Analisis *Failover Multi Controller* Berbasis *Floodlight Controller* Pada *Software Defined Network (SDN)*

Edwart Jhon Pranata[1], Rakhmadhany Primananda, S.T., M.Kom.[2], Widhi Yahya, S.Kom., M.Sc.[3]
Program Studi Informatika, Fakultas Ilmu Komputer, Universitas Brawijaya, Malang
Jl. Veteran No 8, Malang 65145, Indonesia
e-mail: edujhon08@gmail.com[1], rakhmadhany@ub.ac.id[2], widhi.yahya@ub.ac.id[3]

Abstrak - Pada perkembangan jaringan komputer saat ini terdapat sebuah teknologi dimana sistem pengontrol dari arus data dipisahkan dari perangkat kerasnya. *Software defined network* merupakan teknologi jaringan komputer yang memisahkan antara *control plane* dan data *plane* pada *switch. Software defined network* juga merupakan jaringan komputer yang sangat *flexibel* karena dikonfigurasikan dan dikendalikan melalui sebuah *software* terpusat. Cara komunikasi antara perangkat dan *controller* menggunakan sebuah protokol yang disebut dengan *Openflow*.

Sebagai konsep jaringan yang sangat komplek *software defined network* menawarkan *high availability* seperti *fault tolerance multi controller*. *Fault tolerance* merupakan sebuah metode yang memungkinkan suatu sistem tetap berjalan normal meskipun ada komponen yang rusak pada salah satu komponennya. Fault tolerance juga dipakai dalam transmisi data sehingga meskipun ada beberapa data yang gagal diterima pesan dapat diterima secara utuh. Jika *Fault tolerance* diterapkan pada *Software Defined Network*(SDN, maka *Fault Tolerance* adalah metode yang membagi tugas pada *controller* menjadi dua, yaitu *primary* dan *backup. Controller primary* dapat bertugas sebagai *Controller* utama dalam proses dan intruksi terhadap jaringan secara langsung. Sedangkan *controller backup* adalah duplikasi dari *Controller primary* namun memiliki status siaga atau *StandBy* dalam hal proses dan intruksi terhadap jaringan.

Melalui implementasi sistem failover multi controller. Sistem failover multi controller akan memberikan nilai rata-rata sinkronisasi switch yang semakin tinggi apabila menggunakan controller yang banyak. Berbeda dengan nilai rata-rata yang diberikan pada saat kondisi failover multi conroller. Apabila semakin banyak controller yang digunakan maka semakin menurun waktu yang dihasilkan untuk melakukan sistem failover controller.

Kata kunci : *Software Defined Network, High Availability, Fault Tolerance*.

1. PENDAHULUAN

Perkembangan teknologi saat ini berkembang sangat pesat khususnya pada pemanfaatan Internet. Dimana semua orang didunia terhubung dengan satu jaringan komputer yang luas, yang bisa saling berkomukasi satu dengan yang lainnya. Jaringan komputer sendiri saat ini telah menjadi kebutuhan yang sangat mendasar dalam pertukaran informasi. Sulit dibayangkan jika diera modern saat ini pertukaran informasi tanpa menggunakan jaringan komputer. Hal ini dapat kita lihat pada kebutuhan yang sangat banyak akan akses pada jaringan internet, sehingga mengakibatkan timbulnya perkembangan dari jaringan itu sendiri. Salah satu bukti dari perkembangan jaringan yang terjadi muncullah sebuah teknologi yang merupakan teknologi jaringan masa depan yang dikenal dengan *Software Defined Networking* (SDN).

Dalam perkembangan disuatu perusahaan, infrastruktur harus sesuai dengan kebutuhan, baik itu di perusahaan besar maupun kecil dan tentu saja sangat berbeda dalam hal persyaratan dan infrastruktur, khususnya infrastruktur jaringan. Ketersediaan data/layanan sangat dibutuhkan, terutama bagi perusahaan-perusahaan yang sangat membutuhkan data-data ataupun layanan, sangat dibutuhkan adanya server cadangan. Dalam jaringan konvensional untuk mendukung High availibility pada jaringan diperlukan sebuah topologi yang redundan. Sesuai dengan dasar teori dari teknologi *High-Availability*, yaitu ilmu untuk menciptakan redundansi dalam setiap sistem dan subsistem untuk memastikan bahwa layanan tetap tersedia. Sehingga dapat dianalogikan dalam implementasi, bila satu server gagal melayani service tertentu, maka tugas *server* tersebut otomatis akan dilempar ke *server* lainnya. Pada sisi *controller* terdapat mekanisme yang digunakan untuk menangani kegagalan *controller*, yaitu dengan mem*backup controller* yang telah terhubung pada *controller* utamanya. Namun, *backup* tersebut harus secara menyeluruh dan dapat menyebabkan konfigurasi yang sedikit rumit. Agar mengurangi konfigurasi yang rumit tersebut. (Pashkov V, 2014).

Pada penelitian sebelumnya yang dilakukan Mathis Obadia dan kawan-kawan yaitu *Failover Mechanisms for Distributed SDN Controller*, penelitian tersebut melakukan simulasi *failover* pada multi controller menggunakan algoritma *Pre Partitioning Failover*. Penelitian tersebut menggunakan *floodlight* yang berbasis java sebagai kontrolernya. Dalam algoritma *Pre Partitioning Failover*, setiap kontroler utamanya mengirimkan informasi daftar switch yang siap diambil alih ke kontroller cadangan apabila terjadi *failover*. Pengiriman informasi tersebut dilakukan

secara berkala. (Obadia M, 2014). Selain metode *Pre Partitioning Failover* yang dilakukan Mathis Obadia dan kawan kawan, ada juga metode lain yang dapat mencegah terjadinya *failover* pada *multi controller*. Adalah *Active Standby Strategies* yang dilakukan oleh Paskov, V dan kawan kawan. Dalam penelitian *Controller Failover for SDN Enterprise Networks* dimana *controller* utama memiliki switch yang terhubung juga dengan *controller* cadangannya. Secara garis besar metode *Active Standby Strategies* dan *Pre Partitioning Failover* memliki kemiripan pada sisi cara kerjanya, yang membedakan hanya cara pengujiannya saja. (Pashkov, 2014)

Berdasarkan penjelasan di atas, maka penulis akan menggunakan metode *Failover multi controller* dalam mencegah terjadinya *failover* pada *multi controller*. Pada metode ini tugas kontroler utamanya adalah memegang kendali terhadap switch yang terhubung dan memberi perintah bahwa switch tersebut adalah milik dari kontroler utamanya atau bisa disebut *Role MASTER*. Namun switch yang terhubung pada kontroler utama tersebut akan berstatus *SLAVE* pada kontroler cadangannya.

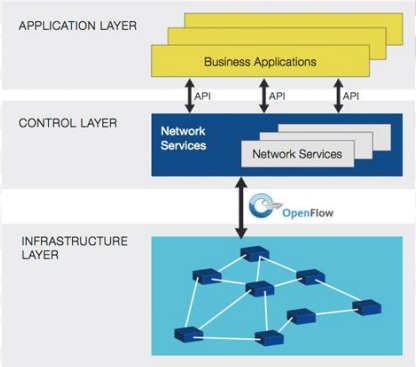
1. DASAR TEORI

1.1 Software Defined Network

Software-defined networking (SDN) adalah sebuah konsep pendekatan jaringan komputer dimana sistem pengkontrol dari arus data dipisahkan dari perangkat kerasnya. Umumnya, sistem pembuat keputusan kemana arus data dikirimkan dibuat menyatu dengan perangkat kerasnya. Sebuah konfigurasi SDN dapat menciptakan jaringan dimana perankat keras pengontrol lalu lintas data secara fisik dipisahkan dari perangkat keras data *forwarding plane*. Konsep ini dikembangkan di UC Berkeley and Stanford University sekitar tahun 2008. Penemu dan penyedia sistem ini mengklaim dapat menyederhanakan jaringan komputer.

Arsitektur SDN dapat dilihat sebagai 3 lapis/bidang:

- **Infrastruktur** (*data-plane / infrastructure layer*) : terdiri dari elemen jaringan yg dapat mengatur SDN Datapath sesuai dengan instruksi yg diberikan melalui *Control-Data-Plane Interface* (CDPI)
- **Kontrol** (*control plane / layer*) : entitas kontrol (SDN *Controller*) mentranslasikan kebutuhan aplikasi dengan infrastruktur dengan memberikan instruksi yg sesuai untuk SDN Datapath serta memberikan informasi yg relevan dan dibutuhkan oleh *SDN Application*
- **Aplikasi** (*application plane / layer*) : berada pada lapis teratas, berkomunikasi dengan sistem via *NorthBound Interface* (NBI)



Gambar 1.1 Arsitektur Software Defined Network

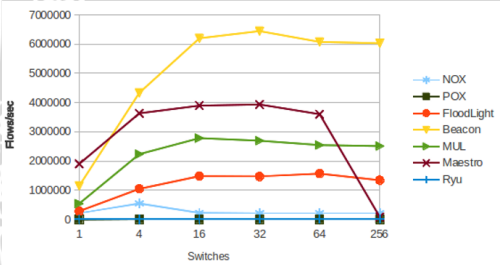
Pada gambar diatas dapat dilihat bahwa control layer memegang peran penting mengendalikan sistem jaringan, setiap perangkat dari berbagai macam vendor dapat beroperasi sesuai perintah dari controllernya. Cara komunikasi antara perangkat dan controllernya sendiri menggunakan suatu protokol yang disebut dengan *OpenFlow*. *OpenFlow* sendiri bisa diibaratkan seperti sebuah CPU pada komputer.

1.2 Floodlight Controller

Software Defined Networking memiliki beberapa controller yang dapat membantu konfigurasi komunikasi antara *application layer* dan *infrastructure layer*. Beberapa *controller* pada SDN yaitu FloodLight, Open Daylight, Trema, POX dan Ryu. *Floodlight Controller* adalah *controller* SDN yang menggunakan bahasa pemrograman java, bersifat *open source*, dan berlisensi *apache*. *Floodlight* merupakan perkembangan dari Beacon yang sudah ada lebih dulu serta didukung oleh komunitas developer dan perusahaan *Big Switch Network* dalam melakukan pengembangannya.

Berikut perbandingan perfoma *floodlight controller* dengan *controller* lainnya: (Alexander Shalimov, 2013)

a. Throughput



Gambar 1.2 Perbandingan Performa Controllers

Pada gambar 1.2 merupakan uji perfoma throughput menggunakan 8 threads dan 32 switch. Terlihat bahwa floodlight lebih unggul dibandingkan POX, NOX, dan RYU. Hal ini dikarenakan *floodlight* menggunakan *multi-threading*.

b. Latency

NOX	91,531
POX	323,443
Floodlight	75,484
Beacon	57,205
MuL	50,082
Maestro	129,321
Ryu	105,58

Gambar 1.3 Waktu Respon minimum (10^{-6} detik/flow)

Pada gambar 1.3 merupakan waktu respon mininimum dalam melakukan flow. Dalam hal *Latency*, floodlight juga lebih bagus dibandingkan POX, NOX, dan RYU.

1.3 *High availability*

High-availability adalah ilmu untuk menciptakan redundansi dalam setiap sistem dan subsistem untuk memastikan bahwa layanan tetap *up* dan tersedia. *High availability* pada dasarnya menempatkan satu atau lebih

server cadangan dalam modus siaga, yang bisa online dalam beberapa saat hanya setelah mereka menemukan kegagalan pada sistem utama.(Charles Bookman, 2002).

High availability cluster, yang juga sering disebut sebagai *Failover Cluster* pada umumnya di implementasikan untuk tujuan meningkatkan ketersediaan layanan yang disediakan oleh kluster tersebut. Elemen kluster akan bekerja dengan memiliki node-node redundan, yang kemudian digunakan untuk menyediakan layanan saat salah satu elemen kluster mengalami kegagalan. Ukuran yang paling umum dari kategori ini adalah dua *node*, yang merupakan syarat minimum untuk melakukan redundansi. Implementasi kluster jenis ini akan mencoba untuk menggunakan redundansi komponen kluster untuk menghilangkan kegagalan di satu titik (*Single Point of Failure*).

Salah satu metode yang mendukung kerja High Availability adalah Fault Tolerance. Secara umum pengertian *fault tolerance* adalah metode yang memungkinkan suatu sistem tetap berjalan normal meskipun ada komponen yang rusak pada salah satu komponennya. *Fault tolerance* juga dipakai dalam transmisi data sehingga meskipun ada beberapa data yang gagal diterima pesan dapat diterima secara utuh. (Proweb Indonesia, 2016)

Jika *Fault tolerance* diterapkan pada jaringan *Software Defined Network*(SDN), maka *Fault Tolerance* adalah metode yang membagi tugas pada kontroler menjadi dua, yaitu *primary* dan *backup*. Kontroler primary dapat bertugas sebagai kontroler utama dalam proses dan intruksi terhadap jaringan secara langsung. Sedangkan kontroler backup adalah duplikasi dari kontroler *primary* namun memiliki status siaga atau *StandBy* dalam hal proses dan intruksi terhadap jaringan. Metode memungkinkan bahwa kontroler primary memliki backup, sehingga jika terjadi kegagalan atau *failure* pada kontroler primary, maka kontroler backup yang menjadi kontroler utama berikutnya.

1.4 Mininet

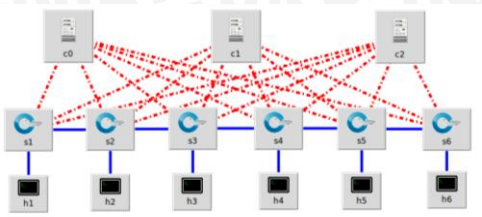
Mininet adalah *software open source* yang digunakan untuk melakukan simulasi jaringan, yang sengaja diciptakan untuk memudahkan dalam melakukan research dan penjelasan tentang sistem *Software Defined Network*(SDN). Mininet menciptakan simulasi jaringan yang memanfaatkan *software real* dari komponen jaringan sehingga dapat digunakan secara interaktif untuk melakukan uji coba *software* jaringan. (B Lantz, 2010)

Mininet sendiri memanfaatkan *Linux Network namespace* untuk menciptakan virtual node pada simulasi jaringan, oleh karenanya proses simulasi yang dijalankan pada *mininet* akan lebih ringan dan cepat. Untuk dapat menggunakan mininet sebenarnya terdapat dua cara yaitu dengan menggunakan (*Virtual Machine*) VM atau dengan menginstall *mininet* secara langsung pada PC atau Laptop.

2. PERANCANGAN DAN IMPLEMENTASI SISTEM

2.1 Perancangan sistem

Pada penelitian sebelumnya yang dilakukan obadia dan kawan kawan, topologi yang digunakan adalah berbentuk linier dengan beberapa variasi jumlah *controller*, *switch*, dan *host*. Dapat dilihat pada gambar 2.2 dimana terdapat tiga buah *controller*, enam buah *openflow switch*, dan enam buah *host*.



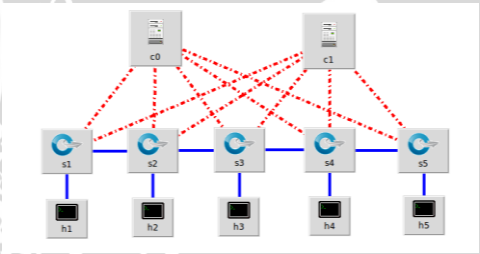
Gambar 2.2 Perancangan topologi 3 *controller* 6 *switch*

Penggunaan variasi jumlah *controller* dimaksudkan untuk beberapa hal, yaitu:

1. Apakah ada dampak yang berbeda apabila menggunakan jumlah controller yang lebih banyak?
2. Apakah ada dampak yang berbeda apabila menggunakan jumlah switch yang lebih banyak?
3. Apakah ada dampak yang berbeda apabila menggunakan jumlah host yang lebih banyak?
4. Apakah ada korelasi antara controller dan switch dalam sistem?

Pada penelitian ini, akan ada beberapa macam variasi topologi yang akan diuji, yaitu:

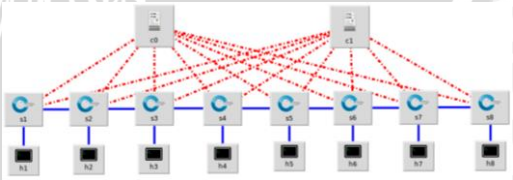
a. Topology 2 *controller* 5 *switch*



Gambar 2.3 Perancangan topologi 2 *controller* 5 *switch*

Pada gambar 2.3 *controller* 1 dapat berperan sebagai *master controller* untuk *switch* satu, dua, tiga, empat, dan lima. *Controller* 2 dapat berperan sebagai *backup controller* untuk *switch* satu, dua, tiga, empat, dan lima. Hal tersebut dapat divariasikan sesuai kebutuhan penelitian.

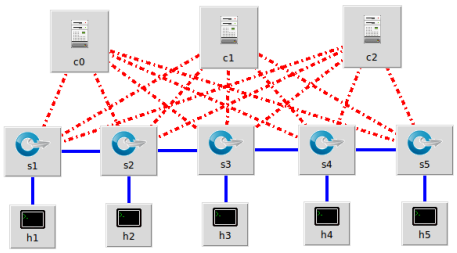
b. Topologi 2 *controller* 8 *switch*



Gambar 2.4 Perancangan topologi 2 *controller* 8 *switch*

Pada gambar 2.4 *controller* 1 dapat berperan sebagai *master controller* untuk switch satu, dua, tiga, empat, lima, enam, tujuh, dan delapan. *Controller* 2 dapat berperan sebagai *backup controller* untuk *switch* satu, dua, tiga, empat, lima, enam, tujuh, dan delapan. Hal tersebut dapat divariasikan sesuai kebutuhan penelitian.

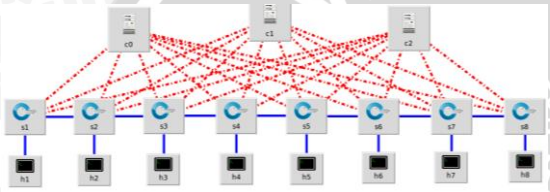
c. Topologi 3 *controller* 5 *switch*



Gambar 2.5 Perancangan topologi 3 *controller* 5 *switch*

Pada gambar 2.5 *controller* 1 dapat berperan sebagai *master controller* untuk *switch* satu, dua, tiga, empat, dan lima. *Controller* 2 dapat berperan sebagai *backup controller* untuk *switch* satu, dua, tiga, empat, dan lima. Controller 3 dapat berperan sebagai *backup controller* untuk *switch* satu, dua, tiga, empat, dan lima. Hal tersebut dapat divariasikan sesuai kebutuhan penelitian.

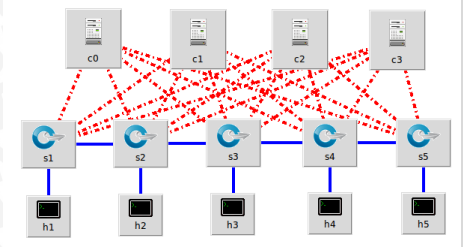
d. Topologi 3 *controller* 8 *switch*



Gambar 2.6 Perancangan topologi 3 *controller* dan 8 *switch*

Pada gambar 2.6 *controller* 1 dapat berperan sebagai *master controller* untuk switch satu, dua, tiga, empat, lima, enam, tujuh, dan delapan. *Controller* 2 dapat berperan sebagai *backup controller* untuk *switch* satu, dua, tiga, empat, lima, enam, tujuh, dan delapan. *Controller* 3 dapat berperan sebagai *backup controller* untuk *switch* satu, dua, tiga, empat, lima, enam, tujuh, dan delapan. Hal tersebut dapat divariasikan sesuai kebutuhan penelitian.

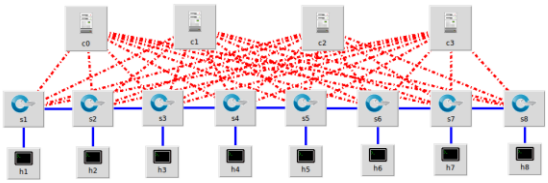
e. Topologi 4 *controller* 5 *switch*



Gambar 2.7 Perancangan topologi 4 *controller* 5 *switch*

Pada gambar 2.7 *controller* 1 dapat berperan sebagai *master controller* untuk *switch* satu, dua, tiga, empat, dan lima. *Controller* 2, 3, dan 4 dapat berperan sebagai *backup controller* untuk *switch* satu, dua, tiga, empat, dan lima. Hal tersebut dapat divariasikan sesuai kebutuhan penelitian.

f. Topologi 4 *controller* 8 *switch*

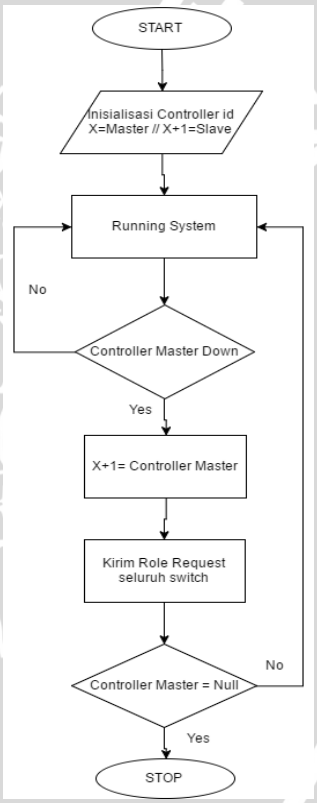


Gambar 2.8 Perancangan topologi 4 *controller* 8 *switch*

Pada gambar 2.8 *controller* 1 dapat berperan sebagai *master controller* untuk *switch* satu, dua, tiga, empat, lima, enam, tujuh, dan delapan. *Controller* 2, 3, dan 4 dapat berperan sebagai *backup controller* untuk *switch* satu, dua, tiga, empat, lima, enam, tujuh, dan delapan. Hal tersebut dapat divariasikan sesuai kebutuhan penelitian.

2.1.1 Flowchart Program

Pada perancangan sistem ini juga menggunakan mekanisme pemilihan *controller* dalam sistem failover controller. Berikut algoritma yang menjelaskan tentang *failover* yang akan diterapkan:



Gambar 2.9 Alur Sistem *Failover Controller*

Berdasarkan gambar diatas dapat diketahui proses cara kerja *failover*. Nanti akan disediakan beberapa *controller* dimana setiap *controller* memiliki beberapa *switch* dan *host*. Setiap *controller* diberi tugas sebagai *Master* dan *Slave*. Tiap-tiap *switch* terhubung dengan *switch* dan terhubung dengan *controller Master* dan *Slave*. Untuk memastikan apakah *controller* berjalan dengan baik, *controller* akan dijalankan dan dilihat *log* pada terminal ubuntu. Untuk membuat *failover controller* akan dilakukan dengan cara menghentikan proses *controller Master*. Selanjutnya untuk mengetahui apakah *switch* yang terhubung dengan *controller* utama tadi tetap berjalan akan dilakukan pengecekan pada *log* terminal *controller backup*. Dan untuk memastikan apakah proses

perubahan status *controller backup* menjadi *master* sudah berjalan, akan dilakukan pengecekan pada *log controller backup*. Sistem *failover* akan berhenti apabila *controller slave* sudah tidak ada lagi, maka *controller* tidak bisa melakukan sistem *failover* kembali. Begitu pula sebaliknya apabila *controller slave* masih tersedia, maka *failover* masih bisa dilakukan. Namun *controller failover* memiliki batas maksimal *controller failover* apabila terjadi *failover*.

2.1.2 Perancangan Topologi

Pada bagian ini akan menjelaskan cara implementasi terhadap variasi topologi yang akan digunakan. Berikut adalah variasi topologi, ip, dan port yang digunakan untuk melakukan implementasi fault tolerance pada multi controller.

Topologi	Controller 1			Controller 2		
	IP	Port Sinkronisasi	Openflow Port	IP	Port Sinkronisasi	Openflow Port
2 controller	192.168.56.101	6642	6653	192.168.56.102	6643	6653
3 controller	192.168.56.101	6642	6653	192.168.56.102	6643	6653
4 controller	192.168.56.101	6642	6653	192.168.56.102	6643	6653

Topologi	Controller 3			Controller 4		
	IP	Port Sinkronisasi	Openflow Port	IP	Port Sinkronisasi	Openflow Port
2 controller	-	-	-	-	-	-
3 controller	192.168.56.103	6644	6653	-	-	-
4 controller	192.168.56.103	6644	6653	192.168.56.104	6645	6653

Gambar 2.10 Rancangan Topologi

Seperti yang sudah dijelaskan pada bab empat tentang teknologi apa saja yang akan penulis pakai dalam implementasi *failover* pada *multi controller*. Dalam pembuatan topologi, penulis memanfaatkan *Appication Programming Interface (API)* dari aplikasi simulasi mininet yang berbahasa pemrograman python. Berikut adalah beberapa tampilan penggalan source code dari topologi yang akan dibuat. Beberapa fungsi dari source code yang dipakai untuk membuat topologi adalah sebagai berikut:

1. addController

```
info( '*** Adding controller\n' )
    c1=net.addController(name='c1',
                         controller=RemoteController,
                         ip='192.168.56.102',
                         protocol='tcp',
                         port=6653)
```

Gambar 2.11 Fungsi addController

Berdasarkan Kode diatas dapat dijelaskan sebagai berikut:

Fungsi ini berfungsi sebagai penambah controller pada topologi, dan juga menjelaskan ip, port, dan protocol apa saja yang akan dipakai dalam pembuatan topologi serta jenis controller yang dipakai. Dalam pembuatan topologi ini penulis menggunakan jenis controller "*RemoteController*". *RemoteController* berfungsi untuk menggunakan controller selain OpenFlow controller yang merupakan controller bawaan mininet.

2. addSwitch

```
info( '*** Add switches\n')
    s1 = net.addSwitch('s1', protocols="OpenFlow13")
    s4 = net.addSwitch('s4', protocols="OpenFlow13")
    s2 = net.addSwitch('s2', protocols="OpenFlow13")
    s3 = net.addSwitch('s3', protocols="OpenFlow13")
```

Gambar 2.12 Fungsi addSwitch

Berdasarkan kode diatas dapat dijelaskan sebagai berikut:

fungsi ini berfungsi sebagai penambah switch yang dipakai dalam pembuatan topologi. Fungsi ini menjelaskan jenis switch dan jenis protocol apa yang akan dipakai. Pada penelitian ini jenis protocol yang digunakan adalah Openflow versi 1.3. Penggunaan Openflow versi 1.3 dikarenakan protocol tersebut sudah mendukung dalam implementasi fault tolerance pada multi controller.

3. addHost dan addLink

```
info( '*** Add hosts\n')
    h2 = net.addHost('h2', cls=Host, ip='10.0.0.2', defaultRoute=None)
    h3 = net.addHost('h3', cls=Host, ip='10.0.0.3', defaultRoute=None)
    h4 = net.addHost('h4', cls=Host, ip='10.0.0.4', defaultRoute=None)
    h1 = net.addHost('h1', cls=Host, ip='10.0.0.1', defaultRoute=None)

    info( '*** Add links\n')
    net.addLink(s1, h1)
    net.addLink(s2, h2)
    net.addLink(s3, h3)
    net.addLink(s4, h4)
    net.addLink(s1, s2)
    net.addLink(s2, s3)
    net.addLink(s3, s4)
```

Gambar 2.13 Fungsi addHost dan addLink

Berdasarkan kode diatas dapat dijelaskan sebagai berikut:

Fungsi addHost berfungsi sebagai penambah host yang akan dipakai dalam pembuatan topologi. Fungsi addLink berfungsi sebagai penghubung antar switch yang dipakai dalam topologi.

4. starting controller dan starting switch

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

    info( '*** Starting switches\n')
    net.get('s1').start([c0,c1,c2])
    net.get('s4').start([c1,c0,c2])
    net.get('s2').start([c0,c1,c2])
    net.get('s3').start([c1,c0,c2])
```

Gambar 2.14 Starting Controller dan Starting Switch

Berdasarkan kode diatas dapat dijelaskan sebagai berikut:

Fungsi starting Controller berfungsi sebagai penanda bahwa controller dapat dimulai atau dijalankan. Fungsi starting switch berfungsi sebagai koneksi antara controller dengan switch.

2.2 Implementasi Modul Floodlight Controller

Dalam mengimplementasikan failover pada multi controller nanti akan terdapat dua cara pada floodlight controller, yaitu cara pertama terletak pada modul forwarding,

simpleFT, dan topology.TopologyManager, dan cara kedua terletak pada modul floodlightdefault.properties dan floodlightNodeBackup.properties. Untuk itu floodlight controller akan dikonfigurasikan sesuai dengan metode yang akan digunakan dalam penelitian ini. Cara melakukan konfigurasi pada floodlight controller terletak pada cara kedua yaitu membuat file .properties sesuai yang diinginkan. Pada file .properties tersebut kita bisa menambahkan modul yang ingin dipakai serta port yang digunakan pada controller.

1. Konfigurasi Port

```
org.sdnplatform.sync.internal.SyncManager.port=6642
net.floodlightcontroller.core.internal.FloodlightProvider.openFlowPort=6653
net.floodlightcontroller.restserver.RestApiServer.httpPort=8080
```

Gambar 2.15 Konfigurasi Port

Port yang akan di konfigurasi adalah port untuk melakukan sinkronasi antar controller dengan mendefinisikan modul SyncManager, lalu port untuk melakukan komunikasi dengan switch melalui openflow protocol dengan mendefinisikan modul FloodlightProvider.openFlowPort, serta mendefinisikan port agar controller dapat diakses melalui Rest Api dengan mendefinisikan modul RestApiServer.httpPort. Konfigurasi port dapat dilihat pada cuplikan kode berikut.

2. Konfigurasi sinkronasi dan failover

Modul yang berperan dalam proses sinkronisasi serta proses failover adalah simpleFT, SyncManager, dan OFSwitchManager. Berikut adalah langkah untuk mengkonfigurasi ketiga modul tersebut:

a. Melakukan pembuatan sebuah kunci terenkripsi yang digunakan untuk autentikasi antar controller yang akan melakukan sinkronisasi. Setelah kunci tersebut dibuat maka lokasi penyimpanan kunci tersebut didefinikan pada mudul SyncManager.keyStorePath.

b. Melakukan konfigurasi ip dan port seluruh controller yang terlibat dalam proses sinkronisasi dengan mendefinisikannya pada modul SyncManager.Nodes.

c. Melakukan inisialisasi peran controller terhadap switch, dengan mendefinisikannya pada modul OFSwitchManager .swtichesInitialState.

Berikut adalah salah satu konfigurasi yang dipakau untuk melakukan sinkronisasi antar controller.

```
org.sdnplatform.sync.internal.SyncManager.keyStorePath=/etc/floodlight/key2.jc
org.sdnplatform.sync.internal.SyncManager.dbPath=/var/lib/floodlight/
org.sdnplatform.sync.internal.SyncManager.keyStorePassword=PassWord
org.sdnplatform.sync.internal.SyncManager.thisNodeId=1
org.sdnplatform.sync.internal.SyncManager.nodes=[\
{"nodeId": 1, "domainId": 1, "hostname": "192.168.1.100", "port": 6642},\
{"nodeId": 2, "domainId": 1, "hostname": "192.168.1.100", "port": 6643}\
]
net.floodlightcontroller.core.internal.OFSwitchManager.switchesInitialState=
{"00:00:00:00:00:00:00:01":"ROLE_MASTER",
"00:00:00:00:00:00:00:02":"ROLE_MASTER",
"00:00:00:00:00:00:00:03":"ROLE_MASTER",
"00:00:00:00:00:00:00:04":"ROLE_MASTER",
"00:00:00:00:00:00:00:05":"ROLE_MASTER"}
```

Gambar 2.16 Konfigurasi Sinkronisasi dan failover

3. Konfigurasi modul forwarding

Modul forwarding adalah modul yang berperan dalam proses forwarding paket yaitu forwarding.Forwarding. Konfigurasi yang dilakukan adalah mendefinisikan field yang akan dicocokkan oleh setiap switch terhadap flow yang akan datang. Pada penelitian ini field yang akan dicocokkan adalah MAC address, port Transport, dan Ip address. Salah satu contoh konfigurasi modul forwarding yang dipakai dapat dilihat pada kode berikut.

```
net.floodlightcontroller.forwarding.Forwarding.match=vlan, mac, ip, transport
```

Gambar 2.17 Konfigurasi Forwarding

2.3 Mekanisme inisialisasi Controller dan koneksi Controller ke Switch

Simulasi ini dilakukan dengan menghidupkan controller yang sudah dikonfigurasi sesuai metode pada bab tiga dengan menggunakan perintah berikut :

```
$- sudo java –jar target/floodlight.jar –cf
src/main/resource/floodlightdefault.properties
```

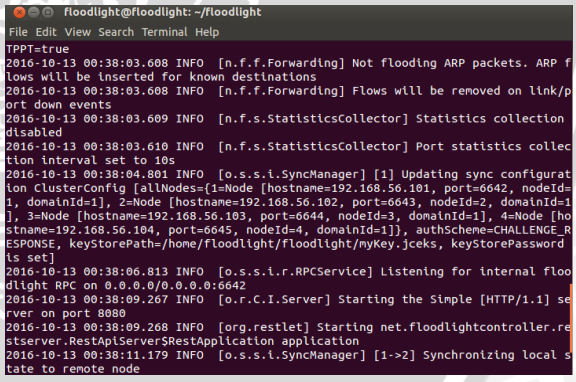
-cf digunakan untuk menjalankan konfigurasi yang akan digunakan oleh controller. Setelah floodlight.controller berjalan pada sistem, dijalankan topologi pada simulasi mininet dengan melakukan perintah berikut:

```
$- sudo pyhton ./nama_topology.py
```

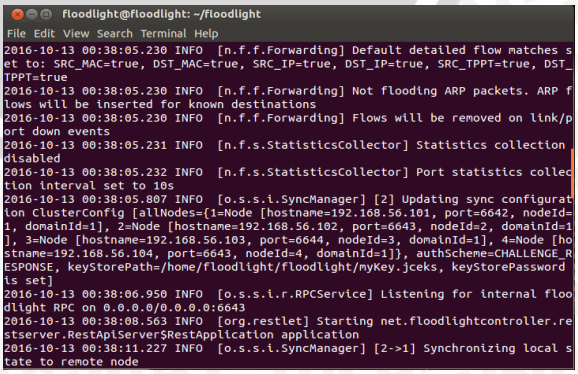
Selain menggunakan perintah diatas, untuk menjalankan topologi juga dapat di lakukan secara langsung pada mininet. Setelah controller dan topologi dijalankan, maka akan dilakukan analisis perhitungan terhadap waktu sinkronasi dengan menggunakan keluaran log pada controller floodlight.

2.4 Mekanisme controller failover

Simulasi ini dimulai ketika jaringan sudah berjalan dengan baik dan benar, berikut adalah contoh mekanisme simulasi controller failover menggunakan 2 controller.



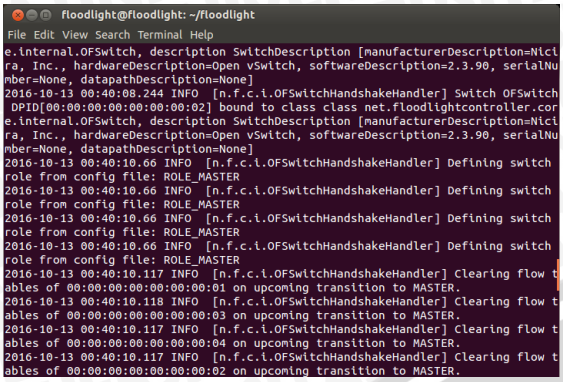
Gambar 2.18 Controller 1 Aktif



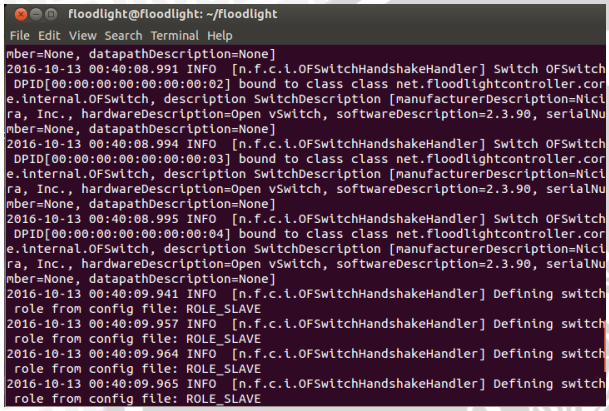
Gambar 2.19 Controller 2 aktif

Terlihat pada gambar kedua controller telah aktif. Kemudian akan dicek apakah *Failover multi controller* telah

terimplementasi dengan melihat peran controller terhadap seluruh switch menggunakan log event pada controller floodlight.



Gambar 2.20 Controller 1 role before failover



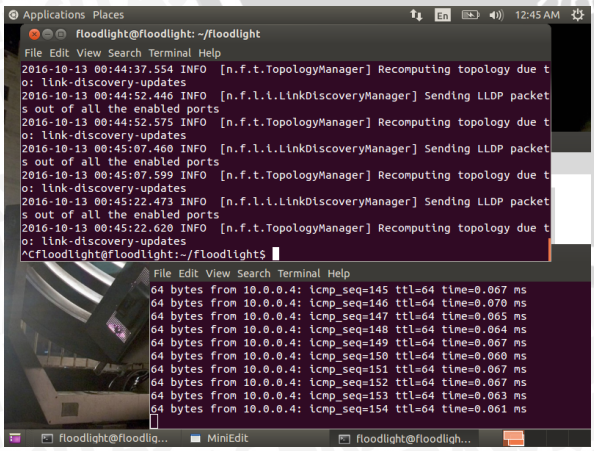
Gambar 2.21 Controller 2 role before failover

Terlihat dari gambar bahwa controller satu bersifat *Master* kepada switch 1,2,3, dan 4, Sedangkan controller dua bersifat *Slave* kepada switch 1,2,3, dan 4 .

Kemudian simulasi dilanjutkan dengan mematikan salah satu controller yang berjalan yaitu controller 2 dengan melakukan perintah :

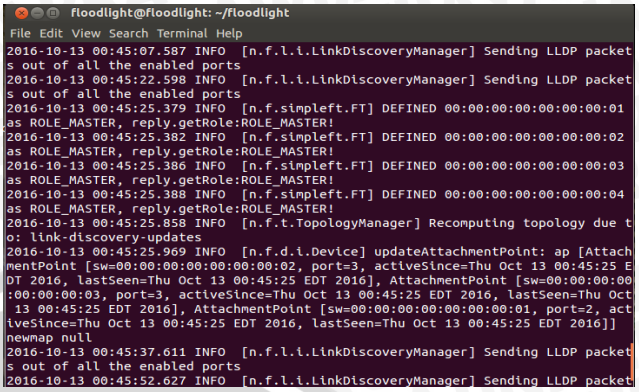
```
$- sudo kill -9 [PID Controller]
```

Cara lain untuk mematikan controller dapat dilakukan dengan menekan tombol *CTRL + C* pada keyboard.



Gambar 2.22 Controller 1 failover

Terlihat pada gambar bahwa controller 1 sudah dimatikan. Untuk memeriksa apakah *Failover multi controller* telah terimplementasi dapat dilihat pada log event controller floodlight.



Gambar 2.23 Controller 2 switch role after failover

Terlihat pada gambar bahwa controller 2 berperan sebagai *Master* terhadap switch 1,2,3 dan 4. Dari simulasi tersebut diatas dapat disimpulkan bahwa *Failover multi controller* telah terimplementasi dengan baik dan benar.

3. PENGUJIAN DAN ANALISA

3.1 Proses Pengambilan Data

1. Waktu Sinkronisasi

Pada penilitian ini akan dihitung waktu sinkronisasi yang dibutuhkan agar jaringan siap menangani controller failover. Perhitungan waktu akan digunakan dengan mengeluarkan waktu pada setiap event controller dalam bentuk log yang berperan dalam aktivitas ini.

Event yang berperan dalam proses sinkronisasi adalah:

(a) New Switch Connection

Event ini memberitahukan bahwa controller mulai melakukan koneksi dengan switch

```
2016-11-03 01:47:43.347 INFO [n.f.c.i.OFChannelHandler] New switch connection from
/192.168.56.101:56956
2016-11-03 01:47:43.380 INFO [n.f.c.i.OFChannelHandler] [[? from 192.168.56.101:56956]]
Disconnected connection
2016-11-03 01:47:45.515 INFO [n.f.c.i.OFChannelHandler] New switch connection from
/192.168.56.101:56958
```

Gambar 3.1 Log Koneksi Switch

(b) Switch DPID connection

Event ini memberitahukan bahwa controller dengan DPID tertentu sudah terkoneksi dengan switch.

```
2016-11-03 01:47:46.320 INFO [n.f.c.i.OFSwitchHandshakeHandler] Switch OFSwitch
DPID[00:00:00:00:00:00:00:01] bound to class class
net.floodlightcontroller.core.internal.OFSwitch, description SwitchDescription
[manufacturerDescription=Nicira, Inc., hardwareDescription=Open vSwitch,
softwareDescription=2.3.90, serialNumber=None, datapathDescription=None]
```

Gambar 3.2 Log Aktivasi Switch

(c) Switch Activated: Defining ROLE

Event ini memberitahukan switch mana saja yang sudah terkoneksi dengan controller dan saling melakukan sinkronisasi. Serta memberi role atau status kepemilikan switch antara controller aktif dan standby.

```
2016-11-03 01:47:47.141 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_MASTER
2016-11-03 01:47:47.147 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_MASTER
2016-11-03 01:47:47.147 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_MASTER
2016-11-03 01:47:47.148 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_MASTER
2016-11-03 01:47:47.404 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_MASTER
```

Gambar 3.3 Log Sinkronisasi Switch Controller Aktif

```
2016-11-03 01:47:47.261 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_SLAVE
2016-11-03 01:47:47.271 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_SLAVE
2016-11-03 01:47:47.274 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_SLAVE
2016-11-03 01:47:47.274 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_SLAVE
2016-11-03 01:47:47.345 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_SLAVE
```

Gambar 3.4 Log Sinkronisasi Switch Controller Stanby

(d) Update Attachment Point

Event ini memberitahukan bahwa terjadi perubahan pada jalur pengiriman paket akibat terjadinya controller failure.

```
2016-11-03 01:47:47.639 INFO  [n.f.d.i.Device] updateAttachmentPoint: ap [AttachmentPoint
[sw=00:00:00:00:00:00:00:01, port=2, activeSince=Thu Nov 03 01:47:47 EDT 2016,
lastSeen=Thu Nov 03 01:47:47 EDT 2016]]  newmap null
```

Gambar 3.5 Log Update Attachment Points

Waktu sinkronisasi adalah selisih waktu antara event New Switch Connection dengan event Switch DPID connection ataupun Switch Defining ROLE (tergantung daripada event yang terakhir terjadi).

2. Waktu Failover

Pada penelitian ini akan dihitung waktu failver yaitu waktu yang dibutuhkan oleh jaringan saat setiap controller yang aktif mengambil alih switch yang tidak memiliki controller akibat controller failure atau down. Perhitungan waktu akan digunakan dengan menggunakan waktu pada setiap event controller dalam bentuk log yang berperan dalam aktivitas ini. Event yang berperan dalam proses sinkronisasi adalah:

(a) Switch Managed

Event ini adalah penanda informasi switch sebelum terjadi controller failure pada salah satu controller pada jaringan. Database sinkronisasi mengirimkan informasi mengenai switch yang dimiliki oleh controller tersebut.

```
2016-11-03 01:47:47.261 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_SLAVE
2016-11-03 01:47:47.271 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_SLAVE
2016-11-03 01:47:47.274 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_SLAVE
2016-11-03 01:47:47.274 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_SLAVE
2016-11-03 01:47:47.345 INFO  [n.f.c.i.OFSwitchHandshakeHandler] Defining switch role
from config file: ROLE_SLAVE
```

Gambar 3.6 Log Sinkronisasi Switch Controller Before Failover

(b) Switch Managed

Event ini adalah penanda informasi switch sesudah terjadi controller failure pada salah satu controller pada jaringan. Database sinkronisasi mengirimkan informasi mengenai switch yang dimiliki oleh controller tersebut.

```
2016-11-03 01:48:36.571 INFO  [n.f.simpleft.FT] DEFINED 00:00:00:00:00:00:00:01 as
ROLE_MASTER, reply.getRole:ROLE_MASTER!
2016-11-03 01:48:36.586 INFO  [n.f.simpleft.FT] DEFINED 00:00:00:00:00:00:00:02 as
ROLE_MASTER, reply.getRole:ROLE_MASTER!
2016-11-03 01:48:36.595 INFO  [n.f.simpleft.FT] DEFINED 00:00:00:00:00:00:00:03 as
ROLE_MASTER, reply.getRole:ROLE_MASTER!
2016-11-03 01:48:36.597 INFO  [n.f.simpleft.FT] DEFINED 00:00:00:00:00:00:00:04 as
ROLE_MASTER, reply.getRole:ROLE_MASTER!
2016-11-03 01:48:36.602 INFO  [n.f.simpleft.FT] DEFINED 00:00:00:00:00:00:00:05 as
ROLE_MASTER, reply.getRole:ROLE_MASTER!
```

Gambar 3.7 Log Sinkronisasi Switch Controller After Failover

(c) Update Attachment Point

Event ini memberitahukan bahwa terjadi perubahan pada jalur pengiriman paket akibat terjadinya controller failure.

```
2016-11-03 01:47:47.639 INFO  [n.f.d.i.Device] updateAttachmentPoint: ap [AttachmentPoint
[sw=00:00:00:00:00:00:00:01, port=2, activeSince=Thu Nov 03 01:47:47 EDT 2016,
lastSeen=Thu Nov 03 01:47:47 EDT 2016]]  newmap null
```

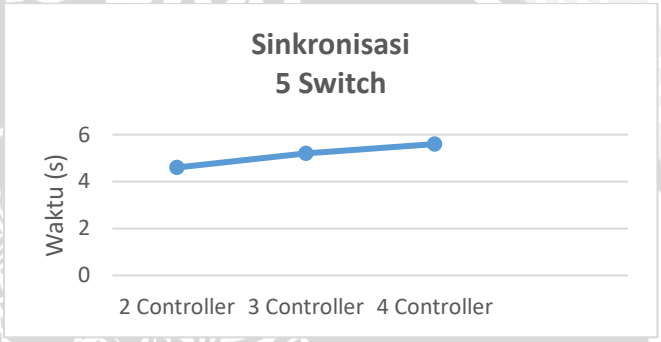
Gambar 3.8 Log Update Attachment Points

Waktu failover adalah selisih waktu antara event Switches Managed dengan update attachment point.

3.2 Kasus Uji 1 – Sinkronisasi Switch

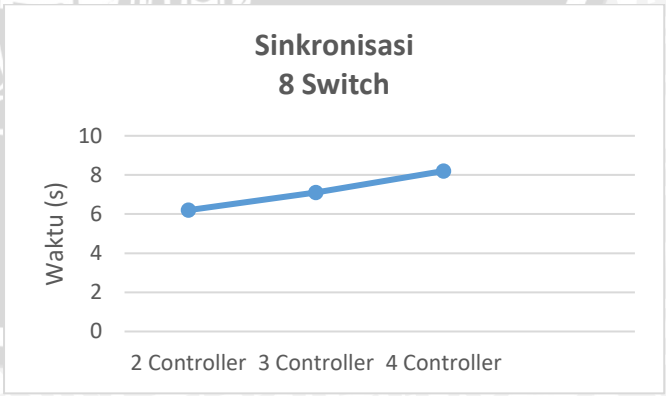
3.2.1 Hasil Perhitungan Waktu Sinkronisasi

Berikut ini adalah hasil perhitungan waktu sinkronisasi yang dibutuhkan controller dengan menggunakan lima dan delapan switch. Hasil yang ditampilkan merupakan waktu rata-rata yang didapat dari melakukan percobaan sebanyak 5 kali.



Gambar 3.9 Grafik Waktu Sinkronisasi 5 Switch

Pada gambar dapat dilihat bahwa dengan menggunakan dua controller waktu sinkroniasi yang dibutuhkan oleh kedua controller sebesar 4,6 detik. Dengan menggunakan tiga controller sebesar 5,2 detik, dan dengan menggunakan empat controller sebesar 5,6 detik. Dapat ditarik kesimpulan bahwa semakin banyak controller maka waktu yang dibutuhkan oleh controller untuk melakukan sinkronisasi akan semakin banyak.



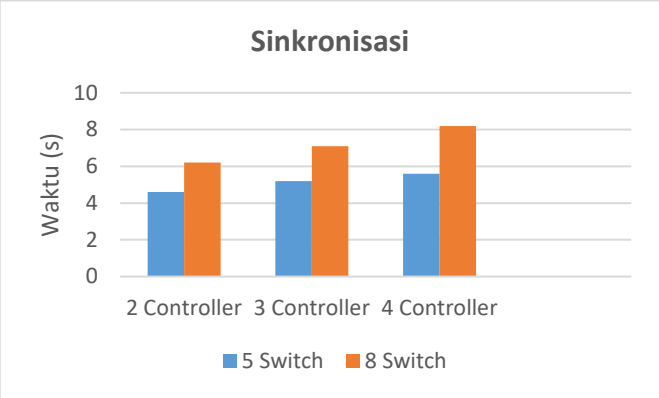
Gambar 3.10 Grafik Waktu Sinkronisasi 8 Switch

Pada gambar dapat dilihat bahwa dengan menggunakan dua controller waktu sinkronisasi yang dibutuhkan oleh kedua controller sebesar 6,2 detik. Dengan menggunakan tiga controller sebesar 7,1 detik, dan dengan menggunakan empat controller sebesar 8,2 detik. Sama seperti penggunaan dengan lima switch, penggunaan dengan delapan swith juga memperlihatkan bahwa semakin banyak controller waktu sinkronisasi yang dibutuhkan juga semakin banyak. Disini

dapat dilihat juga bahwa dengan menggunakan jumlah controller yang sama dan jumlah switch yang berbeda dapat mempengaruhi banyaknya waktu yang dibutuhkan untuk sinkronisasi.

3.2.2 Hasil Perbandingan Waktu Sinkronisasi

Pada perhitungan waktu sinkronisasi yang diuji dapat dibandingkan dengan gambar dibawah ini.



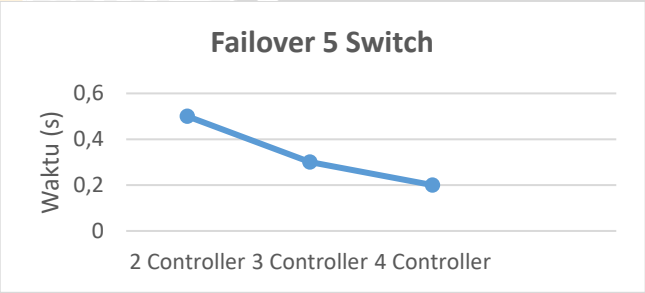
Gambar 3.11 Grafik Perbandingan Waktu Sinkronisasi

Pada gambar diatas dapat dilihat bahwa perbedaan waktu sinkronisasi pada controller yang terjadi. Dengan menggunakan 5 switch waktu sinkronisasi yang dibutuhkan semakin lambat apabila jumlah controller semakin banyak, sedangkan dengan menggunakan 8 switch waktu sinkronisasi yang dibutuhkan semakin lambat apabila jumlah controller semakin banyak. Dari hasil tersebut dapat diasumsikan bahwa faktor yang sangat mempengaruhi dalam waktu sinkronisasi switch dengan controller adalah banyaknya controller yang dipakai.

3.3 Kasus Uji 2 – Failover Controller

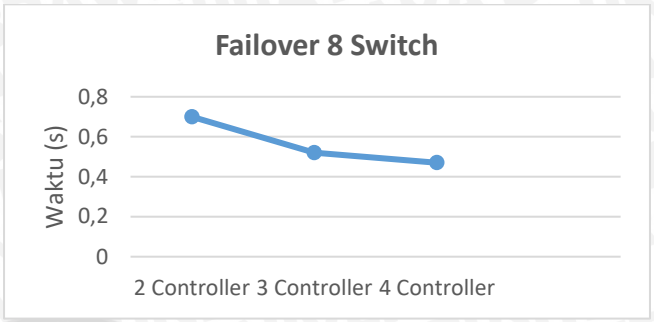
3.3.1 Hasil Perhitungan Waktu Failover

Berikut ini adalah hasil perhitungan waktu failover yang dibutuhkan controller dengan menggunakan lima dan delapan switch. Hasil yang ditampilkan merupakan waktu rata-rata yang didapat dari melakukan percobaan sebanyak 5 kali.



Gambar 3.12 Grafik Waktu Failover 5 switch

Pada gambar diatas dapat dilihat bahwa dengan menggunakan dua controller waktu failover yang dibutuhkan oleh kedua controller sebesar 0,5 detik. Dengan menggunakan tiga controller waktu yang dibutuhkan sebesar 0,3 detik. Sedangkan untuk empat controller waktu yang dibutuhkan masing-masing sebesar 0,2 detik. Dengan hasil tersebut dapat dilihat bahwa terjadi penurunan waktu ketika menggunakan controller lebih dari tiga.

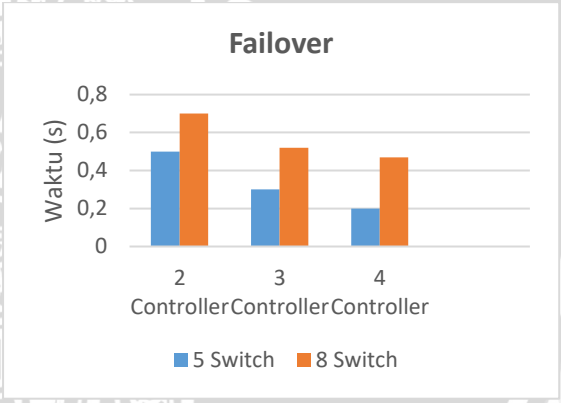


Gambar 3.13 Grafik Waktu Failover 8 switch

Pada gambar dapat dilihat bahwa dengan menggunakan dua controller waktu failover yang dibutuhkan oleh kedua controller sebesar 0,7 detik. Dengan menggunakan tiga controller waktu failover yang dibutuhkan oleh ketiga controller sebesar 0,52 detik, 0,47 detik waktu failover yang dibutuhkan pada empat controller. Pada penggunaan 8 switch grafik selalu menurun yang berarti bahwa waktu failover semakin cepat dengan penggunaan controller yang lebih banyak.

3.3.2 Hasil Perbandingan Waktu Failover

Pada perhitungan waktu failover yang diuji menggunakan dua kasus yang berbeda dapat dibandingkan dengan gambar dibawah ini



Gambar 3.14 Grafik Perbandingan Waktu Failover 2 switch dan 8 switch

Pada gambar diatas dapat dijelaskan bahwa ada perbedaan waktu controller yang terjadi, dengan menggunakan 8 switch yaitu durasi failover yang terjadi semakin cepat apabila jumlah controller semakin banyak, sedangkan dengan menggunakan 5 switch waktu failover yang terjadi dengan menggunakan lebih banyak controller tidak semakin cepat. Dari hasil tersebut dapat disimpulkan bahwa faktor perbedaan yang membuat waktu failover dengan menggunakan 5 switch dan 8 switch adalah perberdaan jumlah switch yang diatur oleh controller yang mati.

Dari proses uji diatas dapat disimpulkan bahwa banyaknya switch yang dikontrol oleh controller yang mati menjadi faktor utama dalam cepat lambatnya waktu failover, hal ini menjelaskan bahwa jika controller ditambah maka waktu failover menjadi semakin sedikit karena banyaknya controller dapat mengurangi jumlah switch yang dikontrol oleh masing-masing controller.

4. PENUTUP

4.1 Kesimpulan

Dengan demikian dari penilitian ini dapat disimpulkan beberapa hal yaitu:

- Semakin banyak switch yang dipakai akan berpengaruh pada proses cepat dan lambatnya sinkronisasi *switch.*
- Semakin banyak controller yang dipakai akan berpengaruh terhadap waktu sinkronisasi yang terjadi. Hal ini disebabkan karena beban controller dalam menangani *switch* akan berubah apabila terjadi failover.
- Dalam hal *failover*, semakin banyak *controller* atau semakin sedikit *switch* maka waktu untuk melakukan proses *failover* akan semakin cepat.

4.2 Saran

Berdasarkan hasil penilitian yang sudah dilakukan, penulis memberikan beberapa saran sebagai berikut:

- Penggunaan *multi controller* memang akan memudahkan kinerja pada jaringan apabila terjadi kegagalan, namun hal itu tidak bisa dilakukan secara sembarangan karena akan membuat penambahan *controller* menjadi sia-sia. Untuk menentukan banyaknya controller harus memperlihatkan kebutuhan jaringan, efisiensi dari penambahan controller dan juga harga yang harus dikeluarkan.
- Dengan menggunakan *Failover multi controller* setiap *controller* dapat bereran terhadap seluruh *switch*, letak penanganan *switch* dapat disesuikan dengan kebutuhan. Namun dalam hal sinkronisasi akan semakin lama apabila terjadi penambahan *controller*. Hal ini disebabkan karena setiap *controller* harus berkomuniasi dengan seluruh switch bukan hanya *switch* yang menjadi *master* tetapi yang menjadi *slave* juga.
- *Floodlight controller* merupakan *Software Defined Network (SDN) controller open source* yang cukup baik untuk menerapkan proses *failover* pada *controller*. Servis yang disediakan cukup banyak ketika ingin membuat sebuah modul program baru. Meski begitu dokumentasi dan informasi dari *controller* ini masih kurang. Para developer floodlight perlu dokumentasi dan cara penggunaan controller ini agar lebih mudah dipakai.

Daftar Pustaka

[1] Akbar, S., Broto, T., Harsono, dan Naning, S. *Analisa perfomansi penanganan kegagalan link pada layer 2 pada jaringan software defined network openflow*.

[2] Beheshti, N., dan Zhang, Y., 2012. *Fast failover for control traffic in software defined networks*.

[3] Chan, Y. C., Wang. K., dan Hsu, Y. H., 2015. *Fast controller failover for multi domain software-defined networks*. In *Networks and Conmmunications (EuCNC), 2015 European Conference on*, pp 370-374.

[4] D., T. N. dan Gray, K., 2013. *SDN: Software Defined Networks. O'Reilly Media, Inc., 1st edition*.

[5] Floodlight., 2016. http://www.projectfloodlight.com/. [Diakses 4 September 2016].

[6] Foundation, O. N., 2013. Openflow switch specification version 1.4.

[7] Metzler, J. dan Asthon, S., 2015. Ten things to look for in an sdn controller.

[8] Mininet., 2016. http://www.mininet.org/. [Diakses 4 September 2016]

[9] Mudha, S., dan Anggara. *Pengujian Perfoma Kontroler Software-defined Network (SDN) : POX dan Floodlight.*

[10] Obadia, M., Bouet, M., Leguay, J., dan Iannone, L., 2015. *A Greedy Approach for Minimizing SDN Control Overhead*.

[11] Obadia, M., Bouet, M., Leguay, J., Phemius, K., dan Iannone, L., 2014. *Failover Mechanisms for Distributed SDN Controllers.*

[12] Pashkov, V., Shalimov, A., dan Smeliansky, R., 2014. *Controller failover enterprise networks.* In *Science and Technology Conference (Modern Networking Technologies) (MoNeTeC), 2014 International*, pp 1-6

[13] Pheims, K., Bouet, M., dan Leguay, J., 2013. *DISCO: Distributed Multi-domain SDN Controller*.

[14] Phemius, K., Bouet, M., dan Leguay, J., 2014. *Distributed SDN for Mission-critical Networks.*