TESIS - IF185401

PENGEMBANGAN MEKANISME *ROUTE DISCOVERY* UNTUK MENGHINDARI *CONGESTION* DENGAN MENGGUNAKAN ALGORITMA *JOHNSON* PADA *SOFTWARE DEFINED NETWORK*

AKBAR PANDU SEGARA

05111850010009

Dosen Pembimbing

Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.

Tohari Ahmad, S.Kom., MIT., Ph.D.

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

TESIS - IF185401

PENGEMBANGAN MEKANISME *ROUTE DISCOVERY* UNTUK MENGHINDARI *CONGESTION* DENGAN MENGGUNAKAN ALGORITMA *JOHNSON* PADA *SOFTWARE DEFINED NETWORK*

AKBAR PANDU SEGARA

05111850010009

Dosen Pembimbing

Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.

Tohari Ahmad, S.Kom., MIT., Ph.D.

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

LEMBAR PENGESAHAN TESIS

Tesis disusun untuk memenuhi salah satu syarat memperoleh gelar
Magister Komputer (M. Kom)
di
Institut Teknologi Sepuluh Nopember
Oleh:
AKBAR PANDU SEGARA
NRP: 05111850010009

Tanggal Ujian: 21 Juli 2020
Periode Wisuda: September 2020

Disetujui oleh:
Pembimbing:

1. Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D. ……
   NIP. 19770824 200604 1 001
2. Tohari Ahmad, S.Kom, MIT., Ph.D. ……
   NIP. 19750525 200312 1 002

Penguji:

1. Waskitho Wibisono, S.Kom, M. Eng, Ph.D ……
   NIP. 19741022 200003 1 001
2. Dr.Eng. Radityo Anggoro, S.Kom., M.Sc. ……
   NIP. 19841016 200812 1 002
3. Ary Mazharuddin Shiddiqi, S.Kom., M.Comp.Sc., Ph.D. ……
   NIP. 19810620 200501 1 003

Kepala Departemen Teknik Informatika
Fakultas Teknologi Elektro dan Informatika Cerdas

Dr. Eng. Chastine Fatichah, S.Kom., M.Kom
NIP. 19751220 200112 2 002

PENGEMBANGAN MEKANISME *ROUTE DISCOVERY* UNTUK MENGHINDARI *CONGESTION* MENGGUNAKAN ALGORITMA *JOHNSON* PADA *SOFTWARE DEFINED NETWORK*

Nama : Akbar Pandu Segara
NRP : 05111850010009
Pembimbing : Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D.
Tohari Ahmad, S.Kom, MIT., Ph.D.

ABSTRAK

Software Defined Network (SDN) merupakan paradigma baru arsitektur jaringan yang menawarkan konsep *control plane* yang terpisah dengan *data plane*. Pada *control plane* diimplementasikan sebuah *controller* yang secara terpusat berfungsi untuk melakukan *control* terhadap perilaku jaringan komputer. Sedangkan perangkat jaringan seperti *switch* atau *router* pada sisi *data plane* menjadi perangkat untuk *forwarding* paket. Dengan model *control plane* terpusat SDN sangat rentan akan terjadinya *congestion* dikarenakan model komunikasi *one-to-many*. Salah satu cara untuk menghindari kongesti adalah dengan menentukan *routing* paket menggunakan suatu algoritma *routing* tertentu.

Penelitian ini bertujuan untuk mengembangkan metode *route discovery* saat untuk menghindari *congestion* pada pada SDN. Dengan menggunakan Algoritma *Johnson* untuk mencari rute alternatif dengan *cost* yang paling minim saat terjadi *congestion*. Dimana algoritma *Johnson* memiliki sifat *all-pair shortest path* dengan membandingkan seluruh pasang *node* pada topologi untuk menentukan rute terpendek. Karena algoritma *Johnson* merupakan penggabungan 2 algoritma *shortest path*, hal ini dapat mempercepat *response time* saat pertama kali algoritma melakukan pencarian rute.

Berdasarkan hasil uji coba, pada skenario topologi *mesh* mengalami penurunan *latency* 0.459 ms menjadi 0.406 ms dan pada topologi *fat-tree* mengalami penurunan 0.781 ms menjadi 0.236 ms setelah diimplentasikannya metode. Hasil uji coba *throughput* skenario topologi *mesh* mengalami kenaikan 8.4% sedangkan pada topologi *fat-tree* mengalami peningkatan 11.6%. Pada uji coba response time pada kedua buah skenario menunjukkan algoritma johnson memiliki kecepetan yang baik dibanding hasil response time algoritma OSPF *Dijkstra*. Hal ini dikarenakan sifat *all-pair shortest path* yang membandingkan seluruh pasang node untuk menentukan rute terpendek sedangkan *Dijkstra* adalah

termasuk algoritma greedy dimana algoritma ini akan mencari nilai sementara antar node pada setiap langkahnya.

Kata kunci: *johnson, routing, shortest path, SDN, congestion avoidance*

DEVELOPMENT OF ROUTE DISCOVERY MECHANISMS FOR AVOID A CONGESTION USING JOHNSON'S ALGORITHM IN SOFTWARE DEFINED NETWORKS

Name	: Akbar Pandu Segara
Student Identity Number	: 05111850010009
Supervisor	: Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D. Tohari Ahmad, S.Kom, MIT., Ph.D.

ABSTRACT

Software Defined Network (SDN) is a new paradigm of network architecture that offers a separate control plane concept with a data plane. In the control plane a controller is centrally implemented which functions to control the behavior of a computer network. While network devices such as switches or routers on the data plane side become devices for packet forwarding. With a centralized control plane model SDN is very vulnerable to congestion due to the one-to-many communication model. One way to avoid congestion is to determine packet routing using a particular routing algorithm.

This study aims to develop a method of route discovery in SDN. By using Johnson's Algorithm to find alternative routes with the least cost when there is congestion. Where Johnson's algorithm has the nature of all-pair shortest path by comparing all node pairs in the topology to determine the shortest route. Because the Johnson algorithm is a combination of two shortest path algorithms, this can speed up response time the first time the algorithm does a route search.

Based on the test results, in the mesh topology skenario the latency decreased 0.459 ms to 0.406 ms and the *fat-tree* topology decreased from 0.781 ms to 0.236 ms after the method implementation. Result of throughput in the mesh topology increased by 8.4% while in the *fat-tree* topology increased by 11.6%. The response time trials in both skenarios show that the Johnson algorithm has a good response time comparing the results of the *Dijkstra* OSPF Algorithm. This is due to the nature of the all-pair shortest path that compares all pairs of nodes to determine the shortest route while *Dijkstra* is a greedy algorithm where this algorithm will look for temporary values between nodes at each step.

Keywords: *johnson, routing, shortest path, SDN, congestion avoidance*

(Halaman ini sengaja dikosongkan)

KATA PENGANTAR

Segala puji bagi Allah SWT yang telah melimpahkan rahmat beserta hidayah-Nya sehingga laporan tesis yang berjudul "Pengembangan Mekanisme Route Discovery Untuk Menghindari *Congestion* Menggunakan Algoritma *Johnson* Pada *Software Defined Network*" dapat diselesaikan dengan baik dan sebagaimana mestinya. Penulis menyadari bahwa penyusunan tesis ini tidak terlepas dari bantuan dari berbagai pihak. Oleh karena itu, penulis ingin menyampaikan rasa hormat dan terima kasih kepada:

1. Allah SWT dan Nabi Agung Muhammad SAW karena atas kehendak, rahmat dan nikmat-Nya laporan tesis ini telah selesai dengan baik.
2. Alm. Supriyanto selaku Ayah dari penulis yang telah memberi motivasi semasa hidup agar penulis dapat melanjutkan studi ke jenjang yang lebih tinggi dan Dra. Titik Zahroh selaku Ibu penulis yang telah memberikan doa restu serta motivasi dan dukungannya selama ini.
3. Elsea Pricilia Roselawati selaku adik penulis yang telah memberikan semangat sehingga penulis merasa sangat terbantu dalam menyelesaikan tesis.
4. Bapak Royyana Muslim Ijtihadie, S.Kom., M.Kom., Ph.D. dan Tohari Ahmad, S.Kom, MIT., Ph.D. selaku dosen pembimbing yang telah memberikan pengarahan dan bimbingan kepada penulis sehingga dapat menyelesaikan tesis ini.
5. Bapak Dr. Ahmad Saikhu, S.Si., M.T., selaku ketua program studi pascasarjana departemen Informatika ITS dan Ibu Dr.Eng. Chastine Fatichah, S.Kom, M.Kom selaku Kepala Departemen Teknik Informatika Fakultas Teknik Elektro dan Informatika Cerdas.
6. Seluruh civitas akademika pascasarjana Departemen Informatika ITS yang telah memberikan bantuan selama penulis menempuh studi dan penyelesaian Tesis.
7. Tim Tesis SDN dan Seluruh teman-teman Pascasarjana Informatika ITS angkatan 2018 dan pihak yang telah memberikan dukungan dan bantuan yang tidak dapat penulis sebutkan satu per satu.

Penulis menyadari bahwa Tesis ini masih banyak kekurangan, oleh karena itu untuk segala kritik dan saran yang membangun penulis ucapkan terima kasih.

Surabaya, 20 Juli 2020

Akbar Pandu Segara

(Halaman ini sengaja dikosongkan)

DAFTAR ISI

LEMBAR PENGESAHAN TESIS ..... **i**
ABSTRAK ..... **ii**
ABSTRACT ..... **iv**
KATA PENGANTAR ..... **ii**
DAFTAR ISI ..... **iv**
DAFTAR GAMBAR ..... **vi**
DAFTAR TABEL ..... **vi**
BAB 1 PENDAHULUAN **1**
1. 1. Latar Belakang 1
1. 2. Rumusan Masalah 4
1. 3. Tujuan Penelitian 4
1. 4. Manfaat Penelitian 5
1. 5. Kontribusi Penelitian 5
1. 6. Batasan Masalah 5
BAB 2 KAJIAN PUSTAKA **7**
2. 1. *Software Defined Network* 7
2. 2. *OpenFlow* 8
2. 3. *Ryu Controller* 10
2. 4. *Shortest Path Algorithm* 11
2. 5. Algoritma *Johnson* 11
2. 6. *Congestion Avoidance* 13
BAB 3 METODOLOGI PENELITIAN **17**
3. 1. Studi Literatur 17
3. 2. Metode yang Diusulkan 17
3.2.1 Perancangan Sistem 18
3.2.2 Perancangan Topologi 20
3.2.3 Perancangan Mekanisme *Route Discovery* 22
3.2.4 Perancangan penghindaran *congestion* dengan Routing 23
3. 3. Pengujian 25

3.3.1 Skenario pengujian 25
3.3.2 Parameter Evaluasi 27
3. 4. Analisa Hasil Pengujian 28
3. 5. Jadwal Penelitian 29
BAB 4 HASIL DAN PEMBAHASAN 31
4.1 Hasil Uji Coba Topologi *Mesh* 31
4.1.1 Uji Coba *Route Discovery* Topologi *Mesh* 31
4.1.2 Uji Coba *Latency* Topologi Mesh 33
4.1.3 Uji Coba *Throughput* Topologi *Mesh* 36
4.1.4 Uji Coba Response Time Topologi *Mesh* 39
4.2 Hasil Uji Coba Topologi *Fat-tree* 41
4.2.1 Uji Coba *Route Discovery Fat-tree* 42
4.2.2 Uji Coba *Latency Fat-tree* 44
4.2.3 Uji Coba *Throughput Fat-tree* 47
4.2.4 Uji Coba Response Time *Fat-tree* 50
BAB 5 KESIMPULAN DAN SARAN 53
5.1 Kesimpulan 53
5.2 Saran 54
DAFTAR PUSTAKA 55
BIODATA PENULIS 59

DAFTAR GAMBAR

Gambar 2.1 Arsitektur SDN 7
Gambar 2.2 Arsitektur OpenFlow 9
Gambar 2.3 Arsitektur Ryu 10
Gambar 2.4 Algoritma Johnson [23] 12
Gambar 3.1 Alur Metodologi Penelitian 17
Gambar 3.2 Flowchart metode penelitian 17
Gambar 3.3 Flowchart tahap perancangan sistem 19
Gambar 3.4 Topologi Mesh 20
Gambar 3.5 Topologi jaringan *Fat-tree* 21
Gambar 3.6 Flowchart Routing Discovery 23
Gambar 3.7 Flowchart Metode yang diusulkan 24
Gambar 3.8 Antarmuka minnet 26
Gambar 4.1 Topologi *Mesh* pada mininet 31
Gambar 4.2 Available path pada Topologi Mesh 32
Gambar 4.3 Grafik perbandingan *Latency* topologi *Mesh* 34
Gambar 4.4 Grafik perbandingan *Latency* dua Algoritma 35
Gambar 4.5 Perbandingan TCP Throughput pada Topologi Mesh 37
Gambar 4.6 Perbandingan *Throughput* dua buah Algoritma 38
Gambar 4.7 Uji coba *response time* Topologi *mesh* 39
Gambar 4.8 Grafik *response time* Topologi *mesh* 40
Gambar 4.9 Skenario uji coba topologi *fat-tree* 41
Gambar 4.10 *Fat-tree Available path pada Ryu controller* 42
Gambar 4.11 Alternate route pada topologi *fat-tree* 44
Gambar 4.12 Grafik perbandingan Latency topologi *Fat-tree* 46
Gambar 4.13 Grafik perbandingan *Latency* dua Algoritma 46
Gambar 4.14 Perbandingan Throughput Topologi *Fat-tree* 49
Gambar 4.14 Perbandingan *Throughput* antar Algoritma 49
Gambar 4.16 Uji coba *response time* Topologi *Fat-tree* 50
Gambar 4.17 Grafik *response time* Topologi *Fat-tree* 52

DAFTAR TABEL

Tabel 3.1 Lingkungan Simulasi 18
Tabel 3.2 Fungsi komponen mininet 25
Tabel 3.3 Jadwal Kegiatan Penelitian 29
Tabel 4.1 Available route pada topologi mesh 32
Tabel 4.2 Hasil Uji Coba Latency Sebelum Routing pada Topologi Mesh 33
Tabel 4.3 Hasil Uji Coba Latency Setelah Routing pada Topologi Mesh 34
Tabel 4.4 TCP Throughput Sebelum Routing pada Topologi Mesh 36
Tabel 4.5 TCP Throughput Setelah Routing padaTopologi Mesh 37
Tabel 4.6 Response Time pada Topologi Mesh 39
Tabel 4.7 Available route pada topologi *fat-tree* 43
Tabel 4.8 Hasil Uji Coba Latency Sebelum Routing pada Topologi *Fat-tree* 45
Tabel 4.9 Hasil Uji Coba Latency Setelah Routing pada Topologi *Fat-tree* 45
Tabel 4.10 TCP Throughput Sebelum CA Topologi *Fat-tree* 47
Tabel 4.11 TCP Throughput Sebelum CA Topologi *Fat-tree* 48
Tabel 4.12 Response Time Topologi *Fat-tree* 51

BAB 1
PENDAHULUAN

Bab ini merupakan penjelasan mengenai beberapa hal dasar dalam pembuatan proposal penelitian. Hal-hal yang dimaksud meliputi latar belakang, perumusan masalah, tujuan penelitian, manfaat penelitian, kontribusi penelitian, serta batasan masalah.

1. 1. Latar Belakang

Perkembangan teknologi jaringan komputer yang semakin maju berdampak pada kebutuhan yang tinggi terhadap kinerja jaringan dan kontrol jaringan yang semakin kompleks. Konfigurasi jaringan, jumlah permintaan, dan metode manajemen jaringan merupakan faktor yang dapat mempengaruhi performansi suatu jaringan. Protokol kontrol terdistribusi dan *transport network* yang berjalan pada *router* dan *switch* yang merupakan komponen utama untuk pertukaran informasi dalam bentuk paket digital. Pada jaringan konvensional konfigurasi perangkat jaringan dengan menggabungkan beberapa vendor yang berbeda dalam satu jaringan membutuhkan usaha tambahan. Selain kompleksitas konfigurasi, lingkungan jaringan harus menyesuaikan diri terhadap perubahan jaringan yang terjadi. Mekanisme rekonfigurasi dan respon otomatis hampir tidak ada dalam jaringan konvensional. Hal ini tentu tidak dapat memenuhi tuntutan operasional saat ini yang memiliki jaringan besar dan perangkat jaringan yang memiliki spesifikasi berbeda [1].

Menurut T. D. Nadeau dan K. Gray [2] pada tahun 2013, *Software Defined Network* (SDN) merupakan paradigma baru arsitektur jaringan yang menawarkan konsep *control plane* yang dipisahkan dari *data plane*. Pemisahan ini mendefinisikan *control plane* akan diletakkan secara terpusat pada sebuah *controller* yang bertugas untuk mengontrol perilaku suatu jaringan komputer dan perangkat jaringan (*switch* atau *router*) yang berada pada *data plane* menjadi perangkat untuk *forwarding* paket data. *Network administrator* dapat mengotomatisasi, memanajemen, dan mengoperasikan jaringan dengan *controller*

agar dapat berkomunikasi dengan perangkat dari vendor yang berbeda. Dengan model *control plane* terpusat SDN sangat rentan akan terjadinya *congestion* dikarenakan model komunikasi *one-to-many* yang rentan terjadi masalah TCP *incast*. *TCP incast* biasanya ditemukan pada *data center* yang menerapkan penyimpanan terdistribusi dan komputasi *framework* dengan pola komunikasi *one-to-many* [3].

Menurut Michael Welzl [4], suatu jaringan dikatakan mengalami *congestion* dari perspektif pengguna jika kualitas layanan dirasakan oleh pengguna menurun karena kenaikan beban jaringan. *Congestion* adalah situasi yang muncul ketika sejumlah besar paket masuk ke lokasi tertentu dari jaringan atau ketika *resource* sistem yang dimiliki jauh lebih sedikit daripada *load* yang dihasilkan. Hal ini cenderung menurunkan kinerja seluruh jaringan, mulai dari *throughput* yang rendah hingga terjadinya *delay*. Salah satu dampak dari *congestion* adalah terjadinya peningkatan jumlah *packet loss*. Jumlah *packet loss* yang tinggi mengakibatkan node melakukan retransmisi data dan menyebabkan penurunan *packet delivery ratio* (PDR). Begitu juga saat paket yang sampai pada node tujuan akan terjadi *delay* dan menyebabkan *end-to-end delay* yang meningkat dan juga mengakibatkan penurunan *throughput*. *Congestion* bukan hanya masalah yang sering terjadi dalam sistem jaringan konvensional tetapi dapat menurunkan kinerja pada *Software Defined Networks*.

Pada survey [5], SDN sendiri terdapat beberapa mekanisme *congestion control*, salah satunya yang diusulkan oleh Yifei Lu dan S. Zhu [6] pada tahun 2015, mekanisme deteksi *congestion* pada jaringan dengan mengurangi ukuran pengiriman paket dari *source* setelah terjadinya congestion. Akan tetapi, hal tersebut dirasa mengakibatkan waktu pengiriman akan menjadi lebih lama karena jumlah paket yang dikirim lebih sedikit. Hal ini mengharuskan administrator jaringan harus dapat mengkonfigurasi sebuah jaringan dengan protokol *routing* tertentu untuk menghindari *congestion*. Protokol *routing* juga menjadi salah satu faktor yang mempengaruhi manajemen jaringan. Protokol *routing* digunakan suatu node untuk terhubung ke node lainnya dengan memberikan informasi *route discovery* dengan algoritma *routing* tertentu [7].

Membangun protokol *routing* yang baik diperlukan algoritma *routing* yang dapat menentukan rute terpendek (*shortest path*). SPA (*Shortest Path Algorithm*) digunakan untuk menemukan rute antara dua buah node pada sebuah *graph* sehingga diperoleh bobot *edge* yang minimum. Terdapat beberapa jenis SPA yang umum digunakan dalam mencari rute terpendek yaitu *Algoritma Djikstra, Bellman-Ford, Floyd-Warshall*, dan *Johnson*. Saat ini, layanan IP menggunakan *Open Shortest Path First* (OSPF) dengan perhitungan yang didasarkan pada algoritma *Dijkstra* yang menghitung *shortest path* dalam jaringan. Namun disatu sisi Algoritma *Dijkstra* merupakan algoritma *greedy* dimana algoritma ini akan mencari nilai sementara antar *node* pada setiap langkahnya. Algoritma ini sering digunakan pada arsitektur jaringan yang sederhana dan tidak terlalu rumit [8].

Dalam perkembangannya, SPA banyak diimplementasikan pada skenario pencarian rute (*route discovery*) terpendek pada *Software Defined Network.* Salah satunya yaitu algoritma *Johnson*, dengan karakteristik *all-pair shortest path* yang dapat diterapkan untuk menentukan melalui rute mana paket akan diteruskan. Algoritma *Johnson* merupakan penggabungan dua buah algoritma *Single-Source Shortest Paths*, yaitu Algoritma *Bellman-Ford* dan Algoritma *Dijkstra* [9]. Dua buah algoritma ini menentukan rute terpendek dengan cara yang berbeda, algoritma *Dijkstra* menentukan rute terpendek berdasarkan jarak antar node sedangkann Algoritma *Bellman-Ford* perhitungannya berdasarkan jumlah hop [10]. Karena merupakan penggabungan dari 2 algoritma *shortest path*, Algoritma *Johnson* memiliki keunggulan yaitu dengan sifat *all-pair shortest path* akan mencari *shortest path* berdasarkan semua pasangan *node* sehingga *runtime* akan lebih cepat [11].

Penelitian ini bertujuan untuk mengembangkan mekanisme *route discovery* untuk menghindari *congestion* pada *Software Defined Network.* Hal ini dirasa perlu karena pada SDN merupakan arsitektur jaringan dengan komunikasi *one-to-many* yang sangat rentan terhadap *congestion*. *Congestion* dapat menurunkan performa jaringan jika tidak ditangani dengan segera maka akan meningkatkan *delay* hingga turunnya *throughput* yang akan berdampak pada proses pengiriman paket. Oleh karena itu, algoritma *routing Johnson* dipilih sebagai usulan karena dirasa dapat melakukan pencarian rute dengan waktu eksekusi yang lebih cepat. Penelitian ini

juga melakukan modifikasi pada metode rute *discovery* dengan menggunakan kemampuan dari algoritma *Johnson* untuk menentukan rute lain dengan *cost minimum* jika *latency* yang terjadi pada rute utama mengalami ketidakstabilan dan kenaikan latensi. *Data plane* akan memberikan informasi pada *controller* terkait *traffic* yang terjadi pada *Openflow switch*. Kemudian *controller* akan merespon informasi dengan memilihan rute alternatif disimpan pada *flow* tabel. Proses diteruskan dengan pengecekan paket, jika paket melewati rute yang sedang terjadi *congestion* maka *controller* akan mendeteksi adanya *delay* yang berisi informasi bahwa pada rute yang dilalui sedang padat dan paket akan dilewatkan melalui rute alternatif terpendek yang lalu lintas datanya tidak padat. Modifikasi algoritma *Johnson* akan diimplementasikan pada sisi *control plane* dengan basis *remote controller Ryu* menggunakan bahasa pemrograman *python*. Dan memanfaatkan kemampuan *OpenFlow switch* dalam menyaring informasi paket yang masuk pada sisi *data plane* sehingga dapat dengan cepat mendapatkan respon jika terjadi perubahan rute. Dengan demikian tidak ada pengurangan ukuran paket dan diharapkan dapat mengurangi *latency* dan mempertahankannya seminimal mungkin.

1. 2. Rumusan Masalah

Rumusan masalah pada penelitian ini adalah sebagai berikut:

1. Bagaimana penerapan *route discovery* pada *control plane Software Defined Network*?
2. Bagaimana perancangan *route discovery* untuk menghindari *congestion* menggunakan algoritma *Johnson* pada *Software Defined Network*?
3. Bagaimana hasil evaluasi mekanisme *route discovery* menggunakan Algoritma *Johnson* dengan pada *Software Defined Network*?

1. 3. Tujuan Penelitian

Tujuan dari penelitian ini adalah sebagai berikut:

1. Menerapkan *route discovery* pada *control plane* pada *Software Defined Network*.

2. Merancang *route discovery* untuk menghindari *congestion* menggunakan algoritma *Johnson* pada *Software Defined Network.*
3. Mengetahui hasil evaluasi mekanisme *route discovery* menggunakan Algoritma *Johnson* dengan pada *Software Defined Network.*

1. 4. Manfaat Penelitian

Manfaat dari penelitian ini diharapkan dapat meningkatkan performansi dalam menghindari kongesti dengan menggunakan algoritma *routing Johnson* untuk mengurangi *latency* dan meningkatkan *throughput* pada *Software Defined Network*.

1. 5. Kontribusi Penelitian

Kontribusi dari penelitian ini adalah mengusulkan mekanisme *route discovery* dengan menambahkan parameter *latency* untuk menentukan rute alternatif dengan *hop minimum* menggunakan algoritma *Johnson* tanpa mengurangi ukuran paket yang dikirimkan pada *Software Defined Network.*

1. 6. Batasan Masalah

Penelitian ini memiliki beberapa batasan antara lain:

1. Algoritma *routing* yang digunakan pada penelitian ini adalah Algoritma *Johnson.*
2. Implementasi dan pengujian dilakukan menggunakan simulator jaringan virtual *mininet*.
3. *Controller* yang digunakan menggunakan *Ryu*.
4. Topologi yang digunakan adalah Topologi *Partially Connected Mesh* dan *Fat-tree*.
5. Parameter yang diuji yaitu *route discovery, latency, throughput,* dan *response time*.

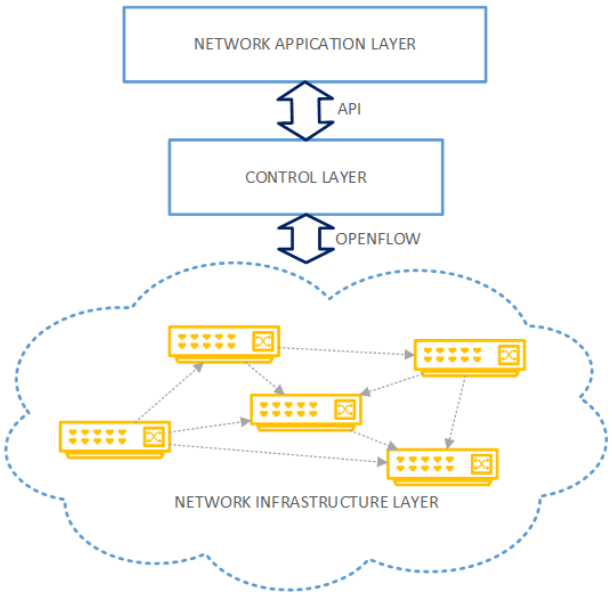
(Halaman ini sengaja dikosongkan)

BAB 2
KAJIAN PUSTAKA

Pada Bab ini akan dijelaskan mengenai referensi terkait penyelesaian masalah dalam penelitian. Beberapa hal yang dijelaskan pada Bab ini yaitu tentang *Software Defined Network*, kemudian dibahas tentang definisi *Software Defined Network*, algoritma *routing* yang digunakan serta mekanisme *congestion avoidance* yang dipakai dalam SDN.

2. 1. *Software Defined Network*

Paradigma *Software Defined Network* dibuat untuk mengatasi keterbatasan suatu infrastruktur jaringan. Dengan konsep memisahkan *network control logic* dari *router* dan *switch* yang meneruskan *traffic.* Dengan memisahkan *control plane* dan *data plane*, *switch* menjadi perangkat untuk meneruskan paket, dan dilakukan *control* secara terpusat pada *network control logic.* Hal ini akan menyederhanakan konfigurasi pada sebuah jaringan dengan banyak perangkat jaringan dari vendor yang berbeda [12]. Untuk penggambaran arsitektur sederhana dari *Software Defined Network* sebagai berikut.



Gambar 2.1 Arsitektur SDN

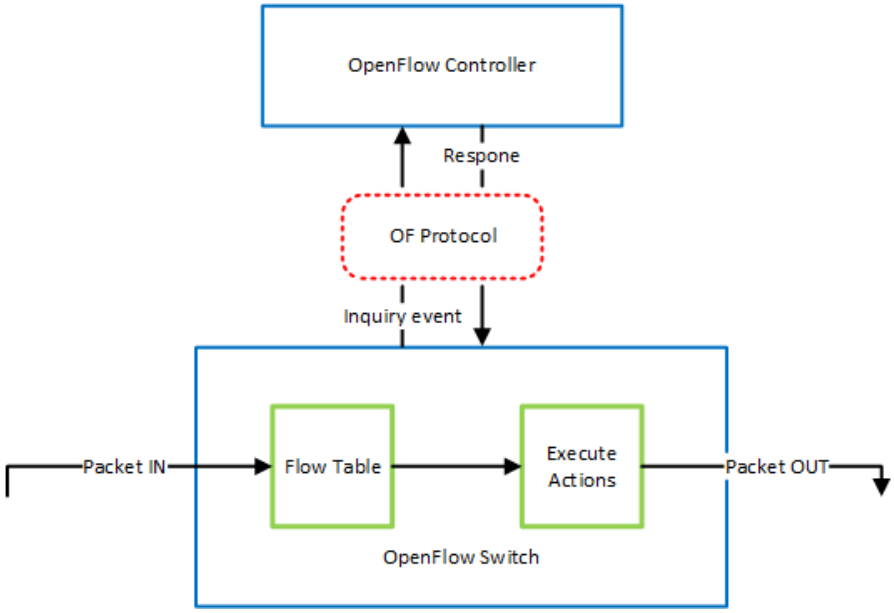
Gambar diatas merupakan arsitektur sederhana dari *Software Defined Network*. Arsitektur SDN tersusun dari tiga layer yang terdiri dari *application layer*, *control layer* dan *infrastructure layer*. *Application layer* berada pada lapisan paling atas, layer ini merupakan letak aplikasi SDN yang diprogram untuk berkomunikasi dengan *controller* SDN. Pada *control layer* terdapat sebuah SDN *controller* yang berfungsi menerima intruksi dari *layer application* dan secara *centralized* mengkontrol sebuah infrastruktur jaringan baik yang sifatnya *centralized* maupun *decentralized* secara fisik. Sedangkan *infrastructure layer* adalah sekumpulan perangkat jaringan yang berfungsi untuk *forwarding* data. Dengan dipisahnya *control plane* dan *data plane* maka *switch* dan *controller* SDN dapat diimplementasikan antarmuka pemrogaman. *Data plane* dapat dikontrol langsung oleh *controller* melalui *application programming interface* (API) [13][14].

2. 2. *OpenFlow*

Pada arsitektur SDN, *open interface* dapat memprogram sebuah *software* berbasis *controller* (*control plane*) dan *network device* menjadi perangkat yang berfungsi untuk meneruskan paket (*data plane*) secara terpusat (*centralized*). Protokol SDN yang umum dipakai adalah protokol *OpenFlow* namun terdapat juga protokol lain yaitu BGP (*Border Gateway Protocol*), OVSDB (*Open vSwitch Database Management Protocol*), XMPP (*Extensible Messaging and Presence Protocol*), NETCONF, MPLS-TP (*Multiprotocol Label Switching-Transport profile*). Untuk menguji dan menerapkan fungsi baru, *OpenFlow* mendukung pengguna untuk mengembangkan fungsi baru terhadap struktur network saat ini melalui sebuah *software*. *OpenFlow* mempunyai performansi keamanan yang bagus, manajemen yang mudah, *OpenFlow* juga berlaku untuk memanajemen jaringan virtual, menyederhanakan struktur jaringan, mengurangi biaya operasi dan konstruksi jaringan [15].

OpenFlow adalah salah satu implementasi dari *open interface*. *OpenFlow* memungkinkan akses langsung untuk memanipulasi *forwarding plane* pada perangkat jaringan seperti *switch* dan *router*. Protokol *OpenFlow* melakukan mekanisme komunikasi terhadap *switch* menggunakan *OpenFlow*. Secara

centralized, *OpenFlow* mengkontrol kerumitan jaringan ke dalam sebuah *controller software*, sehingga administrator dapat melakukan pengaturan jaringan melalui *controller* dengan mudah [16]. Arsitektur *OpenFlow* dapat dilihat pada gambar dibawah ini.

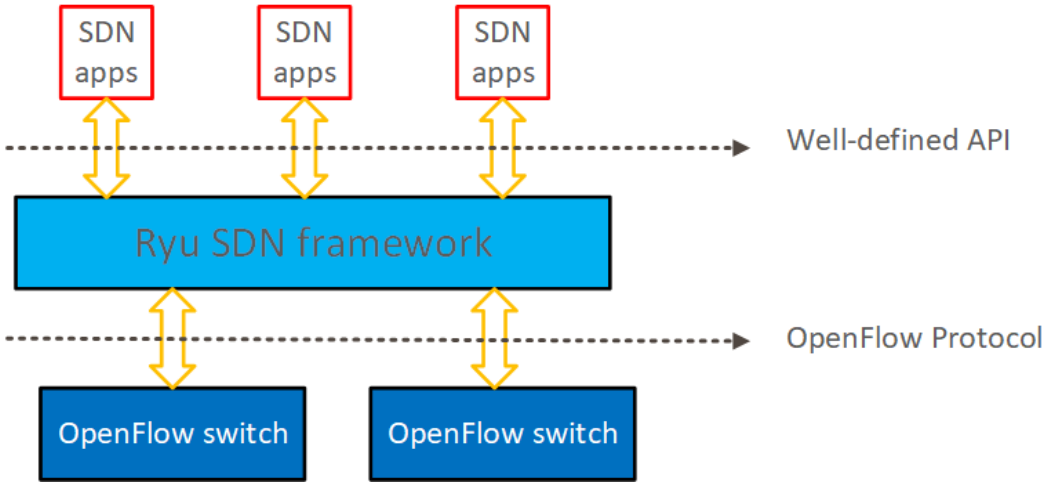


Gambar 2.2 Arsitektur OpenFlow

Secara umum arsitektur *Openflow* terdiri dari bagian, antara lain *Openflow Controller*, Protokol, dan *Switch* [17]. Protokol *Openflow* bertindak sebagai perantara *controller* dan *switch* berkomunikasi secara aman. *Controller* bertindak untuk mengatur perilaku data *forwarding* pada setiap *forwarding device* dengan memberikan rules melalui protokol *OpenFlow*. *Flow* tabel pada *switch* bertindak sebagai penyimpan data *flow entry*. Jadi ketika data sampai pada *OpenFlow switch*, maka *switch* akan mencari *flow entry* yang cocok pada *flow* tabel jika sesuai kemudian akan dieksekusi. Jika tidak sesuai akan dilakukan pemeriksaan kembali dengan mengirimkan sebuah *event* pada *controller* untuk kemudian merespon dengan *flow entry* yang baru untuk menangani antrian paket. Paket *header OpenFlow* berisikan informasi tentang *source* MAC, *destination* MAC, *source* IP, *destination* IP, dan TCP *port* yang akan menentukan kemana arah paket akan diteruskan [18].

2. 3. *Ryu Controller*

Controller merupakan bagian dari SDN yang berfungsi untuk mengontrol logika pada jaringan. Seluruh kebijakan jaringan seperti *routing*, *switching* maupun pengaturan jaringan lainnya diatur oleh *controller*. *Controller* di desain untuk meningkatkan kemampuan jaringan agar dapat secara mudah dilakukan manajemen jaringan dan beradaptasi pada *traffic* jaringan. Pengaturan fungsional tersebut dikembangkan melalui modul-modul pada *controller* yang dapat diprogram menggunakan bahasa pemrograman tertentu. Tiap *controller* menggunakan bahasa pemrogramannya masing-masing. SDN memiliki beberapa *controller* diantaranya *NOX*, *POX*, *OpenDayLight*, *FloodLight*, *Ryu*, *Pyretic*, dan lain-lain. Gambar dari arsitektur Ryu controller adalah sebagai berikut.



Gambar 2.3 Arsitektur Ryu

Ryu merupakan framework berbasis komponen *software defined network* yang menyediakan komponen *software* dengan API yang seluruh kode programnya dapat digunakan secara terbuka (*open access*) yang membuatnya mudah untuk mengembangkan dan mengelola aplikasi. *Ryu* mendukung berbagai protokol untuk mengelola perangkat jaringan, seperti OpenFLow, NetConf, OF-Config dan sebagainya. Ryu berinteraksi langsung dengan perangkat *forwarding* seperti *switch* maupun *router* untuk menghandel bagaimana jaringan mengatasi lalu lintas data flow [19].

2. 4. *Shortest Path Algorithm*

Algoritma *routing* muncul dari perkembangan *dynamic routing*. *Algoritma routing* merupakan metode untuk menentukan rute paket pada *node*. Algoritma *routing* merupakan faktor utama yang mempengaruhi kinerja *routing* pada suatu jaringan. Tujuan dari algoritma *routing* adalah menentukan keputusan *router* untuk menentukan rute terbaik yang akan diambil dari satu *node* ke *node* lainnya. Sehingga ketika terdapat rute yang rusak maka algoritma *routing* akan menentukan rute alternatif yang akan digunakan [20].

Untuk membangun protokol *routing* yang baik diperlukan algoritma *routing* yang dapat menentukan rute terpendek (*shortest path*) pada berbagai macam topologi tanpa melakukan konfigurasi ulang. SPA (*Shortest Path Algorithm*) digunakan untuk menemukan rute antara dua buah node pada sebuah *graph* sehingga diperoleh bobot *edge* yang minimum. Untuk kelebihan dari SPA sendiri diantranya untuk meminimalkan jarak menuju node tujuan dan untuk mengurangi *loss* pada distribusi *power flow* [21]. Terdapat duah buah jenis SPA diantaranya *minimum spanning tree* dan *graph teory*. Salah satu SPA jenis graph teory yang bersifat *all pair shortest path* merupakan Algoritma *Johnson,* yaitu algoritma yang digunakan dalam menentukan *shortest path* berdasarkan semua pasangan *node*, penentuan *shortest path* cepat dan performansinya stabil.

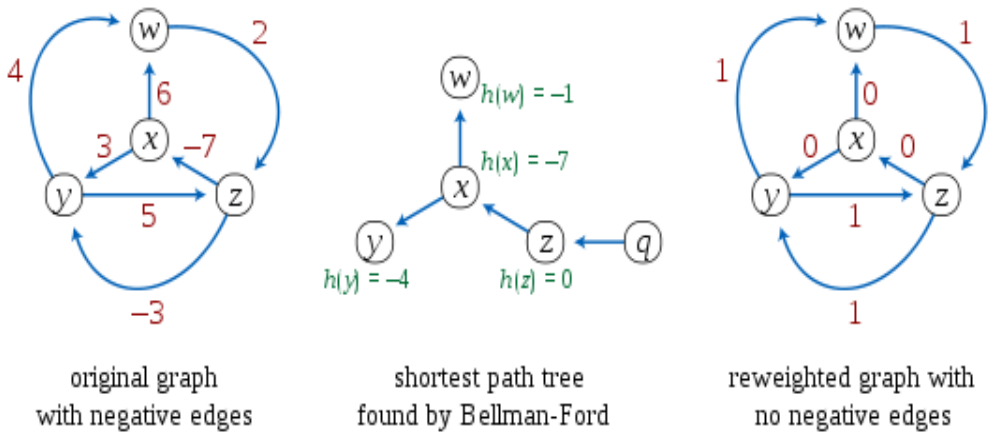
2. 5. Algoritma *Johnson*

Algoritma *Johnson* pertama kali di perkenalkan pada tahun 1977 dimana nama algoritma ini diambil dari pencetusnya sendiri yaitu Donald B. *Johnson* [22]. Algoritma *Johnson* merupakan penggabungan dari Algoritma *Bellman-Ford* dan *Djikstra*. Algoritma *Bellman-Ford* digunakan untuk mengatur ulang input *graph* untuk menghilangkan bobot negatif pada tiap *edge* dan mendeteksi siklus negatif dari *graph*. Kemudian dari *graph* baru yang telah dibuat oleh algoritma *Bellman-Ford.*

Kemudian Algoritma *Djikstra* akan menghitung rute dengan cost minimum dengan membandingkan seluruh pasangan simpul *node*. Algoritma *djikstra* dapat juga dikatakan sebagai algoritma *greedy* dengan mencari nilai minimum pada tiap

edge pada suatu graph. Algoritma *Dijkstra* mencari rute dengan *cost* paling minimum antara titik satu dengan yang lain. Selain itu, algoritma ini juga menghitung total *cost* yang dari jalur lintasan terpendek. Perhitungan cost pada masing-masing edge dengan $\frac{10^8}{Bandwidth}$. Output dari algoritma penggabungan dua algoritma ini berupa himpunan dari seluruh rute terpendek dari original *graph* [8].

Dalam Algoritma *Johnson* terdapat beberapa langkah untuk menghitung jalur terpendek yang dapat dilihat pada gambar 2.4 dibawah yaitu dengan menambahkan sebuah node baru *q* dengan nilai 0 pada sebuah *graph* dengan bobot negatif. *Node* baru *q* dengan jarak *source node* adalah 0 dikarenakan node q diasumsikan sebagai *source node* baru. Kemudian algoritma *Bellman-Ford* akan menghitung bobot minimum dari graph dengan node *q* sebagai titik awal. Setelah bobot minimum dihitung maka akan mendapatkan bobot *graph* yang telah diperbaharui dan mendapatkan bobot positif. Kemudian node *q* dihilangkan lalu algoritma *Djikstra* akan menghitung rute terpendek dari masing-masing *node* pada *graph* yang telah diperbaharui. Berikut adalah gambar dari tahapan algoritma *Johnson* dalam pembobotan ulang sebuah *graph*.



Gambar 2.4 Algoritma *Johnson* [23]

Pada penelitian sebelumnya oleh M.Ilhamsyah, dkk [24] pada tahun 2016, Algoritma *Johnson* dapat diterapkan untuk mendapatkan *shortest path* pada *Software Defined Network*. Algoritma ini diterapkan pada *control plane* SDN agar pada sisi *data plane* yaitu *switch* dapat melakukan *route discovery* dengan mempertimbangkan bobot terendah dari masing-masing hop. Selain itu Algoritma

Johnson dipilih karena dari segi waktu eksekusi lebih baik dibandingkan algoritma jenis *all pair shortest path* lainnya yaitu algoritma *Floyd Warshall*. Pada penelitian lain oleh Anitha dan Ramesh [21] pada tahun 2018 Algoritma *Johnson* digunkan untuk meminimalisasi *loss* dengan cara merekonfigurasi jaringan dengan memilih bobot minimum untuk sampai ketujuan. Kemudian dibandingkan hasil dari jaringan sebelum dan setelah di rekonfigurasi. Untuk lebih jelasnya berikut adalah perhitungan algoritma *Johnson* untuk mencari rute terpendek.

a. Mengkonstruksi graph baru dengan menambahkan node baru dengan persamaan

$$V' = V + \{s\} \text{ dan } E + \{(s, v)v \text{ di } V\} \quad (2.1)$$

b. Setiap node *v* di *V*

$$0 \rightarrow w(s, v) \quad (2.2)$$

$$\infty \rightarrow w(v, s) \quad (2.3)$$

c. Menjalankan algoritma Bellman-Ford pada graph baru
   - Jika bobot negatif maka selesai
   - Jika bobot positif maka akan dihitung $d(s, v), \forall v \in V$ (2.4)

d. Setiap *(u, v)* di *E*

$$W(u, v) = w(u, v) + d(s, v) - d(s, u) \quad (2.5)$$

e. Menjalankan Algoritma Djikstra pada setiap *v* di *V* untuk menghitung *d(u, v)*

$$D' = d(u, v) \quad (2.6)$$

f. Menghitung setiap *(u, v)* di *V* ;

$$d(u, v) = d(u, v) + d(s, v) - d(s, u) \quad (2.7)$$

$$D = d(u, v) \quad (2.8)$$

2. 6. *Congestion Avoidance*

Congestion terjadi ketika suatu jaringan memiliki beban kerja yang banyak sehingga mengakibatkan kinerja suatu jaringan menurun atau dapat dikatakan jumlah permintaan lebih besar dibandingkan *resource* yang tersedia. Untuk menghindari *congestion* maka dibutuhkan sebuah mekanisme *congestion control* digunakan agar suatu jaringan dapat mengalokasikan *resource* yang tersedia dengan

tepat, agar suatu jaringan tetap dapat melakukan kinerja secara baik ketika terjadi permintaan melebihi kapasitas *resource* yang tersedia. *Resource* yang dimaksud ini berupa *bandwidth, memory buffer,* dan *processing capacity* antar *intermediate node* [25]. *Congestion* tidak hanya terjadi pada jaringan konvensional melainkan juga terjadi pada SDN. Dengan model *control plane* terpusat SDN sangat rentan akan terjadinya *congestion* dikarenakan model komunikasi *one to many* yang mirip dengan masalah *TCP incast*. *TCP incast* adalah pola komunikasi *one-to-many* biasanya ditemukan pada pusat data yang menerapkan penyimpanan terdistribusi dan komputasi framework.

Untuk menangani *congestion* pada SDN terdapat beberapa mekanisme, seperti yang dibahas pada survey [5] ada beberepa mekanisme untuk mengatasi *congestion* seperti *SDN-based TCP Congestion Control, Congestion-Aware and Robust multicast TCP, Scalable Congestion Control Protocol*. Dengan metode *congestion control* yang tepat *throughput* jaringan dapat tetap terjaga kestabilannya walaupun dengan beban *traffic* jaringan yang tinggi. *Congestion* dapat terjadi pada sebuah *traffic* jaringan karena disebabkan oleh beberapa faktor diantaranya, terlalu banyak host dalam sebuah *broadcast domain*, terlalu banyak *host* yang mengirimkan informasi *broadcast* pada sebuah jaringan sehingga terjadi *Broadcast Storm*, semakin banyak *host* maka semakin besar pula *broadcast storm*. *Multicasting* terjadi karena banyak *host* yang melakukan *request* pada suatu informasi secara bersamaan sehingga terjadi kemacetan *traffic* jaringan dan berkurangnya *bandwidth* pada jaringan, selain itu *bandwidth* yang kecil dan tidak sesuai dengan banyaknya *traffic* jaringan akan menyebabkan *overload* pada jaringan. *Data Collision* terjadi karena terdapat suatu perangkat yang mencoba mengirimkan sebuah paket data pada sebuah segmen pada jaringan dan kemudian ada perangkat lain yang mencoba melakukan hal yang sama pada segmen yang sama sehingga akan terjadi tabrakan (*collision*) sehingga menyebabkan paket tersebut rusak dan berakibat paket harus dikirim ulang.

Akash S., dkk [26] mengusulkan konsep kendali kongesti dengan melakukan *traffic rerouting* dengan menggunakan algoritma *Dijkstra* sebagai dasar sistem. Penelitian ini menggunakan controller *OpenDayLight* berbasis java yang diuji coba

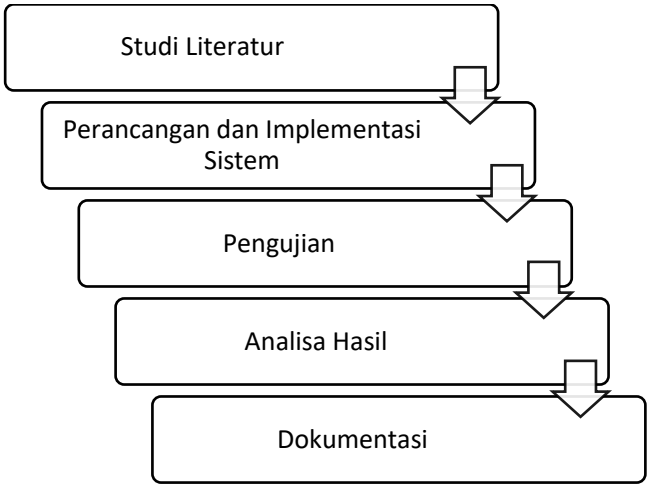
pada jaringan virtual dengan topologi *tree* pada *Software Defined Network*. Seungbeom S.,dkk [27] juga mengusulkan algoritma *flow congestion avoidance* yang dikendalikan oleh jaringan di lingkungan *Software Defined Network*. Peneliti mengusulkan algoritma untuk mencegah masuknya paket baru yang menghasilkan *high congestion* pada switch melalui pemantauan jaringan. Dalam skema yang diusulkan, *link utilization* dihitung dalam *controller* SDN dan algoritma penghitungan ulang rute diterapkan pada switch menggunakan protokol konfigurasi *Openflow*. Hasil dari penelitian ini menunjukkan bahwa dapat mencapai peningkatan ketahanan TCP dengan mendekati *lossless* dengan mengendalikan jaringan.

(Halaman ini sengaja dikosongkan)

BAB 3

METODOLOGI PENELITIAN

Pada Bab ini akan dijelaskan alur metodologi yang digunakan pada penelitian. Metodologi yang dilakukan terdiri dari studi literatur, perancangan metode yang diusulkan, implementasi, pengukuran performa, analisa hasil, dan terakhir merupakan penyusunan laporan. Gambar 3.1. merupakan ilustrasi alur yang dilakukan.



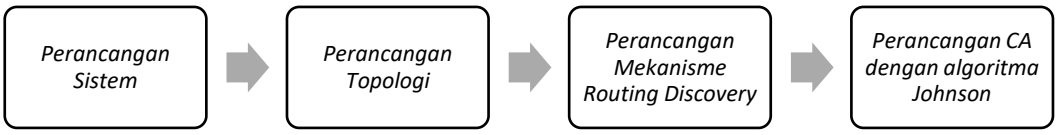
Gambar 3.1 Alur Metodologi Penelitian

Pada Sub-bab berikut ini akan dibahas tahapan metodologi penelitian secara lebih rinci.

3. 1. Studi Literatur

Tahap pertama yang dilakukan pada penelitian ini merupakan pengkajian literatur yang sesuai dengan topik penelitian. Literatur yang digunakan yaitu tentang *Software Defined Network*, Algoritma *Johnson*, metode *route discovery* pada SDN, serta metode untuk melaukan evaluasi pada metode yang diusulkan.

3. 2. Metode yang Diusulkan



Gambar 3.2 *Flowchart* metode penelitian

Pada penelitian ini terdapat empat bagian utama yaitu yang pertama adalah perancangan sistem, perancangan topologi, perancangan algoritma *routing*, dan yang terakhir merupakan penerapkan algoritma *routing* untuk menghindari *congestion*. Berikut merupakan penjelasan tiap bagian dalam *flowchart* yang ditunjukkan pada Gambar 3.2 diatas.

3.2.1 Perancangan Sistem

Perancangan sistem merupakan tahapan yang dilakukan untuk membangun sistem dari penelitian setelah melakukan studi literatur. Tujuan dilakukan perancangan sistem agar implementasi sistem berjalan sistematis dan terstruktur. Spesifikasi perangkat keras yang digunakan untuk perancangan penelitian ini dapat dilihat pada tabel 3.1 berikut.

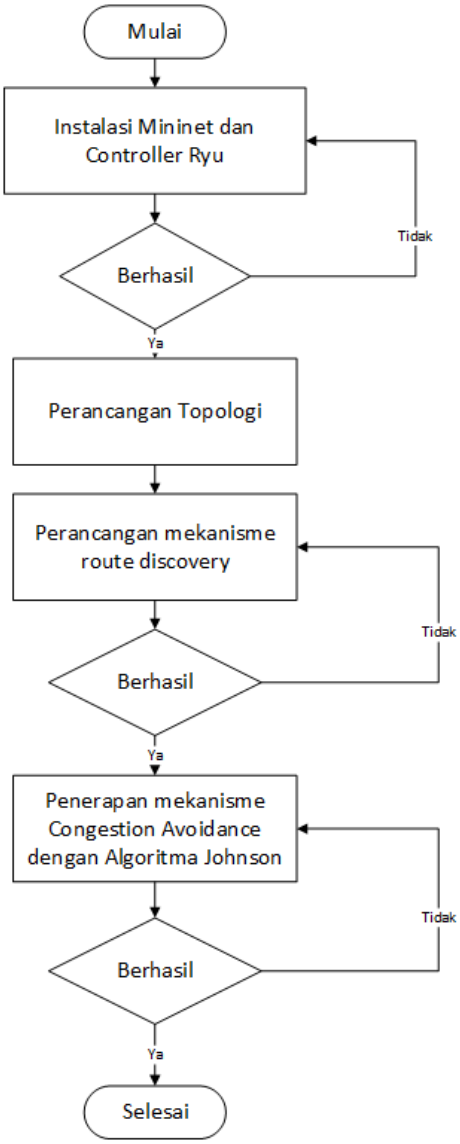
Tabel 3.1 Lingkungan Simulasi

Komponen	Spesifikasi
Sistem Operasi	Ubuntu 14.04 64bit
CPU	Intel® Core™ i7-3537U
RAM	RAM DDR3 1600 MHz 2x4GB
Penyimpanan	1 TB

Sedangkan perangkat lunak yang digunakan dalam penelitian ini adalah:

1. *Mininet* sebagai simulator pembuatan topologi virtual berbasis *Software Defined Network*
2. *Miniedit* sebagai visualisasi topologi SDN
3. *Ryu* sebagai controller pada jaringan SDN
4. NetworkX sebagai library untuk implementasikan algoritma *routing*.

Beberapa tahapan perancangan sistem dapat dilihat pada diagram alir pada gambar 3.3 dibawah ini.



Gambar 3.3 *Flowchart* tahap perancangan sistem

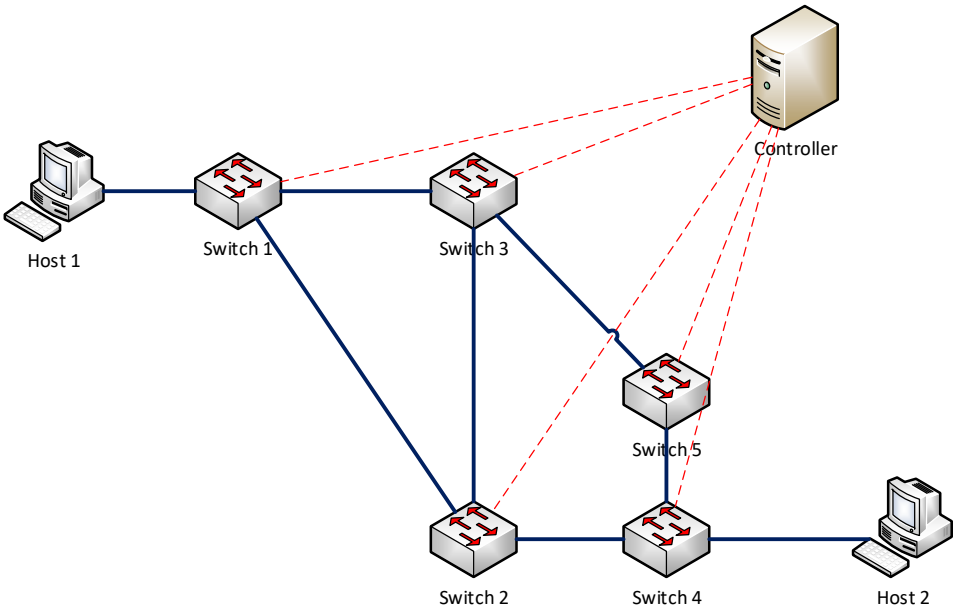
Perancangan sistem yang dimulai dengan melakukan instalasi *mininet* sebagai simulator jaringan *Software Defined Network* dan *Ryu* sebagai *controller*. Apabila proses instalasi telah selesai akan dilakukan proses pengecekan terlebih dahulu, jika sudah dapat berjalan dan simulator bisa digunakan maka akan berlanjut ke perancangan topologi.

3.2.2 Perancangan Topologi

Perancangan topologi akan menjelaskan topologi yang dipilih dalam penelitian. Dua topologi yang berbeda dirancang dengan tujuan untuk mengetahui performansi dan untuk mengetahui skalabilitas algoritma *routing* pada 2 buah topologi yang berbeda.

3.2.2.1 Perancangan Topologi Mesh

Topologi *mesh* dipilih karena topologi ini sering digunakan pada implementasi *Software Defined Network* dalam berbagai macam skenario. Karena topologi ini penggambaran arsitektur jaringan yang sederhana dan tidak terlalu rumit. Topologi ini juga memiliki rute alternatif sehingga jika salah satu rute utama tidak tersedia maka rute lain masih tetap tersedia [8]. Perancangan topologi akan dibuat menggunakan *mininet* sebagai simulator. Berikut adalah gambar 3.4 dari rancangan topologi yang akan digunakan.



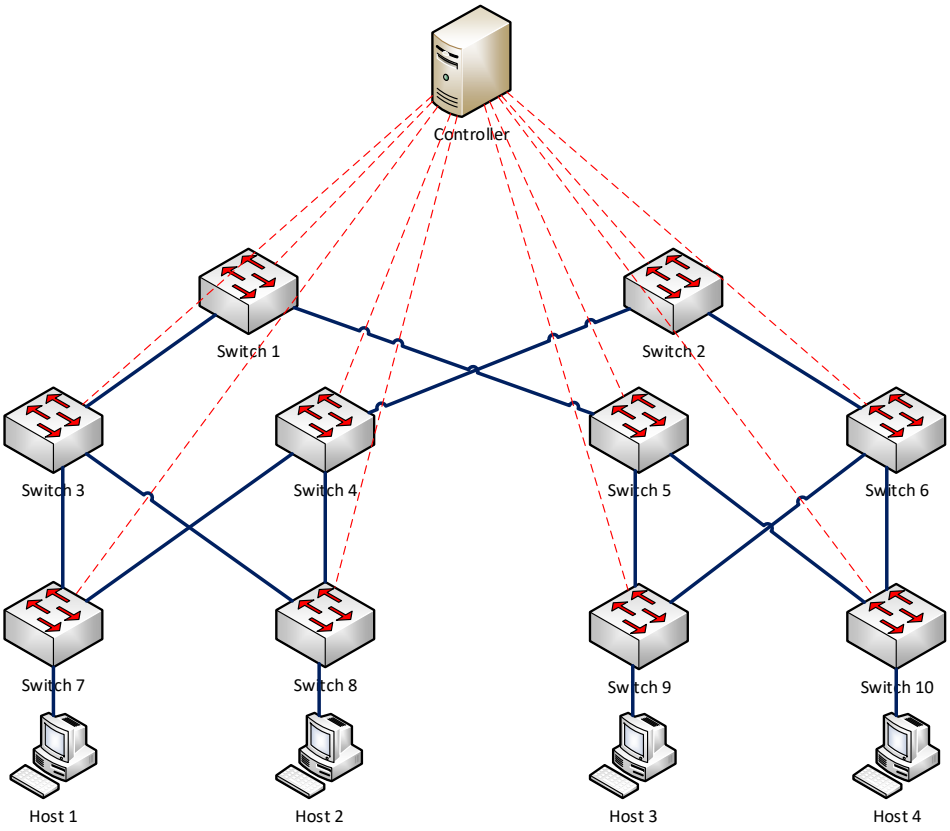
Gambar 3.4 Topologi Mesh

Pada simulasi jaringan pertama yang digunakan yaitu dengan menggunakan 5 *Openflow Switch* dengan 2 buah *host* dan 1 buah

controller. Topologi yang digunakan adalah topologi *partially connected mesh* dengan skala kecil dikarenakan pada topologi ini terdapat beberapa pilihan rute alternatif.

3.2.2.2 Perancangan Topologi *Fat-tree*

Topologi *fat-tree* dipilih karena topologi ini sering digunakan pada implementasi *Software Defined Network* dalam skenario *data center* [28]. Selain itu topologi ini juga memiliki rute alternatif sehingga jika salah satu rute utama tidak tersedia maka rute lain masih tetap tersedia. Perancangan topologi akan dibuat menggunakan *mininet* sebagai simulator. Untuk simulasi topologi jaringan kedua dapat dilihat pada gambar 3.5 berikut.



Gambar 3.5 Topologi jaringan *Fat-tree*

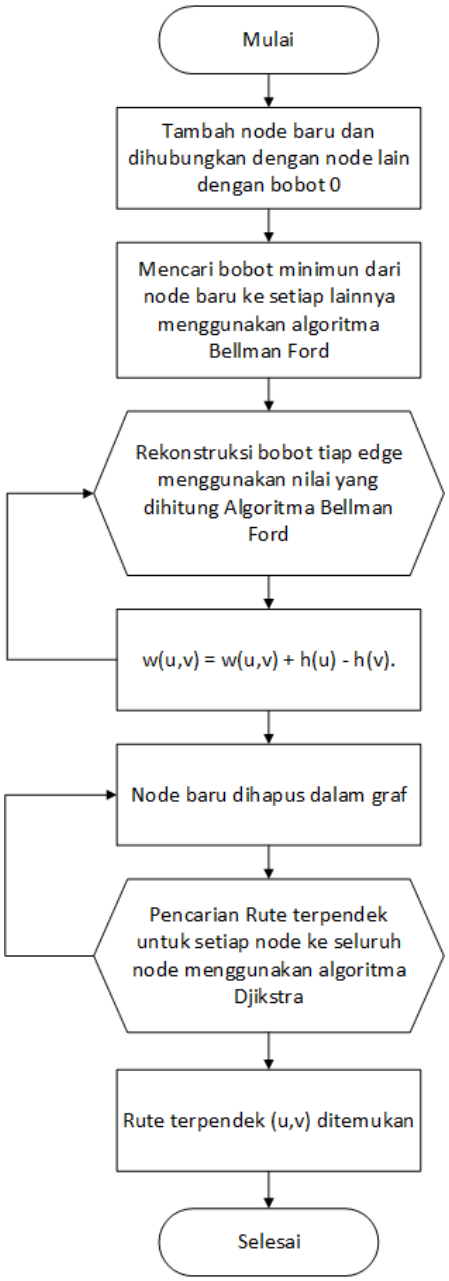
Pada simulasi jaringan kedua menambah jumlah entitas jaringan dengan dengan menggunakan 10 *Openflow Switch*

dengan 4 buah *host* dan 1 buah *controller*. Selanjutnya kedua topologi ini akan di simulasikan pada simulator mininet dilakukan perancangan algoritma *routing* dengan membuat program *shortest path* dengan menggunakan algoritma *Johnson*.

3.2.3 Perancangan Mekanisme *Route Discovery*

Perancangan mekanisme *route discovery* akan menjelaskan alur untuk menentukan rute minimum pada suatu algoritma *routing*. Pada penelitian ini menggunakan algoritma yaitu Algoritma *Johnson*. Algoritma ini memiliki mekanisme menentukan *shortest path* pada sebuah topologi *routing*. Untuk menentukan *shortest path*, pada Algoritma *Johnson* menambahkan *node* baru pada graf. Selanjutnya adalah merekonstruksi bobot pada semua *edge node* menggunakan Algoritma *Bellman-Ford* sebagai subrutin. Tiap edge pada topologi memiliki *cost*. *Cost* akan direpresentasikan sebagai jarak antar node dalam satuan meter. Kemudian Algoritma *Dijkstra* digunakan untuk mencari *shortest path* dari tiap *node* ke semua *node* lain dengan *cost* yang paling minimum.

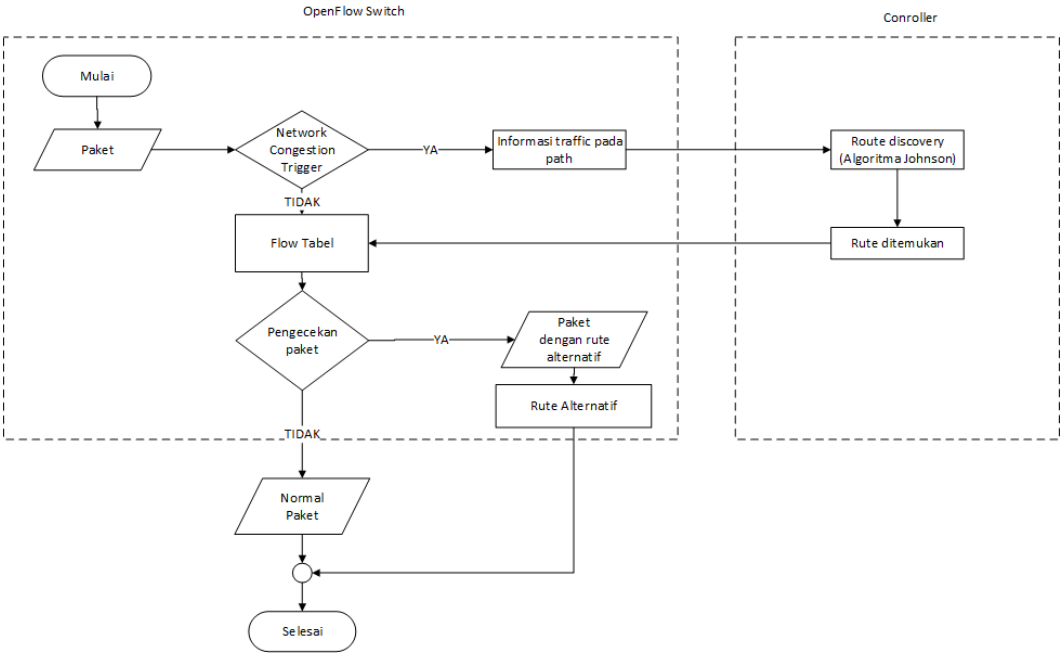
Mekanisme *route discovery* pada Algoritma *Johnson* diawali dengan menambahkan sebuah *node* baru yang dihubungkan pada setiap *node* lainnya dengan bobot 0. Kemudian Algoritma *Bellman-Ford* dijalankan untuk menghitung bobot minimum dengan *node* baru sebagai titik awal. Tahap selanjutnya adalah merekonstruksi bobot pada setiap *node* dengan persamaan $w(u,v) = w(u,v) + h(u) - h(v)$. Lalu *node* baru tersebut dihapus dari topologi dan *Algoritma Dijkstra* dijalankan untuk menghitung *shortest path* dari setiap *node* ke *node* lainnya dengan bobot yang diberikan menggunakan perasamaan $D(u,v) = D(u, v) + h(v) - h(u)$. Hasil dari perancangan ini adalah program berekstensi .py yang akan dijalankan pada *controller*. Untuk *flowchart* dari algoritma *Johnson* dapat dilihat pada Gambar 3.6 berikut.



Gambar 3.6 *Flowchart Routing Discovery*

3.2.4 Perancangan penghindaran *congestion* dengan Routing

Perancangan mekanisme penghindaran *congestion* dilakukan apabila perancangan *routing discovery* telah berhasil dilakukan. Mekanisme penghindaran *Congestion* pada SDN yang telah dijelaskan pada sub-bab 2.6. Berikut adalah Gambar 3.7 dibawah ini adalah metode yang diusulkan.



Gambar 3.7 *Flowchart* Metode yang diusulkan

Ketika paket melewati sebuah *switch* akan selalu dilakukan pendeteksian *congestion* jika terjadi *congestion* pada rute pengiriman. Jika paket melewati rute yang sedang terjadi *congestion* maka *controller* akan mendeteksi adanya *traffic* dengan *delay*, maka *data plane* akan menginformasikan *controller* untuk menggunakan algoritma *Johnson* sebagai pemilihan rute alternatif yang akan dilalui paket untuk sampai ke tujuan dengan pertimbangan rute alternatif dengan *cost* yang paling minimum. Sehingga paket berikutnya akan berisi informasi bahwa pada rute tersebut sedang terjadi *congestion* dan paket akan dilewatkan melalui rute alternatif yang yang tersedia pada proses *route discovery* yang lalu lintas datanya tidak padat. Kemudian *controller* akan menginformasikan kepada *data plane* agar informasi pemilihan rute alternatif disimpan pada *flow* tabel. Proses dilanjutkan dengan pengecekan paket, apakah paket yang akan dikirimkan berisi informasi *route discovery*. Jika paket tidak melewati rute yang sedang terjadi *congestion* maka paket tersebut termasuk paket normal yang akan langsung dikirimkan dari *source* ke *destination*. Saat pertama kali melakukan pencarian rute suatu algoritma *routing* membutuhkan waktu saat pertama kali menentukan rute.

3. 3. Pengujian

Tahap pengujian pada penelitian ini dilakukan untuk menunjukkan bahwa sistem telah mampu bekerja sesuai perancangan. Penelitian ini mengusulkan metode Congestion control dengan mencari *shortest path* dengan Algoritma *Johnson* pada *Software Defined Network*. Pengujian sistem akan dilakukan pada sebuah antarmuka jaringan virtual yaitu mininet. Masing-masing *switch* pada topologi akan terhubung *controller* pusat menggunakan *platform* kontrol *Ryu*. *Controller Ryu* menggunakan basis pemrograman *python*, yang akan diimplementasikan sebagai *remote controller*. *OpenFlow* versi 1.3 digunakan sebagai penghubung antara *control plane* dan *data plane*. Simulasi jaringan dilakukan pada mininet dengan antar muka miniedit untuk visualisasi topologi. Masing-masing host didefinisikan dengan nama H1-H4 dengan alamat IP 10.0.0.1-10.0.0.4 pada masing-masing host. Berikut adalah tabel fungsionalitas dari masing-masing komponen mininet.

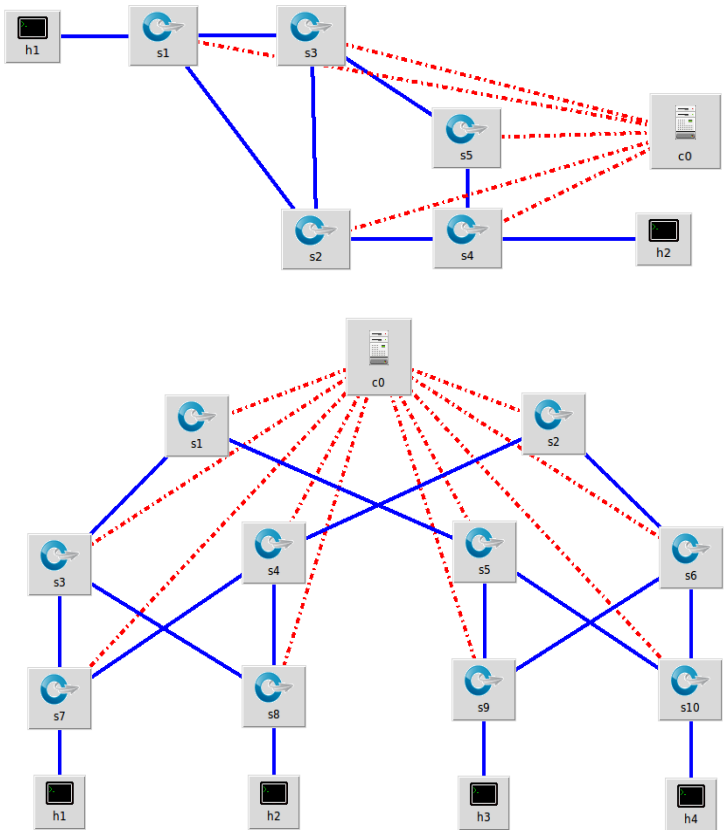
Tabel 3.2 Fungsi komponen mininet

No.	Icon	Nama	Fungsi
1.		*Controller*	Bertindak sebagai *Controller* SDN yang diakses oleh *Mininet* pada port tertentu.
2.		*Host*	Bertindak sebagai *host/end device* yang menggunakan layanan jaringan.
3.		*Openflow Switch*	Bertindak sebagai perangkat jaringan *OpenFlow* yang bertindak sebagai *data plane* SDN.

3.3.1 Skenario pengujian

1. Skenario dua topologi yang berbeda tujuannya untuk mengetahui performansi, fleksibilitas, dan skalabilitas algoritma *routing* jika diimplementasikan pada topologi yang berbeda. Topologi *mesh* dipilih karena topologi ini sering digunakan pada implementasi *Software Defined Network* dalam berbagai macam skenario.

Topologi *fat-tree* dipilih karena topologi ini sering digunakan pada implementasi *Software Defined Network* dalam skenario *data center*. Antarmuka jaringan dapat dilihat pada gambar 3.8 berikut.



Gambar 3.8 Antarmuka minnet

2. Skenario pengujian *route discovery* dengan menggunakan tool *ping* pada host asal yang ditujukan ke host tujuan. Tujuannya adalah untuk mengetahui apakah algoritma *routing Johnson* sudah dapat menemukan semua rute dan menentukan *shortest path* dari node asal ke node tujuan. Hasil *route discovery* sebelum dan sesudah algoritma *routing* diterapkan akan dibandingkan.
3. Skenario yang digunakan untuk menguji *latency* dengan tool *ping* antar *host* dengan variasi menambah variasi paket yang dikirmkan sebanyak 100 kali pengiriman. Hasil *latency* sesudah dan sebelum algoritma diterapkan akan dibandingkan. Tujuan dari pengujian

latency adalah untuk mengetahui waktu yang dibutuhkan oleh suatu paket menempuh jarak dari node pengirim ke node penerima sebelum dan sesudah menggunakan Algoritma *Johnson.*

4. Skenario uji coba *throughput* dengan tool *iperf* yang dijalankan pada masing-masing host sebagai *server* dan *client*. untuk mengetahui *bandwidth* yang telah digunakan saat proses komunikasi dan juga untuk mengetahui apakah ada *packet loss* saat terjadi komunikasi antar host. Pada pengujian ini dilakukan selama 100 detik dengan interval waktu 1 detik. Hasil *throughput* sesudah dan sebelum algoritma diterapkan akan dibandingkan.
5. Skenario pengujian *response time* juga dapat dilakukan dengan tool *ping* dengan melakukan *filtering* paket yang pertama masuk pada *switch* yang terhubung langsung pada *host destination*. Skenario ini dilakukan dengan sample 30 kali percobaan. Hal ini dilakukan untuk mengetahui performansi algoritma pada saat pertama kali melakukan pencarian rute. Hasil *response time* Algoritma *Johnson* akan dibandingkan dengan algoritma OSPF konvensional yaitu algoritma *Dijkstra*.

3.3.2 Parameter Evaluasi

Pengujian akan mengamati beberapa parameter evaluasi yang telah ditentukan, antara lain *route discovery*, *latency*, *throughput*, dan *response time* pada sebuah topologi.

1. *Route discovey*

   Tujuan dari pengujian *Route Discovery* adalah untuk mengetahui apakah algoritma *Johnson* sudah dapat menemukan semua rute dan menentukan *shortest path* dari node asal ke node tujuan dengan *cost* yang minimal. Pada pengujian ini, mengimplementasikan Algoritma *Johnson* pada *controller Ryu* yang berbasis bahasa pemrogograman python untuk menentukan rute terpendek (*shortest path)*.

2. *Latency*

Tujuan dari pengujian *latency* adalah untuk mengetahui waktu yang dibutuhkan oleh suatu paket menempuh jarak dari node pengirim ke node penerima sebelum dan sesudah menggunakan Algoritma *Johnson*. Jika nilai *latency* yang didapatkan semakin kecil, maka semakin baik pula sistem yang digunakan.

3. *Throughput*

Tujuan dari pengujian *bandwidth* dan *throughput* dalam suatu *link* untuk mengetahui berapa kecepatan dan bandwidth yang digunakan saat pengiriman paket. Untuk menghitung *throughput* dapat menggunakan perhitungan sebagai berikut.

$$Throughput(mbps) = \frac{jumlah\ data\ yang\ dikirim}{waktu\ pengiriman\ data} \quad (3.1)$$

4. *Response Time*

Response Time adalah waktu yang dibutuhkan algoritma *routing* saat pertama kali melakukan pencarian rute dari *source* ke *destination* host. Semakin cepat *response time* suatu algoritma *routing* maka akan semakin handal suatu jaringan dalam menentukan rute baru.

3. 4. Analisa Hasil Pengujian

Proses analisa hasil dilakukan dengan menganalisa parameter seperti *throughput, packet loss, latency, response time* dan kemampuan algoritma *Johnson* dalam melakukan *route discovery* dan menentukan *shortest path* pada sebuah topologi virtual. Analisa dilakukan dengan membandingkan beberapa parameter tersebut sebelum dan sesudah diimplementasikannya *route discovery*. Dan membandingkan performansi algoritma OSPF konvensional yaitu Algoritma *Dijkstra*.

3. 5. Jadwal Penelitian

Tabel 3.3 Jadwal Kegiatan Penelitian

No.	Kegiatan	Bulan																			
		Februari				Maret				April				Mei				Juni			
1.	Studi Literatur																				
2.	Perancangan Sistem																				
3.	Implementasi Metode																				
4.	Pengujian dan Analisa Hasil																				
5.	Penyusunan Laporan																				

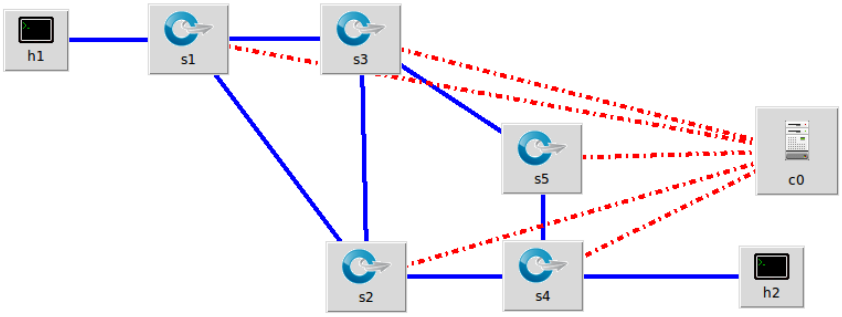
(Halaman ini sengaja dikosongkan)

BAB 4
HASIL DAN PEMBAHASAN

Analisa hasil penelitian ini membahas skenario simulasi jaringan virtual sebelum dan sesudah diimplementasikannya *route discovery*. Terdapat dua skenario pengujian yang pertama pada topologi *mesh* dan topologi *fat-tree*. Setiap hasil simulasi mencakup pembahasan tentang analisa implementasi algoritma *routing Johnson* dengan *route discovery* dan perbandingan hasil dari parameter uji coba yakni *shortest path*, *response time*, *latency*, dan *throughput.*

4.1 Hasil Uji Coba Topologi *Mesh*

Pada skenario pertama topologi mesh dibuat dengan entitas kecil 5 *switch* dengan 2 *host* dan 1 *controller*. Dengan menggunakan konfigurasi *openflow* versi 1.3 sebagai penghubung antara *controller* dan *switch*. Uji coba dilakukan H1 melakukan pengiriman data menuju H2. Untuk lebih jelasnya visualisasi topologi dapat dilihat pada gambar 4.1.



Gambar 4.1 Topologi *Mesh* pada mininet

4.1.1 Uji Coba *Route Discovery* Topologi *Mesh*

Pada uji coba ini yang pertama kali dilakukan pengujian kemampuan algoritma *routing* dalam melakukan *route discovery* dan *path available* dari *host source* ke *host destination*. Algoritma *routing* diupload pada *ryu controller* berbasis *remote control plane* sehingga dapat dengan mudah dilakukan konfigurasi. *Path*

available yang termonitor pada sisi control plane dapat dilihat pada gambar 4.2 berikut.

```
Available paths from  1  to  4  :  [[1, 3, 5, 4], [1, 3, 2, 4], [1, 2, 4], [1, 2
, 3, 5, 4]]
[1, 2, 4] cost =  2
```

Gambar 4.2 *Available path* pada *Topologi Mesh*

Secara *default* algoritma *routing* akan memilih *shortest path* dari topologi untuk pengiriman paket. Pada uji coba dilakukan pengujian dua buah algoritma yaitu algoritma *Johnson* sebagai algoritma yang diusulkan dan algoritma *Dijkstra* sebagai pembanding. Pada pengujian *route discovery* dua buah algoritma ini sama-sama dapat menemukan seluruh rute yang tersedia untuk dilalui paket dan rute yang dipilih sebagai rute alternatif melewati rute yang sama. Untuk lebih jelasnya dapat dilihat pada tabel 4.1 berikut.

Tabel 4.1 *Available route* pada topologi mesh

Node Asal	Node Tujuan	*Route Available*	*Shorthest Path*	*Alternate Route*
H1	H2	1-3-5-4	1-2-4	1-3-5-4
		1-3-2-4		
		1-2-4		
		1-2-3-5-4		

Sebelum diimplementasikan *route discovery* paket melewati rute terpendek dengan path 1-2-4. Hal ini dikarenakan sifat dari algoritma *shortest path* yaitu memilih rute yang paling pendek. Saat *congestion* terdeteksi ditandai dengan terjadinya peningkatan *latency* maka paket akan dilewatkan rute alternatif dimana rute yang dipilih adalah rute dengan jarak sama dengan rute sebelumnya. Untuk pengujian *latency* akan dijelaskan pada sub-bab tersendiri setelah pengujian *shortest path*. Namun dalam kondisi ini *controller* tidak dapat menemukan rute alternatif pada topologi *mesh* yang hopnya sama dengan rute sebelumnya sehingga paket mencari rute lain untuk pengiriman paket. Pada kasus ini controller akan memilih rute yang lebih jauh yang terpaut satu hop dengan rute sebelumnya. Dari hasil uji coba diatas terdapat dua buah alternatif rute yang tersedia yaitu melalui

switch 1-3-5-4, 1-3-2-4, dan 1-2-3-5-4. Namun dari sini *controller* akan memilih rute 1-3-5-4 sebagai rute alternatif karena dirasa lebih baik performansinya karena tidak melewati switch 2 karena sebelumnya paket sudah melewati switch pada pengiriman sebelumnya. Sehingga dapat disimpulkan bahwa algoritma *Johnson* dapat melakukan pencarian rute yang tersedia dan dapat memilih rute terpendek beserta rute alternatif yang akan digunakan untuk *packet forwarding* pada simulasi skenario pengujian.

4.1.2 Uji Coba *Latency* Topologi Mesh

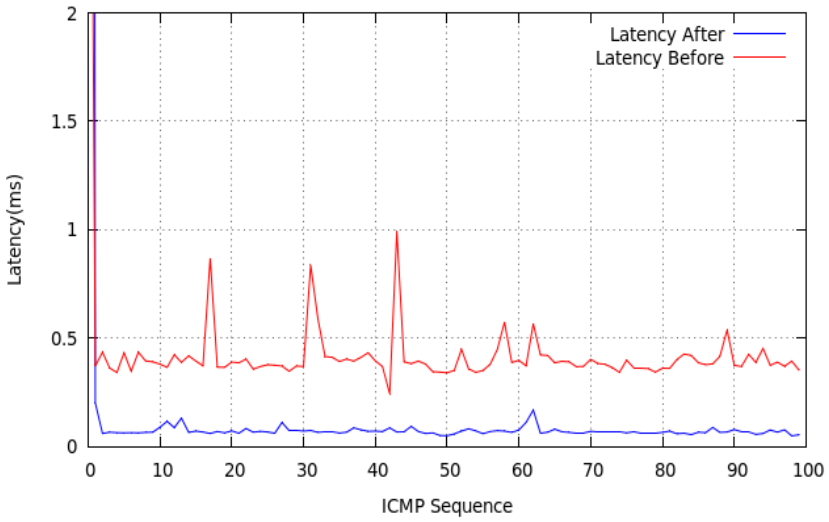
Pada pengujian *latency* dilakukan pengujian dengan melakukan perbandingan algoritma *Johnson* sebelum dan setelah diimplementasikannya *route discovery*. Pada skenario ini host H1 akan melakukan pengiriman data menuju *host destination* H2 dengan ukuran maksimal paket yang dikirim sebesar 20000 byte untuk memberikan beban pada saat pengiriman. Pengiriman dilakukan sebanyak 100 kali menggunakan tool *ping*. Berikut adalah data yang didapatkan saat sebelum dan sesudah metode dapat dilihat pada tabel 4.2 dan 4.3 dan disajikan dalam bentuk grafik pada gambar 4.3.

Tabel 4.2 Hasil Uji Coba *Latency* Sebelum *Routing* pada Topologi *Mesh*

Min (ms)	Avg (ms)	Max (ms)	Mdev (ms)
0.342	0.911	5.590	1.645
0.366	0.435	0.869	0.154
0.348	0.376	0.404	0.016
0.370	0.466	0.837	0.144
0.244	0.424	0.992	0.205
0.341	0.399	0.574	0.073
0.369	0.409	0.567	0.058
0.343	0.370	0.403	0.021
0.361	0.408	0.538	0.052
0.354	0.390	0.453	0.029
Avg 0.344	Avg 0.459	Avg 1.123	Avg 0.240

Tabel 4.3 Hasil Uji Coba *Latency* Setelah *Routing* pada Topologi Mesh

Min (ms)	Avg (ms)	Max (ms)	Mdev (ms)
0.061	3.412	33.4	10.537
0.061	0.083	0.132	0.024
0.062	0.075	0.112	0.015
0.063	0.072	0.087	0.007
0.052	0.071	0.094	0.012
0.051	0.068	0.083	0.009
0.062	0.083	0.169	0.034
0.062	0.067	0.071	0.003
0.056	0.067	0.089	0.009
0.050	0.066	0.079	0.010
Avg 0.058	Avg 0.406	Avg 3.432	Avg 1.066

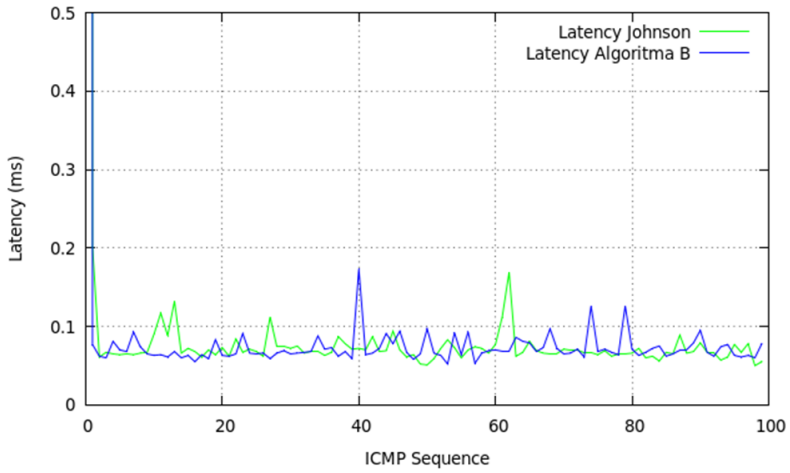


Gambar 4.3 Grafik perbandingan *Latency* topologi *Mesh*

Pada hasil uji coba didapatkan peningkatan performansi dari *latency* sebelum dan sesudah diimplementasikannya *route discovery* menggunakan algoritma *Johnson*. Sebelum implementasi metode seluruh paket akan melewati rute terpendek yang sama sehingga menyebabkan terjadinya kepadatan pada rute yang dilalui. Dapat dilihat grafik pada gambar 4.3 sebelum implementasi *latency* yang dihasilkan cenderung tidak stabil. Sebelum metode diimplementasikan rata-rata *latency* yang didapatkan 0.459 ms dengan maksimal *latency* 5.590 ms. Namun berbeda saat setelah implementasi walau rute yang dilalui cenderung lebih jauh

dibanding sebelumnya namun *latency* yang dihasilkan cenderung stabil dan mengalami penurunan dengan rata-rata *latency* sebesar 0.406 ms.

Dari hasil uji coba tersebut penurunan *latency* dengan presentase sebesar 11.5%. Nilai maksimal *latency* setelah metode sebesar 33.4 ms, hal ini dikarenakan pengaruh dari *response time* pada algoritma *routing* untuk mencari rute baru pada saat terjadi *congestion* pada rute terpendek, sehingga waktu yang dibutuhkan untuk pengirman paket cenderung lebih lama karena *controller* memerlukan waktu untuk mengadaptasi rute yang baru. Ada kemungkinan jika *controller* tidak dapat menemukan rute alternatif yang sama dengan rute sebelumnya dan melalui rute yang lebih jauh maka *latency* yang dihasilkan akan semakin tinggi.



Gambar 4.4 Grafik perbandingan *Latency* dua Algoritma

Perbandingan hasil uji coba *latency* setelah metode diimplementasikan juga akan dibandingkan dengan Algoritma OSPF konvensional dengan dasar Algoritma *Dijkstra* dengan *route discovery*. Yang hasil uji coba grafiknya dapat dilihat pada gambar 4.4. Rata-rata *latency* yang dihasilkan algoritma pembanding sebesar 0.472 ms. Rata-rata *latency* yang dihasilkan hampir sama dengan algoritma *Johnson*. Perbedaan waktu yang dibutuhkan algoritma pembanding lebih lama dengan 40.3 ms saat pertama kali pengirimkan paket. *Latency* pada pengiriman paket selanjutnya yang dihasilkan tergolong rendah karena rute yang dilalui tidak dilewati oleh paket lain. Sehingga dapat disimpulkan mekanisme *route discovery* dengan dapat mengurangi *latency* dan mempertahankan *latency* agar lebih stabil saat pengiriman paket.

4.1.3 Uji Coba *Throughput* Topologi *Mesh*

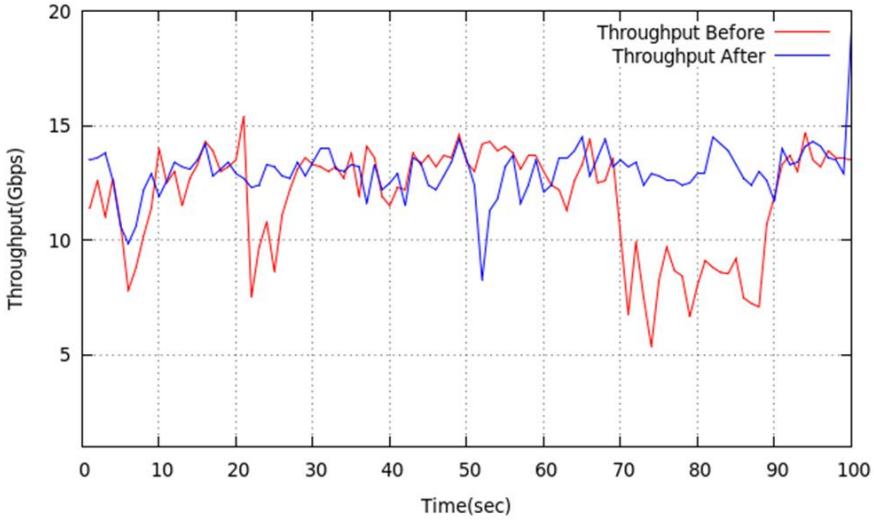
Pada uji coba *throughput* dilakukan pengujian dengan melakukan perbandingan sebelum dan setelah diimplementasikannya metode. Akan dibandingkan juga performansi setelah metode diimplementasikan dengan algoritma OSPF konvensional. Pada skenario ini host H1 bertindak sebagai *client* dan host H2 bertindak *server*. Uji coba dilakukan dengan waktu 100 detik dengan interval pengiriman 1 detik sekali menggunakan tool *iperf*. Berikut adalah sample data 20 detik pertama yang didapatkan saat sebelum dan sesudah *routing discovery* dapat dilihat pada tabel 4.4 dan 4.5.

Tabel 4.4 TCP *Throughput* Sebelum *Routing* pada Topologi *Mesh*

Interval (Sec)	Transfer Data (Mbytes-GBytes)	Bandwidth (Gbits/sec)
0.0- 1.0	1.33	11.4
1.0- 2.0	1.47	12.6
2.0- 3.0	1.28	11.0
3.0- 4.0	1.47	12.7
4.0- 5.0	1.25	10.7
5.0- 6.0	930M	7.80
6.0- 7.0	1.02	8.80
7.0- 8.0	1.19	10.2
8.0-9.0	1.32	11.4
9.0- 10.0	1.63	14.0
10.0- 11.0	1.45	12.5
11.0- 12.0	1.52	13.0
12.0- 13.0	1.34	11.5
13.0- 14.0	1.48	12.7
14.0- 15.0	1.55	13.3
15.0- 16.0	1.67	14.3
16.0- 17.0	1.62	13.9
17.0- 18.0	1.51	13.0
18.0- 19.0	1.53	13.2
19.0- 20.0	1.58	13.5

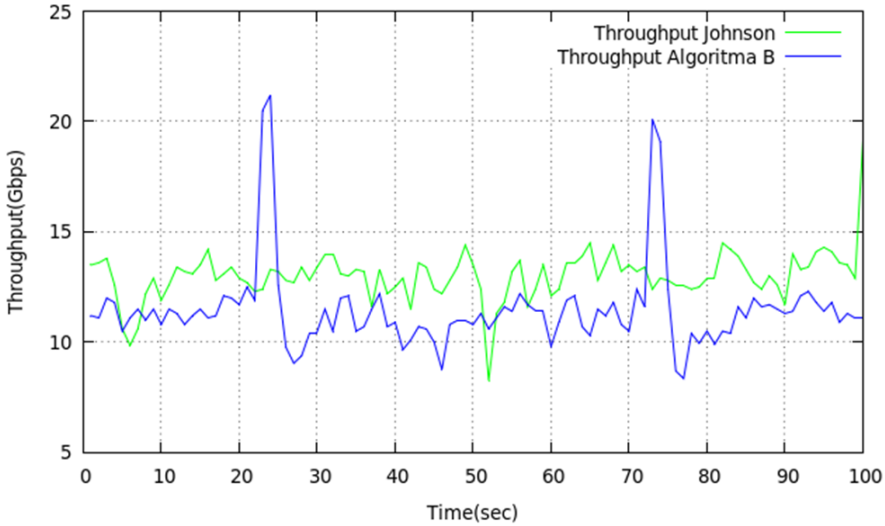
Tabel 4.5 TCP *Throughput* Setelah *Routing* padaTopologi *Mesh*

Interval (Sec)	Transfer Data (GBytes)	*Bandwidth (Gbits/sec)*
0.0- 1.0	1.57	13.5
1.0- 2.0	1.59	13.6
2.0- 3.0	1.6	13.8
3.0- 4.0	1.46	12.6
4.0- 5.0	1.24	10.6
5.0- 6.0	1.15	9.85
6.0- 7.0	1.24	10.6
7.0- 8.0	1.42	12.2
8.0-9.0	1.51	12.9
9.0- 10.0	1.38	11.9
10.0- 11.0	1.47	12.6
11.0- 12.0	1.56	13.4
12.0- 13.0	1.53	13.2
13.0- 14.0	1.52	13.1
14.0- 15.0	1.57	13.5
15.0- 16.0	1.65	14.2
16.0- 17.0	1.49	12.8
17.0- 18.0	1.53	13.1
18.0- 19.0	1.56	13.4
19.0- 20.0	1.5	12.9



Gambar 4.5 Perbandingan TCP *Throughput* pada Topologi Mesh

Pada hasil uji coba didapatkan kenaikan performansi dari *throughput* sebelum dan sesudah implementasi metode. Sebelum implementasi seluruh paket akan melewati rute terpendek. Sebelum *route discovery* diimplementasikan rata-rata *throughput* mencapai 11.9 Gbits/sec dan setelah diimplementasikan naik menjadi rata-rata 13.0 Gbits/sec. Untuk lebih jelasnya dapat dilihat pada grafik perbandingan pada gambar 4.5. Dari analisa hasil uji coba tersebut terjadi kenaikan *throughput* sebesar 8.4%. Untuk Hasil perbandingan dengan Algoritma OSPF *Dijkstra* dapat dilihat pada gambar 4.6 berikut.



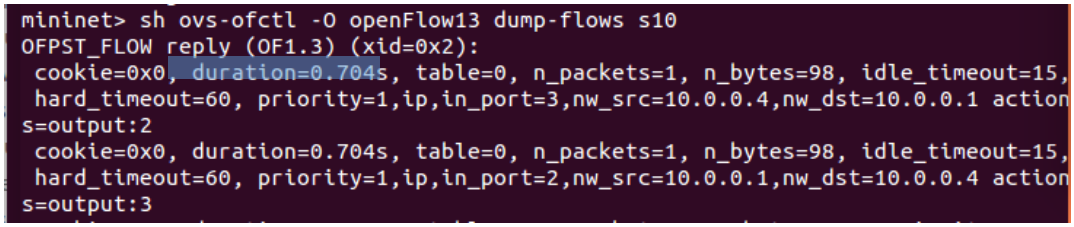
Gambar 4.6 Perbandingan *Throughput* dua buah Algoritma

Pada hasil uji coba dengan algoritma pembanding, algoritma *Dijkstra* menghasilkan rata-rata *throughput* 11.5 Gbps. *Throughput* rata-rata yang dihasilkan algoritma *Johnson* setelah pemilihan rute alternatif masih lebih unggul. Algoritma OSPF konvensional menunjukkan performa yang cukup baik ditandai dengan minimnya *throughput* yang turun secara signifikan dalam uji coba, walaupun secara rata-rata *throughput* mengalami penurunan. Hal dikarenakan rute alternatif lain yang dipilih tidak sedang digunakan untuk pengiriman paket. Paket akan dilewatkan melalui rute yang lebih jauh dari sebelumnya yang hanya berbanding 1 *hop* sehingga kenaikan *throughput* tidak terlalu signifikan. Sehingga dapat disimpulkan metode *route discovery* dengan algoritma *Johnson* lebih unggul dari

algoritma OSPF konvensional. Kenaikan *throughput* dapat terjadi jika rute alternatif yang dilalui tidak terlalu jauh dibandingkan rute sebelumnya. Sangat mungkin jika *controller* tidak menemukan alternatif rute terpendek dan harus melewati rute yang jaraknya lebih jauh maka dapat mengakibatkan penurun *throughput* yang signifikan.

4.1.4 Uji Coba Response Time Topologi *Mesh*

Pada uji coba ini akan diketahui berapa waktu yang dibutuhkan masing-masing algoritma *routing* saat melakukan pengiriman paket pertama kali dari *source* ke *destination*. *Response Time* diambil dari *latency* dari pengiriman *ping* paket pertama yang dapat ditunjukkan bahwa hasil *routing* berhasil terinstall di *switch*, dimana hasil instalasi berupa *flow* antara H1 (10.0.0.1) dan H2 (10.0.0.2). Uji coba *response time* dapat dilihat pada gambar 4.7 berikut.



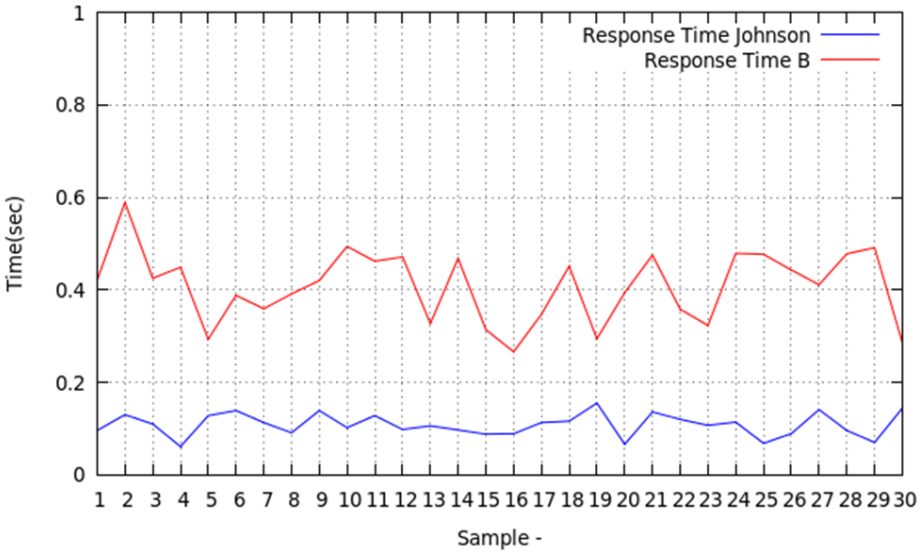
Gambar 4.7 Uji coba *response time* Topologi *mesh*

Untuk pengujian *response time* dilakukan filtering paket yang masuk pada *switch* yang sudah ditentukan. Pada pengujian ini melalui *OpenFlow switch* 4 (S4) dimana *switch* 4 terkoneksi langsung dengan *host destination*. Pengujian *Response Time* dilakukan 30 kali pada topologi. Untuk perbandingan hasil uji coba dapat dilihat pada tabel 4.3 dan akan diolah dalam bentuk grafik pada gambar 4.8 berikut.

Tabel 4.6 *Response Time* pada Topologi *Mesh*

Sampel ke-	Algoritma *Johnson* (sec)	Algoritma B (sec)
1	0.096	0.421
2	0.130	0.589
3	0.110	0.425
4	0.061	0.449
5	0.128	0.293
6	0.139	0.388

7	0.113	0.359
8	0.091	0.391
9	0.139	0.420
10	0.102	0.494
11	0.128	0.462
12	0.098	0.471
13	0.106	0.327
14	0.097	0.468
15	0.088	0.314
16	0.089	0.266
17	0.113	0.347
18	0.116	0.451
19	0.155	0.294
20	0.066	0.394
21	0.136	0.475
22	0.120	0.358
23	0.107	0.323
24	0.114	0.479
25	0.068	0.477
26	0.089	0.443
27	0.141	0.411
28	0.096	0.478
29	0.070	0.491
30	0.144	0.286
Rata-rata	0.108	0.408

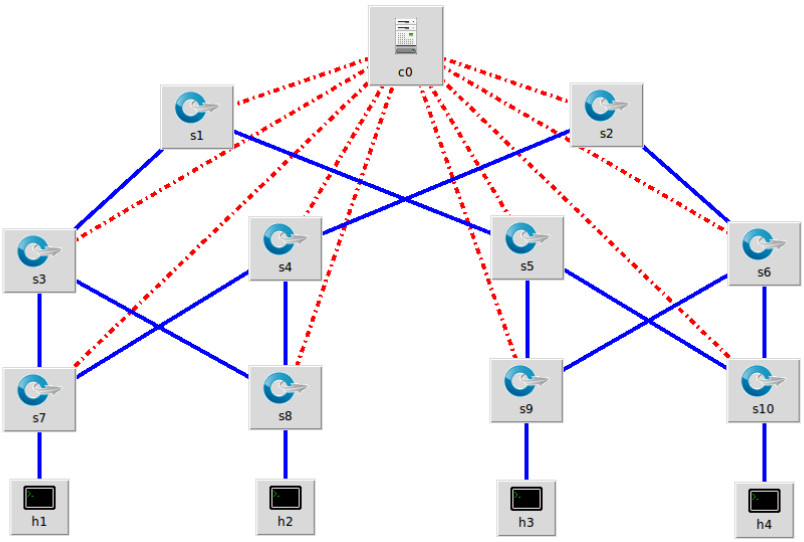


Gambar 4.8 Grafik *response time* Topologi *mesh*

Dari hasil uji coba *response time* pada masing-masing algoritma yang dilakukan sebanyak 30 kali, dapat dilihat performansi algoritma *Johnson* lebih cepat dibanding algoritma B yaitu *Dijkstra* saat pertama kali melakukan pencarian rute. Dengan rata-rata waktu yang dibutuhkan algoritma *Dijkstra* selama 0.408 detik dengan waktu paling lama yaitu pada 0.589 detik. Sedangkan algoritma *Johnson* membutuhkan waktu rata-rata 0.108 detik dengan waktu paling lama 0.155 detik. Algoritma *Johnson* dengan sifat *all pair shortest path* yang memandingkan semua pasang *node* terlebih dahulu. Sedangkan algoritma *Dijkstra* merupakan algoritma *greedy* dimana algoritma ini akan mencari nilai sementara antar *node* pada setiap langkahnya. Hal ini dapat disimpulkan performa algoritma *Johnson* lebih baik saat melakukan pertama kali mengirimkan paket dikarenakan memiliki *response time* lebih cepat.

4.2 Hasil Uji Coba Topologi *Fat-tree*

Pada skenario kedua dengan topologi *fat-tree* dibuat dengan entitas yang lebih besar dari sebelumnya dengan 10 *switch* dengan 4 *host* dan 1 *controller*. Dengan menggunakan konfigurasi *openflow* versi 1.3 sebagai penghubung antara *ryu controller* dan *OpenFlow switch*. Uji coba dilakukan pada *host* H1 dengan pengiriman data menuju *host* H4, karena jarak antar dua buah *host* saling berjauhan. Untuk lebih jelasnya visualisasi topologi dapat dilihat pada gambar 4.9 berikut.



Gambar 4.9 Skenario uji coba topologi *fat-tree*

4.2.1 Uji Coba *Route Discovery Fat-tree*

Pada uji coba ini sama dengan pengujian pada skenario pertama dilakukan pengujian kemampuan algoritma *routing* dalam melakukan *route discovery* dan *path available* dari seluruh *host*. Pada topologi ini terdapat banyak alternatif *path* antar *host*. Contoh *Path available* yang termonitor pada sisi *control plane* dari *host* H1, H2, H3 menuju *host* H4 dapat dilihat pada gambar 4.10 berikut.

```
Available paths from  7  to  10  :  [[7, 4, 2, 6, 10], [7, 4, 2, 6, 9, 5, 10], [
7, 4, 8, 3, 1, 5, 10], [7, 4, 8, 3, 1, 5, 9, 6, 10], [7, 3, 1, 5, 10], [7, 3, 1,
 5, 9, 6, 10], [7, 3, 8, 4, 2, 6, 10], [7, 3, 8, 4, 2, 6, 9, 5, 10]]
[7, 4, 2, 6, 10] cost =  4
Available paths from  8  to  10  :  [[8, 4, 7, 3, 1, 5, 10], [8, 4, 7, 3, 1, 5,
9, 6, 10], [8, 4, 2, 6, 10], [8, 4, 2, 6, 9, 5, 10], [8, 3, 7, 4, 2, 6, 10], [8,
 3, 7, 4, 2, 6, 9, 5, 10], [8, 3, 1, 5, 10], [8, 3, 1, 5, 9, 6, 10]]
[8, 4, 2, 6, 10] cost =  4
Available paths from  9  to  10  :  [[9, 6, 10], [9, 6, 2, 4, 7, 3, 1, 5, 10], [
9, 6, 2, 4, 8, 3, 1, 5, 10], [9, 5, 10], [9, 5, 1, 3, 7, 4, 2, 6, 10], [9, 5, 1,
 3, 8, 4, 2, 6, 10]]
[9, 6, 10] cost =  2
```

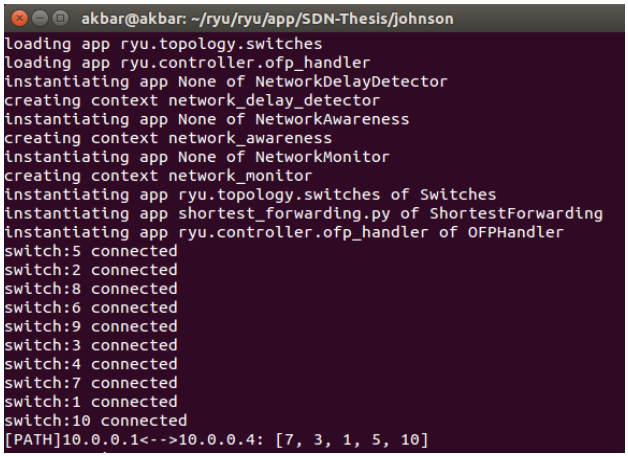
Gambar 4.10 *Fat-tree Available path pada Ryu controller*

Secara *default* algoritma *routing* akan memilih *shortest path* dari topologi. Dikarenakan link pada topologi *fat-tree* lebih banyak dibandingkan topologi sebelumnya maka rute yang tersedia untuk *source host* menuju ke *destination host* akan semakin banyak pula. Pada uji coba ini penulis mengambil sampel rute yang tersedia seperti pada gambar 4.10, dikarenakan skenario pengujian akan dilakukan komnunikasi antara H1 sebagai *host source* dan H4 sebagai *host destination*. *Host* H2 dan H3 menjadi *host* yang men-*trigger congestion* dengan mengirimkan sejumlah paket pada *host destination*. Pada uji coba tetap dilakukan pengujian fungsional dua buah algoritma yaitu algoritma *Johnson* sebagai algoritma yang diusulkan dan algoritma *Dijkstra* sebagai pembanding. Pada pengujian *route discovery* kedua algoritma ini tetap sama, dapat menentukan rute yang akan dilalui paket dan sama-sama memilih rute terpendek dengan memilih rute *switch* yang sama. Untuk lebih jelasnya dapat dilihat pada tabel 4.7 berikut.

Tabel 4.7 *Available route* pada topologi *fat-tree*

Node Asal	Node Tujuan	*Route Available*	*Shorthest Path*	*Alternate Route*
H1	H4	7-4-2-6-10	7-4-2-6-10	7-3-1-5-10
		7-3-1-5-10		
		7-3-1-5-9-6-10		
		7-3-8-4-2-6-10		
		7-4-2-6-9-5-10		
		7-4-8-3-1-5-10		
		7-3-8-4-2-6-9-5-10		
		7-4-8-3-1-5-9-6-10		
H2	H4	8-4-2-6-10	8-4-2-6-10	-
		8-3-1-5-10		
		8-3-1-5-9-6-10		
		8-3-7-4-2-6-10		
		8-4-2-6-9-5-10		
		8-4-7-3-1-5-10		
		8-3-7-4-2-6-9-5-10		
		8-4-7-3-1-5-9-6-10		
H3	H4	9-6-10	9-6-10	-
		9-5-10		
		9-6-2-4-7-3-1-5-10		
		9-6-2-4-8-3-1-5-10		
		9-5-1-3-7-4-2-6-10		
		9-5-1-3-8-4-2-6-10		

Sebelum diimplementasikan *route discovery host source* H1 melakukan pengiriman data menuju *host* H4. *Controller* akan memilih secara *random path* awal karena ada 2 *path* dengan *cost* yang sama yakni 7-4-2-6-10 dan 7-3-1-5-10. Dalam uji coba *path* awal yang dipilih H1 adalah 7-4-2-6-10. Pada pengujian ini juga dilakukan pengiriman data dari *host* H2 menuju H4 dan mendapatkan hasil pencarian rute 8-4-2-6-10, dan *host* H3 menuju H4 dengan rute 9-6-10. Pada skenario ini rute 4-2-6-10 terjadi *congestion* dikeranakan *host* mengirim pada melalui rute dan melalui *switch* tersebut. Saat *controller* menentukan rute alternatif dapat dapat dilihat pada gambar 4.11 berikut.



Gambar 4.11 *Alternate route* pada topologi fat-tree

Dari hasil uji coba diatas *controller* dapat menemukan rute alternatif pada topologi *fat-tree* yang sama dengan rute sebelumnya sehingga paket dilewatkan rute alternatif yang tersedia yaitu melalui switch 7-3-1-5-10. Pada kasus ini *controller* akan mendeteksi terjadinya *congestion* ditandai dengan terjadinya peningkatan *latency* maka paket akan dilewatkan rute yang telah dipilih dengan ketentuan rute yang jarak dan *cost* sama dengan rute sebelum terjadi *congestion*. Sehingga dapat disimpulkan bahwa algoritma *Johnson* dapat melakukan pencarian rute pada topologi *fat-tree* yang tersedia dan dapat memilih rute terpendek beserta rute alternatif yang akan digunakan pada simulasi skenario pengujian.

4.2.2 Uji Coba *Latency Fat-tree*

Pada pengujian *latency* dilakukan pengujian dengan melakukan perbandingan sebelum dan setelah diimplementasikannya *route discovery* menggunakan algoritma *Johnson*. Pada skenario ini *host* H1, akan melakukan pengiriman data menuju *host destination* H4 dengan ukuran maksimal paket yang dikirim sebesar 20000-byte untuk memberikan beban pada saat pengiriman paket. H2 dan H3 juga akan melakukan pengiriman yang akan memadati rute sehingga akan terjadi *congestion* pada rute tertentu. Pengiriman dilakukan sebanyak 100 kali menggunakan tool *ping*. Berikut adalah data yang didapatkan saat sebelum dan sesudah metode dapat dilihat pada tabel 4.8 dan 4.9 dan disajikan dalam bentuk grafik pada gambar 4.12.

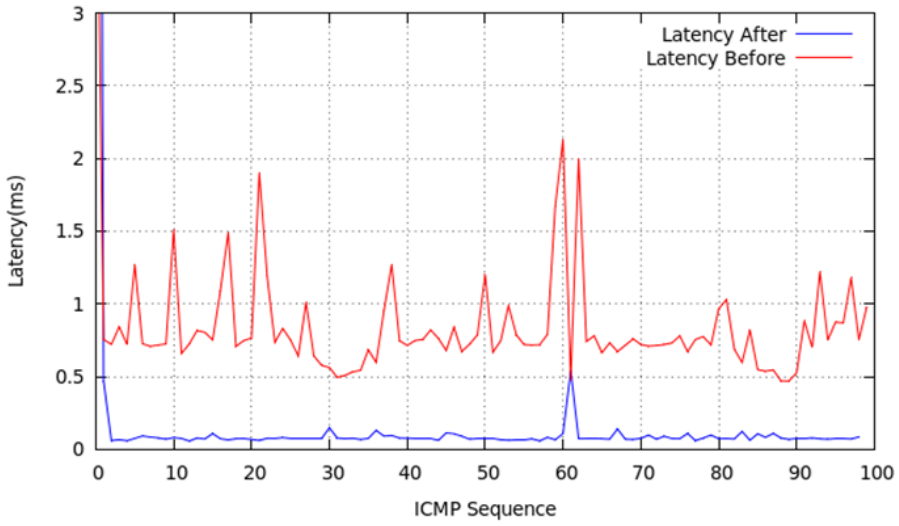
Tabel 4.8 Hasil Uji Coba *Latency* Sebelum *Routing* pada Topologi *Fat-tree*

Min (ms)	Avg (ms)	Max (ms)	Mdev (ms)
0.708	1.202	4.830	1.286
0.659	0.930	1.510	0.322
0.577	0.905	1.900	0.395
0.495	0.689	1.270	0.248
0.672	0.750	0.839	0.055
0.667	0.901	1.670	0.315
0.479	0.967	2.130	0.585
0.671	0.730	0.780	0.033
0.469	0.667	1.030	0.204
0.050	0.067	0.079	0.010
Avg 0.545	Avg 0.781	Avg 1.604	Avg 0.345

Tabel 4.9 Hasil Uji Coba *Latency* Setelah *Routing* pada Topologi *Fat-tree*

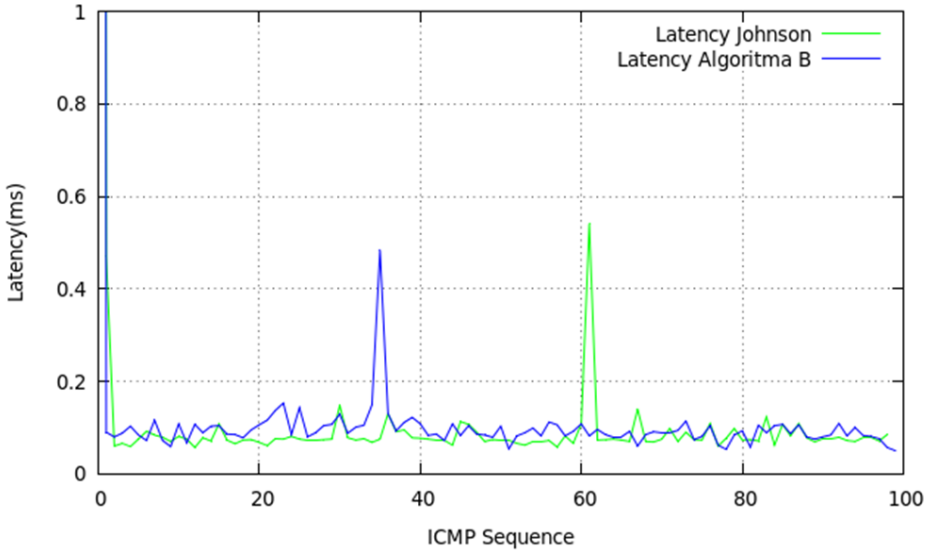
Min (ms)	Avg (ms)	Max (ms)	Mdev (ms)
0.059	1.596	14.9	4.676
0.057	0.076	0.108	0.013
0.061	0.073	0.081	0.005
0.068	0.091	0.148	0.027
0.062	0.082	0.113	0.017
0.058	0.069	0.082	0.007
0.069	0.129	0.541	0.146
0.060	0.082	0.109	0.015
0.062	0.085	0.123	0.021
0.070	0.076	0.085	0.005
Avg 0.063	Avg 0.236	Avg 1.629	Avg 0.493

Pada hasil uji coba didapatkan peningkatan performansi dari *latency* sebelum dan sesudah diimplementasikannya *route discovery*. Sebelum implementasi metode seluruh paket akan melewati rute terpendek yang sama sehingga menyebabkan terjadinya kepadatan pada rute yang dilalui. Dapat dilihat grafik pada gambar 4.12 sebelum implementasi grafik cenderung tidak stabil. Sebelum diimplementasikan rata-rata *latency* yang didapatkan 0.781 ms dengan maksimal *latency* 4.830 ms.



Gambar 4.12 Grafik perbandingan *Latency* topologi *Fat-tree*

Namun berbeda saat setelah metode diimplementasikan *latency* yang dihasilkan cenderung lebih stabil dan mengalami penurunan dengan rata-rata *latency* sebesar 0.236 ms. Dari hasil uji coba tersebut penurunan *latency* berbanding 69.7% dari *latency* sebelumnya. Nilai maksimal *latency* setelah metode sebesar 14.9 ms, hal ini dikarenakan pengaruh dari *response time* pada algoritma *routing* untuk mencari rute baru pada saat terjadi pada rute terpendek, sehingga waktu yang dibutuhkan cenderung lebih lama karena *controller* memerlukan waktu untuk mengadaptasi rute yang baru.



Gambar 4.13 Grafik perbandingan *Latency* dua Algoritma

Perbandingan hasil uji coba *latency* setelah metode diimplementasikan juga akan dibandingkan dengan Algoritma OSPF konvensional dengan dasar Algoritma *Dijkstra* sebagai *route discovery*. Yang hasil uji coba grafiknya dapat dilihat pada gambar 4.13. Rata-rata *latency* yang dihasilkan algoritma pembanding sebesar 0.304 ms. Rata-rata *latency* yang dihasilkan lebih tinggi dari algoritma *Johnson* namun lebih rendah dari sebelum *route discovery* diterapkan. *Latency* pada pengiriman paket yang dihasilkan tergolong rendah dan stabil karena rute yang dilalui tidak dilewati oleh paket lain. Sehingga dapat disimpulkan mekanisme *route discovery* dengan dapat mengurangi *latency* dan mempertahankan *latency* agar lebih stabil saat pengiriman paket. Pada uji coba ini controller dapat menemukan rute alternatif dengan hop yang sama sehingga performansi meningkat karena pada rute sebelumnya dengan hop yang sama terjadi kepadatan sehingga terjadi peningkatan jumlah *latency*. Ada kemungkinan jika *controller* tidak dapat menemukan rute alternatif yang sama dengan rute sebelumnya dan terpaksa melalui rute yang lebih jauh maka *latency* yang dihasilkan akan semakin tinggi.

4.2.3 Uji Coba *Throughput Fat-tree*

Pada uji coba *throughput* dilakukan pengujian dengan melakukan perbandingan sebelum dan setelah diimplementasikannya metode. Akan dibandingkan juga performansi setelah metode diimplementasikan dengan algoritma OSPF konvensional. Pada skenario ini *host* H1 bertindak sebagai *client* dan *host* H4 bertindak *server*. Uji coba dilakukan dengan waktu 100 detik dengan interval pengiriman 1 detik sekali menggunakan tool *iperf*. Berikut adalah data sampel 20 detik pertama yang didapatkan saat sebelum dan sesudah metode dapat dilihat pada tabel 4.10 dan 4.11.

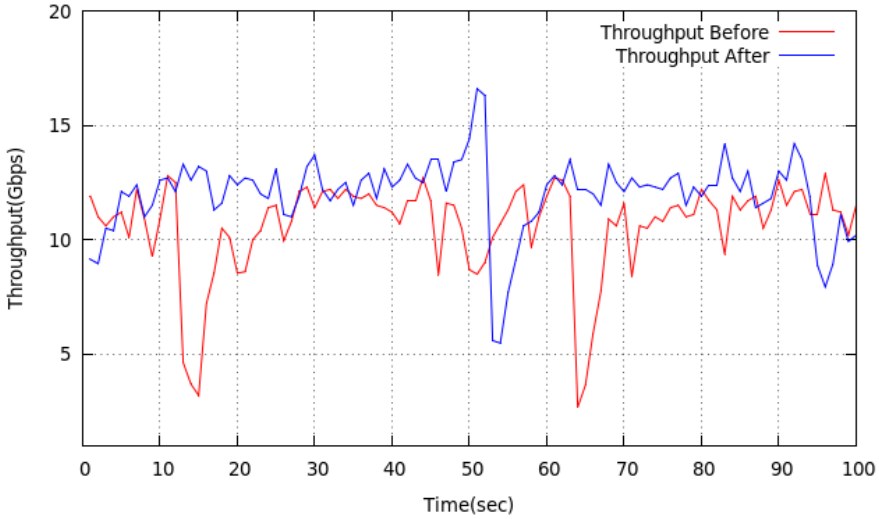
Tabel 4.10 TCP *Throughput* Sebelum CA Topologi *Fat-tree*

Interval (Sec)	Transfer Data (MBytes-GBytes)	Bandwidth (Gbits/sec)
0.0- 1.0	1.39	11.9
1.0- 2.0	1.28	11.0
2.0- 3.0	1.23	10.6

3.0- 4.0	1.28	11.0
4.0- 5.0	1.30	11.2
5.0- 6.0	1.17	10.1
6.0- 7.0	1.43	12.2
7.0- 8.0	1.27	10.9
8.0-9.0	1.08	9.26
9.0- 10.0	1.27	10.9
10.0- 11.0	1.49	12.8
11.0- 12.0	1.45	12.5
12.0- 13.0	550M	4.62
13.0- 14.0	440M	3.69
14.0- 15.0	380M	3.19
15.0- 16.0	858M	7.20
16.0- 17.0	1.23	10.5
17.0- 18.0	1.18	10.1
18.0- 19.0	1.39	11.9
19.0- 20.0	1.28	11.0

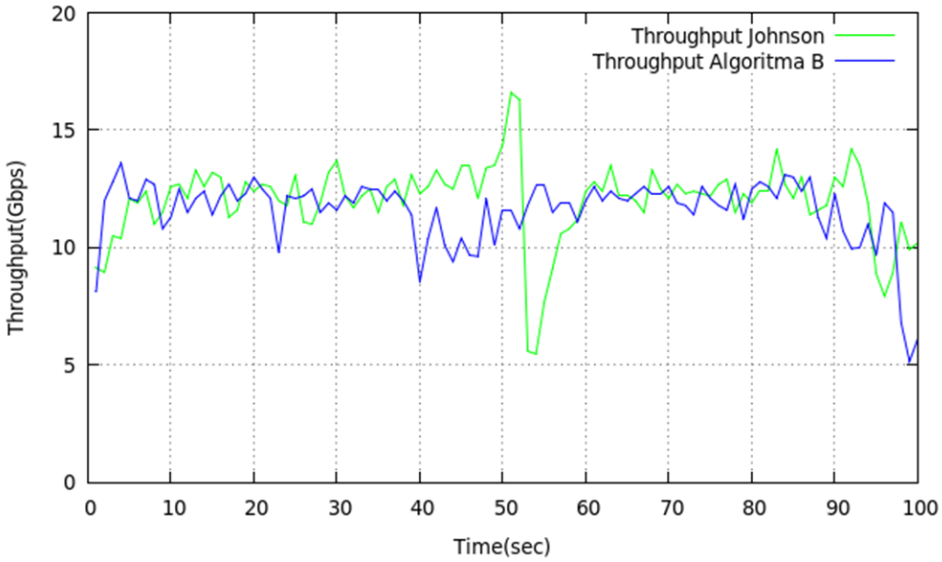
Tabel 4.11 TCP *Throughput* Sebelum CA Topologi *Fat-tree*

Interval (Sec)	Transfer Data (Gbytes)	Bandwidth (Gbits/sec)
0.0- 1.0	1.06	9.14
1.0- 2.0	1.04	8.95
2.0- 3.0	1.22	10.5
3.0- 4.0	1.21	10.4
4.0- 5.0	1.41	12.1
5.0- 6.0	1.39	11.9
6.0- 7.0	1.44	12.4
7.0- 8.0	1.28	11.0
8.0-9.0	1.33	11.5
9.0- 10.0	1.47	12.6
10.0- 11.0	1.48	12.7
11.0- 12.0	1.41	12.1
12.0- 13.0	1.55	13.3
13.0- 14.0	1.46	12.6
14.0- 15.0	1.54	13.2
15.0- 16.0	1.51	13.0
16.0- 17.0	1.31	11.3
17.0- 18.0	1.35	11.6
18.0- 19.0	1.49	12.8
19.0- 20.0	1.44	12.4



Gambar 4.14 Perbandingan *Throughput* Topologi *Fat-tree*

Pada hasil uji coba didapatkan peningkatan performansi dari *throughput* sebelum dan sesudah diimplementasikannya *route discovery*. Pada gambar 4.14 Sebelum diimplementasikan rata-rata *throughput* mencapai 10.6 Gbites/sec dan meningkat menjadi rata-rata 12.0 Gbites/sec. Dari analisa hasil uji coba terjadi peningkatan presentase *throughput* sebesar 11.6%. Peningkatan ini karena rute alternatif yang dipilih yang jarak hopnya sama dengan rute terpendek sebelumnya. Untuk Hasil perbandingan dengan Algoritma OSPF *Dijkstra* dapat dilihat pada gambar 4.15 berikut.

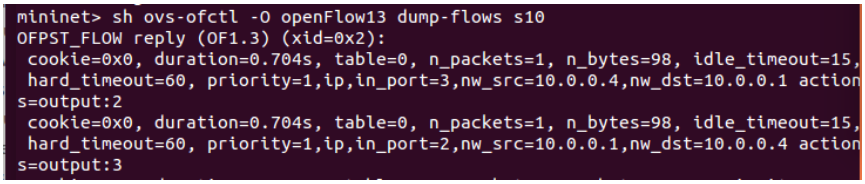


Gambar 4.15 Perbandingan *Throughput* antar Algoritma

Pada hasil uji coba dengan algoritma pembanding, algoritma *Dijkstra* menghasilkan rata-rata *throughput* 11.6 Gbps. *Throughput* rata-rata yang dihasilkan algoritma *Johnson* setelah pemilihan rute alternatif masih lebih unggul. Algoritma OSPF konvensional menunjukkan performa yang cukup baik ditandai dengan minimnya *throughput* yang turun secara signifikan dalam uji coba, walaupun secara rata-rata *throughput* mengalami penurunan disbanding algoritma *Johnson*. Karena rute alternatif yang dipilih tidak dilewati oleh paket yang dikirim dari *host* lain maka terjadilah peningkatan *throughput*. Sehingga dapat disimpulkan *route discovery* dengan algoritma *shortest path* dapat meningkatkan kecepatan *throughput* jika rute alternatif yang dipilih tepat dan sesuai. Karena itu pemilihan arsitektur dan topologi jaringan juga akan mempengaruhi performansi.

4.2.4 Uji Coba Response Time *Fat-tree*

Pada uji coba ini akan diketahui berapa waktu yang dibutuhkan masing-masing algoritma *routing* saat melakukan pengiriman paket pertama kali dari *source* ke *destination*. *Response Time* diambil dari *latency* dari pengiriman *ping* paket pertama yang dapat ditunjukkan bahwa hasil *routing* berhasil terinstall di *switch*, dimana hasil instalasi berupa *flow* antara H1 (10.0.0.1) dan H4 (10.0.0.4). Uji coba *response time* dapat dilihat pada gambar 4.16 berikut.

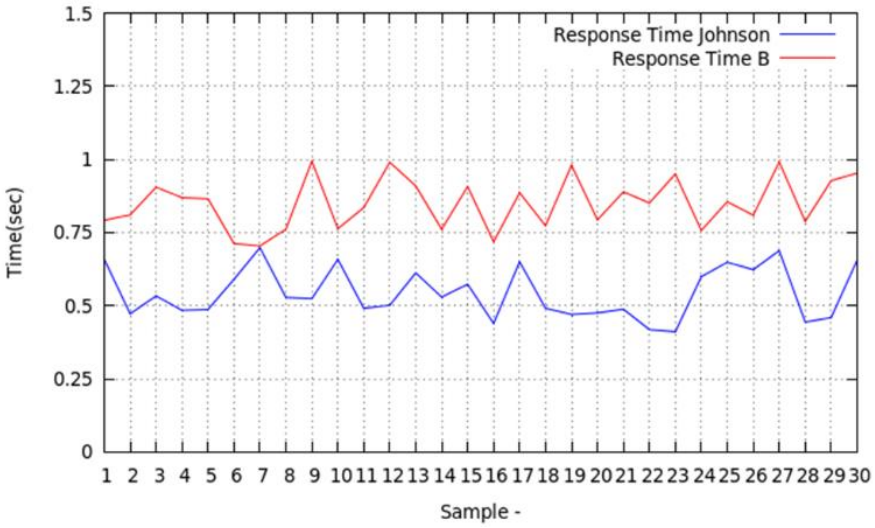

```
mininet> sh ovs-ofctl -O openFlow13 dump-flows s10
OFPST_FLOW reply (OF1.3) (xid=0x2):
 cookie=0x0, duration=0.704s, table=0, n_packets=1, n_bytes=98, idle_timeout=15,
 hard_timeout=60, priority=1,ip,in_port=3,nw_src=10.0.0.4,nw_dst=10.0.0.1 action
s=output:2
 cookie=0x0, duration=0.704s, table=0, n_packets=1, n_bytes=98, idle_timeout=15,
 hard_timeout=60, priority=1,ip,in_port=2,nw_src=10.0.0.1,nw_dst=10.0.0.4 action
s=output:3
```

Gambar 4.16 Uji coba *response time* Topologi *Fat-tree*

Untuk pengujian *response time* dilakukan menggunakan kemampuan *OpenFlowSwitch* untuk filtering paket yang masuk pada *switch* yang sudah ditentukan. Pada pengujian ini melalui *OpenFlow switch* 10 (S10) dimana *switch* 10 terkoneksi langsung dengan *host destination*. Pengujian *Response Time* dilakukan 30 kali pada topologi. Untuk perbandingan hasil uji coba dapat dilihat pada tabel 4.12 dan akan diolah dalam bentuk grafik pada gambar 4.17 berikut.

Tabel 4.12 *Response Time* Topologi *Fat-tree*

Sampel ke-	Algoritma *Johnson* (sec)	Algoritma B (sec)
1	0.659	0.792
2	0.472	0.810
3	0.533	0.905
4	0.484	0.869
5	0.487	0.865
6	0.590	0.712
7	0.698	0.704
8	0.528	0.761
9	0.524	0.994
10	0.658	0.763
11	0.491	0.835
12	0.501	0.990
13	0.612	0.909
14	0.529	0.761
15	0.573	0.907
16	0.439	0.718
17	0.650	0.886
18	0.491	0.773
19	0.470	0.980
20	0.475	0.793
21	0.488	0.889
22	0.418	0.851
23	0.411	0.950
24	0.599	0.757
25	0.648	0.855
26	0.623	0.809
27	0.688	0.991
28	0.444	0.788
29	0.459	0.927
30	0.655	0.953
Rata-rata	0.543	0.850



Gambar 4.17 Grafik *response time* Topologi *Fat-tree*

Dari hasil uji coba *response time* pada masing-masing algoritma yang dilakukan sebanyak 30 kali, dapat dilihat performansi algoritma *Johnson* lebih cepat dengan dibanding algoritma B yaitu *Dijkstra* saat pertama kali melakukan pengiriman paket. Dengan rata-rata waktu yang dibutuhkan algoritma *Dijkstra* selama 0.850 detik dengan waktu paling lama yaitu pada 0.994 detik. Sedangkan algoritma *Johnson* membutuhkan waktu rata-rata 0.543 detik dengan waktu paling lama 0.698 detik. Perbedaan yang terjadi pada pengujian dengan jumlah *switch* yang lebih banyak menunjukkan performansi algoritma *Johnson* lebih cepat dengan penurunan 17.8% dibandingkan algoritma *Dijkstra*. Algoritma *Johnson* dengan sifat *all pair shortest path* yang memandingkan semua pasang *node* terlebih dahulu. Sedangkan algoritma *Dijkstra* merupakan algoritma *greedy* dimana algoritma ini akan mencari nilai sementara antar *node* pada setiap langkahnya. Hal ini dapat disimpulkan performa algoritma *Johnson* lebih baik saat melakukan pengiriman paket dikarenakan memiliki *response time* lebih cepat walaupun dengan jumlah *switch* dan rute yang lebih banyak.

BAB 5
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Pengujian dan analisis yang telah dilakukan menghasikan beberapa kesimpulan penelitian, yaitu

1. *Control plane* merupakan *remote controller* yang dirancang menggunakan suatu algoritma *routing* untuk menentukan *route discovery*. Algoritma *Johnson* dipilih karena menggabungkan dua algoritma, yaitu Algoritma *Bellman-Ford* yang perhitungannya berdasarkan jumlah hop dan Algoritma *Dijkstra* menentukan rute terpendek berdasarkan jarak antar node. Algoritma ini dijalankan dengan menambahkan sebuah node baru dengan nilai 0. Algoritma *Bellman-Ford* dijalankan untuk menghitung *hop* dengan node baru sebagai titik awal. Setelah ditemukan Lalu node baru tersebut dihapus dari topologi dan Algoritma *Dijkstra* dijalankan untuk menghitung *shortest path* dari setiap node ke node lainnya dengan bobot yang diberikan.
2. Mekanisme *route discovery* dapat diimplementasikan pada lingkungan *Software Defined Network* dengan menggunakan *Shortest Path Algorithm*. Algoritma *Johnson* digunakan sebagai algoritma *routing* dengan mempertimbangkan parameter *delay* pada saat pengiriman paket. Pemilihan rute alternatif yang dilalui paket saat terjadi *congestion* adalah rute yang *cost* dan *hop*-nya sama dengan rute utama sebelum terjadi *congestion*. Jika *controller* tidak dapat menemukan rute lain dengan *cost* yang sama maka algoritma *routing* akan memilih rute lain dengan pertimbangan rute yang dilalui tidak terlalu jauh atau rute terpendek kedua dari seluruh rute yang sudah ditemukan pada saat *route discovery*.
3. Hasil uji coba *response time* pada kedua skenario menunjukkan bahwa performansi dari Algoritma *Johnson* dalam pertama kali mengirimkan paket lebih cepat dibanding protokol OSPF konvensional yaitu Algoritma

Dijkstra dengan rata-rata 0.108 detik pada topologi *mesh* dan 0.543 detik pada topologi *fat-tree*. Hasil uji coba *latency* pada kedua skenario yang berbeda mendapatkan hasil yang sama yaitu terjadi penurunan *latency* pada saat pengiriman paket. Pada kedua skenario terjadi penurunan *latency*, pada topologi *mesh* presentase penurunan 11.5%. Pada topologi *Fat-tree* dengan persentase penurunan sebesar 69.7%. Pada grafik uji coba kedua buah skenario juga menunjukkan *latency* yang dihasilkan lebih stabil setelah metode diimplementasikan. Hasil perbandingan *latency* dengan protokol OSPF konvensional yaitu Algoritma *Dijkstra* menunjukkan algoritma *Johnson* masih tetap unggul. Hasil uji coba *throughput* pada kedua skenario mendapatkan hasil yang sama yaitu kenaikan *throughput*. Pada skenario topologi *mesh* presentase kenaikan sebesar 8.4% dan pada topologi *fat-tree* dengan presentase kenaikan 11.6%. Hasil perbandingan *Throughput* dengan protokol OSPF konvensional yaitu Algoritma *Dijkstra* menunjukkan algoritma *Johnson* masih tetap unggul.

5.2 Saran

Saran yang dapat diberikan dari hasil uji coba dan evaluasi adalah sebagai berikut:

1. Penelitian selanjutnya dapat menggunakan algoritma *routing* lain sebagai dasar pemilihan rute.
2. Algoritma *routing* juga dapat diimplementasikan pada *controller* yang berbeda untuk mengetahu performansi dari tiap-tiap *controller*.
3. Dapat menerapkan *Multipath routing* dengan menggunakan algoritma *Johnson* sebagai pencarian rutenya.

DAFTAR PUSTAKA

[1] D. Kreutz, F. M. V. Ramos, P. E. Verissimo, C. E. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-Defined Networking : A Comprehensive Survey," *Proc. IEEE*, vol. 103, no. 1, pp. 14–76, 2015.

[2] T. D. Nadeau and K. Gray, *SDN: Software Defined Networks*, First Edit. O'Reilly Media, Inc., 2013.

[3] W. Chen, F. Ren, J. Xie, C. Lin, K. Yin, and F. Baker, "Comprehensive Understanding of TCP Incast Problem," *IEEE Conf. Comput. Commun.*, pp. 1688–1696, 2015.

[4] M. Welzl, *Network Congestion Control Managing Internet Traffic*. Chichester: John Wiley & Sons, Ltd, 2005.

[5] F. H. Saputra and R. M. Ijtihadie, "Survei mekanisme congestion kontrol pada transmission control protocol di software defined network," *JUTI J. Ilm. Teknol. Inf.*, vol. 6, no. 1, pp. 1–9, 2018.

[6] Y. Lu and S. Zhu, "SDN-based TCP Congestion Control in Data Center Networks," in *IEEE 34th International Performance Computing and Communications Conference (IPCCC)*, 2015, p. 15.

[7] H. Xin, "Introduction of Centralized and Distributed Routing Protocols," *2011 Int. Conf. Electron. Commun. Control*, pp. 2698–2701, 2011.

[8] S. Waleed, M. Faizan, M. Iqbal, and M. I. Anis, "Demonstration of Single Link Failure Recovery using Bellman Ford and Djikstra Algorithm in SDN," pp. 0–3, 2017.

[9] H. M. Abu-Ryash and A. Tamimi A., "Comparison Studies for Different Shortest path Algorithms," *Int. J. Comput. Technol.*, vol. 14, no. May, pp. 5980–5986, 2015.

[10] A. Nastiti, A. Rakhmatsyah, and M. A. Nugroho, "Link Failure Emulation with Dijkstra and Bellman-Ford Algorithm in Software Defined Network Architecture (Case Study : Telkom University Topology)," *2018 6th Int. Conf. Inf. Commun. Technol.*, vol. 0, no. c, pp. 135–140, 2018.

[11] M. Glabowski, B. Musznicki, and P. Zwierzykowski, "Review and Performance Analysis of Shortest Path Problem Solving Algorithms," *Int. J. Adv. Softw.*, vol. 7, no. July, pp. 20–30, 2014.

[12] H. Kim and N. Feamster, "Improving Network Management with Software Defined Networking," *IEEE Communications Magazine*, no. February, pp. 114–119, 2013.

[13] M. S. Olimjonovich, “Software Defined Networking : Management of network resources and data flow,” *2016 Int. Conf. Inf. Sci. Commun. Technol.*, pp. 1–3, 2016.

[14] J. H. Cox *et al.*, “Advancing Software-Defined Networks : A Survey,” vol. 5, pp. 25487–25526, 2017.

[15] A. Prajapati, A. Sakadasariya, and J. P. Computer, “Software Defined Network : Future of Networking,” *2018 2nd Int. Conf. Inven. Syst. Control*, no. Icisc, pp. 1351–1354, 2018.

[16] S. Azodolmolky, *Software Defined Networking with OpenFlow*, First Edit. Birmingham, Mumbai: Packt Publishing Ltd., 2013.

[17] S. Sharma, D. Staessens, D. Colle, M. Pickavet, and P. Demeester, “OpenFlow : Meeting carrier-grade recovery requirements q,” *Comput. Commun.*, vol. 36, no. 6, pp. 656–665, 2013.

[18] N. Mckeown, T. Anderson, L. Peterson, J. Rexford, S. Shenker, and S. Louis, “OpenFlow : Enabling Innovation in Campus Networks,” *ACM SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[19] R. project Team, “Ryu SDN Framework,” *http://osrg.github.io/ryu-book/en/html/*, 2014. .

[20] J. F. Kurose and K. W. Ross, *COMPUTER NETWORKING A Top-Down Approach*. 2013.

[21] B. . R. S Anitha, “Network Reconfiguration for Loss Minimization by Using Johnson’s Algorithm,” *2018 4th Int. Conf. Electr. Energy Syst.*, pp. 680–684, 2018.

[22] D. B. Johnson, “Efficient Algorithms for Shortest Paths in Sparse Networks,” *J. Assoc. Comput. Mach.*, vol. 24, no. 1, pp. 1–13, 1977.

[23] Wikipedia, “Johnson’s algorithm,” *Wikipedia.org*, 2019. [Online]. Available: https://en.wikipedia.org/wiki/Johnson%27s_algorithm#cite_note-3. [Accessed: 22-Oct-2019].

[24] M. Ilhamsyah, R. M. Rumani, and S. N. Hertiana, “Simulasi dan Uji Kinerja Algoritma Johnson untuk Penentuan Rute Terbaik Pada Jaringan Software Defined Network,” *J. Sist. Komput.*, vol. 6, no. 2, pp. 77–79, 2016.

[25] Raj Jain, “Congestion control in computer networks: issues and trends,” *IEEE Netw. Mag.*, pp. 24–30, 1990.

[26] A. Srikanth, P. Varalakshmi, V. Somasundaram, and P. Ravichandiran, “Congestion Control Mechanism in Software Defined Networking by Traffic

Rerouting," *2018 Second Int. Conf. Comput. Methodol. Commun.*, no. Iccmc, pp. 55–58, 2018.

[27] S. Song, J. Lee, K. Son, H. Jung, and J. Lee, "A Congestion Avoidance Algorithm in SDN Environment," *2016 Int. Conf. Inf. Netw.*, pp. 420–423, 2016.

[28] A. M. Abdelmoniem and B. Bensaou, "Enforcing Transport-Agnostic Congestion Control in SDN-based Data Centers," in *IEEE 42nd Conference on Local Computer Networks*, 2017, pp. 120–136.

BIODATA PENULIS

Akbar Pandu Segara, lahir di Banyuwangi, 24 Agustus 1995. Penulis adalah anak Pertama dari dua bersaudara. Penulis menempuh pendidikan sekolah dasar di SDN 1 Karangsari - Banyuwangi pada tahun 2001-2006 lalu melanjutkan pendidikan sekolah menengah pertama di SMPN 1 Genteng – Banyuwangi pada tahun 2007-2009 dan penulis menempuh pendidikan menengah atas di SMA Negeri 2 Genteng – Banyuwangi pada tahun 2010-2012. Selanjutnya penulis melanjutkan pendidikan sarjana di Program Studi Teknik Komputer pada Jurusan Teknik Informatika, Fakultas Ilmu Komputer, Universitas Brawijaya Malang pada tahun 2013 dan menyelesaikan studi pada tahun 2017. Saat pendidikan sarjana, penulis mengambil bidang minat Jaringan. Pada saat pendidikan sarjana penulis aktif dalam organisasi dan menjadi staff Kementrian Administrasi BEM TIIK Kabinet Bersatu III periode 2015/2016 Fakultas Ilmu Komputer, Universitas Brawijaya Malang. Kemudian penulis melanjutkan studi pascasarjana di Departemen Teknik Informatika, Fakultas Teknologi Elektro dan Informatika Cerdas, Institut Teknologi Sepuluh Nopember Surabaya pada tahun 2018. Dalam menyelesaikan pendidikan pascasarjana, penulis mengambil bidang minat Komputasi Berbasis Jaringan (KBJ). Penulis dapat dihubungi melalui *email*: akbarpandu94@gmail.com.