ANALISIS *FAIL PATH* PADA ARSITEKTUR *SOFTWARE DEFINED NETWORK* MENGGUNAKAN *DIJKSTRA ALGORITHM*

SKRIPSI

KEMINATAN TEKNIK KOMPUTER

Untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun oleh:
Eka Putri Aprilianingsih
NIM: 125150301111012

PROGRAM STUDI TEKNIK INFORMATIKA
JURUSAN TEKNIK INFORMATIKA
FAKULTAS ILMU KOMPUTER
UNIVERSITAS BRAWIJAYA
MALANG
2017

PENGESAHAN

ANALISIS *FAIL PATH* PADA ARSITEKTUR *SOFTWARE DEFINED NETWORK* MENGGUNAKAN *DIJKSTRA ALGORITHM*

SKRIPSI

KEMINATAN TEKNIK KOMPUTER

Diajukan untuk memenuhi sebagian persyaratan
memperoleh gelar Sarjana Komputer

Disusun Oleh :
Eka Putri Aprilianingsih
NIM: 125150301111012

Skripsi ini telah diuji dan dinyatakan lulus pada
30 Maret 2017
Telah diperiksa dan disetujui oleh:

Dosen Pembimbing I	Dosen Pembimbing II
Rakhmadhany Primananda, S.T., M.Kom.	Aswin Suharsono, S.T., M.T.
NIK. 201609 860406 1 001	NIK. 201102 840919 1 001

Mengetahui
Ketua Jurusan Teknik Informatika

Tri Astoto Kurniawan, S.T.,M.T. Ph.D.
NIP. 19710518 200312 1 001

PERNYATAAN ORISINALITAS

Saya menyatakan dengan sebenar-benarnya bahwa sepanjang pengetahuan saya, di dalam naskah skripsi ini tidak terdapat karya ilmiah yang pernah diajukan oleh orang lain untuk memperoleh gelar akademik di suatu perguruan tinggi, dan tidak terdapat karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis disitasi dalam naskah ini dan disebutkan dalam daftar pustaka.

Apabila ternyata didalam naskah skripsi ini dapat dibuktikan terdapat unsur-unsur plagiasi, saya bersedia skripsi ini digugurkan dan gelar akademik yang telah saya peroleh (sarjana) dibatalkan, serta diproses sesuai dengan peraturan perundang-undangan yang berlaku (UU No. 20 Tahun 2003, Pasal 25 ayat 2 dan Pasal 70).

Malang, 30 Maret 2017

Eka Putri Aprilianingsih
NIM: 125150301111012

KATA PENGANTAR

Puji dan syukur penulis panjatkan kehadirat Allah SWT Yang Maha Pengasih lagi Maha Penyayang yang telah melimpahkan rahmat dan karunia-Nya, sehingga penulis dapat menyelesaikan skripsi yang berjudul "Analisis *Fail Path* Pada Arsitektur *Software Defined Network* Menggunakan *Dijkstra Algorithm*".

Keberhasilan penulis dalam menulis skripsi ini tentu tidak lepas dari pihak-pihak yang telah meluangkan waktu untuk memberikan bantuan dan dukungannya. Pada kesempatan ini penulis ingin menyampaikan terima kasih kepada:

1. Kedua Orangtua dan adik yang selalu mendukung, memberi doa dan memberi semangat agar penulis dapat menyelesaikan skripsi dengan lancar.
2. Rakhmadhany Primananda, S.T, M.Kom. dan Aswin Suharsono, S.T., M.T. selaku dosen pembimbing skripsi yang telah membimbing penulis dalam pengerjaan skripsi.
3. Seluruh civitas akademika Fakultas Ilmu Komputer Universitas Brawijaya yang telah banyak memberi bantuan dan dukungan selama penulis menempuh studi Informatika di Universitas Brawijaya dan selama penyelesaian skripsi.
4. Semua teman-teman Teknik Komputer dan saudara yang selalu mendukung, memberi semangat dan bantuan pikiran untuk penulis.

Penulis menyadari masih banyak kekurangan dan kelemahan dalam penyusunan skripsi ini. Oleh karena itu, penulis dengan kerendahan hati menerima kritik maupun saran yang sifatnya membangun dan berguna bagi kita semua.

Malang, 30 Maret 2017

Penulis

eka.put3.april@gmail.com

ABSTRAK

Software Defined Network (SDN) merupakan sebuah konsep baru yang digunakan untuk mengatasi masalah jaringan tradisional dengan melakukan pemisahan antara *control plane* dan *data plane* dalam suatu perangkat yang berbeda. Untuk komunikasi antara *control plane* dan *data plane* menggunakan protokol *openflow. SDN* memiliki beberapa kemampuan dalam banyak metode teknologi jaringan dan sudah banyak diimplementasikan antara lain untuk mekanisme *routing*. Di dalam *routing* terdapat beberapa masalah di antaranya kegagalan link (*fail path*). Ketika terjadi *fail path* maka sistem akan mencari jalur terpendek lain. Untuk menentukan jalur terpendek lain dengan menggunakan *dijkstra algorithm*. *Dijkstra algorithm* akan diimplementasikan menggunakan SDN kemudian dilakukan analisis. Implementasi ini menggunakan topologi mesh dan pyretic sebagai *controller*. Setelah dilakukan implementasi, selanjutnya program algoritma akan diuji dengan 3 skenario.

Didalam skenario terdapat 4 pengujian, yaitu pengujian topologi, *throughput*, *latency* dan *convergence*. Pengujian topologi dilakukan untuk menghasilkan jalur terpendek. Skenario 1, yaitu h1, h4, h3, h7 dengan total *cost* 10. Skenario 2, yaitu h1, h2, h3, h7 dengan *cost* 13. Skenario 3, yaitu h1, h4, h6, h7 dengan *cost* 18. Pengujian *throughput* dilakukan untuk menghitung paket yang sampai pada tujuan. Skenario 1 menghasilkan *throughput* paling banyak, yaitu 12.0 Gbits/sec. Skenario 2 yaitu 9.56 Gbits/sec. Skenario 3 yaitu 9.68 Gbits/sec. Pengujian latency untuk menghitung waktu yang diperoleh dari sebuah paket sampai ke tujuan. Skenario 1 menghasilkan *latency* paling tinggi, yaitu 0.162ms. Skenario 2 yaitu 0.190ms. Skenario 3 yaitu 0.150ms. Pengujian *convergence* untuk mengetahui perubahan waktu yang dibutuhkan ketika ada pemutusan *link.* Skenario 1 tidak terjadi *fail path* sehingga tidak ada hasil *convergence*. Skenario 2 yaitu 2.009ms. Skenario 3 yaitu 3.01ms.

Kata kunci: *software defined network*, *dijkstra algorithm*, *fail path*

ABSTRACT

Software Defined Network (SDN) is a new concept that is used to fix traditional network's issues by making the separation between the control plane and the data plane in many different network devices vendor. The communication between the control plane and the data plane use the OpenFlow Protocol. SDN has some abilitiesin many network technology methods, and has been widely implemented such as network routing. There are some problems in routing, for example a network link failure (file path). When there is a fail path arises, the system will look for another shortest paths. To define another shortest paths by using Dijkstra Dlgorithm. Dijkstra Algorithm will be implemented using SDN then will be analyzed. This implementation uses a mesh topology and Pyretic as the controller. After implementing the program, algorithms will be tested with three scenarios.

In these scenarios, there are four tests: the testing topology, throughput, latency and convergence. Tests conducted topology to generate the shortest path. Scenario 1, is h1, h4, h3, h7 with cost 10. Scenario 2, is h1, h2, h3, h7 with cost 13. Scenario 3, is h1, h4, h6, h7 with cost 18. The purpose of throughput testing is to count packets arrive at the destination side. Scenario 1 had maximum throughput at 12.0 Gbits / sec. Scenario 2 is 9.56 Gbits / sec. Scenario 3 is 9.68 Gbits / sec. Latency test is to calculate the time taken from a package to its destination. Scenario 1 produced the highest latency at 0.162*ms. Scenario 2 had 0.190ms. Scenario 3 had 0.150ms. Convergence test is to determine the changes in the time required when there is a link termination. Scenario 1 has no file path so there were no results in convergence. Scenario 2 had 2.009ms. Scenario 3 had 3.01ms.*

Keywords: software defined network, dijkstra algorithm, fail path

DAFTAR ISI

DAFTAR TABEL

DAFTAR GAMBAR

DAFTAR LAMPIRAN

BAB 1 PENDAHULUAN

1.1 Latar belakang

Saat ini perkembangan teknologi di indonesia sudah semakin modern, khususnya perkembangan internet. Pengguna internet di Indonesia hingga saat ini telah mencapai 82 juta orang. Dengan capaian tersebut, Indonesia berada pada peringkat ke-8 di dunia (Kemkominfo, 2014). Internet mempermudah pengguna dalam mengakses berbagai informasi dan mempermudah komunikasi tanpa mengenal jarak. Salah satu faktor yang menentukan kinerja dari sebuah internet adalah jaringan. Jaringan merupakan sebuah infrastruktur yang digunakan untuk menunjang proses transfer data.

Dalam infrastruktur jaringan tradisional bisa menyebabkan dampak kompleksitas seiring dengan bertambahnya kebutuhan jaringan karena masing-masing perangkat mempunyai konfigurasi yang berbeda, yaitu *control plane* dan *data plane* terletak dalam satu perangkat. Konsep *software defined network* dapat menjadi solusi untuk mengatasi masalah jaringan tradisional, yaitu dengan melakukan pemisahan antara *control plane* dan *data plane* dalam suatu perangkat yang berbeda. *Software Defined Network* adalah suatu konsep atau paradigma baru yang memberikan kemudahan kepada peneliti, administrator dan operator untuk mengontrol jaringan dengan perangkat lunak khusus dan menyediakan *Application Programming Interface* (API) terhadap tabel *forwarding* dari *switch* dari vendor yang berbeda sehingga dapat dikonfigurasi dan dikendalikan melalui *software* terpusat (Appelman et al, 2012). *Software defined network* mendukung kebutuhan dan inovasi dalam bidang jaringan, dimana *data plane* berada pada perangkat jaringan dan *control plane* berada pada sebuah entitas terpisah bernama *controller*. *Control plane* berfungsi untuk mengatur logika didalam perangkat jaringan seperti *routing table* dan pemetaan jaringan. Sedangkan *data plane* berfungsi untuk meneruskan paket-paket yang masuk ke suatu *port* pada perangkat jaringan menuju *port* keluar dengan berkonsultasi pada *control plane*. Untuk komunikasi antara *control plane* dan *data plane* menggunakan protokol *openflow*.

Software defined network memiliki beberapa kemampuan teknologi di antaranya *load balancing*, *multimedia multicast, Intrusion detection,* dan *routing*. *Routing* digunakan untuk menunjang proses transfer data. Algoritma *routing* untuk menentukan jalur terpendek di antaranya *dijkstra algorithm*. *Dijkstra algorithm* menentukan minimum *path* menggunakan cara dengan membandingan jarak masing-masing jalur antara node dengan *weight* node. Melalui hasil perbandingan perhitungan maka *rute* yang terpilih adalah jumlah node *weight* dan *edge weight* yang terkecil. (Gupta et al, 2016)

Di dalam *routing* terdapat beberapa masalah di antaranya kegagalan link (fail path). Ketika terjadi fail path mengakibatkan sistem mencari jalur lain. Dalam proses mencari jalur lain dilakukan oleh *controller* yang memonitor status setiap port dari masing-masing switch (Ying-Dar Lin et al, 2016). Ketika seluruh jalur

node dan *weight* node sudah terbentuk, maka melalui *dijkstra algorithm* dipilih minimum *path* sebagai mekanisme menentukan jalur lain. *Dijkstra algorithm* mengeksekusi *fail path* dan melakukan *routing* ulang melalui hasil perbandingan perhitungan jumlah *weight* terkecil (Radwan et al, 2011). Jika jalur yang menghubungkan antara switch pada jaringan terjadi *fail path* maka akan mempengaruhi proses transfer data dan jalur sebelumnya tidak dapat dilewati. *Fail path* terjadi ketika jalur yang menghubungkan antara switch mengalami *down*. Saat salah satu terjadi *fail path*, maka *routing protocol* akan melakukan *update* ulang dengan menentukan jalur terpendek lain untuk dilewati.

Berdasarkan penjelasan diatas untuk analisis *fail path* diperlukan pengujian dengan memutuskan jalur antara *switch* menggunakan *link down*. Ketika terjadi *fail path* maka jalur tersebut tidak dapat dilewati. Untuk menyelesaikan permasalahan *fail path* akan dilakukan dengan mencari jalur terpendek lain untuk dilewati menggunakan *dijkstra algorithm*. *Dijkstra algorithm* adalah metode yang digunakan untuk menentukan jalur terpendek berdasarkan *cost* terkecil pada suatu rute dari node asal ke node tujuan. ketika dalam sebuah jaringan terjadi *fail path* maka *dijkstra algorithm* akan melakukan routing ulang dengan mencari jalur terpendek lain. Proses mencari jalur terpendek saat terjadi fail path dilakukan untuk menghasilkan jalur terpendek lain, yaitu dengan melihat hasil perbandingan antara jarak terdekat sebelum dan sesudah terjadi *fail path*.

1.2 Rumusan masalah

Berdasarkan uraian latar belakang, maka dapat dirumuskan beberapa permasalahan sebagai berikut:

1. Bagaimana menganalisis *fail path* pada arsitektur *Software Defined Network* menggunakan *dijkstra algorithm*?
2. Bagaimana melakukan pengujian *dijkstra algorithm* dengan menggunakan *fail path* untuk menghasilkan jalur terpendek dengan hasil *cost* yang berbeda?
3. Bagaimana melakukan pengujian topologi, *throughput, latency* dan *convergence* pada *dijkstra algorithm* dengan terjadi *fail path*?

1.3 Tujuan

Berikut tujuan yang bisa dihasilkan dari penelitian ini yaitu:

1. Menganalisis *fail path* pada arsitektur *Software Defined Network* menggunakan *dijkstra algorithm*.
2. Melakukan pengujian *dijkstra algorithm* dengan menggunakan *fail path* untuk menghasilkan jalur terpendek dengan hasil *cost* yang berbeda.
3. Melakukan pengujian topologi, *throughput, latency* dan *convergency* pada *dijkstra algorithm* dengan menggunakan *fail path.*

1.4 Manfaat

Penelitian ini bermanfaat untuk menganalisis ketika terjadi *fail path* dengan menggunakan *dijkstra algorithm*. *Djikstra algorithm* akan diimplementasikan pada arsitektur *software defined network* dengan melakukan routing ulang untuk menghasilkan jalur terpendek lain ketika terjadi fail path, yaitu dengan melakukan pengujian topologi, *throughput, latency* dan *convergence*.

1.5 Batasan masalah

1. Melakukan *routing* ulang ketika terjadi *fail path* menggunakan *dijkstra algorithm*.
2. Pengujian jaringan dilakukan dengan menggunakan desain topologi mesh.
3. *Controller* yang digunakan adalah *pyretic*.
4. Bahasa pemrograman menggunakan bahasa *python*.
5. Implementasi dan pengujian dilakukan dengan menggunakan simulator mininet.

1.6 Sistematika pembahasan

Berikut adalah struktur penulisan skripsi berserta uraiannya dari masing-masing bab:

BAB 1 PENDAHULUAN

Bab ini menjelaskan latar belakang, rumusan masalah, tujuan, manfaat, batasan masalah dan sistematika pembahasan dari penulisan penelitian yang mengimplementasikan *dijkstra algorithm* untuk analisis *fail path* pada arsitektur *software defined network*.

BAB 2 LANDASAN KEPUSTAKAAN

Bab ini berisi beberapa dasar teori dan referensi penelitian yang pernah ada yang memiliki hubungan dengan analisis *fail path* pada arsitektur *software defined network* menggunakan *dijkstra algorithm*.

BAB 3 METODE PENELITIAN

Bab ini membahas langkah kerja yang dilakukan dalam penelitian, diantaranya alur metodologi penelitian, analisis kebutuhan sistem, perancangan, spesifikasi sistem, dan perancangan pengujian sistem.

BAB 4 REKAYASA KEBUTUHAN

Bab ini membahas semua kebutuhan sistem dan penjabaran tentang fungsional dan non-fungsional sistem.

BAB 5 PERANCANGAN DAN IMPLEMENTASI

Bab ini menjelaskan tentang rancangan sistem berserta implementasinya dari masing-masing sub sistem sehingga menjadi satu sistem utuh sesuai dengan rancangannya.

BAB 6 PENGUJIAN

Bab ini berisi tentang proses pengujian setelah dilakukan implementasi, setelah itu dianalisis untuk mengetahui keberhasilannya.

BAB 7 PENUTUP

Bab ini berisi kesimpulan hasil dari penelitian berserta saran untuk perkembangan penelitian selanjutnya.

BAB 2 LANDASAN KEPUSTAKAAN

Pada bab ini berisikan landasan pustaka yang akan menjabarkan tentang tinjauan pustaka dan dasar teori dalam mengerjakan penelitian ini. Tinjauan pustaka terdiri dari penelitian yang sudah ada, dan penelitian yang akan diusulkan, berisi kajian teori dari komponen dan materi terkait yang dapat mendukung penelitian.

2.1 Tinjauan Pustaka

No.	Nama penulis, Tahun dan Judul	Persamaan	Perbedaan	
			Penelitian Terdahulu	Rencana Penilitian
1	Henny Syahriza Lubis., 2009. Perbandingan Algoritma Greedy dan Algoritma Dijkstra Untuk Menentukan Lintasan Terpendek	Menerapkan algoritma untuk menghasilkan jalur terpendek	Melakukan perbandingan antara algoritma greedy dan algoritma dijkstra. Menggunakan algoritma dijkstra dalam menentukan jalur terpendek akan lebih akurat dan tepat.	Implementasi menggunakan algoritma dijkstra untuk menghasilkan jalur terpendek.
2	Gupta Nitin., 2016. Applying Dijkstra's Algorithm in Routing Process	Menggunakan algoritma djikstra untuk menentukan rute terpendek	Menggunakan algoritma djikstra untuk menentukan rute terpendek dengan memilih minimum weight	Menggunakan algoritma djikstra untuk menentukan rute terpendek dengan memilih minimum weight yang diterapkan pada infrastruktur SDN.
3	Radwan S.	Melakukan	Routing ulang	Routing ulang

	Abujassar., 2011. Efficient Algorithms to Enhance Recovery Schema in Link State Protocols	routing ulang ketika terjadi kegagalan link	dengan membuat backup routing tabel	dengan backup path

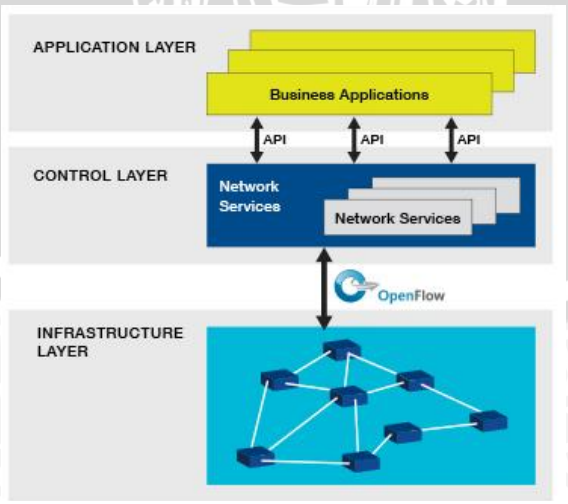
2.2 Dasar Teori

Dasar teori terdiri dari semua teori yang akan diperlukan berisi pembahasan komponen sebagai penunjang untuk penyelesaian sistem ini yaitu *software defined network*, *openflow*, *pyretic*, mininet, Teori rute terpendek, *dijkstra algoritma*.

2.2.1 *Software Defined Network*

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

Konsep SDN adalah sebuah konsep pendekatan jaringan komputer dimana sistem pengkontrol dari arus data (*control plane*) secara fisik dipisahkan dari perangkat kerasnya (data atau *forwarding plane*). *Control Plane* adalah bagian yang berfungsi untuk mengatur logika pada perangkat jaringan seperti *routing table*, pemetaan jaringan, dan sebagainya. *Data Plane* adalah bagian yang berfungsi untuk meneruskan paket-paket yang masuk ke suatu *port* pada perangkat jaringan menuju *port* keluar dengan komunikasi pada *Control Plane*.



Gambar 2.1 *Open Networking Foundation*
Sumber: www.opennetworking.org

Pada gambar 2.1 terdapat tiga layer yang menyusun arsitektur jaringan pada SDN yaitu *application layer, control layer* dan *infrastructure layer*. *Application layer* merupakan layer yang berisikan aplikasi-aplikasi seperti *mail server*, *web server* maupun lainnya. *Control layer* memiliki peranan penting dalam mengendalikan sistem, pada layer ini merupakan bagian saat *controller* dipisahkan dari data *plane*. Sedangkan *data plane* merupakan sekumpulan dari perangkat *networking* yang akan dikendalikan oleh *control plane*. Setiap perangkat *networking* pada *infrastructure layer* dapat beroperasi sesuai dengan perintah dari *controller*. Komunikasi antara perangkat dengan *controller* menggunakan protokol yang disebut *openflow*. (Astuto et al, 2014)

Perbedaan mendasar jaringan SDN dengan jaringan tradisional adalah penempatan fungsi *control* dan *forward*. Pada jaringan tradisional fungsi *control* dan *forward* ditempatkan pada *device* yang sama yaitu *router*. Sedangkan pada jaringan SDN fungsi *control* ditempatkan pada *software* terpusat (*controller*) dan fungsi *forward* ditempatkan pada suatu perangkat kosong berupa *switch* (*forwarding decive*). SDN memerlukan beberapa metode agar *control plane* dapat berkomunikasi dengan *data plane* yaitu dengan menggunakan suatu protocol yang disebut *openflow*. (Nishtha, 2014)

2.2.2 *Openflow*

Openflow adalah protokol atau standar komunikasi antarmuka yang berada diantara *control plane* dan *data plane*. *Openflow* bertujuan untuk mengontrol *data plane* pada *switch*, yang telah dipisahkan secara fisik dari *control plane* menggunakan perangkat lunak pengendali (*controller*) pada sebuah server. *Control Plane* berkomunikasi dengan *data plane* melalui protokol *openflow*. *OpenFlow* memungkinkan pengaturan *routing* dan pengiriman paket ketika melalui sebuah *switch*. Dalam sebuah jaringan, setiap *switch* berfungsi meneruskan paket yang melalui suatu *port*. *OpenFlow* dapat mengakses dan memanipulasi *forwarding plane (data plane)* secara langsung dari perangkat-perangkat jaringan seperti *switch* dan *router* baik secara fisik maupun *virtual*.

Mekanisme aliran paket data pada *openflow* adalah ketika sebuah paket tiba di *switch openflow*, bagian *header* paket diperiksa berdasarkan entri *flow table*, jika entri yang sesuai ditemukan, *switch* kemudian menerapkan instruksi-intruksi terkait berdasarkan aliran paket data yang sesuai dan jika tidak ada yang sesuai dengan prosedur *flow table*, maka aliran paket akan diarahkan ke entri *table-miss*. Entri *table-miss* adalah entri yang diperlukan untuk menentukan set instruksi yang akan diterapkan terhadap paket yang masuk ketika tidak ada yang cocok atau sesuai dengan prosedur *flow table*. Intruksi dalam entri *table-miss* adalah menghapus (*dropping*) paket, mengirim paket pada semua *interface*, dan meneruskan paket ke *controller* (Nick McKeown, 2008).

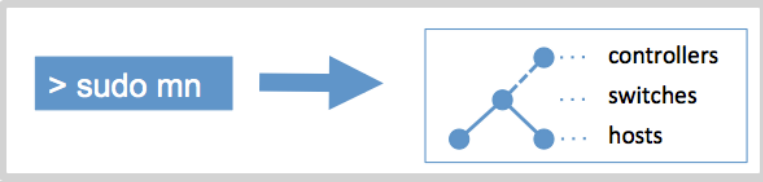
2.2.3 *Pyretic*

Pyretic adalah sebuah cara baru dalam mengelola jaringan melalui *software* yang mengatur jaringan. *Pyretic* memungkinkan seorang *programmer* untuk menulis dan membuat program jaringan secara benar untuk mendapatkan performa yang terbaik (Project, 2016). *Pyretic* adalah *platform* berbasis *python* yang dapat mewujudkan banyak konsep dan memungkinkan sistem *programmer* untuk membuat aplikasi SDN canggih. *Pyretic* merupakan perkembangan dari *frenetic* (JOSHUA REICH, 2013).

Pyretic memiliki dua operator komposisi kebijakan, yaitu komposisi paralel dan komposisi sekuensial. Sehingga memungkinkan *programmer* untuk dapat menggabungkan beberapa kebijakan. Komposisi ini yang membuat *programmer* dapat membuat program SDN yang canggih. Secara konseptual, kebijakan *pyretic* adalah fungsi yang mengambil paket *input* dan mengembalikan satu set paket. Fungsi ini menjelaskan apa yang harus dilakukan *switch* pada jaringan dengan paket yang masuk. Demikian juga fungsi yang memforward paket dari lokasi tertentu (*switch* dan *port*) yang hanya menuju ke tujuannya (Project, 2016).

2.2.4 Mininet

Mininet adalah perangkat lunak emulator jaringan yang dapat digunakan untuk mensimulasikan jaringan, baik *switch, host,* dan *controller* SDN dalam satu perintah (seperti pada single komputer atau laptop maupun *Virtual Machine*). Pada mininet dapat melakukan simulasi jumlah node (dalam hal ini *switch*) yang banyak sesuai dengan topologi yang diinginkan. Mininet digunakan untuk mengemulasi *data plan* atau *infrastructure layer* dalam perancangan jaringan SDN. Secara sederhana mininet berfungsi untuk emulasi pada bagian data *path* untuk melakukan test konfigurasi jaringan SDN. Sedangkan untuk melakukan testing pada mininet dapat dilakukan dengan *command* "*sudo mn*". Dengan *command* ini mininet akan mengemulasikan konfigurasi jaringan SDN yang terdiri dari 1 *controller*, 1 *switch* dan 2 *host*.



Gambar 2.2 *Single Command*
Sumber: http://mininet.org/

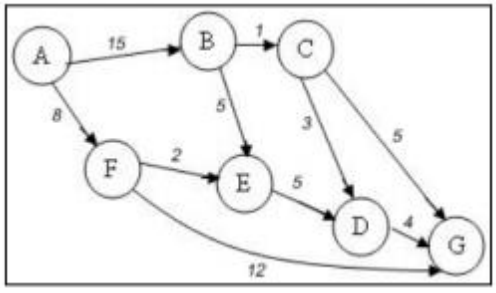
Mininet diciptakan dengan tujuan untuk mendukung riset di bidang SDN dan *openflow*. Mininet dapat menciptakan jaringan virtual yang realistis, menjalankan *real kernel, switch* dan kode aplikasi, pada *single machine* (baik berupa *physical machine* atau *virtual machine*). Mininet sangat berguna untuk pengembangan riset, pengajaran, serta penelitian. Pada penelitian ini, mininet diinstal dan dikonfigurasikan pada *VirtualBox* (*virtual machine*) sehingga kontroler akan berada pada PC yang berbeda dengan mininet. Kontroler diletakkan atau diinstal pada PC atau laptop yang juga menjalankan *virtualbox*.

2.2.5 *Dijkstra Algorithm*

Dijkstra Algorithm merupakan algoritma yang paling sering digunakan dalam memecahkan permasalahan pencarian jalur terpendek untuk sebuah graf berarah *(directed graph)* dengan bobot-bobot sisi *(edge weights)* yang bernilai tak-negatif, penggunaannya dengan menggunakan simpul-simpul sederhana pada sebuah jaringan yang tidak rumit (Juan, 2006). *Dijkstra algorithm* ditemukan oleh seorang ilmuwan *computer* berkebangsaan Belanda yang bernama Edsger Dijkstra.

Dalam mencari solusi, *dijkstra algorithm* menggunakan prinsip greedy, yaitu mencari solusi optimum pada setiap langkah yang dilalui, dengan tujuan untuk mendapatkan solusi optimum pada langkah selanjutnya yang akan mengarah pada solusi terbaik. Hal ini membuat kompleksitas waktu *dijkstra algorithm* menjadi cukup besar, yaitu sebesar O (V * Log (v + e)), dimana v dan e adalah simpul dan sisi pada graf yang digunakan. *Input* dari *dijkstra algorithm* berupa sebuah graf berbobot G(e,v), sedangkan *output* berupa rute terpendek dari simpul awal (*start*) ke masing-masing simpul yang ada pada graf.

Simpul pada graf dapat dinomori dengan huruf, seperti a, b, c atau bilangan asli 1, 2, 3 atau gabungan dari keduanya. Sedangkan sisi yang menghubungkan simpul u dengan simpul v dinyatakan dengan pasangan (u, v) atau dinyatakan dengan lambang e1, e2. Jika e adalah sisi yang menghubungkan simpul u dengan simpul v, maka e dapat ditulis sebagai e = (u, v). Secara geometri graf digambarkan sebagai sekumpulan simpul yang dihubungkan dengan sekumpulan garis (sisi) (Fitria, 2013).

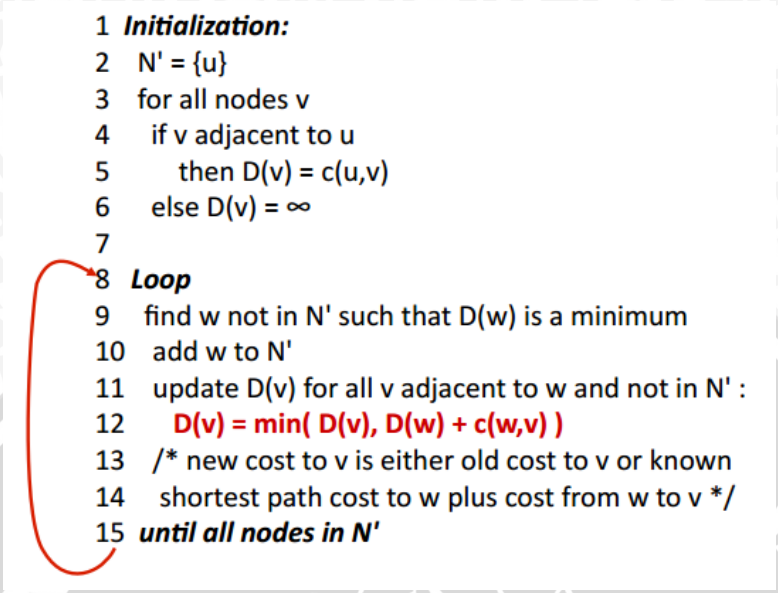


Gambar 2.3 Graph
Sumber: Rahayu Ningati, 2014

Gambar 2.3 simpul atau node A akan dianggap sebagai node awal dan node G dianggap sebagai node tujuan. Bobot (*cost*) dari sebuah sisi dapat dianggap sebagai jarak antara dua node, yaitu jumlah jarak semua sisi dalam jalur tersebut.

Cara kerja *dijkstra algorithm* menggunakan prinsip antrian (*queue*), akan tetapi antrian yang digunakan *dijkstra algorithm* adalah antrian berprioritas (*priority queue*). Jadi hanya simpul yang memiliki prioritas tertinggi yang akan dilewati. Dalam menentukan simpul yang berprioritas, algoritma ini membandingkan setiap nilai (bobot) dari simpul yang berada pada satu level. Selanjutnya bobot dari setiap simpul tersebut disimpan untuk dibandingkan

dengan bobot dari jalur yang baru ditemukan, begitu seterusnya sampai ditemukan simpul yang di cari dengan menghasilkan jalur terpendek.



Gambar 2.4 *Pseudocode Dijkstra Algorithm*

Gambar 2.4 merupakan *pseudocode dijkstra algorithm* dalam mencari jalur terpendek. Untuk penjelasan gambar 2.4 adalah sebagai berikut:

1. Baris 1-2 adalah melakukan inisialisasi kemudian mencatat node asal pada array N sebagai u.
2. Baris 3 adalah untuk V mewakili node-node lain selain u (karena node v berfungsi sebagai tujuan), dari tahap ini dimulai perulangan (untuk setting cost awal) dengan mencoba semua kemungkinan v (nilai v berganti-ganti sesuai node yang ada).
3. Baris 4-6 adalah Jika v dan u bertetangga (terhubung langsung), selanjutnya melakukan setting cost antara u dan v sebagai c(u,v) {misal, cost node 5 dan 6 adalah 10, maka c(5,6)=10} tetapi jika tidak bertetangga maka lakukan setting nilai cost menjadi *infinite*, ulangi ke baris 1 sampai semua node dicatat costnya.
4. Baris 8-15 adalah perulangan kedua (untuk algoritma Dijkstra). Melakukan perhitungan jarak selain dari u (selain di daftar N), yaitu v ke tetangganya (dilambangkan w), cari nilai paling kecil, jika sudah dapat nilai terkecil antara v dan w, masukkan ke daftar N. Kemudian melakukan *update* nilai v ke w jika ada nilai yang lebih kecil dengan menggunakan rumus Dijkstra (hitung jarak terkecil antara u ke v ke w). Selanjutnya kembali ke langkah 8 sampai semua node diperhitungkan jaraknya.

2.2.6 *Throughput*

Throughput adalah *bandwidth* sebenarnya (aktual) yang di ukur dengan satuan waktu tertentu dalam *bit per second* (bps) dan digunakan untuk

melakukan transfer data dengan ukuran tertentu. Waktu *download* terbaik adalah ukuran file di bagi dengan *bandwidth*. Sedangkan waktu aktual atau sebenarnya adalah ukuran file di bagi dengan *throughput* (William S. Bobanto, 2014).

Menurut (Rahmad Saleh Lubis, 2014) *throughput* yaitu kecepatan (*rate*) transfer data efektif, yang diukur dalam bps (*bit per second*). Untuk mengukur nilai *Throughput* adalah dengan mengamati jumlah paket terkirim yang datang pada destination selama interval waktu tertentu yang dibagi oleh durasi interval waktu tersebut. Throughput biasanya selalu dikaitkan dengan bandwidth.

$$\text{Throughput} = \frac{\text{Jumlah data yang dikirim}}{\text{Waktu Pengiriman Data}}$$

2.2.7 *Latency*

Latency dalah waktu yang dibutuhkan paket untuk menempuh jarak dari awal sampai ke tujuan. *Delay* dapat dipengaruhi oleh jarak, kongesti atau juga waktu proses yang lama (Yanto, 2013).

Waktu yang dibutuhkan paket untuk mencapai tujuan, karena adanya antrian, atau mengambil rute yang lain untuk menghindari kemacetan. Untuk pengujian dapat dilakukan dengan menggunakan perintah ping dari PC *client* ke PC server.

- Ping –s 250 –c 7 10.0.0.1

  Keterangan: -s adalah sebagai *server*

  250 adalah ukuran paket adalah 250 byte

  -c adalah sebagai *client*

  7 adalah paket yang dikirim sebanyak 7 kali

  10.0.0.1 adalah *no ip server*

2.2.8 *Convergence*

Convergence adalah waktu yang dibutuhkan sebuah jaringan untuk menemukan jalur yang baru ketika terjadi pemutusan jalur. Pengukuran *convergence* dilakukan dengan mengirimkan paket ping dari satu host ke host lain, lalu ditengah pengiriman paket ping dilakukan pemutusan jalur pada rute yang dipilih (Aris Saputra Ihsan, 2016).

Nilai *convergence* dapat diketahui ketika terdapat perubahan pada jaringan. Dalam pengujian waktu *convergence* didapatkan waktu dengan rata-rata nilai detik. Pengujian *convergence* dilakukan untuk mengukur seberapa cepat sebuah jaringan mendapatkan jalur lain untuk menggantikan jalur yang lama.

2.2.9 Teori *Fail Path*

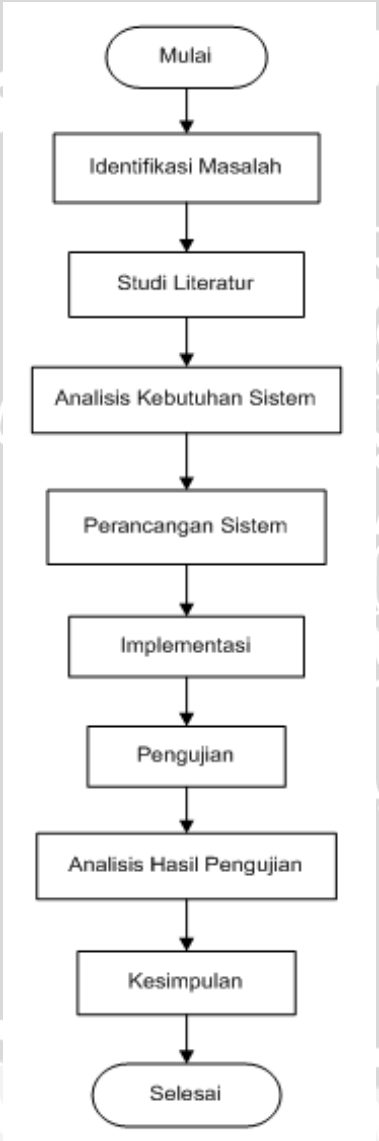
Fail path terjadi jika di dalam *routing* terdapat beberapa masalah di antaranya kegagalan link. Mekanisme *fail path* mengakibatkan sistem mencari jalur lain untuk di lewati. Jika jalur yang menghubungkan antara switch pada jaringan terjadi *fail path* maka akan mempengaruhi proses transfer data dan kinerja seluruh jaringan terganggu sehingga membuat kinerja jaringan kurang efisien.

Fail path terjadi ketika jalur yang menghubungkan antara switch mengalami *down*. Ketika salah satu terjadi *fail path*, maka *routing* ulang akan melakukan *update* ulang dengan menentukan jalur terpendek lain untuk dilewati.

BAB 3 METODOLOGI

3.1 Metode Penelitian

Pada bab ini akan menjelaskan tentang alur pengerjaan skripsi berupa perancangan dan pengujian dari simulasi yang akan dijalankan. Langkah-langkah yang akan dilakukan adalah mengumpulkan teori-teori pendukung yang akan dikumpulkan menjadi satu dalam studi literatur. Lalu dilakukan perancangan dengan melakukan implementasi yang sesuai dengan perancangan. Kemudian dilanjutkan dengan pengujian pada sistem yang telah dibuat. Pada gambar 3.1 menjelaskan tentang diagram alir yang akan digunakan dalam pengerjaan skripsi.



Gambar 3.1 Diagram Alir Penelitian

3.1.1 Studi literatur

Studi literatur menjelaskan tentang dasar teori yang digunakan untuk mendukung dalam analisis *fail path* pada arsitektur *software defined network* menggunakan *dijkstra algorithm*. Adapun yang dapat dijadikan dasar teori pendukung sebagai bahan dalam studi literatur pada penelitian ini meliputi:

1. *Software Defined Network*
2. *Pyretic*
3. *Mininet*
4. *Dijkstra algorithm*

3.1.2 Analisa Kebutuhan

Analisa kebutuhan bertujuan untuk menganalisis semua kebutuhan yang diperlukan untuk membangun sistem pada penelitian yang dilakukan, sehingga dapat diperoleh kebutuhan yang sesuai dengan yang diharapkan.

3.1.2.1 Kebutuhan sistem

Perangkat lunak yang dibutuhkan untuk membangun sistem antara lain, yaitu:

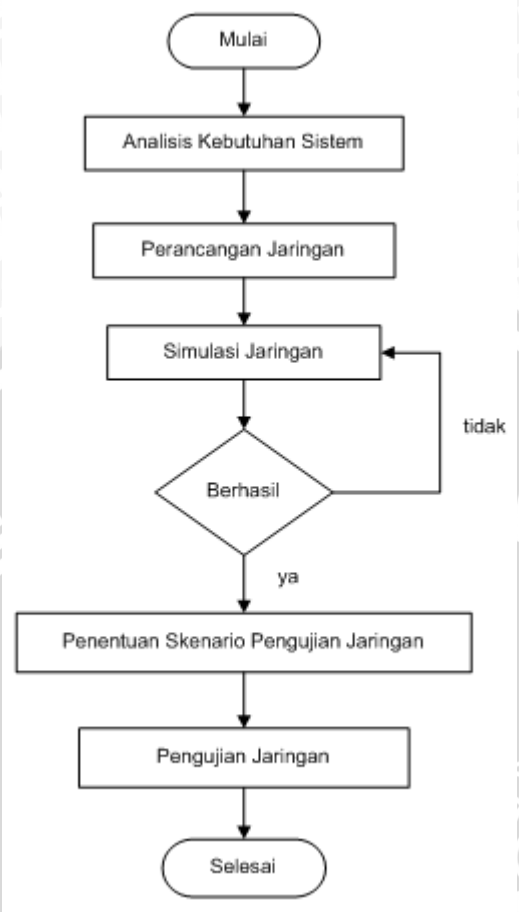
1. mininet versi 2.2.1
2. Linux ubuntu 14.04
3. *controler* menggunakan *pyretic* dengan bahasa *python*
4. *Xming* digunakan untuk x server
5. *Putty* digunakan untuk SSH

3.1.2.2 Kebutuhan software

Perangkat lunak yang dibutuhkan untuk membangun sistem diantaranya mininet-VM *operating system ubuntu* 14.04, *controler* menggunakan *pyretic* dengan bahasa pemrograman *python*, *Xming, putty*.

3.1.3 Perancangan Sistem

Pada tahap ini dilakukan perancangan sistem untuk menggambarkan bagaimana suatu sistem dibangun. Perancangan dilakukan agar dapat terarah dan terstruktur. Perancangan sistem pada penelitian ini dapat dilihat pada Gambar 3.2 dijelaskan tentang perancangan sistem dari tahap penentuan topologi sampai penentuan skenario dan pengujian jaringan yang dilakukan pada sistem.



Gambar 3.2 Diagram Alir Perancangan Sistem

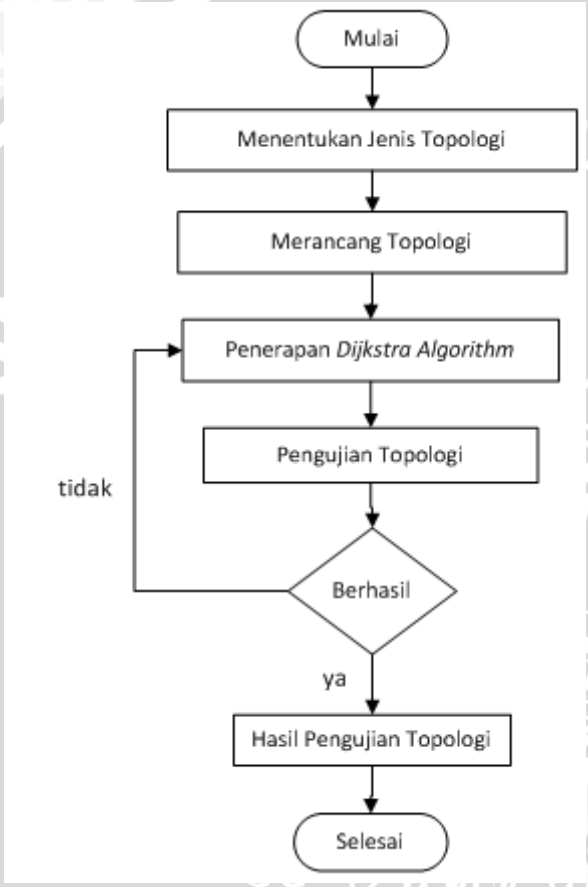
Perancangan sistem dimulai dari tahapan pada analisis kebutuhan sistem adalah merancang perangkat lunak apa saja yang diperlukan oleh sistem, seperti mininet, *pyretic*. Perancangan jaringan adalah menentukan topologi jaringan apa yang akan digunakan seperti jenis topologi yang akan dibangun. Simulasi jaringan adalah membangun topologi pada emulator berbasis jaringan *software defined network*, seperti menginstal mininet dengan versi 2.2.0, *install controller pyretic*. Pengecekan adalah jika simulasi jaringan yang dilakukan berhasil maka dapat dilanjutkan dengan penentuan skenario pengujian jaringan, jika tidak berhasil maka dilakukan proses ulang simulasi jaringan. Penentuan skenario pengujian jaringan adalah dengan menerapkan skenario apa yang akan digunakan untuk melakukan pengujian jaringan.

3.1.4 Implementasi

Pada tahap implementasi sistem dilakukan proses pengujian terhadap sistem berdasarkan perancangan kebutuhan sistem dan perancangan sistem yang telah dibuat. Pada tahapan implementasi sistem dimulai dengan menginstal mininet sebagai emulator dalam menjalankan simulasi jaringan. Setelah mininet berhasil diinstal, maka akan dilanjutkan dengan membuat program *routing* dalam jaringan *openflow*. Setelah berhasil membuat *routing*, maka akan dilanjutkan

membuat program *dijkstra algorithm*. Program *routing* adalah program utama yang akan dijalankan dan program *routing* yang akan mengimpor file pada program *dijkstra algorithm*. Program *dijkstra algorithm* digunakan untuk menghasilkan jalur terpendek lain saat terjadi *fail path* pada *software defined network* sehingga jika *switch* mengalami masalah, seperti *fail path* maka jalur yang dilewati sebelumnya tidak dapat dilewati. Dengan program *dijkstra algorithm* hasil yang didapatkan dapat menentukan jalur terpendek lain.

3.1.5 Pengujian



Gambar 3.3 Diagram Alir Sistem

Gambar 3.3 pengujian akan dilakukan dengan menentukan jenis topologi apa yang akan dibuat dalam pengujian, yaitu menggunakan topologi mesh, kemudian merancang topologi mesh sesuai dengan perancangan yang akan dibuat, selanjutnya melakukan penerapan *dijkstra algorithm* pada pengujian topologi untuk mencari jalur terpendek. Jika pengujian topologi berhasil maka akan menghasilkan jalur terpendek tetapi jika terjadi *fail path* maka akan kembali melakukan pengujian jaringan yang nantinya akan menghasilkan jalur terpendek lain.

3.1.6 Analisis

Dari pengujian yang sudah dilakukan, didapatkan hasil analisis pada kinerja sistem apakah sistem sudah berjalan dengan baik dan sesuai dengan tujuan penelitian. Analisis dilakukan untuk mengetahui apakah sistem dapat menentukan jalur terpendek dengan menerapkan *dijkstra algorithm* pada *software defined network* dan apakah sistem dapat menentukan jalur terpendek lain dengan melakukan *routing* ulang ketika terjadi *fail path*.

3.1.7 Kesimpulan

Penarikan kesimpulan dapat dilakukan setelah semua tahapan perancangan, implementasi dan pengujian sistem yang dirancang telah selesai dilakukan. Hasil kesimpulan dapat diambil dari hasil pengujian dan analisis yang telah dirancang. Pembuatan kesimpulan akan memudahkan penelitian untuk menilai dan mengerti dari penelitian yang telah dilakukan. Dari kesimpulan akan menimbulkan reaksi atau saran atau pendapat yang bertujuan untuk memberi masukan agar dapat menunjang untuk penelitian berikutnya.

BAB 4 PERANCANGAN DAN IMPLEMENTASI

Pada bab ini menjelaskan tentang rancangan sistem yang akan dibuat penulis untuk mencapai tujuan penelitian. Setelah perancangan sistem dilakukan, selanjutnya akan dibuat skenario implementasi sistem dari penelitian ini.

4.1 Perancangan Sistem

Pada tahap ini akan dijelaskan tentang perancangan sistem secara umum. Perancangan yang dilakukan dimulai dari arsitektur dan alur kerja sistem. Kemudian dilanjutkan pada perancangan perangkat keras, perangkat lunak dan pada masing-masing sub sistem.

4.1.1 Kebutuhan Perangkat Keras

Dalam perancangan yang dilakukan pada penelitian ini, akan membutuhkan perangkat keras untuk implementasi sistem. Perangkat Keras yang digunakan untuk implementasi sistem adalah sebuah laptop dengan spesifikasi, antara lain:

1. CPU = intel core i3-2350M, 2,30Ghz
2. RAM = 2,00GB
3. Harddisk = 500GB

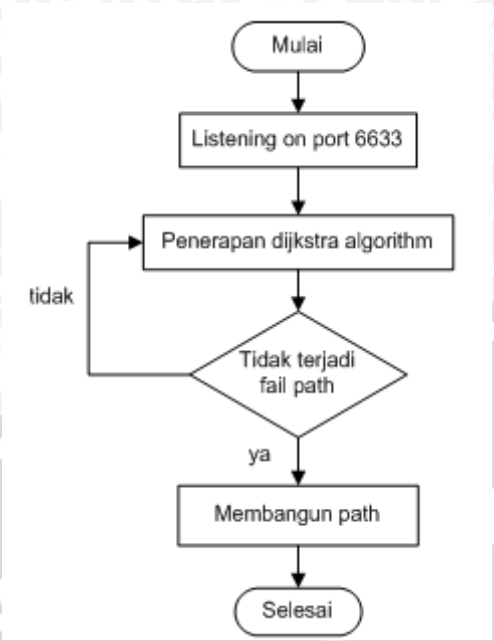
4.1.2 Kebutuhan Perangkat Lunak

Dalam perancangan yang dilakukan selain yang dibutuhkan perangkat keras maka perangkat lunak juga dibutuhkan pada penelitian ini untuk menunjang perancangan sistem dan membutuhkan perangkat lunak untuk implementasi sistem. Perangkat lunak yang digunakan untuk implementasi sistem adalah sebagai berikut:

1. *Linux ubuntu* 14.04
2. Mininet versi 2.2.0
3. *Pyretic*

4.1.3 Alur Kerja Sistem

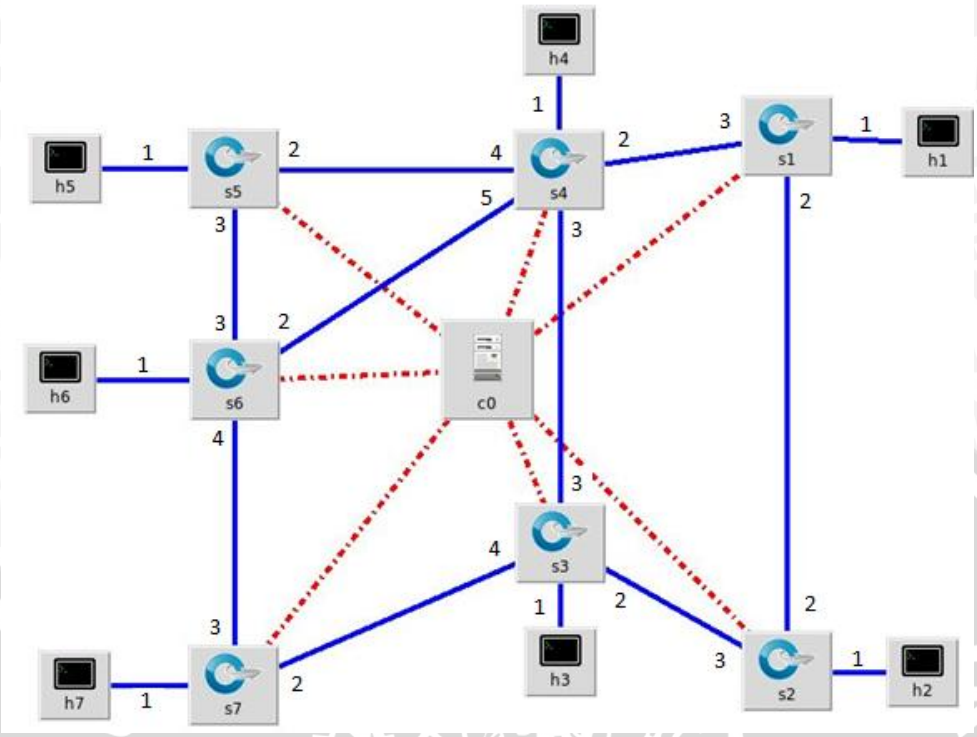
Proses pada cara kerja ketika terjadi *fail path* pada arsitektur *software defined network* akan menggunakan *pyretic* sebagai *controller setting*. Diagram *flow* pada sistem ini digambarkan pada Gambar 4.1. Pada saat melakukan koneksi maka pertama kali yang akan dilakukan, yaitu *setting port* 6633 untuk *pyretic controller*. Selanjutnya akan dilakukan penerapan *dijkstra algorithm* sebagai mekanisme implementasi deteksi *fail path* saat menangani gangguan pada *switch*. *Dijkstra algorithm* membangun path dengan menghasilkan jalur terpendek. Tetapi ketika jalur antar *switch* terjadi *fail path* maka akan dilakukan *routing* ulang dengan penerapan *dijkstra algorithm*.



Gambar 4.1 Alur Kerja Sistem

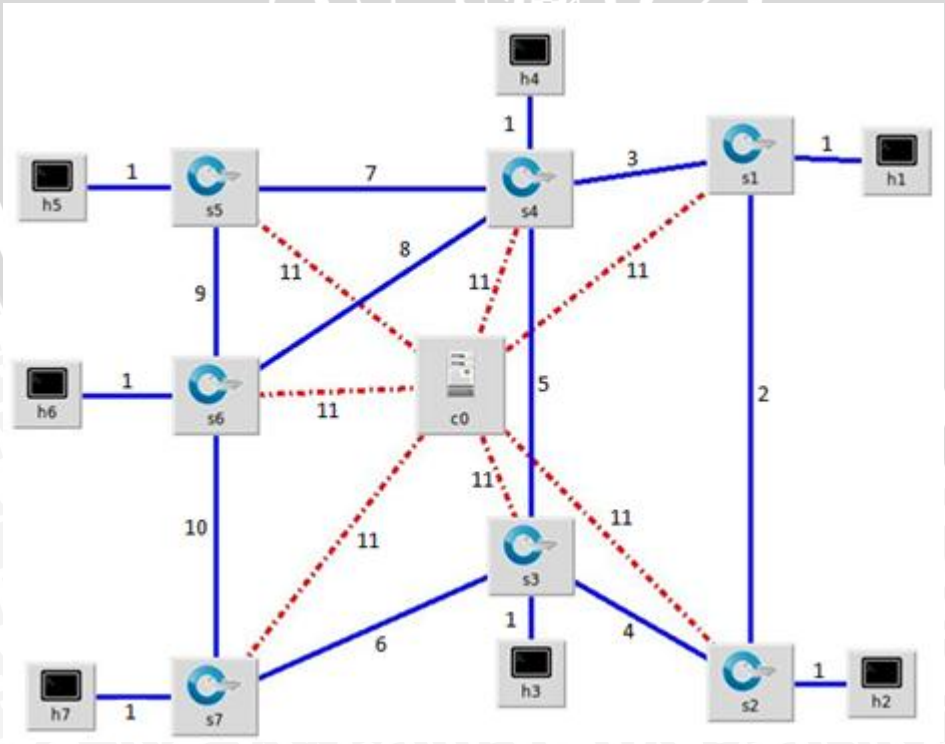
4.2 Perancangan Topologi

Perancangan topologi yang dibuat nantinya akan dilakukan pengujian dengan menerapkan metode *dijkstra algorithm* pada arsitektur *software defined network*. Topologi yang digunakan berupa topologi mesh dengan menggunakan 7 *host* dan 7 *switch*. Topologi mesh akan diterapkan pada emulator mininet. Topologi yang dirancang ini dapat menghasilkan jalur terpendek lain ketika terjadi *fail path* karena antara *switch* tidak terhubung. Untuk menghasilkan jalur terpendek adalah dengan menerapkan metode *djikstra algorithm* pada perancangan topologi dan sebelum melakukan perancangan topologi maka terlebih dahulu harus mengetahui nomor *port* yang diperoleh *switch* dari urutan pembentukan *link*. Urutan pembentukan *link* dapat dilihat pada gambar 4.3. Urutan pembentukan link dapat mempengaruhi nomer *port* yang didapat oleh *switch.* Pembentukan *link* yang disambungkan terlebih dahulu akan menghasilkan *port* 1. Pembentukan link selanjutnya menghasilkan *port* 2 dan seterusnya. Nomer port yang dihasilkan dari urutan pembentukan link dapat dilihat pada gambar 4.2.



Gambar 4.2 Nomor Port pada Switch

Untuk lebih lengkapnya dalam membangun perancangan topologi dilakukan urutan pembentukan jalur seperti pada gambar 4.3.



Gambar 4.3 Urutan Pembentukan Jalur pada Topologi

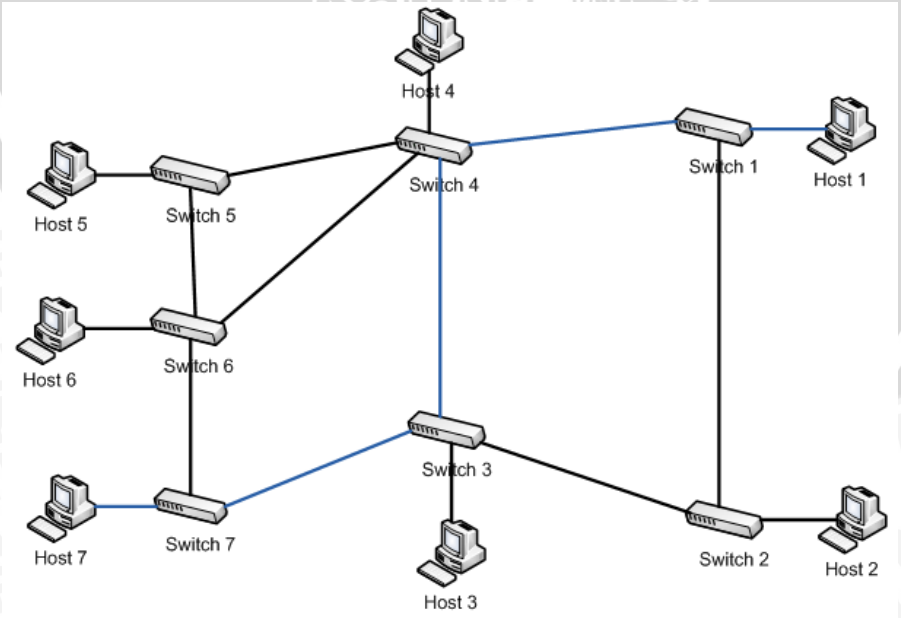
4.3 Ilustrasi Sistem

Ilustrasi sistem menjelaskan tentang bagaimana membuat topologi dengan menerapkan *dijkstra algorithm* untuk mengatasi masalah gangguan pada jaringan, seperti terjadi *fail path* pada jaringan. Ketika terjadi *fail path* pada arsitektur *software defined network* analisis dilakukan dengan menjalankan *dijkstra algorithm*. *Dijkstra algorithm* digunakan untuk menentukan jalur terpendek. Perancangan sistem juga akan dibuat untuk mengatasi permasalahan gangguan pada jalur yang terjadi fail path, yaitu bagaimana membuat perancangan sistem untuk menentukan jalur terpendek lain meskipun terjadinya *fail path*. Setelah membuat perancangan topologi maka dapat dilihat gambaran yang akan diterapkan pada perancangan sistem untuk kinerja sistem nantinya. Gambaran kinerja sistem yang akan dirancang adalah sebagai berikut:

Ketika *host* 1 menuju *host* 7 maka akan dijalankan *routing* dengan *dijkstra algorithm* untuk menentukan jalur terpendek. Dari *host* 1 menuju *host* 7 nantinya didapatkan hasil dengan *host* 1 melalui *host* 4 kemudian menuju *host* 3 dan akhirnya sampai ke *host* 7 dengan menerapkan *dijkstra algorithm* untuk mendapatkan jalur terpendek. Pada gambar 4.4 dapat dilihat jalur terpendek dari *host* 1 menuju *host* 7.

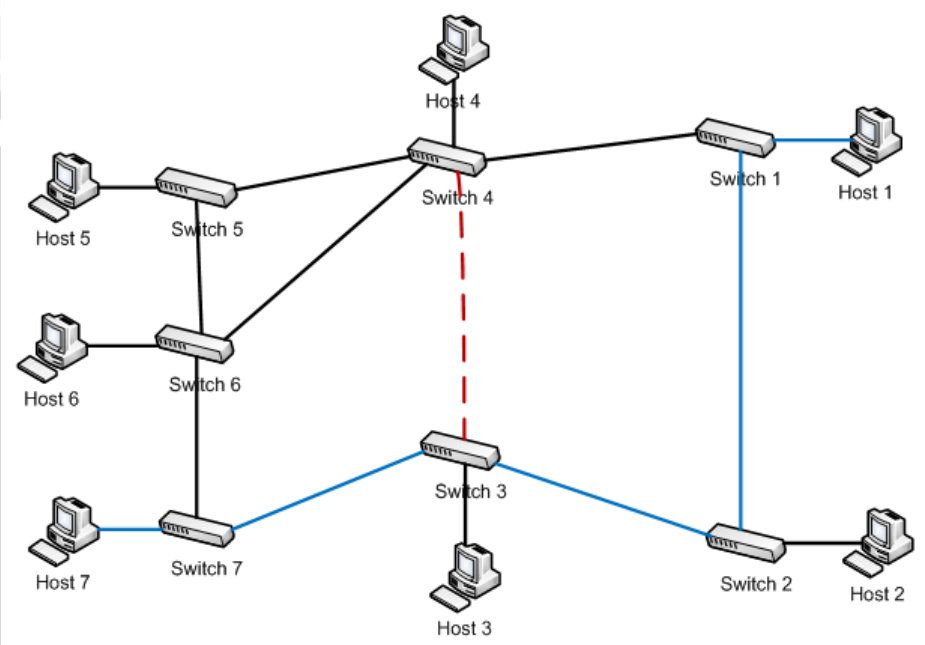
Tabel 4.1 Jalur pada Ilustrasi Sistem

1	———	Jalur yang menghubungkan antara *host* dan *switch*, *switch* dan *switch*
2	———	Hasil dari jalur terpendek
3	- - - - - - - - -	*Link down* atau *fail path*



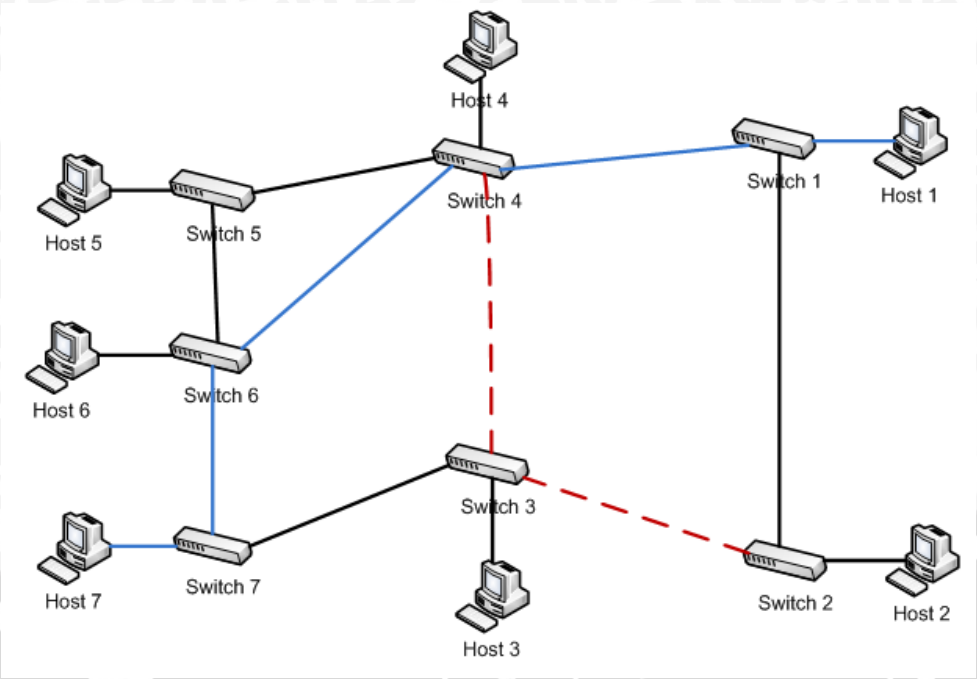
Gambar 4.4 Ilustrasi Sistem 1

Pada kinerja sistem gambar 4.5 juga diterapkan *dijkstra algorithm* untuk menghasilkan jalur terpendek tetapi dilakukan dengan perancangan sistem dengan adanya gangguan pada jaringan, yaitu *fail path*. Gangguan jaringan yang disebabkan oleh *fail path* terjadi antara *host* 3 dan *host* 4. Sehingga pencarian pada jalur yang dilewati sebelumnya tidak dapat dilakukan karena terjadi *fail path*. Dengan menggunakan *dijkstra algorithm* maka dapat dilakukan pencarian jalur terpendek lain. Meskipun terjadi *fail path* tetapi masih dapat menghasilkan jalur terpendek lain untuk dilewati dari *host* 1 menuju *host* 7. Hasil yang didapatkan dari jalur terpendek lain adalah host 1 melewati *host* 2 kemudian melewati *host* 3 dan akhirnya sampai ke *host* 7 dengan menerapkan *dijkstra algorithm*.



Gambar 4.5 Ilustrasi Sistem 2

Pada kinerja sistem gambar 4.6 juga akan diterapkan *dijkstra algorithm* untuk menghasilkan jalur terpendek lain dengan perancangan sistem yang mengalami gangguan pada jaringan, yaitu *fail path*. Gangguan di jaringan topologi terjadi pada 2 jalur, yaitu *fail path* terjadi antara *host* 3 dan *host* 4 juga terjadi antara *host* 2 dan *host* 3. Sehingga pencarian di jalur terpendek sebelumnya tidak dapat dilewati karena terjadi *fail path*. Dengan menggunakan *dijkstra algorithm* maka dapat dilakukan pencarian jalur terpendek lain. Meskipun terjadi *fail path* tetapi masih dapat menghasilkan jalur terpendek lain untuk dilewati dari *host* 1 menuju *host* 7. Hasil yang didapatkan dari jalur terpendek lain adalah host 1 melewati *host* 4 kemudian melewati *host* 6 dan akhirnya sampai ke *host* 7 dengan menerapkan *dijkstra algorithm*.

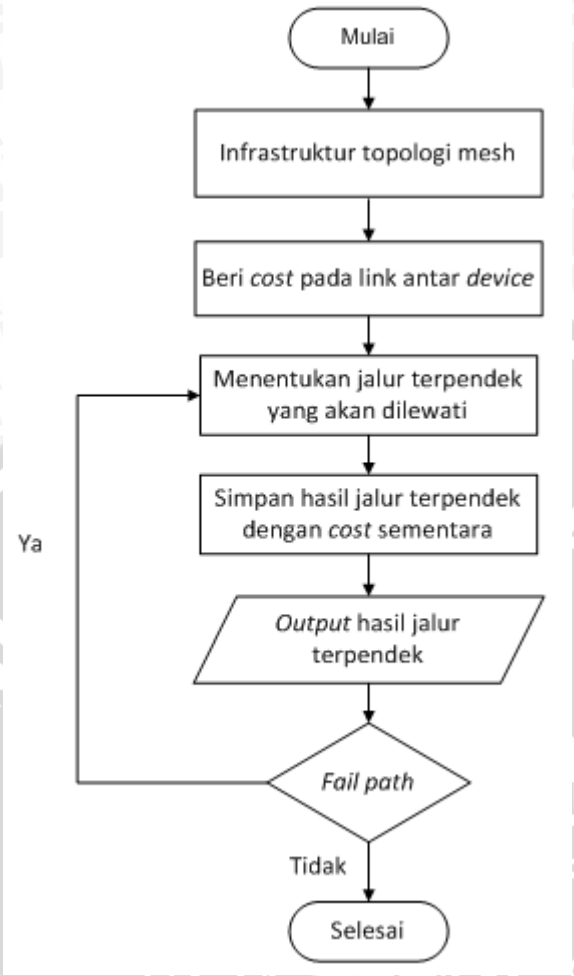


Gambar 4.6 Ilustrasi Sistem 3

4.4 Perancangan *Routing*

Perancangan *routing* akan menjelaskan tentang bagaimana cara membangun sebuah fungsi *routing* pada jaringan *openflow*. Pada pencarian jalur terpendek nantinya akan menggunakan *dijkstra algorithm*. Dengan *dijkstra algorithm* dapat menentukan jalur mana yang akan dilewati mulai dari node asal ke node tujuan.

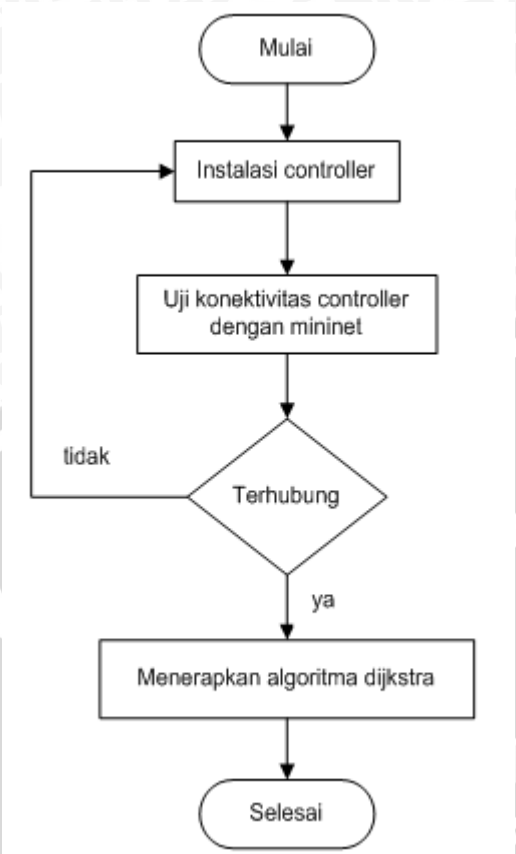
Untuk menentukan jalur terpendek yang akan dilewati dengan melihat bobot terkecil yang menghubungkan sebuah node dari node asal ke node tujuan pada sebuah graf. Dengan menghitung nilai–nilai dari setiap *cost* pada setiap jalur sehingga dapat menentukan jalur terpendek mana yang akan dilewati. Menentukan jalur terpendek dengan melihat *cost* terkecil dan akan dijelaskan pada gambar 4.7. Dimulai dari inisialisasi titik awal pada infrastruktur topologi mesh dengan menentukan *node* asal, selanjutnya beri *cost* untuk jarak terpendek yang dilewati, selanjutnya simpan *cost* untuk sementara yang diperoleh dari jarak antar *device* yang sudah dilewati sebelumnya, selanjutnya menampilkan hasil dari jalur terpendek, tetapi jika terjadi *fail path* maka mencari jalur terpendek lain untuk di lewati.



Gambar 4.7 Perancangan *Routing*

4.5 Implementasi Sistem

Pada bab ini akan membahas tentang tahapan-tahapan dalam implementasi sistem berdasarkan perancangan yang sudah dijelaskan pada sub bab sebelumnya. Pada gambar 4.8 akan menjelaskan tentang tahapan implementasi pada sistem dimulai dari instalasi kontroller *pyretic*, setelah instalasi kontroler *pyretic* selesai, selanjutnya akan dilakukan pengujian terhadap konectivitas *controller* dengan simulator mininet untuk menguji keberhasilan pada instalasi *controller*. Mininet adalah sebuah emulator yang menciptakan jaringan virtual dan merupakan salah satu emulator untuk pengembangan *software defined network*. Jika *controller* dapat menghubungkan beberapa komponen *virtual* seperti *switch* dan *host* pada emulator mininet, maka *instalasi controller* sudah dianggap berhasil sehingga *controller* terhubung dengan mininet. Tetapi jika *controller* tidak dapat menghubungkan atau menjalankan komponen virtual pada mininet, maka *instalasi controller* belum berhasil sehingga *controller* tidak terhubung dengan mininet dan perlu dilakukan *trubleshoot* untuk melakukan instalasi ulang pada *controller*. Ketika *controller* sudah dapat berjalan pada mininet, selanjutnya akan menerapkan *dijkstra algorithm* untuk menentukan jalur terpendek.



Gambar 4.8 Implementasi Sistem

4.5.1 Instalasi *Pyretic*

Instalasi *pyretic* akan dipasang pada sebuah *personal computer* dengan sistem operasi linux ubuntu 14.04. Tahap-tahap yang dilakukan untuk instalasi *controller pyretic* adalah sebagai berikut:

1. Menginstal python dependensi yang dibutuhkan *pyretic* dengan menggunakan perintah:

```
$ sudo apt-get install python-dev python-pip python netaddr screen hping3 ml-lpt graphviz ruby1.9.1-dev libboost-dev libboost-test-dev libboost-program-options-dev libevent-dev automake libtool flex bison pkg-config g++ libssl-dev python-all python-all-dev python-all-dbg
```

```
$ sudo pip install networkx bitarray netaddr ipaddr pytest ipdb sphinx pyparsing==1.5.7 yappi
```

```
$ sudo gem install jekyll
```

2. *Melakukan patch asynchat python dependency* dengan menggunakan perintah:

```
$ wget https://raw.github.com/frenetic-lang/pyretic/master/pyretic/backend/patch/asynchat.py
```

```
$ sudo mv asynchat.py /usr/lib/python2.7/
```

```
$ sudo chown root:root /usr/lib/python2.7/asynchat.py
```

3. Menginstal *git subtree* dengan menggunakan perintah:

```
$ git clone https://github.com/git/git.git
$ pushd git/contrib/subtree/
$ make
$ mv git-subtree.sh git-subtree
$ sudo install -m 755 git-subtree /usr/lib/git-core
$ popd
$ rm -rf git
```

4. Melakukan *cloning repositori pyretic* ke direktori utama milik anda dengan menggunakan perintah:

```
$ cd ~
$ git clone http://github.com/frenetic-lang/pyretic.git
```

5. Melakukan *setting variabel environment* dengan menambahkan baris dibawah ini ke akhir *.profile*:

```
export PATH=$PATH:$HOME/pyretic:$HOME/pox
export PYTHONPATH=$HOME/pyretic:$HOME/mininet:$HOME/pox
```

Setelah tahap-tahap diatas sudah selesai dilakukan, maka *controller pyretic* sudah dapat dijalankan. Untuk menjalankan *controller pyretic* dengan menggunakan perintah:

`cd pyretic` digunakan untuk masuk terminal *directory* pyretic kemudian dilanjutkan dengan menggunakan perintah:

```
$ pyretic.py –m p0 pyretic.modules.mac_learner
```

- `-m` merupakan indikasi mode apa yang dipakai dalam pyretic, di dalam pyretic terdapat 3 mode macam, yaitu `p0, r0, i`.
- `P0` merupakan mode yang digunakan untuk mode proaktif.
- `pyretic.modules` merupakan letak dimana `mac_learner` ditempatkan.

```
mininet@mininet-vm:~$ cd pyretic
mininet@mininet-vm:~/pyretic$ pyretic.py -m p0 pyretic.modules.cob
POX 0.2.0 (carp) / Copyright 2011-2013 James McCauley, et al.
Connected to pyretic frontend.
INFO:core:POX 0.2.0 (carp) is up.
INFO:openflow.of_01:[00-00-00-00-00-05 1] connected
INFO:openflow.of_01:[00-00-00-00-00-06 5] connected
INFO:openflow.of_01:[00-00-00-00-00-02 4] connected
INFO:openflow.of_01:[00-00-00-00-00-04 3] connected
INFO:openflow.of_01:[00-00-00-00-00-01 6] connected
INFO:openflow.of_01:[00-00-00-00-00-03 2] connected
INFO:openflow.of_01:[00-00-00-00-00-07 7] connected
```

Gambar 4.9 *Running Pyretic*

Gambar 4.9 menjelaskan bahwa *controller pyretic* sudah dapat dijalankan dan terdapat informasi bahwa *switch* 1, *switch* 2, *switch* 3, *switch* 4, *switch* 5, *switch* 6, *switch* 7 sudah terhubung.

4.5.2 Uji Konektivitas *Controller Pyretic*

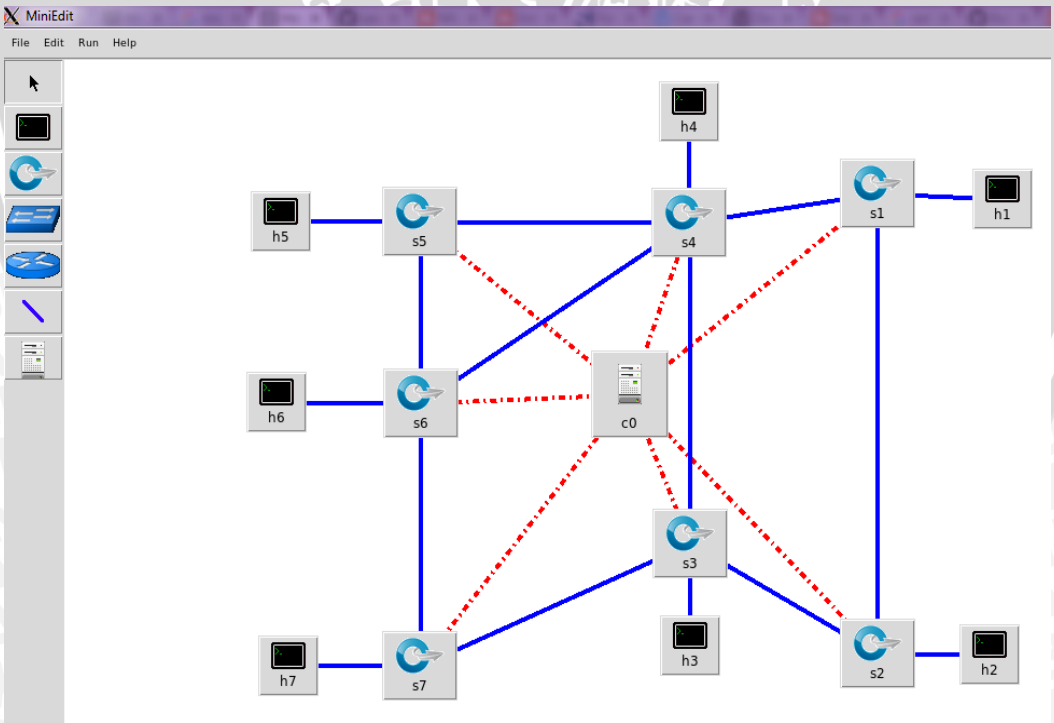
Setelah instalasi *pyretic* telah dilakukan maka langkah selanjutnya adalah melakukan pengujian apakah *controller pyretic* dapat bekerja di dalam emulator mininet. Pada pengujian konectivitas controller berbeda dengan pengujian sistem karena pada pengujian ini akan memastikan apakah *controller pyretic* dapat bekerja didalam emulator mininet. Langkah-langkah untuk melakukan pengujian konektivitas *controller pyretic* pada emulator mininet adalah sebagai berikut:

1. Menjalankan emulator mininet melalui miniedit berbasis GUI dengan menggunakan perintah `miniedit.py` pada direktori `/mininet/examples/` seperti tampilan pada gambar 4.10.

```
mininet@mininet-vm:~$ sudo ~/mininet/examples/miniedit.py
MiniEdit running against Mininet 2.2.0
topo=none
```

Gambar 4.10 *Running* Mininet

Setelah menjalankan `miniedit.py` maka akan muncul tampilan seperti pada gambar 4.11.



Gambar 4.11 Emulator Miniedit

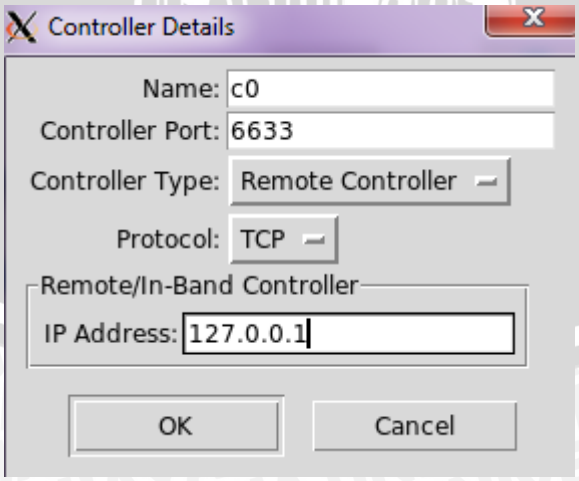
2. Kemudian melakukan konfigurasi emulator mininet dengan menarik komponen yang ada pada *tools* miniedit untuk membuat topologi sesuai

keinginan, kemudian menghubungkan beberapa komponen yang telah dipilih dengan komponen jalur yang terdapat pada *tools* miniedit, seperti pada gambar 4.11 dengan menggunakan 1 *controller*, 7 *switch* dan 7 *host* yang masing-masing komponen dihubungkan dengan jalur agar semuanya dapat terhubung. Komponen emulator miniedit pada mininet dapat dilihat pada gambar 4.12.

1		Kursor	Digunakan untuk memindahkan komponen
2		Host	Digunakan sebagai host
3		Openflow Switch	Digunakan sebagai openflow switch
4		Legacy Switch/Switch Tradisional	Adalah komponen untuk switch tradisional
5		Legacy Router/Router Tradisional	Adalah komponen untuk router tradisional
6		NetLink	Digunakan sebagai penghubung antar komponen
7		Controller	Adalah komponen yang disambungkan dengan openflow switch dan digunakan untuk menjalankan perintah sesuai keinginan.

Gambar 4.12 Komponen Emulator Miniedit

3. Pengaturan pada *controller* juga perlu dilakukan, yaitu dengan cara klik kanan pada komponen *controller* kemudian pilih properties lalu pada *controller port* isi dengan *port* 6633 karena *pyretic* berjalan pada port 6633. Untuk jenis *controller* pilih *remote controller*. Untuk protocol pilih TCP dan untuk *IP Address* konfigurasi menggunakan alamat *localhost*. Pengaturan pada *controller* dapat dilihat pada gambar 4.13.



Gambar 4.13 Pengaturan *Controller*

4. Setelah pengaturan selesai dilakukan. Selanjutnya jalankan *emulator miniedit* dengan menekan *Run* pada bagian kiri bawah emulator dan untuk berhenti menjalankan emulator mininet dengan menekan *Stop* pada bagian kiri bawah pada bawahnya *Run*.
5. Setelah menjalankan mininet, jalankan *controller pyretic* dengan menggunakan perintah:

```
$ pyretic.py –m p0 pyretic.modules.mac_learner.
```

1. Setelah menjalankan *pyretic*, lakukan percobaan konektivitas antara h1 dan h7 dengan menggunakan perintah *ping* seperti pada gambar 4.14 dan 4.15.

```
mininet> h1 ping h7
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_seq=1 ttl=64 time=1197 ms
64 bytes from 10.0.0.7: icmp_seq=2 ttl=64 time=197 ms
64 bytes from 10.0.0.7: icmp_seq=3 ttl=64 time=0.796 ms
64 bytes from 10.0.0.7: icmp_seq=4 ttl=64 time=0.172 ms
64 bytes from 10.0.0.7: icmp_seq=5 ttl=64 time=0.150 ms
^C
--- 10.0.0.7 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 0.150/279.335/1197.708/465.512 ms, pipe 2
```

Gambar 4.14 Hasil dari h1 ping h7

```
mininet> h7 ping h1
PING 10.0.0.1 (10.0.0.1) 56(84) bytes of data.
64 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=1.13 ms
64 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.082 ms
64 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.080 ms
64 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.078 ms
64 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.059 ms
64 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=0.081 ms
^C
--- 10.0.0.1 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 5006ms
rtt min/avg/max/mdev = 0.059/0.251/1.131/0.393 ms
```

Gambar 4.15 Hasil dari h7 ping h1

6. Pada gambar diatas menjelaskan bahwa h1 mempunyai alamat ip 10.0.0.1 dan host 7 mempunyai alamat ip 10.0.0.7. Gambar 4.14 dan 4.15 dapat memberikan balasan pesan icmp yang dikirim oleh host masing-masing sehingga instalasi *controller* yang telah dilakukan dapat dikatakan berhasil.

4.5.3 Instalasi Sistem

Instalasi sistem dilakukan dengan membangun sistem pemilihan jalur routing adaptif berdasarkan jalur terpendek dengan menggunakan *dijkstra algorithm* pada jaringan *openflow* untuk membangun suatu sistem.

4.5.3.1 Implementasi Routing

Implementasi *routing* dilakukan berdasarkan bagaimana membuat *routing* dalam jaringan openflow menggunakan *pyretic*. *Path* dibuat berdasarkan nomor *switch* dan nomor *port* dan disesuaikan dengan alamat IP yang dituju atau *host* yang hendak dibuat *path*nya. Karena dalam topologi ini terdapat 7 *host* maka ada 7 kondisi if dan elif yang ada. Gambar 4.16 merupakan penjelasan tentang program *routing*, yaitu:

1. Baris 39-42 adalah jika ada paket dari host manapun yang menuju host 10.0.0.1. simpan isi *class build_path* di p1 lalu panggil fungsi *dij_routing*. *Path1* digunakan untuk membuat *source path*.
2. Baris 45 adalah *dictionary* untuk mencatat *source*, *destination* dan *path* yang dilalui.
3. Baris 47-51 adalah jika *dictionary* sudah memiliki catatan host 10.0.0.1 maka *update* data tetapi jika *dictionary* belum memiliki catatan host 10.0.0.1 maka tambahkan isi data untuk host 10.0.0.1.
4. Baris 53-55 adalah simpan isi dari *class build_path* di p2 kemudian panggil fungsi *dij_routing*. Pada p2 digunakan untuk *destination path.*

```
39          if pkt['dstip']==IPAddr("10.0.0.1"):
40              p1=build_path(self.topology,1,
41  pkt['switch'],pkt['inport'])
42              path1=p1.dij_routing()
43
44              #self.update_wG(self.wG,path1)
45              dc=build_dic(IPAddr("10.0.0.1"),
46  pkt['srcip'],path1)
47              if self.dic.has_key(IPAddr("10.0.0.1")):
48                  self.dic[IPAddr("10.0.0.1")].update
49  (dc[IPAddr("10.0.0.1")])
50              else:
51                  self.dic.update(dc)
52
53              p2=build_path(self.topology,
54  pkt['switch'],1,1)
55              path2=p2.dij_routing()
56
57              dc=build_dic(pkt['srcip'],IPAddr
58  ("10.0.0.1"),path2)
59              if self.dic.has_key(pkt['srcip']):
60                  self.dic[pkt['srcip']].update
61  (dc[pkt['srcip']])
62              else:
63                  self.dic.update(dc)
64              self.forward =build_flowtable(self.dic)
```

Gambar 4.16 Kode Program *Routing*

Membuat *routing* dibutuhkan *ip address*, *switch* dan *egress port* sebagai *input*. Pada *routing* dilakukan proses untuk mengetahui nilai tiap *edge*, algoritma untuk menentukan jalur terpendek, dan *forwarding rules*.

Untuk proses implementasi *routing* dapat dilihat pada gambar 4.17. pada proses gambar 4.17akan dilakukan dengan memberikan nilai pada tiap jalur.

```
1   .............
2   for u in G.edge:
3                   dist[u]={}
4                   for v in G.edge:
5                       #print type(v)
6                       #print G.has_edge
7                       if G.has_edge(u,v):
8               if u== 1 and v==2 or u== 2 and v==1 :
9                                   dist[u][v]= 8 #seting weight
10              elif u== 1 and v==4 or u== 4 and v==1 :
11                  dist[u][v]= 2
12              elif u== 2 and v==3 or u== 3 and v==2 :
13                  dist[u][v]= 2
14              elif u== 3 and v==4 or u== 4 and v==3 :
15                  dist[u][v]= 5
16              elif u== 3 and v==7 or u== 7 and v==3 :
17                  dist[u][v]= 3
18              elif u== 4 and v==5 or u== 5 and v==4 :
19                  dist[u][v]= 10
20              elif u== 4 and v==6 or u== 6 and v==4 :
21                  dist[u][v]= 7
22              elif u== 5 and v==6 or u== 6 and v==5 :
23                  dist[u][v]= 3
24              elif u== 6 and v==7 or u== 7 and v==6 :
25                  dist[u][v]= 9
26
27                      else:
28                          dist[u][v]=huge
29                  dist[u][u]=0
30  ...................
```

Gambar 4.17 *Dijkstra Algorithm*

Implementasi *routing* dilakukan dengan memberi nilai pada tiap *jalur* dimana tiap *jalur* akan dikonfigurasi dengan memberi bobot weight antar *switch*. Sehingga antar *switch* akan diberi bobot sesuai keinginan untuk *switch* terdekat dan jika tujuan *switch* adalah *switch* itu sendiri, maka nilainya adalah 0.

Gambar 4.17 penjelasan untuk memberi nilai di tiap link, yaitu:

1. Baris 8-9 adalah jika *switch1* dan *switch2* atau sebaliknya akan diberi bobot 8.
2. baris 10-11 adalah jika *switch1* dan *switch4* atau sebaliknya akan diberi bobot 2.

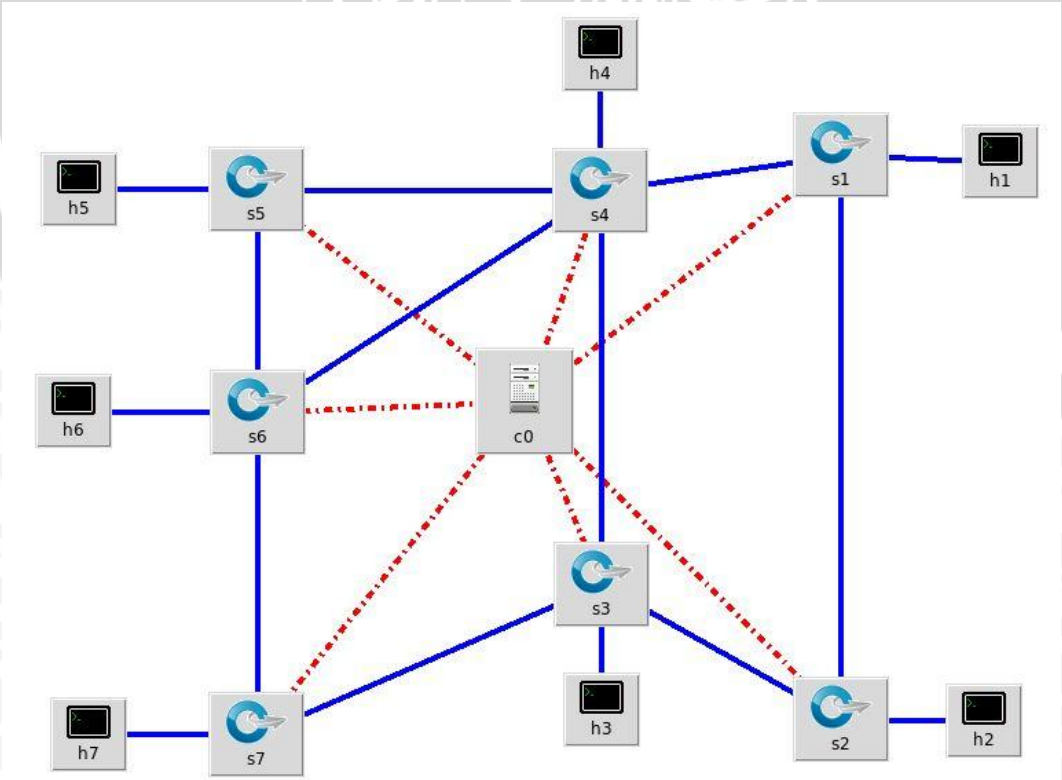
3. Baris 28 adalah jika tidak dekat atau bukan tetangga dekat maka bernilai *infinite*.
4. Baris 29 tujuan *switch* adalah *switch* itu sendiri jadi bobotnya 0.

BAB 5 PENGUJIAN DAN ANALISIS

Pada bab ini menjelaskan tentang pengujian sistem yang telah melalui tahap implementasi dan melakukan analisis terhadap hasil pengujian sistem tersebut. Pengujian yang dilakukan bertujuan untuk mengetahui semua kebutuhan pada pengujian perangkat keras dan perangkat lunak yang digunakan pada penelitian. Untuk melakukan pengujian akan dilakukan perubahan pada program agar dapat disesuaikan dengan kebutuhan pengujian. Dalam setiap pengujian akan dilakukan sesuai dengan tahapan pada skenario. Analisis dari hasil pengujian yang dilakukan akan ditarik kesimpulan pada setiap pengujian.

5.1 Pengujian

Pengujian dilakukan berdasarkan pada perancangan topologi sebelumnya dengan membuat topologi pada emulator mininet. Topologi yang digunakan adalah topologi mesh dengan menggunakan penerapan *dijkstra algorithm*. Pengujian dilakukan dengan menggunakan 3 skenario yang berbeda, yaitu sebelum dan ketika terjadi *fail path*. Didalam 3 skenario akan dilakukan pengujian topologi, *throughput*, *latency* dan *convergence*. Pengujian dilakukan untuk melihat apakah sistem telah bekerja dengan baik dan dapat menghasilkan jalur terpendek lain ketika terjadi fail path. Topologi yang dibuat dapat dilihat pada gambar 5.1.



Gambar 5.1 Topologi

Controller yang digunakan adalah *controller pyretic*. Implementasi penentuan jalur terpendek menggunakan *pyretic*. Dengan *virtual ip address*. h pada gambar topologi diatas adalah akronim dari *host*, s adalah akronim dari *switch* dan c0 adalah akronim dari *controller*. Sehingga h1 adalah *host*1.

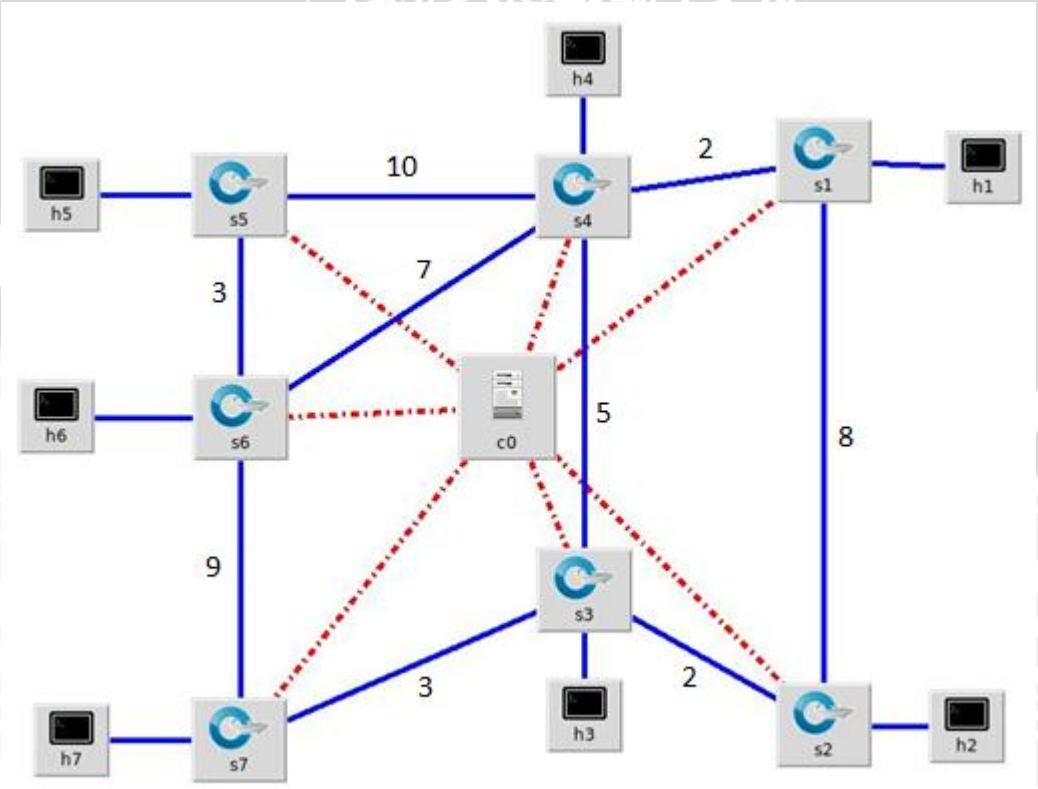
5.1.1 Pengujian Skenario 1

Gambar 5.2 dilakukan pengujian dengan menerapkan *dijkstra algorithm* pada *software defined network* menggunakan topologi dengan 7 *switch* dan 7 *host*. Topologi yang dibuat diberi nilai pada masing-masing jalur yang menghubungkan antara *switch*, sehingga dapat diketahui *cost* dengan nilai terkecil dan *cost* yang bernilai *infinite* karena *switch* tidak terhubung. Untuk membuat *switch* saling terhubung dengan menerapkan *dijkstra algorithm*.

Tabel 5.1 merupakan keterangan yang ada pada jalur di topologi.

Tabel 5.1 *Link* Topologi

1	———	Untuk jalur yang terhubung
2	– – – – – –	Untuk *link down* atau *fail path*
3	.....	Untuk *switch* yang terhubung ke *controller*



Gambar 5.2 Topologi Skenario 1

5.1.1.1 Prosedur Pengujian Skenario 1

1. Pengujian topologi untuk menghasilkan jalur terpendek.

   Menghasilkan jalur terpendek dilakukan dengan tidak terjadi *fail path*. Pengujian dilakukan dengan menentukan jalur terpendek dari h1 menuju h7 dengan melakukan *ping*.

2. Pengujian *throughput* untuk menghitung paket yang sampai pada tujuan.

   Mengirimkan paket *iperf* dilakukan dengan h7 sebagai *client* menuju ke h1 sebagai *server* dengan tidak adanya gangguan *fail path*. Pengujian dilakukan dengan mengamati jumlah paket terkirim yang datang pada destination selama interval waktu tertentu yang dibagi oleh durasi interval waktu tersebut.

3. Pengujian *latency* untuk menghitung waktu sebuah paket sampai ke tujuan.

   Pengujian dilakukan dengan tidak terjadi *fail path*. Pengujian menggunakan perintah ping dengan h7 mengirimkan paket sebesar 250 byte pada h1.

4. Pengujian *convergence* untuk menghitung waktu yang dibutuhkan untuk menemukan jalur yang baru ketika terjadi *fail path.*

   Pengujian dilakukan dengan mengirimkan paket *ping*, yaitu h1 ping h7 dengan tidak terjadi *fail path*.

5.1.1.2 Hasil Pengujian Skenario 1

1. Hasil Pengujian Topologi

Dari hasil percobaan *ping* pada gambar 5.3 tidak ada jalur yang terputus atau bisa terhubung.

```
mininet> h1 ping h7
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_seq=1 ttl=64 time=1204 ms
64 bytes from 10.0.0.7: icmp_seq=2 ttl=64 time=204 ms
64 bytes from 10.0.0.7: icmp_seq=3 ttl=64 time=0.124 ms
64 bytes from 10.0.0.7: icmp_seq=4 ttl=64 time=0.057 ms
64 bytes from 10.0.0.7: icmp_seq=5 ttl=64 time=0.088 ms
^C
--- 10.0.0.7 ping statistics ---
5 packets transmitted, 5 received, 0% packet loss, time 4001ms
rtt min/avg/max/mdev = 0.057/281.842/1204.411/468.030 ms, pipe 2
```

Gambar 5.3 Skenario 1 h1 ping h7

Gambar 5.4 merupakan *cost* dari *host* 1 menuju *host* 7. Jika antar *switch* tidak saling terhubung maka bernilai *infinite*. *Cost* di jalur dinotasikan dengan Q. Gambar 5.4 menjelaskan bahwa *switch* 1 terhubung langsung dengan *switch* 2 dan *switch* 4 sehingga *cost* menuju *switch* 3, *switch* 5, *switch* 7 dan *switch* 6 adalah *infinite*. Pada iterasi kedua diambil jalur menuju *switch* 3 lewat *switch* 4 sehingga *cost* menuju *switch* 3 adalah 7, *switch* 5 lewat *switch* 4 sehingga *cost*

yang menuju *switch* 5 adalah 12 dan *switch* 6 lewat *switch* 4 sehingga *cost* yang menuju *switch* 6 adalah 9. Pada iterasi ketiga diambil jalur menuju *switch* 7 lewat *switch* 3 sehingga *cost* yang menuju *switch* 7 adalah 10.

```
Q :  {'1': 0}
Q :  {'3': inf, '2': 8, '5': inf, '4': 2, '7': inf, '6': inf}
Q :  {'3': 7, '2': 8, '5': 12, '7': inf, '6': 9}
Q :  {'2': 8, '5': 12, '7': 10, '6': 9}
Q :  {'5': 12, '7': 10, '6': 9}
Q :  {'5': 12, '7': 10}
Q :  {'5': 12}
```

Gambar 5.4 Jalur *Cost* Skenario 1

Gambar 5.5 *cost* yang dihasilkan dari *host* 1 menuju *host* 7. *Switch* 1 bernilai 0 karena *switch* 1 merupakan *source* jadi bernilai 0. *Switch* 3 bernilai 7 karena *switch* 3 sebelumnya lewat *switch* 4 jadi bernilai 2 ditambahkan dengan nilai menuju *switch* 3 maka bernilai 7. *Switch* 2 bernilai 8 karena *switch* 1 menuju *switch* 2 bernilai 8. *Switch* 5 bernilai 12 karena *switch* 1 menuju *switch* 4 bernilai 2 ditambah dengan *cost* dari *switch* 4 menuju *switch* 5, yaitu 10 sehingga bernilai 12. *Switch* 4 bernilai 2 karena *switch* 1 menuju *switch* 4 bernilai 2. *Switch* 7 bernilai 10 karena jalur yang dilewati sebelumnya, yaitu *switch* 1 menuju *switch* 4 bernilai 2, *switch* 4 munuju *switch* 3 bernilai 5 dan *switch* 3 menuju *switch* 7 bernilai 3 jadi hasil menuju *switch* 7 bernilai 10. *Switch* 6 bernilai 9 karena jalur yang dilewati sebelumnya, yaitu *switch* 1 menuju *switch* 4 bernilai 2 dan *switch* 4 menuju *switch* 6 bernilai 7 jadi nilai yang dihasilkan 9.

```
D :  {'1': 0, '3': 7, '2': 8, '5': 12, '4': 2, '7': 10, '6': 9}
```

Gambar 5.5 *Current Value* Skenario 1

Gambar 5.6 *switch* yang harus dilewati dari *source* menuju *destination*. Ketika menuju *switch* 2 harus melalui *switch* 1, menuju *switch* 3 melalui *switch* 4, menuju *switch* 4 melalui *switch* 1, menuju *switch* 5 melalui *switch* 4, menuju *switch* 6 melalui *switch* 4 dan menuju *switch* 7 melalui *switch* 3.

```
P  :  {2: 1, 3: 4, 4: 1, 5: 4, 6: 4, 7: 3}
```

Gambar 5.6 *Predecessor* Skenario 1

Gambar 5.7 hasil dari pencarian jalur terpendek dari *host* 1 menuju *host* 7. Dapat dilihat bahwa *switch* yang dilalui dari *source* menuju *destination* yaitu 1 dengan nomor port [3], 4 dengan nomor port [3], 3 dengan nomor port [4], 7 dengan nomor port [1] dan hasil yang didapat dari jarak terdekat adalah 10.

```
pathlist untuk switch yang dilalui:
1[3]
4[3]
3[4]
7[1]
Hasil Jarak Terdekat:
10
```

Gambar 5.7 Jalur Terpendek Skenario 1

Dari pengujian topologi skenario 1 dengan h1 ping h7 hasilnya sudah dapat menentukan jalur terpendek. Jalur tependek dari h1 menuju h7, yaitu melalui h1, h4, h3, h7 dengan hasil jarak terdekat 10.

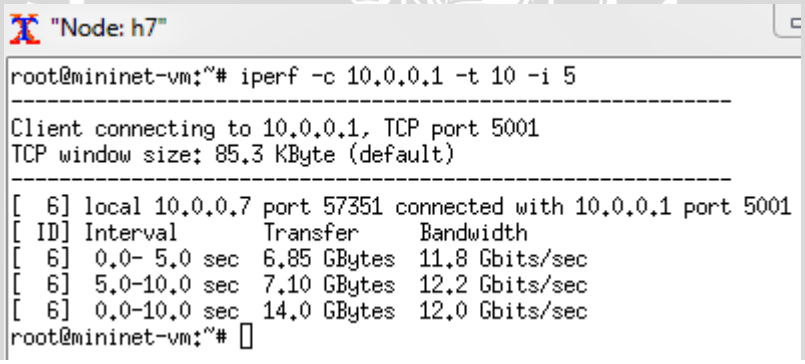
2. Hasil Pengujian *Throughput*

Pengujian *throughput* dilakukan dengan tidak terjadi *fail path*. Host 7 mengirimkan paket *iperf* menuju host 1 dengan *timing* 10 dan *interval* 5.

Tabel 5.2 Sintaks pada *iperf test*

-c	sebagai client
10.0.0.1	no ip server
-t	*Timing*
-i	*Interval*

Tabel 5.2 merupakan penjelasan dari sintaks yang digunakan pada gambar 5.8 untuk pengujian *throughput*.



Gambar 5.8 Uji *Throughput* Skenario 1

Gambar 5.8 untuk pengujian *throughput* hasil yang diperoleh merupakan jumlah paket terkirim yang datang pada destination selama interval waktu tertentu dibagi dengan durasi interval waktu, yaitu dalam waktu 10 detik data yang terkirim di transfer sebanyak 14 GBytes. Jadi jumlah throughputnya adalah 1,4 GBytes/sec.

3. Hasil Pengujian *Latency*

Pengujian *latency* dilakukan dengan tidak terjadi *fail path*. Dengan perintah *ping*, yaitu h7 mengirimkan paket sebesar 250*byte* pada h1 dan pengiriman paket sebanyak 7 kali.

Tabel 5.3 Sintaks Uji *Latency*

-s	Sebagai *server*
250	Ukuran paket adalah 250 *byte*
-c	Sebagai *client*
7	Paket yang dikirim sebanyak 7 kali
10.0.0.1	no ip *server*

Tabel 5.3 merupakan penjelasan sintaks yang digunakan pada gambar 6.9 untuk pengujian latency.

```
root@mininet-vm:~# ping -s 250 -c 7 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 250(278) bytes of data.
258 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.038 ms
258 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.110 ms
258 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.049 ms
258 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.103 ms
258 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.333 ms
258 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=0.099 ms
258 bytes from 10.0.0.1: icmp_seq=7 ttl=64 time=0.403 ms

--- 10.0.0.1 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6006ms
rtt min/avg/max/mdev = 0.038/0.162/0.403/0.134 ms
```

Gambar 5.9 Uji *Latency* Skenario 1

Gambar 5.9 untuk nilai *latency* diperoleh dari menghitung waktu yang dibutuhkan paket untuk mencapai tujuan, yaitu jumlah seluruh *time* kemudian bagi dengan banyaknya paket yang sampai. Nilai *latency* adalah 0.162ms.

4. Hasil Pengujian *Convergence*

Pengujian dilakukan dengan tidak terjadi *fail path.* Dengan perintah *ping,* yaitu h1 ping h7.

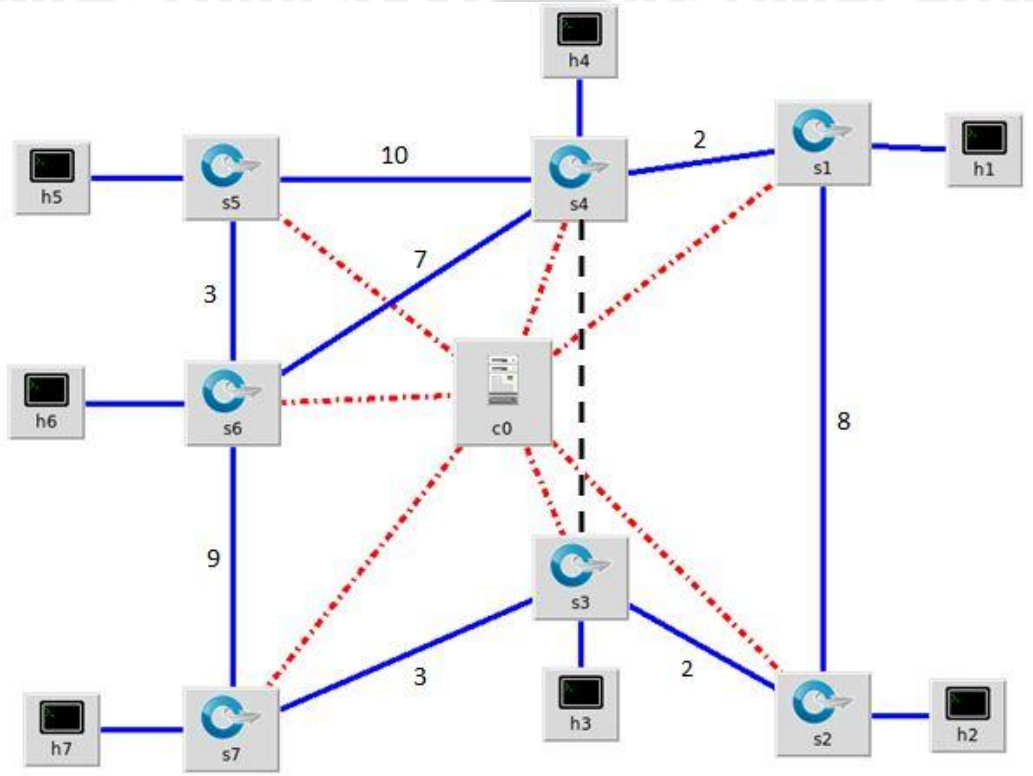
401	14.026350000	10.0.0.1	10.0.0.7	ICMP	98 Echo (ping) request id=0x227b, seq=1/256, ttl=64 (reply in 403)
402	14.026353000	10.0.0.1	10.0.0.7	ICMP	98 Echo (ping) request id=0x227b, seq=2/512, ttl=64 (reply in 404)
403	14.026717000	10.0.0.7	10.0.0.1	ICMP	98 Echo (ping) reply id=0x227b, seq=1/256, ttl=64 (request in 401)
404	14.026728000	10.0.0.7	10.0.0.1	ICMP	98 Echo (ping) reply id=0x227b, seq=2/512, ttl=64 (request in 402)
409	14.026350000	10.0.0.1	10.0.0.7	ICMP	100 Echo (ping) request id=0x227b, seq=1/256, ttl=64 (reply in 411)
410	14.026353000	10.0.0.1	10.0.0.7	ICMP	100 Echo (ping) request id=0x227b, seq=2/512, ttl=64 (reply in 412)
411	14.026717000	10.0.0.7	10.0.0.1	ICMP	100 Echo (ping) reply id=0x227b, seq=1/256, ttl=64 (request in 409)
412	14.026728000	10.0.0.7	10.0.0.1	ICMP	100 Echo (ping) reply id=0x227b, seq=2/512, ttl=64 (request in 410)
419	14.510201000	10.0.0.1	10.0.0.7	ICMP	100 Echo (ping) request id=0x227b, seq=3/768, ttl=64 (reply in 420)
420	14.510542000	10.0.0.7	10.0.0.1	ICMP	100 Echo (ping) reply id=0x227b, seq=3/768, ttl=64 (request in 419)
425	14.510201000	10.0.0.1	10.0.0.7	ICMP	98 Echo (ping) request id=0x227b, seq=3/768, ttl=64 (reply in 426)
426	14.510542000	10.0.0.7	10.0.0.1	ICMP	98 Echo (ping) reply id=0x227b, seq=3/768, ttl=64 (request in 425)
456	15.512961000	10.0.0.1	10.0.0.7	ICMP	100 Echo (ping) request id=0x227b, seq=4/1024, ttl=64 (reply in 457)
457	15.513059000	10.0.0.7	10.0.0.1	ICMP	100 Echo (ping) reply id=0x227b, seq=4/1024, ttl=64 (request in 456)

Gambar 5.10 Uji *Convergence* Skenario 1

Gambar 5.10 tidak menghasilkan nilai *convergence* karena tidak terjadi *fail path* sehingga tidak ada waktu yang dibutuhkan untuk menemukan jalur baru.

5.1.2 Pengujian Skenario 2

Pada pengujian skenario 2 dilakukan dengan terjadi *fail path* pada topologi jaringan. Gambar 5.11 terjadi *fail path* pada jalur antara s3 dan s4.



Gambar 5.11 Topologi Skenario 2

5.1.2.1 Prosedur Pengujian Skenario 2

1. Pengujian topologi untuk menghasilkan jalur terpendek.

   Menghasilkan jalur terpendek dilakukan dengan terjadi *fail path* antara switch 3 dan switch 4. Pengujian dilakukan dengan menentukan jalur terpendek dari h1 menuju h7 dengan melakukan *ping*.

2. Pengujian *throughput* untuk menghitung paket yang sampai pada tujuan.

   Mengirimkan paket *iperf* dilakukan dengan h7 sebagai *client* menuju ke h1 sebagai *server* dengan terjadi *fail path* antara switch 4 dan switch 3. Pengujian dilakukan dengan mengamati jumlah paket terkirim yang datang pada destination selama interval waktu tertentu yang dibagi oleh durasi interval waktu tersebut.

3. Pengujian *latency* untuk menghitung waktu sebuah paket sampai ke tujuan.

Pengujian dilakukan dengan terjadi *fail path* antara switch 3 dan switch 4. Pengujian menggunakan perintah *ping* dengan mengirimkan paket sebesar 250 byte dengan pengiriman paket sebanyak 7 kali.

4. Pengujian *convergence* untuk menghitung waktu yang dibutuhkan untuk menemukan jalur yang baru ketika terjadi *fail path.*

Pengujian dilakukan dengan mengirimkan paket *ping*, yaitu h1 ping h7 dengan terjadi *fail path* antara switch 4 dan switch 3.

5.1.2.2 Hasil Pengujian Skenario 2

1. Hasil Pengujian Topologi

Pada pengujian skenario 2 dilakukan dengan terjadi *fail path* pada jaringan. Pengujian dilakukan dengan menentukan jalur terpendek dari h1 menuju h7 dengan melakukan ping. Dari hasil percobaan ping pada gambar 5.12 tidak ada jalur yang terputus atau bisa terhubung.

```
mininet> h1 ping h7
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_seq=2 ttl=64 time=184 ms
64 bytes from 10.0.0.7: icmp_seq=3 ttl=64 time=1.00 ms
64 bytes from 10.0.0.7: icmp_seq=4 ttl=64 time=0.102 ms
64 bytes from 10.0.0.7: icmp_seq=5 ttl=64 time=0.052 ms
64 bytes from 10.0.0.7: icmp_seq=6 ttl=64 time=0.071 ms
^C
--- 10.0.0.7 ping statistics ---
6 packets transmitted, 5 received, 16% packet loss, time 5010ms
rtt min/avg/max/mdev = 0.052/37.059/184.072/73.507 ms
```

Gambar 5.12 Skenario 2 dengan h1 ping h7

Dengan adanya fail path sehingga hasil dari jalur terpendek sebelumnya pada gambar 5.7 tidak dapat terhubung dan harus mencari jalur terpendek yang lain. Dengan menggunakan algoritma dijkstra dapat dilakukan *routing* ulang untuk menentukan jalur terpendek lain yang akan dilewati.

Gambar 5.13 merupakan cost dari *host* 1 menuju *host* 7. Jika antar *switch* tidak saling terhubung maka bernilai *infinite*. *Cost* pada jalur dinotasikan dengan Q. Gambar 5.13 menjelaskan bahwa *switch* 1 terhubung langsung dengan *switch* 2 dan *switch* 4 sehingga *cost* menuju *switch* 3, *switch* 5, *switch* 7 dan *switch* 6 adalah *infinite*. Pada iterasi kedua diambil jalur menuju *switch* 5 lewat *switch* 4 sehingga *cost* menuju *switch* 5 adalah 12 dan *switch* 6 lewat *switch* 4 sehingga cost yang menuju *switch* 6 adalah 9. Pada iterasi ketiga diambil jalur menuju *switch* 3 lewat *switch* 2 sehingga *cost* yang menuju *switch* 3 adalah 10. Pada iterasi keempat diambil jalur menuju *switch* 7 lewat *switch* 6 sehingga *cost* yang menuju *switch* 7 adalah 18.

```
Q :  {'1': 0}
Q :  {'3': inf, '2': 8, '5': inf, '4': 2, '7': inf, '6': inf}
Q :  {'3': inf, '2': 8, '5': 12, '7': inf, '6': 9}
Q :  {'3': 10, '5': 12, '7': inf, '6': 9}
Q :  {'3': 10, '5': 12, '7': 18}
Q :  {'5': 12, '7': 13}
Q :  {'7': 13}
```

Gambar 5.13 Jalur *Cost* Skenario 2

Gambar 5.14 *cost* yang dihasilkan dari *host* 1 menuju *host* 7. *Switch* 1 bernilai 0 karena *switch* 1 merupakan *source* jadi bernilai 0. *Switch* 3 bernilai 10 karena *switch* 3 sebelumnya lewat *switch* 2 jadi bernilai 2 ditambahkan dengan nilai menuju *switch* 3 maka bernilai 10. *Switch* 2 bernilai 8 karena *switch* 1 menuju *switch* 2 bernilai 8. *Switch* 5 bernilai 12 karena *switch* 1 menuju *switch* 4 bernilai 2 ditambah dengan *cost* dari *switch* 4 menuju *switch* 5, yaitu 10 sehingga bernilai 12. *Switch* 4 bernilai 2 karena *switch* 1 menuju *switch* 4 bernilai 2. *Switch* 7 bernilai 13 karena jalur yang dilewati sebelumnya, yaitu *switch* 1 menuju *switch* 2 bernilai 8, *switch* 2 munuju *switch* 3 bernilai 2 dan *switch* 3 menuju *switch* 7 bernilai 3 jadi hasil menuju *switch* 7 bernilai 13. *Switch* 6 bernilai 9 karena jalur yang dilewati sebelumnya, yaitu *switch* 1 menuju *switch* 4 bernilai 2 dan *switch* 4 menuju *switch* 6 bernilai 7 jadi nilai yang dihasilkan 9.

```
D :  {'1': 0, '3': 10, '2': 8, '5': 12, '4': 2, '7': 13, '6': 9}
```

Gambar 5.14 *Current Value* Skenario 2

Gambar 5.15 *switch* yang harus dilewati dari *source* menuju *destination*. Ketika menuju *switch* 2 harus melalui *switch* 1, menuju *switch* 3 melalui *switch* 2, menuju *switch* 4 melalui *switch* 1, menuju *switch* 5 melalui *switch* 4, menuju *switch* 6 melalui *switch* 4 dan menuju *switch* 7 melalui *switch* 3.

```
P  :  {2: 1, 3: 2, 4: 1, 5: 4, 6: 4, 7: 3}
```

Gambar 5.15 *Predecessor* Skenario 2

Gambar 5.16 hasil dari pencarian jalur terpendek dari *host* 1 menuju *host* 7. Dapat dilihat bahwa *switch* yang dilalui dari *source* menuju *destination* yaitu 1 dengan nomor *port* [2] , 2 dengan nomor *port* [3], 3 dengan nomor *port* [4], 7 dengan nomor *port* [1] dan hasil yang didapat dari jarak terdekat adalah 13.

```
pathlist untuk switch yang dilalui:
1[2]
2[3]
3[4]
7[1]
Hasil Jarak Terdekat:
13
```

Gambar 5.16 Jalur Terpendek skenario 2

Dari pengujian topologi skenario 2 ketika terjadi *link down* antara h4 dan h3 dengan melakukan *ping*, yaitu h1 *ping* h7 sudah dapat melakukan pengujian

untuk menentukan jalur tependek lain dengan melalui h1, h2, h3, h7 dan hasil yang didapat dari jarak terdekat adalah 13.

2. Hasil Pengujian *Throughput*

Pengujian *throughput* dilakukan dengan terjadi *fail path* antara *switch* 3 dan *switch* 4. *Host* 7 mengirimkan paket *iperf* menuju *host* 1 dengan *timing* 10 dan *interval* 5.

```
root@mininet-vm:~# iperf -c 10.0.0.1 -t 10 -i 5
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 10.0.0.7 port 57352 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0- 5.0 sec  6.69 GBytes  11.5 Gbits/sec
[  6]  5.0-10.0 sec  4.43 GBytes  7.62 Gbits/sec
[  6]  0.0-10.0 sec  11.1 GBytes  9.56 Gbits/sec
root@mininet-vm:~#
```

Gambar 5.17 Uji *Throughput* Skenario 2

Gambar 5.17 untuk pengujian *throughput* hasil yang diperoleh merupakan jumlah paket terkirim yang datang pada destination selama interval waktu tertentu dibagi dengan durasi interval waktu, yaitu dalam waktu 10 detik data yang terkirim di transfer sebanyak 11,1 GBytes. Jadi jumlah throughputnya adalah 1,11 GBytes/sec.

3. Hasil Pengujian *Latency*

Pengujian *latency* dilakukan dengan terjadi *fail path* antara switch 3 dan switch 4. Dengan perintah *ping*, yaitu h7 mengirimkan paket sebesar 250byte pada h1 dan pengiriman paket sebanyak 7 kali.

```
root@mininet-vm:~# ping -s 250 -c 7 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 250(278) bytes of data.
258 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.534 ms
258 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.110 ms
258 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.112 ms
258 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.136 ms
258 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.120 ms
258 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=0.109 ms
258 bytes from 10.0.0.1: icmp_seq=7 ttl=64 time=0.212 ms

--- 10.0.0.1 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6009ms
rtt min/avg/max/mdev = 0.109/0.190/0.534/0.144 ms
```

Gambar 5.18 Uji *Latency* Skenario 2

Gambar 5.18 untuk nilai *latency* diperoleh dari menghitung waktu yang dibutuhkan paket untuk mencapai tujuan, yaitu jumlah seluruh *time* kemudian bagi dengan banyaknya paket yang sampai. Nilai *latency* adalah 0.190ms.

4. Hasil Pengujian *Convergence*

Pengujian dilakukan dengan terjadi *fail path* antara *switch* 3 dan *switch* 4. Dengan perintah *ping*, yaitu h1 ping h7

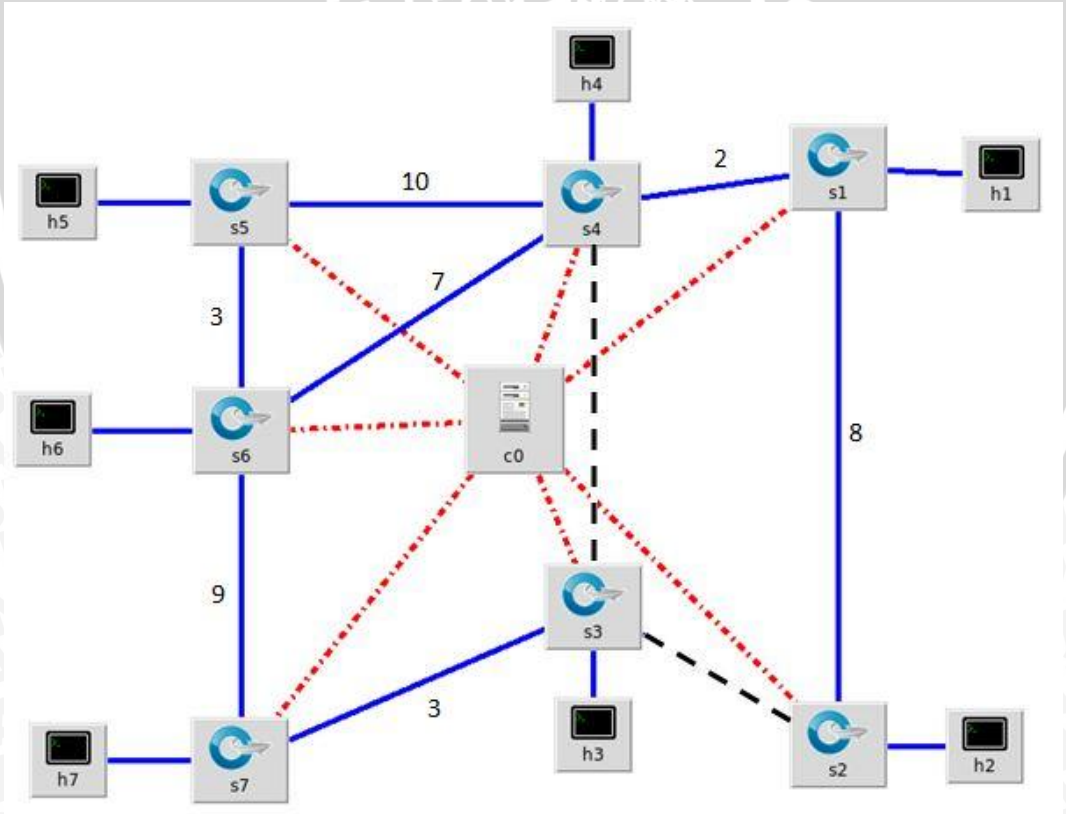
461	15.512961000	10.0.0.1	10.0.0.7	ICMP	98 Echo (ping) request id=0x227b, seq=4/1024, ttl=64 (reply in 462)
462	15.513059000	10.0.0.7	10.0.0.1	ICMP	98 Echo (ping) reply id=0x227b, seq=4/1024, ttl=64 (request in 461)
471	16.513573000	10.0.0.1	10.0.0.7	ICMP	100 Echo (ping) request id=0x227b, seq=5/1280, ttl=64
472	16.513573000	10.0.0.1	10.0.0.7	ICMP	98 Echo (ping) request id=0x227b, seq=5/1280, ttl=64
480	17.522913000	10.0.0.1	10.0.0.7	ICMP	100 Echo (ping) request id=0x227b, seq=6/1536, ttl=64
482	17.522913000	10.0.0.1	10.0.0.7	ICMP	98 Echo (ping) request id=0x227b, seq=6/1536, ttl=64 (reply in 483)
483	17.756617000	10.0.0.7	10.0.0.1	ICMP	98 Echo (ping) reply id=0x227b, seq=6/1536, ttl=64 (request in 482)

Gambar 5.19 Uji *Convergence* Skenario 2

Gambar 5.19 waktu rata-rata yang dibutuhkan sebuah jaringan untuk menemukan jalur yang baru ketika terjadi *fail path*. Nilai *convergence* diperoleh dari perhitungan waktu sebelum terjadi *fail path* dikurangi dengan waktu menemukan jalur baru. Nilai *convergence*, yaitu 2,009ms.

5.1.3 Pengujian Skenario 3

Pada pengujian skenario 3 dilakukan dengan adanya *fail path* pada topologi jaringan. Gambar 5.20 *fail path* terjadi pada jalur antara s3 dan s4 dan pada jalur antara s2 dan s3.



Gambar 5.20 Topologi Skenario 3

5.1.3.1 Prosedur Pengujian Skenario 3

1. Pengujian Topologi untuk menghasilkan jalur terpendek.

   Menghasilkan jalur terpendek dilakukan dengan adanya *fail path* antara h3, h4 dan h2, h3. Pengujian dilakukan dengan menentukan jalur terpendek dari h1 menuju h7 dengan melakukan *ping*.

2. Pengujian *Throughput* untuk menghitung paket yang sampai pada tujuan.

   Mengirimkan paket *iperf* dilakukan dengan h7 sebagai *client* menuju ke h1 sebagai *server* dengan terjadi *fail path* antara h3, h4 dan h2, h3. Pengujian dilakukan dengan mengamati jumlah paket terkirim yang datang pada destination selama interval waktu tertentu yang dibagi oleh durasi interval waktu tersebut.

3. Pengujian *Latency* untuk menghitung waktu sebuah paket sampai ke tujuan.

   Pengujian dilakukan dengan terjadi *fail path* antara h3, h4 dan h2, h3. Pengujian menggunakan perintah *ping* dengan mengirimkan data sebesar 250 *byte* dengan pengiriman paket sebanyak 7 kali.

4. Pengujian *convergence* untuk menghitung waktu yang dibutuhkan untuk menemukan jalur yang baru ketika terjadi *fail path.*

   Pengujian dilakukan dengan mengirimkan paket *ping*, yaitu h1 ping h7 dengan terjadi *fail path* antara h3, h4 dan h2, h3.

5.1.3.2 Hasil Pengujian Skenario 3

1. Hasil pengujian Topologi

Pengujian skenario 3 dilakukan dengan adanya terjadi *fail path* pada topologi jaringan. Pengujian dilakukan dengan menentukan jalur terpendek dari h1 menuju h7 dengan melakukan *ping*. Dari hasil percobaan *ping* pada gambar 5.21 tidak ada jalur yang terputus atau bisa terhubung.

```
mininet> h1 ping h7
PING 10.0.0.7 (10.0.0.7) 56(84) bytes of data.
64 bytes from 10.0.0.7: icmp_seq=2 ttl=64 time=174 ms
64 bytes from 10.0.0.7: icmp_seq=3 ttl=64 time=1.08 ms
64 bytes from 10.0.0.7: icmp_seq=4 ttl=64 time=0.081 ms
64 bytes from 10.0.0.7: icmp_seq=5 ttl=64 time=0.058 ms
64 bytes from 10.0.0.7: icmp_seq=6 ttl=64 time=0.107 ms
^C
--- 10.0.0.7 ping statistics ---
6 packets transmitted, 5 received, 16% packet loss, time 5010ms
rtt min/avg/max/mdev = 0.058/35.107/174.206/69.550 ms
```

Gambar 5.21 Skenario 3 dengan h1 ping h7

Dengan adanya *fail path* sehingga hasil dari jalur terpendek sebelumnya pada gambar 5.7 dan gambar 5.13 tidak dapat terhubung dan harus mencari jalur terpendek yang lain. Dengan menggunakan *dijkstra algorithm* akan dapat

dilakukan *routing* ulang untuk menentukan jalur terpendek lain yang akan dilewati.

Gambar 5.22 merupakan jalur *cost* dari *host* 1 menuju *host* 7. Jika antar *switch* tidak saling terhubung maka bernilai *infinite*. *Cost* pada jalur dinotasikan dengan Q. Gambar 5.22 menjelaskan bahwa *switch* 1 terhubung langsung dengan *switch* 2 dan *switch* 4 sehingga *cost* menuju *switch* 3, *switch* 5, *switch* 7 dan *switch* 6 adalah *infinite*. Pada iterasi kedua diambil jalur menuju *switch* 5 lewat *switch* 4 sehingga *cost* menuju *switch* 5 adalah 12 dan *switch* 6 lewat *switch* 4 sehingga *cost* yang menuju *switch* 6 adalah 9. Pada iterasi ketiga diambil jalur menuju *switch* 7 lewat *switch* 6 sehingga *cost* yang menuju *switch* 7 adalah 18.

```
Q :  {'1': 0}
Q :  {'3': inf, '2': 8, '5': inf, '4': 2, '7': inf, '6': inf}
Q :  {'3': inf, '2': 8, '5': 12, '7': inf, '6': 9}
Q :  {'3': inf, '5': 12, '7': inf, '6': 9}
Q :  {'3': inf, '5': 12, '7': 18}
Q :  {'3': inf, '7': 18}
Q :  {'3': 21}
```

Gambar 5.22 Jalur *Cost* Skenario 3

Gambar 5.23 *cost* yang dihasilkan dari *host* 1 menuju *host* 7. *Switch* 1 bernilai 0 karena *switch* 1 merupakan *source* jadi bernilai 0. *Switch* 3 bernilai 21 karena jalur yang dilewati sebelumnya, yaitu *switch* 1 menuju *switch* 4 bernilai 2, *switch* 4 menuju *switch* 6 bernilai 7, *switch* 6 menuju *switch* 7 bernilai 9 dan *switch* 7 menuju *switch* 3 bernilai 3 jadi nilai yang dihasilkan 21. *Switch* 2 bernilai 8 karena *switch* 1 menuju *switch* 2 bernilai 8. *Switch* 5 bernilai 12 karena *switch* 1 menuju *switch* 4 bernilai 2 ditambah dengan *cost* dari *switch* 4 menuju *switch* 5, yaitu 10 sehingga bernilai 12. *Switch* 4 bernilai 2 karena *switch* 1 menuju *switch* 4 bernilai 2. *Switch* 7 bernilai 18 karena jalur yang dilewati sebelumnya, yaitu *switch* 1 menuju *switch* 4 bernilai 2, *switch* 4 munuju *switch* 6 bernilai 7 dan *switch* 6 menuju *switch* 7 bernilai 9 jadi hasil menuju *switch* 7 bernilai 18. *Switch* 6 bernilai 9 karena jalur yang dilewati sebelumnya, yaitu *switch* 1 menuju *switch* 4 bernilai 2 dan *switch* 4 menuju *switch* 6 bernilai 7 jadi nilai yang dihasilkan 9.

```
D :  {'1': 0, '3': 21, '2': 8, '5': 12, '4': 2, '7': 18, '6': 9}
```

Gambar 5.23 *Current Value* Skenario 3

Gambar 5.24 *switch* yang harus dilewati dari *source* menuju *destination*. Ketika menuju *switch* 2 harus melalui *switch* 1, menuju *switch* 3 melalui *switch* 7, menuju *switch* 4 melalui *switch* 1, menuju *switch* 5 melalui *switch* 4, menuju *switch* 6 melalui *switch* 4 dan menuju *switch* 7 melalui *switch* 6.

```
P  :  {2: 1, 3: 7, 4: 1, 5: 4, 6: 4, 7: 6}
```

Gambar 5.24 *Predecessor* Skenario 3

Gambar 5.25 hasil dari pencarian jalur terpendek dari *host* 1 menuju *host* 7. Dapat dilihat bahwa *switch* yang dilalui dari *source* menuju *destination* yaitu 1

dengan nomor *port* [3], 4 dengan nomor *port* [5], 6 dengan nomor *port* [4], 7 dengan nomor *port* [1] dan hasil yang didapat dari jarak terdekat adalah 18.

```
pathlist untuk switch yang dilalui:
1[3]
4[5]
6[4]
7[1]
Hasil Jarak Terdekat:
18
```

Gambar 5.25 Jalur Terpendek skenario 3

Dari hasil pengujian topologi skenario 3 ketika terjadi *fail path* antara h4, h3 dan h2, h3 dengan melakukan ping, yaitu h1 ping h7 sudah dapat melakukan pengujian untuk menentukan jalur terpendek lain dari h1 menuju h7 adalah dengan melalui h1, h4, h6, h7 dan hasil yang didapat dari jarak terdekat adalah 18.

2. Hasil Pengujian *Throughput*

Pengujian *throughput* dilakukan dengan adanya *fail path* antara *switch* 3 dan *switch* 4, *switch* 2 dan *switch* 3. *Host* 7 mengirimkan paket *iperf* menuju *host* 1 dengan *timing* 10 dan *interval* 5.

```
root@mininet-vm:~# iperf -c 10.0.0.1 -t 10 -i 5
------------------------------------------------------------
Client connecting to 10.0.0.1, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 10.0.0.7 port 57353 connected with 10.0.0.1 port 5001
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0- 5.0 sec  6.36 GBytes  10.9 Gbits/sec
[  6]  5.0-10.0 sec  4.92 GBytes  8.46 Gbits/sec
[  6]  0.0-10.0 sec  11.3 GBytes  9.69 Gbits/sec
root@mininet-vm:~#
```

Gambar 5.26 Uji *Throughput* Skenario 3

Gambar 5.26 untuk pengujian *throughput* hasil yang diperoleh merupakan jumlah paket terkirim yang datang pada destination selama interval waktu tertentu dibagi dengan durasi interval waktu, yaitu dalam waktu 10 detik data yang terkirim di transfer sebanyak 11,3 GBytes. Jadi jumlah throughputnya adalah 1,13 GBytes/sec.

3. Hasil Pengujian *Latency*

Pengujian *latency* dilakukan dengan terjadi *fail path* antara switch 3 dan switch 4, switch 2 dan switch 3. Dengan perintah *ping*, yaitu h7 mengirimkan paket sebesar 250*byte* pada h1 dan pengiriman paket sebanyak 7 kali.

```
root@mininet-vm:~# ping -s 250 -c 7 10.0.0.1
PING 10.0.0.1 (10.0.0.1) 250(278) bytes of data.
258 bytes from 10.0.0.1: icmp_seq=1 ttl=64 time=0.050 ms
258 bytes from 10.0.0.1: icmp_seq=2 ttl=64 time=0.108 ms
258 bytes from 10.0.0.1: icmp_seq=3 ttl=64 time=0.105 ms
258 bytes from 10.0.0.1: icmp_seq=4 ttl=64 time=0.109 ms
258 bytes from 10.0.0.1: icmp_seq=5 ttl=64 time=0.453 ms
258 bytes from 10.0.0.1: icmp_seq=6 ttl=64 time=0.110 ms
258 bytes from 10.0.0.1: icmp_seq=7 ttl=64 time=0.117 ms

--- 10.0.0.1 ping statistics ---
7 packets transmitted, 7 received, 0% packet loss, time 6009ms
rtt min/avg/max/mdev = 0.050/0.150/0.453/0.125 ms
```

Gambar 5.27 Uji *Latency* Skenario 3

Gambar 5.27 untuk nilai *latency* diperoleh dari menghitung waktu yang dibutuhkan paket untuk mencapai tujuan, yaitu jumlah seluruh *time* kemudian bagi dengan banyaknya paket yang sampai. Nilai *latency* adalah 0.150ms.

4. Hasil Pengujian *Convergence*

Pengujian dilakukan dengan terjadi *link down* antara *switch* 3 *dan switch* 4*, switch* 2 *dan switch* 3. Dengan perintah *ping, yaitu* h1 ping h7.

572	21.530086000	10.0.0.7	10.0.0.1	ICMP	100	Echo (ping) reply id=0x227b, seq=10/2560, ttl=64 (request in 571)
576	21.529989000	10.0.0.1	10.0.0.7	ICMP	98	Echo (ping) request id=0x227b, seq=10/2560, ttl=64 (reply in 577)
577	21.530086000	10.0.0.7	10.0.0.1	ICMP	98	Echo (ping) reply id=0x227b, seq=10/2560, ttl=64 (request in 576)
588	22.530245000	10.0.0.1	10.0.0.7	ICMP	100	Echo (ping) request id=0x227b, seq=11/2816, ttl=64
589	22.530245000	10.0.0.1	10.0.0.7	ICMP	98	Echo (ping) request id=0x227b, seq=11/2816, ttl=64
611	23.539084000	10.0.0.1	10.0.0.7	ICMP	100	Echo (ping) request id=0x227b, seq=12/3072, ttl=64
612	23.539084000	10.0.0.1	10.0.0.7	ICMP	98	Echo (ping) request id=0x227b, seq=12/3072, ttl=64
634	24.540102000	10.0.0.1	10.0.0.7	ICMP	100	Echo (ping) request id=0x227b, seq=13/3328, ttl=64
638	24.540102000	10.0.0.1	10.0.0.7	ICMP	98	Echo (ping) request id=0x227b, seq=13/3328, ttl=64 (reply in 639)
639	24.788936000	10.0.0.7	10.0.0.1	ICMP	100	Echo (ping) reply id=0x227b, seq=13/3328, ttl=64 (request in 638)

Gambar 5.28 Uji *Convergence* Skenario 3

Gambar 5.28 waktu yang dibutuhkan sebuah jaringan untuk menemukan jalur yang baru ketika terjadi *fail path*. Nilai *convergence* diperoleh dari perhitungan waktu sebelum terjadi *fail path* dikurangi dengan waktu menemukan jalur baru. Nilai *convergence*, yaitu 3,01ms.

5.2 Analisis Pengujian Sistem

Didalam skenario terdapat 3 pengujian, yaitu pengujian topologi, *throughput*, *latency* dan *convergence*. Berdasarkan 3 pengujian yang telah dilakukan akan dihasilkan analisis kesimpulan dengan melakukan perbandingan pada skenarionya. Analisis kesimpulan yang dihasilkan adalah sebagai berikut:

1. Pengujian topologi untuk menghasilkan jalur terpendek

   a. Skenario 1

   Jalur yang menghubungkan antara switch tidak ada yang terjadi *fail path*. Pengujian h1 menuju h7 melewati h4 dan h3. Dan menghasilkan jalur terpendek dengan *cost* 10.

Perhitungan dijkstra dapat dibuktikan dengan melihat tabel 5.4 dibawah. Dapat dilihat langsung pada kolom h7, dimana kolom h7 terdapat 10h3 artinya h1 melewati h3 sebelum menuju h7 dengan menghasilkan jalur terpendek 10. Selanjutnya pada kolom h3, dimana terdapat 7h4 artinya h1 melewati h4 sebelum menuju h3 dan h7 dengan menghasilkan jalur terpendek 7. Pada kolom h4 terdapat 2h1 artinya h1 sebelum menuju h4, h3 dan h7 bernilai 2. Jadi, terbukti bahwa jalur terpendek h1 menuju h7 adalah h1, h4, h3 dan h7 dengan *cost* 10.

Tabel 5.4 Perhitungan *Dijkstra* Skenario 1

	H2	H3	H4	H5	H6	H7
H1	8h1	∞	2h1	∞	∞	∞
H1h4	8h1	7h4		12h4	9h4	∞
H1h4h3	8h1			12h4	9h4	10h3
H1h4h3h2				12h4	9h4	10h3
H1h4h3h2h6				12h4		10h3
H1h4h3h2h6h7				12h4		

b. Skenario 2

Terjadi *fail path* pada jalur antara h4 dan h3. Pengujian h1 menuju h7 melewati jalur yang berbeda dari skenario 1, yaitu melewati h2 dan h3. Dan menghasilkan jalur terpendek lain dengan *cost* 13.

Perhitungan dijkstra dapat dibuktikan dengan melihat tabel 5.5 dibawah. Dapat dilihat langsung pada kolom h7, dimana kolom h7 terdapat 13h3 artinya h1 melewati h3 sebelum menuju h7 dengan menghasilkan jalur terpendek 13. Selanjutnya pada kolom h3, dimana terdapat 10h2 artinya h1 melewati h2 sebelum menuju h3 dan h7 dengan menghasilkan jalur terpendek 10. Pada kolom h2 terdapat 8h1 artinya h1 sebelum menuju h2, h3 dan h7 bernilai 8. Jadi, terbukti bahwa jalur terpendek h1 menuju h7 yang dilewati berbeda dengan skenario 1, yaitu h1, h2, h3 dan h7 dengan *cost* 13.

Tanda x pada kolom h3 menjelaskan bahwa h3 terjadi *fail path* pada jalur antara h3 menuju h4.

Tabel 5.5 Perhitungan *Dijkstra* Skenario 2

	H2	H3	H4	H5	H6	H7
H1	8h1	∞	2h1	∞	∞	∞
H1h4	8h1	×		12h4	9h4	∞
H1h4h2		10h2		12h4	9h4	∞
H1h4h2h6		10h2		12h4		18h6
H1h4h2h6h3				12h4		13h3
H1h4h2h6h3h5						13h3

c. Skenario 3

Terjadi *fail path* pada jalur antara h4,h3 dan h2,h3. Pengujian h1 menuju h7 melewati h4 dan h6. Dan menghasilkan jalur terpendek lain dengan *cost* 18.

Perhitungan dijkstra dapat dibuktikan dengan melihat tabel 5.6 dibawah. Dapat dilihat langsung pada kolom h7, dimana kolom h7 terdapat 18h6 artinya h1 melewati h6 sebelum menuju h7 dengan menghasilkan jalur terpendek 18. Selanjutnya pada kolom h6, dimana terdapat 9h4 artinya h1 melewati h4 sebelum menuju h6 dan h7 dengan menghasilkan jalur terpendek 9. Pada kolom h4 terdapat 2h1 artinya h1 sebelum menuju h4, h6 dan h7 bernilai 2. Jadi, terbukti bahwa jalur terpendek h1 menuju h7 yang dilewati berbeda dengan skenario 1 dan skenario 2, yaitu h1, h4, h6 dan h7 dengan *cost* 18.

Tanda x pada kolom h3 menjelaskan bahwa h3 terjadi *fail path* pada jalur antara h3, h4 dan antara h3, h2.

Tabel 5.6 Perhitungan *Dijkstra* Skenario 3

	H2	H3	H4	H5	H6	H7
H1	8h1	∞	2h1	∞	∞	∞
H1h4	8h1	×		12h4	9h4	∞
H1h4h2		×		12h4	9h4	∞
H1h4h2h6				12h4		18h6
H1h4h2h6h5						18h6

Dari pengujian skenario 1, 2 dan 3 dapat dilihat di tabel 5.7 bahwa terbukti benar dan didapatkan hasil yang berbeda untuk pencarian jalur terpendek antara sebelum dan ketika terjadi *fail path*.

Tabel 5.7 Hasil Jalur Terpendek

Skenario 1	H1, h4, h3, h7	10
Skenario 2	H1, h2, h3, h7	13
Skenario 3	H1, h4, h6, h7	18

2. Pengujian *throughput* untuk menghitung paket yang sampai pada tujuan.

   a. Skenario 1

   Pengujian dilakukan dengan tidak terjadi *fail path* sehingga *throughput* yang dihasilkan 1,4 GBytes/sec. *Throughput* skenario 1 lebih besar dari pada skenario 2 dan 3 karena melewati jalur yang tidak terjadi *fail path* sehingga jumlah paket yang terkirim menuju destination lebih besar.

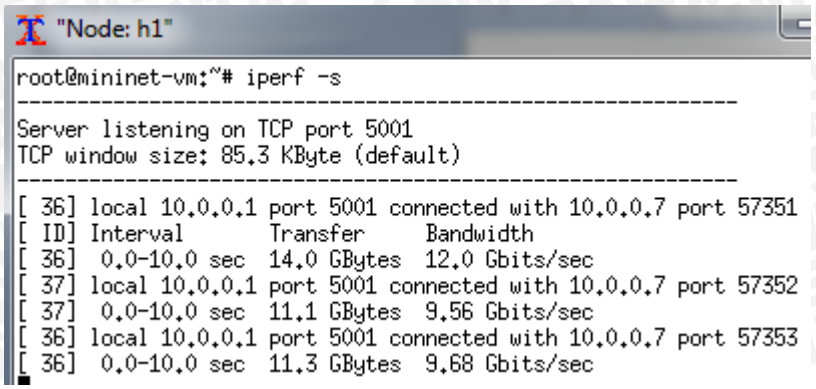
   b. Skenario 2

   Pengujian dilakukan dengan terjadi *fail path* pada salah satu jalur antara h1 dan h4 sehingga *throughput* yang dihasilkan lebih kecil dari skenario 1, yaitu 1,4 GBytes/sec. *Throughput* yang dihasilkan pada skenario 2 lebih kecil karena jalur yang dilewati sebelumnya terjadi *fail path* sehingga jumlah paket yang terkirim menuju *destination* lebih kecil.

   c. Skenario 3

   Pengujian yang dilakukan terjadi *fail path* sehingga *throughput* yang dihasilkan lebih kecil dari skenario 1, yaitu 1,4 GBytes/sec. *Throughput* pada skenario 3 lebih kecil karena melewati jalur yang terjadi *fail path* sehingga jumlah paket yang terkirim menuju destination lebih kecil.

Dari pengujian skenario yang telah dilakukan didapatkan hasil jumlah paket yang terkirim dari h1 menuju h7 dapat dilihat pada gambar 5.29 dan untuk hasil throughput dapat dilihat pada tabel 5.8. Pada skenario 1 nilai *throughput* yang sampai lebih banyak karena tidak terjadi *fail path*.



Gambar 5.29 *Iperf* Pada Server

Tabel 5.8 Perbandingan *Throughput*

Skenario Ke-	*Throughput*
Skenario 1	1,4 GBytes/sec
Skenario 2	1,11 GBytes/sec
Skenario 3	1,13 GBytes/sec

3. Pengujian *latency* untuk menghitung waktu yang diperoleh dari sebuah paket sampai ke tujuan.

   a. Skenario 1

   Waktu yang ditempuh sebuah paket dari *source* menuju *destination* dibutuhkan pencarian paling lama karena harus melakukan *routing* awal untuk membandingkan *switch* mana yang harus dilewati yang memiliki *cost* terkecil.

   b. Skenario 2

   Terjadi *packet loss* dan tidak membutuhkan waktu lama dalam pencarian jalur terpendek karena terjadi *fail path* sehingga untuk melakukan *routing* lebih cepat.

   c. Skenario 3

   Terjadi *packet loss* dan tidak membutuhkan waktu lama dalam pencarian jalur terpendek karena terjadi *fail path* sehingga untuk melakukan *routing* lebih cepat.

   Tabel 5.9 hasil dari pengujian *latency* yang telah dilakukan bahwa ketika terjadi *fail path* maka sebuah paket lebih cepat untuk sampai ke tujuan karena *routing* ulang dengan *backup path*.

Tabel 5.9 Perbandingan *Latency*

Skenario 1	0.162ms
Skenario 2	0.190ms
Skenario 3	0.150ms

4. Pengujian *convergence* untuk menghitung waktu yang dibutuhkan untuk menemukan jalur lain ketika terjadi *fail path*.

   a. Skenario 1

   Tidak ada hasil *convergence* karena tidak terjadi *fail path* ketika mencari jalur terpendek.

   b. Skenario 2

   Terjadi *fail path* antara h4 dan h3 sehingga membutuhkan waktu *convergence* untuk melakukan *routing* ulang.

   c. Skenario 3

   Terjadi *fail path* antara h4, h3 dan h2, h3 sehingga membutuhkan waktu *convergence* untuk melakukan *routing* ulang.

   Tabel 5.10 hasil dari pengujian *convergence* yang telah dilakukan bahwa untuk *convergence* menghasilkan waktu dengan melihat lamanya terjadi *fail path*.

Tabel 5.10 Perbandingan *Convergence*

Skenario 1	-
Skenario 2	2.009ms
Skenario 3	3.01ms

BAB 6 PENUTUP

6.1 Kesimpulan

Berdasarkan rumusan masalah yang diangkat dan sistem telah melalui tahap perancangan, implementasi dan telah dilakukan pengujian maka dapat ditarik kesimpulan bahwa sistem telah berhasil. Berikut adalah hasil penelitian yang didapat:

1. Untuk *routing* ulang dilakukan dengan menggunakan *dijkstra algorithm* pada arsitektur *software defined network* melalui *update path* berdasarkan *cost* terkecil.
2. *fail path* terjadi pada salah satu jalur yang menghubungkan tiap-tiap *switch*. Ketika terjadi *fail path* maka penerapan *dijkstra algorithm* akan menghasilkan jalur terpendek lain.
3. Untuk proses implementasi *dijkstra algorithm* dilakukan dengan memberi *cost* pada tiap jalur. Dengan memberi bobot *weight* antar switch untuk menghasilkan jalur terpendek
4. Ketika terjadi *fail path* maka pada pengujian topologi menghasilkan jalur terpendek lain yang dilewati. Skenario 1, yaitu h1, h4, h3, h7 dengan total *cost* 10. Skenario 2, yaitu h1, h2, h3, h7 dengan *cost* 13. Skenario 3, yaitu h1, h4, h6, h7 dengan *cost* 18.
5. Pada pengujian *throughput* ketika terjadi *fail path* mengalami penurunan dibandingkan saat belum terjadi *fail path*. Skenario 1, yaitu 12.0 Gbits/sec. Skenario 2 yaitu 9.56 Gbits/sec. Skenario 3 yaitu 9.68 Gbits/sec.
6. Pada pengujian *Latency* membutuhkan waktu lebih sedikit ketika terjadi *fail path* dibandingkan dengan sebelum terjadi *fail path* yang membutuhkan waktu lama. Skenario 1, yaitu 0.183ms. Skenario 2 yaitu 0.172ms. Skenario 3 yaitu 0.257ms..
7. Pada pengujian *convergence* untuk skenario 1 tidak ada hasil yang di peroleh karena tidak terjadi *fail path* sehingga tidak ada waktu yang dibutuhkan untuk menemukan jalur lain. Skenario 2, yaitu 2.009ms. Skenario 3 yaitu 3.01ms.

6.2 Saran

Saran dari penulis untuk pengembangan selanjutnya adalah

1. Diperlukan penelitian lebih lanjut terkait konsep *software defined network* dengan menggunakan *controller* yang berbeda.
2. Diperlukan penelitian lebih lanjut terkait konsep *software defined network* berdasarkan algoritma yang digunakan untuk mencari jalur terpendek lain ketika terjadi *fail path*.

DAFTAR PUSTAKA

Appelman Michiel, D. B. (2012). Performance Analysis of Openflow Hardware. *University of Amsterdam* .

Aris Saputra Ihsan, R. R. (2016). Uji Performansi Algoritma Floyd-Warshall pada Jaringan Software Defined Network (SDN).

Astuto Bruno A. Nunes, M. M.-N. (2014). A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks. *IEEE COMMUNICATIONS SURVEYS & TUTORIALS, VOL. 16, NO. 3, THIRD QUARTER* .

Fitria, A. T. (2013). Implementasi Algoritma Dijkstra Dalam Aplikasi Untuk Menentukan Lintasan. *Jurnal Sistem Informasi (JSI), VOL. 5, NO. 2, Oktober 2013,* , 611-621.

Gupta Nitin, M. K. (2016). International Journal of New Technology and Research (IJNTR). *Applying Dijkstra's Algorithm in Routing Process* , 122-124.

JOSHUA REICH, C. M. (2013). Modular SDN Programming with Pyretic. *OCTOBER 2013 VOL. 38 NO. 5* , 40-47. Juan, C. (2006). *Dijkstra's Algorithm As a Dynamic Programming strategy* .

Kemkominfo. (2014, Mei 8). *KEMENTERIAN KOMUNIKASI DAN INFORMATIKA REPUBLIK INDONESIA*. [Online]
Available at: https://kominfo.go.id: https://kominfo.go.id/index.php/content/detail/3980/Kemkominfo%3A+Pengguna+Internet+di+Indonesia+Capai+82+Juta/0/berita_satker
[Diakses 4 Juli 2016]

Lady Silk M, S. (2011). PENGARUH MODEL JARINGAN TERHADAP OPTIMASI ROUTING OPEN SHORTEST PATH FIRST (OSPF). *TEKNOLOGI, VOL. 1, NO. 2, JULI 2011* , 68-80.

Lubis, H. S. (2009). Perbandingan Algoritma Greedy dan Dijkstra Untuk Menentukan Lintasan Terpendek.

Nick McKeown, T. A. (2008). OpenFlow: Enabling Innovation in Campus Networks.

Niels L. M. van Adrichem, B. J. (2014). Fast Recovery in Software-Defined Networks. *2014 Third European Workshop on Software-Defined Networks* , 61-66.

NINGATI, R. (2014). APLIKASI PENCARIAN RUTE TERPENDEK DAERAH WISATA KOTA.

Nishtha, M. S. (2014). Software Defined Network - Architectures. *2014 International Conference on Parallel, Distributed and Grid Computing* , 451-456.

Project, T. F. (2016). *Frenetic*. [Online]
Available at: The Frenetic Project: http://frenetic-lang.org/pyretic/
[Diakses 25 Agustus 2016]

Rahmad Saleh Lubis, M. P. (2014). ANALISIS QUALITY OF SERVICE (QoS) JARINGAN. *VOL. 7 NO. 3/ Juni 2014* , 131-136.

William S. Bobanto, A. S. (2014). Analisis Kualitas Layanan Jaringan Internet (Studi Kasus PT. Kawanua Internetindo Manado). *e-journal Teknik Elektro dan Komputer (2014), ISSN: 2301-8402* , 80-87.

Yanto. (2013). ANALISIS QOS (QUALITY OF SERVICE) PADA JARINGAN INTERNET (STUDI KASUS: FAKULTAS TEKNIK UNIVERSITAS TANJUNGPURA).

Ying-Dar Linl, H.-Y. T.-R.-C.-C. (2016). *Fast Failover and Switchover for Link Failures and Congestion in Software Defined Network* .

LAMPIRAN

Skenario 1

No	Topologi	*Throughput*	*Latency*	*Convergence*
1	H1, h4, h3, h7 Cost: 10	1,4 GBytes/sec	0,156ms	-
2	H1, h4, h3, h7 Cost: 10	1,21 GBytes/sec	0,189ms	-
3	H1, h4, h3, h7 Cost: 10	1,42 GBytes/sec	0,245ms	-
4	H1, h4, h3, h7 Cost: 10	1,42 GBytes/sec	0,114ms	-
5	H1, h4, h3, h7 Cost: 10	1,49 GBytes/sec	0,147ms	-

Skenario 2

No	Topologi	*Throughput*	*Latency*	*Convergence*
1	H1, h2, h3, h7 Cost: 13	1,11 GBytes/sec	0,248ms	2.009ms
2	H1, h2, h3, h7 Cost: 13	1,17 GBytes/sec	0,156ms	2,01ms
3	H1, h2, h3, h7 Cost: 13	1,39 GBytes/sec	0,154ms	2,001ms
4	H1, h2, h3, h7 Cost: 13	1,38 GBytes/sec	0,154ms	2,32ms
5	H1, h2, h3, h7 Cost: 13	0,977 GBytes/sec	0,232ms	1,99ms

Skenario 3

No	Topologi	*Throughput*	*Latency*	*Convergence*
1	H1, h4, h6, h7 Cost: 18	1,13 GBytes/sec	0,187ms	3.01ms

2	H1, h4, h6, h7 Cost: 18	1,16 GBytes/sec	0,164ms	3,3ms
3	H1, h4, h6, h7 Cost: 18	1,18 GBytes/sec	0,184ms	2,33ms
4	H1, h4, h6, h7 Cost: 18	1,41 GBytes/sec	0,129ms	3,23ms
5	H1, h4, h6, h7 Cost: 18	1,17 GBytes/sec	0,132ms	2ms