendah dibandingkan dengan modifikasi DFS pada *multipath*. Secara khusus, *multipath* modifikasi DFS konvensional memiliki delay lebih rendah daripada DFS konvensional pada *multipath* routing yaitu sebesar 44,385 ms. Pada *throughput* modifikasi *multipath* DFS memiliki *throughput* lebih tinggi dibandingkan dengan DFS konvensional pada kedua jenis *routing*, baik *single-path* maupun *multipath*. Pada *multipath routing* modifikasi DFS memiliiki *throughput* lebih tinggi daripada DFS konvensional yaitu sebesar 563,8 Mbps. Pada *jitter multipath* modifikasi DFS memiliki *jitter* lebih rendah dibandingkan dengan DFS konvensional pada kedua jenis *routing*, sebesar 0,0053 ms. Secara khusus, modifikasi DFS memiliki tingkat kehilangan paket lebih rendah daripada DFS konvensional pada *multipath routing* yaitu sebesar 0,20 % atau selisih 0,02%. Dengan demikian pada algoritma modifikasi DFS pada *multipath routing*, memberikan peningkatan performa dalam hal *throughput* yang lebih tinggi, *jitter* yang lebih stabil, dan tingkat kehilangan paket yang lebih rendah dibandingkan dengan DFS konvensional.

BAB V
PENUTUP

5.1 Kesimpulan

Berdasarkan hasil analisis dan pengujian, penulis memperoleh kesimpulan yang dapat diambil dari pengujian optimalisasi *Routing* menggunakan modifikasi algoritma depth first search pada software defined network sebagai berikut :

1. Pemanfaatan multipath routing dapat mengatasi kemacetan data karena memanfaatkan lebih dari satu jalur yang aktif sehingga mengurangi terjadinya kemacetan data pada jaringan.
2. Penggunaan algoritma DFS yang dimodifikasi pada multipath routing dapat menemukan jalur – jalur yang tidak memiliki kesamaan.
3. Berdasarkan hasil pengujian antara *singlepath* dan *Multipath* berdasarkan parameter beban 100 – 1000 selama 10 s pada *delay, throughput, jitter, dan packet loss*. Dapat disimpulkan bahwa algoritma DFS modifikasi mengalami peningkatan dalam hal throughput, dan packet loss yang lebih sedikit dibandingkan dengan DFS konvensional.

5.2 Saran

Adapun saran – saran yang dapat digunakan untuk pengembangan pada penelitian selanjutnya :

1. Perlu adanya suatu kondisi atau Batasan *threshold* apakah perlu menggunakan algoritma *singlepath* atau *Multipath* DFS pada topologi yang digunakan secara dynamis tanpa perlu user yang mengaturnya.
2. Adanya tampilan GUI (Graphical User Interface) dalam menampilkan hasil pencarian atau hasil QOS sehingga dapat lebih mudah dipahami.
3. Program controller perlu adanya pengembangan dalam penemuan jalur untuk mengatasi jika ada kegagalan link dalam topologi.

DAFTAR PUSTAKA

Chiang, Y.-R.et al., (2017). *A Multipath Transmission Scheme for the Improvement of Throughput over SDN.Proceedings of the 2017 IEEEInternational Conference on Applied System Innovation*, IEEE.

Contributors, M. P. (2023). *Mininet: An Instant Virtual Network on Your Laptop (or Other PC) - Mininet*. http://mininet.org/

Firdausi, A., & Wardani, H. W. (2020). *Simulasi Dan Analisa QoS Dalam Jaringan VPN Site To Site Berbasis IPSec Dengan Routing Dynamic*. Jurnal Telekomunikasi Dan Komputer, 10(2), 49. https://doi.org/10.22441/incomtech.v10i2.8131

Gueant, V. (2023). *iPerf - The TCP, UDP and SCTP network bandwidth measurement tool*. https://iperf.fr/

Mulyana, E., & Arif, M. (2023). *Buku Komunitas SDN-RG*.

Rangkuty, M. F., Ijtihadie, R. M., & Ahmad, T. (2020). *DEVELOPMENT OF LOAD BALANCING MECHANISMS IN SDN DATA PLANE FAT TREE USING MODIFIED DIJKSTRA'S ALGORITHM*. JUTI: Jurnal Ilmiah Teknologi Informasi, 18(2), 197. https://doi.org/10.12962/j24068535.v18i2.a1008

Saputra, R. H., & Subardono, A. (2020). *PENGARUH FAILOVER PADA JARINGAN SOFTWARE-DEFINED NETWORK DAN KONVENSIONAL*. In *Journal of Internet and Software Engineering* (Vol. 1, Issue 1). https://www.opennetworking.org/

Sembiring, A. C., Kurniawan, W., & Yahya, W. (2018). *Mencari Jalur K Terpendek Menggunakan Yen Algoritm Untuk Multipath Routing Pada Openflow Software-Defined Network* (Vol. 2, Issue 9). http://j-ptiik.ub.ac.id

Suryo Wicaksono, I., & Hari Trisnawan, P. (2021). *Implementasi Multipath Routing menggunakan Algoritme Iterative Deepening*

Depth First Search pada OpenFlow Software-Defined Networking (Vol. 5, Issue 2). http://j-ptiik.ub.ac.id

Sutawijaya, B., Basuki, A., & Abdurrachman Bachtiar, F. (2020). *ANALISIS KINERJA ALGORITME TCP CONGESTION CONTROL BERDASARKAN SINGLE DAN MULTIPLE FLOW PADA MULTI-PATH ROUTING*. 7(5), 961–970. https://doi.org/10.25126/jtiik.202072402

Turmudi, A., & Abdul Majid, F. (2019). *SIGMA-Jurnal Teknologi Pelita Bangsa ANALISIS QOS (QUALITY OF SERVICE) DENGAN METODE TRAFFI SHAPING PADA JARINGAN INTERNET (STUDI KASUS : PT TOYONAGA INDONESIA)* (Vol. 9).

Ubuntu. (n.d.). (2023). *Enterprise Open Source and Linux | Ubuntu*. https://ubuntu.com/

Wibowo, M. A., Yahya, W., & Kartikasari, D. P. (2018). *Implementasi Link Fast-Failover Pada Multipath Routing Jaringan Software-Defined Network* (Vol. 2, Issue 10). http://j-ptiik.ub.ac.id

Yudha, U., Wirawan, T., Yahya, W., & Basuki, A. (2018a). *Implementasi Multipath Routing Berbasis Algoritme DFS yang Dimodifikasi* (Vol. 2, Issue 11). http://j-ptiik.ub.ac.id

Yudha, U., Wirawan, T., Yahya, W., & Basuki, A. (2018b). *Implementasi Multipath Routing Berbasis Algoritme DFS yang Dimodifikasi* (Vol. 2, Issue 11). http://j-ptiik.ub.ac.id

RIWAYAT PENULIS

DATA PRIBADI

Nama	: Haris Abdul Afif
Tempat/tgl lahir	: Surabaya, 14 Desember 2000
Jenis Kelamin	: Laki – Laki
Agama	: Islam
Alamat	: Dsn Ketapan, Ds Pekoren, Kec Rembang, Kab Pasuruan
Telepon	: 0895335975807
Alamat Email	: abdulab9090@gmail.com

DATA KELUARGA

Nama Ayah	:	Tumino
Pekerjaan Ayah	:	Karyawan Swasta
Nama Ibu	:	Sunarsih Irmalasari
Pekerjaan Ibu	:	Ibu Rumah Tangga (IRT)
Alamat	:	Jl. Ketapan, Pekoren, Rembang, Pasuruan

RIWAYAT PENDIDIKAN

2007 – 2012	:	SDN PEKOREN III
2012 - 2015	:	SMPN III BANGIL
2016 - 2019	:	SMKN 1 BANGIL
2019 – Sekarang	:	Institut ASIA Malang

LAMPIRAN A
SCRIPT CONTROLLER

```
from ipaddress import AddressValueError
from ryu.base import app_manager
from ryu.controller import ofp_event
from ryu.controller.handler import
CONFIG_DISPATCHER, MAIN_DISPATCHER, DEAD_DISPATCHER
from ryu.controller.handler import set_ev_cls
from ryu.ofproto import ofproto_v1_3
from ryu.lib.mac import haddr_to_bin
from ryu.lib.packet import packet
from ryu.lib.packet import arp
from ryu.lib.packet import ethernet
from ryu.lib.packet import ipv4
from ryu.lib.packet import ipv6
from ryu.lib.packet import ether_types
from ryu.lib import mac, ip
from ryu.topology.api import get_switch, get_link
from ryu.app.wsgi import ControllerBase
from ryu.topology import event

from collections import defaultdict
from operator import itemgetter
from datetime import datetime

import setting
from ryu.lib import hub
from ryu.lib.packet import packet

import csv
import os
import random
import time
import threading

import setting

from requests import get
from subprocess import check_output
from _thread import start_new_thread
```

```
from operator import itemgetter
import logging
import socket
import time
import os
import shlex
import json
import re
import random

byte = defaultdict(lambda: 0)
clock = defaultdict(lambda: 0)
thr = defaultdict(lambda: defaultdict(lambda: 0))

# switches
switches = defaultdict(dict)

# myhost[srcmac]->(switch, port)
mymac = {}

topology_map = defaultdict(dict)

min_route = defaultdict(dict)

# adjacency map [sw1][sw2]->port from sw1 to sw2
adjacency = defaultdict(dict)

multipath_group_ids = {}

group_ids = []

# Cisco Reference bandwidth = 1 Gbps
REFERENCE_BW = 10000000

DEFAULT_BW = 10000000

collector = '127.0.0.1'

def getIfInfo(ip):
    '''
    Get interface name of ip address (collector)
    '''
```

```
    s = socket.socket(socket.AF_INET,
socket.SOCK_DGRAM)
    s.connect((ip, 0))
    ip = s.getsockname()[0]
    ifconfig = check_output(['ifconfig'])
    ifs = re.findall(r'^(\S+).*?inet
addr:(\S+).*?', ifconfig, re.S | re.M)
    for entry in ifs:
        if entry[1] == ip:
            return entry

def measure_link():
    '''
    Measure outgoing traffic per second for all
switch ports
    '''
    while True:
        try:
            for switch in switches:
                for port in
switches[switch]['ports']:
                    url = 'http://' + collector +
':8008/metric/' + \
                        collector + '/' +
switches[switch]['ports'][port]['ifindex'] + \
                        '.ifoutoctets/json'
                    r = get(url)
                    response = r.json()
                    # print response
                        # Bps to Kbps
                    thr[switch][port] =
response[0]['metricValue'] * 8 / 1000
                    # print switch,thr[switch]
        except:
            pass
        hub.sleep(1)


def path_cost(route):
    cost = 0
    for s, p in route:
        for i in thr[s]:
```

```
            cost += thr[s][i]
    return cost


def measure_path(thread):
    while True:
        for src in list(topology_map.keys()):
            for dst in
list(topology_map[src].keys()):
                try:
                    min_route[src][dst] = min(
                        topology_map[src][dst],
key=path_cost)
                except KeyError:
                    pass
        time.sleep(0.1)


class ProjectController(app_manager.RyuApp):
    OFP_VERSIONS = [ofproto_v1_3.OFP_VERSION]
    def __init__(self, *args, **kwargs):
        super(ProjectController,
self).__init__(*args, **kwargs)
        self.mac_to_port = {}
        self.topology_api_app = self
        self.flow_stats = {}
        self.flow_speed = {}
        self.datapath_list = {}
        self.datapaths = {}
        self.arp_table = {}
        self.switches = []
        self.hosts = {}
        self.stats = {}
        self.overflow = []
        self.need_two = True
        self.multipath_group_ids = {}
        self.group_ids = []
        self.adjacency = defaultdict(dict)
        self.bandwidths = defaultdict(lambda:
defaultdict(lambda: DEFAULT_BW))
        self.paths = []
```

```
    def get_paths(self, src, dst):
        computation_start = time.time()
        if src == dst:
            return [[src]]
        paths = []
        stack = [(src, [src])]
        i = 0
        while stack:
            (node, path) = stack.pop()
            for next in
set(self.adjacency[node].keys()) - set(path):
                if next is dst:
                    paths.append(path + [next])
                else:
                    stack.append((next, path +
[next]))
            i = i + 1
        paths.sort(key=len)
        return paths

    def get_link_cost(self, s1, s2):
        e1 = self.adjacency[s1][s2]
        e2 = self.adjacency[s2][s1]
        bl = min(self.bandwidths[s1][e1],
self.bandwidths[s2][e2])
        ew = REFERENCE_BW/bl
        return ew

# calculate link cost
    def get_link_cost(self, s1, s2):
        e1 = self.adjacency[s1][s2]
        e2 = self.adjacency[s2][s1]
        bl = min(self.bandwidths[s1][e1],
self.bandwidths[s2][e2])
        ew = REFERENCE_BW/bl
        return ew

# calculate path cost
    def get_path_cost(self, path):
        cost = 0
        for i in range(len(path) - 1):
```

```
            cost += self.get_link_cost(path[i],
path[i+1])
        return cost

    def get_optimal_paths(self, src, dst):
        paths = self.get_paths(src, dst)
        if self.need_two == True:
            MAX_PATHS = float('Inf')
        else:
            MAX_PATHS = 1
        paths_count = len(paths) if len(paths) <
MAX_PATHS else MAX_PATHS
        #print('available paths  :', sorted(paths))
        return sorted(paths, key=lambda x:
self.get_path_cost(x))[1:3]

    def add_ports_to_paths(self, paths, first_port,
last_port):
        paths_p = []
        for path in paths:
            p = {}
            in_port = first_port
            for s1, s2 in zip(path[:-1], path[1:]):
                out_port = self.adjacency[s1][s2]
                p[s1] = (in_port, out_port)
                in_port = self.adjacency[s2][s1]
            p[path[-1]] = (in_port, last_port)
            paths_p.append(p)
        return paths_p

    def generate_openflow_gid(self):
        n = random.randint(0, 2**32)
        while n in self.group_ids:
            n = random.randint(0, 2**32)
        return n

# Installing paths
    def install_paths(self, src, first_port, dst,
last_port, ip_src, ip_dst):
        computation_start = time.time()
        paths = self.get_optimal_paths(src, dst)
        pw = []
```

```
        for path in paths:
            pw.append(self.get_path_cost(path))
            print("Chosen paths from:", src, " to
",dst, " = ", path, "cost = ", pw[len(pw) -1 ])

# saving to csv
        header = ['timestamp', 'Available paths
from : ', 'to : ', 'path : ', 'cost : ', 'Path
installation finished in :']
        data = [datetime.now(), src, dst, path, pw,
time.time() - computation_start]

        with
open('./data_export/multipath_dfs_modif.csv', 'a+',
newline='') as f:

            writer = csv.writer(f)

            writer.writerow(data)

        sum_of_pw = sum(pw) * 1.0
        paths_with_ports =
self.add_ports_to_paths(paths, first_port,
last_port)
        switches_in_paths = set().union(*paths)

        for node in switches_in_paths:

            dp = self.datapath_list[node]
            ofp = dp.ofproto
            ofp_parser = dp.ofproto_parser

            ports = defaultdict(list)
            actions = []
            i = 0

            for path in paths_with_ports:
                if node in path:
                    in_port = path[node][0]
                    out_port = path[node][1]
                    if (out_port, pw[i]) not in
ports[in_port]:
```

```
ports[in_port].append((out_port, pw[i]))
                i += 1

            for in_port in ports:

                match_ip = ofp_parser.OFPMatch(
                    eth_type=0x0800,
                    ipv4_src=ip_src,
                    ipv4_dst=ip_dst
                )
                match_arp = ofp_parser.OFPMatch(
                    eth_type=0x0806,
                    arp_spa=ip_src,
                    arp_tpa=ip_dst
                )

                out_ports = ports[in_port]
                # print out_ports

                if len(out_ports) > 1:
                    group_id = None
                    group_new = False

                    if (node, src, dst) not in
self.multipath_group_ids:
                        group_new = True
                        self.multipath_group_ids[
                            node, src, dst] =
self.generate_openflow_gid()
                    group_id =
self.multipath_group_ids[node, src, dst]

                    buckets = []
                    #print ("node at ",node," out
ports : ",out_ports)
                    for port, weight in out_ports:
                        bucket_weight =
int(round((1 - weight/sum_of_pw) * 10))
                        bucket_action =
[ofp_parser.OFPActionOutput(port)]
                        buckets.append(
```

```
                                ofp_parser.OFPBucket(
weight=bucket_weight,
                                    watch_port=port,
watch_group=ofp.OFPG_ANY,
actions=bucket_action
                                )
                            )

                        if group_new:
                            req =
ofp_parser.OFPGroupMod(
                                dp, ofp.OFPGC_ADD,
ofp.OFPGT_SELECT, group_id,
                                buckets
                            )
                            dp.send_msg(req)
                        else:
                            req =
ofp_parser.OFPGroupMod(
                                dp, ofp.OFPGC_MODIFY,
ofp.OFPGT_SELECT,
                                group_id, buckets)
                            dp.send_msg(req)

                        actions =
[ofp_parser.OFPActionGroup(group_id)]

                        self.add_flow(dp, 32768,
match_ip, actions)
                        self.add_flow(dp, 1, match_arp,
actions)

                    elif len(out_ports) == 1:
                        actions =
[ofp_parser.OFPActionOutput(out_ports[0][0])]

                        self.add_flow(dp, 32768,
match_ip, actions)
```

```
                    self.add_flow(dp, 1, match_arp,
actions)
        print("Path installation finished in ",
time.time() - computation_start)
        print("============================")
        return paths_with_ports[0][src][1]


    def add_flow(self, datapath, priority, match,
actions, buffer_id=None):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        inst =
[parser.OFPInstructionActions(ofproto.OFPIT_APPLY_A
CTIONS,

actions)]
        if buffer_id:
            mod =
parser.OFPFlowMod(datapath=datapath,
buffer_id=buffer_id,

priority=priority, match=match,

instructions=inst)
        else:
            mod =
parser.OFPFlowMod(datapath=datapath,
priority=priority,
                                      match=match,
instructions=inst)
        datapath.send_msg(mod)

    @set_ev_cls(ofp_event.EventOFPSwitchFeatures,
CONFIG_DISPATCHER)
    def _switch_features_handler(self, ev):
        print("switch_features_handler invoked")
        datapath = ev.msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match = parser.OFPMatch()
```

```
        actions =
[parser.OFPActionOutput(ofproto.OFPP_CONTROLLER,ofp
roto.OFPCML_NO_BUFFER)]
        self.add_flow(datapath, 0, match, actions)


@set_ev_cls(ofp_event.EventOFPPortDescStatsReply,
MAIN_DISPATCHER)
    def port_desc_stats_reply_handler(self, ev):
        switch = ev.msg.datapath
        for p in ev.msg.body:
            self.bandwidths[switch.id][p.port_no] =
p.curr_speed

    @set_ev_cls(ofp_event.EventOFPPacketIn,
MAIN_DISPATCHER)
    def _packet_in_handler(self, ev):
        msg = ev.msg
        datapath = msg.datapath
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser
        in_port = msg.match['in_port']

        pkt = packet.Packet(msg.data)
        eth = pkt.get_protocol(ethernet.ethernet)
        arp_pkt = pkt.get_protocol(arp.arp)

        if eth.ethertype == 35020:
            return

        if pkt.get_protocol(ipv6.ipv6):  # Drop the
IPV6 Packets.
            match =
parser.OFPMatch(eth_type=eth.ethertype)
            actions = []
            self.add_flow(datapath, 1, match,
actions)
            return None

        dst = eth.dst
        src = eth.src
        dpid = datapath.id
```

```
        if src not in self.hosts:
            self.hosts[src] = (dpid, in_port)

        out_port = ofproto.OFPP_FLOOD

        if arp_pkt:
            src_ip = arp_pkt.src_ip
            dst_ip = arp_pkt.dst_ip
            if arp_pkt.opcode == arp.ARP_REPLY:
                self.arp_table[src_ip] = src
                h1 = self.hosts[src]
                h2 = self.hosts[dst]
                out_port =
self.install_paths(h1[0], h1[1], h2[0], h2[1],
src_ip, dst_ip)
                self.install_paths(h2[0], h2[1],
h1[0], h1[1], dst_ip, src_ip) # reverse
            elif arp_pkt.opcode == arp.ARP_REQUEST:
                if dst_ip in self.arp_table:
                    self.arp_table[src_ip] = src
                    dst_mac =
self.arp_table[dst_ip]
                    h1 = self.hosts[src]
                    h2 = self.hosts[dst_mac]
                    out_port =
self.install_paths(h1[0], h1[1], h2[0], h2[1],
src_ip, dst_ip)
                    self.install_paths(h2[0],
h2[1], h1[0], h1[1], dst_ip, src_ip) # reverse

        actions =
[parser.OFPActionOutput(out_port)]

        data = None
        if msg.buffer_id == ofproto.OFP_NO_BUFFER:
            data = msg.data

        out = parser.OFPPacketOut(
            datapath=datapath,
buffer_id=msg.buffer_id, in_port=in_port,
            actions=actions, data=data)
```

```
        datapath.send_msg(out)

    @set_ev_cls(event.EventSwitchEnter)
    def switch_enter_handler(self, ev):
        switch = ev.switch.dp
        ofp_parser = switch.ofproto_parser

        if switch.id not in self.switches:
            self.switches.append(switch.id)
            self.datapath_list[switch.id] = switch

            req =
ofp_parser.OFPPortDescStatsRequest(switch)
            switch.send_msg(req)

    @set_ev_cls(event.EventSwitchLeave,
MAIN_DISPATCHER)
    def switch_leave_handler(self, ev):
        switch = ev.switch.dp.id
        if switch in self.switches:
            self.switches.remove(switch)
            del self.datapath_list[switch]
            del self.adjacency[switch]

    @set_ev_cls(event.EventLinkAdd,
MAIN_DISPATCHER)
    def link_add_handler(self, ev):
        s1 = ev.link.src
        s2 = ev.link.dst
        self.adjacency[s1.dpid][s2.dpid] =
s1.port_no
        self.adjacency[s2.dpid][s1.dpid] =
s2.port_no

    @set_ev_cls(event.EventLinkDelete,
MAIN_DISPATCHER)
    def link_delete_handler(self, ev):
        print("link deleted, calling func")
        s1 = ev.link.src
        s2 = ev.link.dst
        try:
            del self.adjacency[s1.dpid][s2.dpid]
```

```
            del self.adjacency[s2.dpid][s1.dpid]
        except KeyError:
            pass

        for i in self.paths:
            if s1.dpid in list(i[0][0].keys()) and
s2.dpid in list(i[0][0].keys()):
                self.install_paths(i[1], i[2],
i[3], i[4], i[5], i[6])

    def del_flow(self, datapath, dst):
        ofproto = datapath.ofproto
        parser = datapath.ofproto_parser

        match =
parser.OFPMatch(dl_dst=AddressValueError.mac.text_t
o_bin(dst))
        mod = parser.OFPFlowMod(
            datapath=datapath, match=match,
cookie=0,
            command=ofproto.OFPFC_DELETE)
        datapath.send_msg(mod)
```

LAMPIRAN B
SCRIPT TOPOLOGI

```
#!/usr/bin/python

from mininet.net import Mininet
from mininet.node import Controller,
RemoteController, OVSController
from mininet.node import CPULimitedHost, Host, Node
from mininet.node import OVSKernelSwitch,
UserSwitch
from mininet.node import IVSSwitch
from mininet.cli import CLI
from mininet.log import setLogLevel, info
from mininet.link import TCLink, Intf
from subprocess import call

def myNetwork():

    net = Mininet( topo=None,
                   build=False,
                   ipBase='10.0.0.0/8')

    info( '*** Adding controller\n' )
    c0=net.addController(name='c0',
                         controller=RemoteController,
                         ip='127.0.0.1',
                         protocol='tcp',
                         port=6633)

    info( '*** Add switches\n')
    s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
    s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
    s3 = net.addSwitch('s3', cls=OVSKernelSwitch)
    s4 = net.addSwitch('s4', cls=OVSKernelSwitch)
    s5 = net.addSwitch('s5', cls=OVSKernelSwitch)
    s6 = net.addSwitch('s6', cls=OVSKernelSwitch)
    s7 = net.addSwitch('s7', cls=OVSKernelSwitch)
    s8 = net.addSwitch('s8', cls=OVSKernelSwitch)
    s9 = net.addSwitch('s9', cls=OVSKernelSwitch)
```

```
    s10 = net.addSwitch('s10', cls=OVSKernelSwitch)
    s11 = net.addSwitch('s11', cls=OVSKernelSwitch)
    s12 = net.addSwitch('s12', cls=OVSKernelSwitch)
    s13 = net.addSwitch('s13', cls=OVSKernelSwitch)
    s14 = net.addSwitch('s14', cls=OVSKernelSwitch)
    s15 = net.addSwitch('s15', cls=OVSKernelSwitch)
    s16 = net.addSwitch('s16', cls=OVSKernelSwitch)
    s17 = net.addSwitch('s17', cls=OVSKernelSwitch)
    s18 = net.addSwitch('s18', cls=OVSKernelSwitch)
    s19 = net.addSwitch('s19', cls=OVSKernelSwitch)
    s20 = net.addSwitch('s20', cls=OVSKernelSwitch)

    info( '*** Add hosts\n')
    h1 = net.addHost('h1', cls=Host,
ip='10.0.0.1/8', defaultRoute=None)
    h2 = net.addHost('h2', cls=Host,
ip='10.0.0.2/8', defaultRoute=None)
    h3 = net.addHost('h3', cls=Host,
ip='10.0.0.3/8', defaultRoute=None)
    h4 = net.addHost('h4', cls=Host,
ip='10.0.0.4/8', defaultRoute=None)
    h5 = net.addHost('h5', cls=Host,
ip='10.0.0.5/8', defaultRoute=None)
    h6 = net.addHost('h6', cls=Host,
ip='10.0.0.6/8', defaultRoute=None)
    h7 = net.addHost('h7', cls=Host,
ip='10.0.0.7/8', defaultRoute=None)
    h8 = net.addHost('h8', cls=Host,
ip='10.0.0.8/8', defaultRoute=None)

    info( '*** Add links\n')
    net.addLink(s5, s1)
    net.addLink(s7, s1)
    net.addLink(s9, s1)
    net.addLink(s11, s1)
    net.addLink(s5, s2)
    net.addLink(s7, s2)
    net.addLink(s2, s9)
    net.addLink(s2, s11)
    net.addLink(s3, s6)
    net.addLink(s3, s8)
    net.addLink(s3, s10)
```

```
net.addLink(s3, s12)
net.addLink(s4, s6)
net.addLink(s4, s8)
net.addLink(s4, s10)
net.addLink(s4, s12)
net.addLink(s5, s13)
net.addLink(s6, s14)
net.addLink(s6, s13)
net.addLink(s5, s14)
net.addLink(s13, h1)
net.addLink(s7, s15)
net.addLink(s8, s16)
net.addLink(s8, s15)
net.addLink(s7, s16)
net.addLink(s9, s17)
net.addLink(s10, s18)
net.addLink(s10, s17)
net.addLink(s9, s18)
net.addLink(s11, s19)
net.addLink(s12, s20)
net.addLink(s12, s19)
net.addLink(s11, s20)
net.addLink(s14, h2)
net.addLink(s15, h3)
net.addLink(s16, h4)
net.addLink(s17, h5)
net.addLink(s18, h6)
net.addLink(s19, h7)
net.addLink(s20, h8)

info( '*** Starting network\n')
net.build()
info( '*** Starting controllers\n')
for controller in net.controllers:
    controller.start()

info( '*** Starting switches\n')
net.get('s15').start([c0])
net.get('s5').start([c0])
net.get('s7').start([c0])
net.get('s13').start([c0])
net.get('s2').start([c0])
```

```
    net.get('s19').start([c0])
    net.get('s14').start([c0])
    net.get('s16').start([c0])
    net.get('s4').start([c0])
    net.get('s17').start([c0])
    net.get('s1').start([c0])
    net.get('s6').start([c0])
    net.get('s10').start([c0])
    net.get('s8').start([c0])
    net.get('s12').start([c0])
    net.get('s20').start([c0])
    net.get('s11').start([c0])
    net.get('s18').start([c0])
    net.get('s3').start([c0])
    net.get('s9').start([c0])

    info( '*** Post configure switches and
hosts\n')

    CLI(net)
    net.stop()

if __name__ == '__main__':
    setLogLevel( 'info' )
    myNetwork()
```