BAB 2 LANDASAN KEPUSTAKAAN

Bab ini membahas teori penunjang yang digunakan penulis sebagai bahan referensi mengenai *Software Defined Network* (SDN), *Openflow*, beserta bagian pendukung lainnya seperti kontroler yang digunakan, Floodlight, Maestro, RYU, POX, dan ONOS.

2.1 Kajian Pustaka

Berdasarkan judul skripsi yang dibahas, penulis menggunakan penelitian-penelitian yang telah dilakukan sebelumnya, penelitian mengenai perbandingan performansi pada kontroler *Software Defined Network* (SDN). Dimana penelitian sebelumnya bebagai berikut:

Tabel 2.1 Penelitian terdahulu

No	Nama Penulis, Tahun, dan Judul	Persamaan	Pebedaan
1	Wikansetya Luhur Pambudi, Ferry Wahyu Wibowo. 2015. "Uji *Throughput* Kontroler Floodlight Dan Beacon Menggunakan Emulator Mininet"	Melakukan analisis performa kontroler dengan menggunakan parameter t*hroughput*	Skenario pengujian. Tanpa melakukan analisis perfoma kontroler dengan menggunakan parameter *latency*
2	Rikie Kartadie. 2015. "Uji Performa Kontroler Floodlight dan Opendaylight Sebagai Komponen Utama Arsitektur *Software*-Defined Network".	Melakukan analisis perfoma kontroler dengan menggunakan parameter t*hroughput* dan l*atency*	Skenario pengujian. Jumlah *switch* dan *host* yang diberikan pada pengujian
3	Sawung Murdha Anggara. 2015. "Pengujian Performa Kontroler *Software*-Defined Network (SDN): POX dan Floodlight".	Melakukan analisis perfoma kontroler dengan menggunakan parameter throughput dan l*atency*	Skenario pengujian. Jumlah *switch* dan *host* yang diberikan pada pengujian. Mengetahui perbedaan performa dari kontroler yang dibangun dari dua bahasa yang berbeda dan dibangun dari dua kontroler yang berbeda

Penelitian pertama adalah penelitian yang dilakukan oleh Wikansetya Luhur Pambudi dengan judul "Uji *Throughput* Kontroler Floodlight dan Beacon Menggunkan Emulator Mininet". Penelitian tersebut bertujuan untuk melakukan perbandingan terhadap kontroler *Software Defined Network* (SDN), yaitu Floodlight dan Beacon. Dari hasil penelitian yang dilakukan ditarik kesimpulan bahwa cara pengujian Floodlight dan Beacon menggunakan emulator mininet dengan *tool* benchmark bernama Cbench. Kemudian setiap penambahan jumlah *host* pada pengujian *throughput* kontroler Floodlight menghasilkan nilai *throughtput* yang terus meningkat, sedangkan pada pengujian *throughput* kontroler Beacon menghasilkan nilai *throughput* yang tidak stabil. Tetapi hasil nilai *throughput* kontroler Beacon memiliki jumlah *flow* tiap detiknya lebih besar dari hasil nilai *throughput* kontroler Floodlight (Pambudi, 2015).

Penelitian kedua dilakukan oleh Rikie Kartadie dengan judul "Uji Performa Kontroler Floodlight dan Opendaylight Sebagai Komponen Utama Arsitektur Software-Defined Network". Penelitian tersebut bertujuan untuk mengetahui tingkat *throughput* dan *latency* dari kontroler, sehingga akan diperoleh informasi yang tepat kemampuan dari kontroler yang akan digunakan. Dalam penelitian ini menggunakan Cbench atau sebuah bechmaking *tool* yang yang digunakan untuk mengevaluasi parameter *throughput* dan *latency* dari kontroler. Dengan didapatkannya kesimpulan bahwa kontroler Floodlight lebih baik performanya dibandingkan dengan kontroler Opendaylight untuk jumlah *switch* yang besar, dan juga kontroler Floodlight mampu memberikan *throughtput* yang lebih besar dibandingkan dengan Opendaylight (Kartadie, 2016).

Penelitian ketiga adalah penelitian yang dilakukan oleh Sawung Murdha Anggara dengan judul penelitian "Pengujian Performa Kontroler *Software*-Defined Network (SDN): POX dan Floodlight". Penelitian tersebut bertujuan untuk mengetahui perbedaan performa dari kontroler yang dibangun dari dua bahasa yang berbeda dan dibangun dari dua kontroler yang berbeda. POX adalah kontroler yang dibangun berdasarkan bahasa pemrograman python sementara floodlight dibangun berdasar bahasa pemrograman java. Kedua kontroler ditesting dengan alat uji performa yang sering digunakan dalam penelitian yaitu collective benchmark *tool* atau Cbench. Dengan menggunakan Cbench akan dihasilkan nilai *throughput* dan *latency* dari hasil masing-masing kontroler, yang memungkinkan untuk mengetahui nilai performa dari kinerja kontroler tersebut. Kemudian hasil dari penelitian yang dilakukan diperoleh hasil bahwa kontroler floodlight mampu melakukan penanganan aliran data dalam jumlah yang sangat besar daripada POX hingga batasan jumlah *host* tertentu. Menunjukkan bahwa nilai performa floodlight dalam pengelolaan aliran data lebih tinggi dari POX, sehingga disimpulkan bahwa floodlight lebih menjamin pengelolaan data dalam jumlah besar dan membutuhkan pengaturan aliran data yang sangat tinggi daripada apabila data itu dikelola dengan POX (Anggara, 2015).

2.2 Software defined network

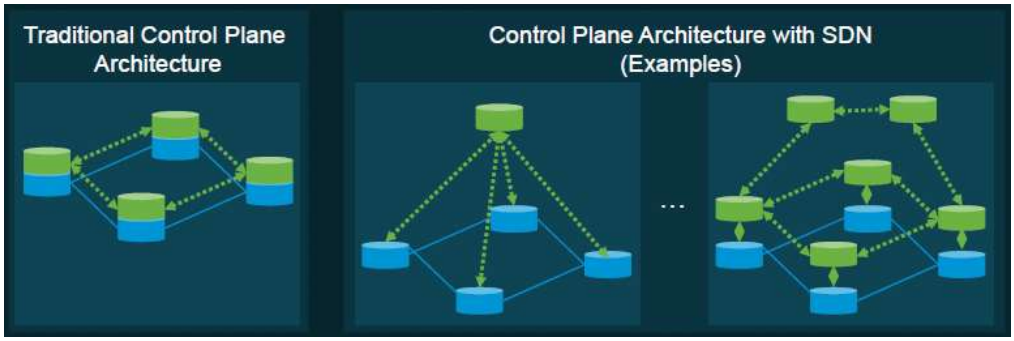
Software Defined Network (SDN) adalah istilah yang merujuk pada konsep atau paradigma baru dalam merancang, mengelola dan mengimplementasikan jaringan, terutama untuk mendukung kebutuhan dan inovasi di bidang ini yang semakin lama semakin kompleks. Konsep dasar *Software Defined Network* (SDN) adalah dengan melakukan pemisahan antara *control plane* dan *data plane*, serta kemudian melakukan abstraksi sistem dan mengisolasi kompleksitas yang ada pada komponen atau sub-sistem dengan mendefinisikan *interface* yang standar.

Terdapat beberapa aspek penting di dalam SDN antara lain sebagai berikut:

1. Adanya pemisah secara fisik antara *control plane* dan *data plane*.
2. *Interface* standar (*vendor-agnosic*) untuk memprogram perangkat jaringan.
3. *Control plane* yang terpusat atau adanya sistem operasi jaringan yang mampu membentuk *logical map* dari seluruh jaringan dan kemudian mempresentasikan melalui *Application Programming Inteface* (API).
4. Virtualisasi dimana beberapa sitem operasi jaringan dapat mengkontrol bagian-bagian *slices* atau *substrates* dari perangkat yang sama.

Software Defined Network (SDN) hadir dalam menghadapi kompleksitas dan tantangan pada jaringan saat ini. Dalam hal ini terkait dengan kebutuhan inovasi untuk bidang jaringan yang semakin kompleks. Termasuk diantaranya adalah fakta-fakta dan kebutuhan berikut:

1. Virtualisasi dan *cloud*: Komponen dan entitas jaringan *hybrid* – antara fisik bare mental dan yang virtual.
2. *Orchestration* dan *Scalability*: Kemampuan untuk mengatur dan mengelola ribuan perangkat melalui *sebuah point of management*.
3. *Programmability* dan *Automation*: Kemampuan untuk mengubah *behaviour* secara otomatis.
4. *Visibility:* Kemampuan untuk dapat memonitor jaringan, baik dari sisi sumber data, konektivitas dan lain-lain.
5. Kinerja: Kemampuan untuk memaksimalkan penggunaan perangkat jaringan. Misalkan optimasi *bandwidth, load balancing, traffic enginering* dan lain lain yang menghubungkan dengan *programmability* dan *scalability* (Risdianto, Arif dan Mulyana, 2016).



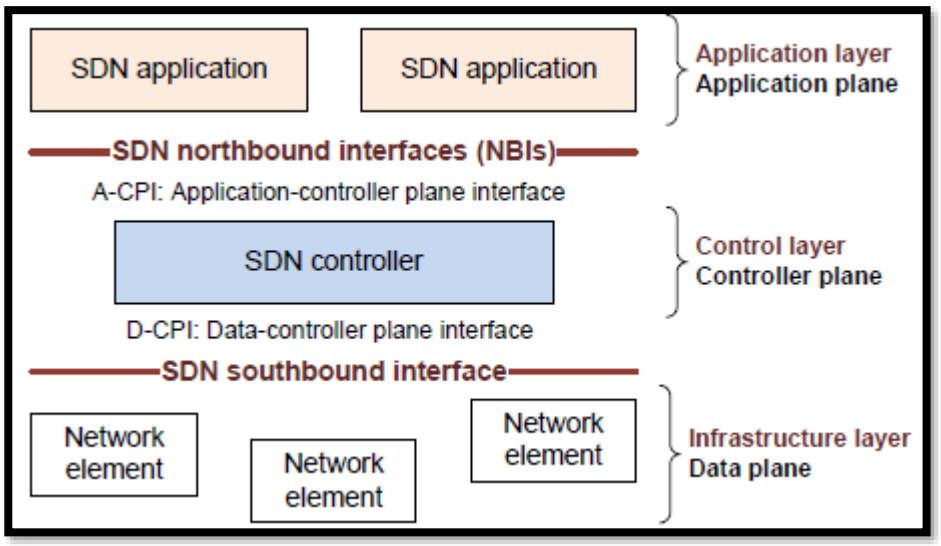
Gambar 2.1 Arsitektur jaringan tradisional dan SDN

Sumber: Network Operating Systems Group

Seperti yang ditunjukkan pada **Gambar 2.1**, dengan adanya pemisahan antara *control plane* dan *data plane* juga dengan kemampuan memusatkan logika jaringan pada satu entitas tertentu, arsitektur jaringan *Software Defined Network* (SDN) diharapkan mampu untuk mengatasi kompleksitas yang terjadi pada jaringan tradisional (Pratama, 2015).

2.2.1 Arsitektur Software defined network

Dalam konsep *Software Defined Network* (SDN), tersedia *open interface* yang memungkinkan sebuah entitas *software* untuk mengendalikan konektivitas yang disediakan oleh sejumlah sumber daya jaringan, mengendalikan aliran trafik yang melewatinya dan juga melakukan inspeksi terhadap trafik tersebut.



Gambar 2.2 SDN Komponen

Sumber: Open Networking Foundation

Dari **Gambar 2.2** tersebut menunjukkan arsitektur SDN dapat dilihat sebagai tiga lapis komponen, yaitu:

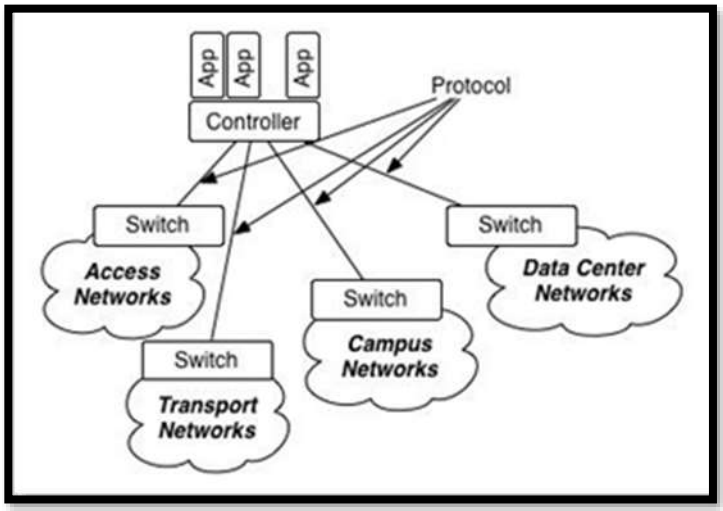
1. Infrastruktur (*data plane/infrastructure layer*): Terdiri dari elemen jaringan yang dapat mengatur SDN *Datapath* sesuai dengan instruksi yang diberikan melalui *Control Data plane Interface* (CDPI).
2. Kontrol (*control plane/layer*): Entitas kontrol (SDN *Controller*) mentranslasikan kebutuhan aplikasi dengan infrastruktur dengan memberikan intruksi yang sesuai untuk SDN *Datapath* serta memberikan informasi yang relevan.
3. Aplikasi (*application plane/layer*): Berada pada lapis teratas, berkomunikasi dengan sistem melalui *North Bound Interface* (NBI).

Bidang management dan admin bertanggungjawab dalam inisiasi elemen jaringan, memasangkan SDN *Datapath* dengan SDN *Controller*, atau menkonfigurasi cakupan (*Coverage*) dari SDN *Controller* dan SDN *App* (Risdianto, Arif dan Mulyana, 2016).

2.3 Openflow

Openflow adalah sebuah protokol yang memungkinkan pengaturan penjaluran dan pengiriman paket ketika melalui sebuah *switch*. Pada Gambar 2.3 dijelaskan mengenai topologi pada *Openflow*. Dari gambar tersebut dapat diketahui bahwa pada sebuah jaringan konvensional, setiap *switch* hanya berfungsi meneruskan paket yang lewat *port* yang sesuai tanpa dapat membedakan tipe protokol data yang dikirim.

Dengan menggunakan *Openflow* dapat dilakukannya *flow forwarding* berbasis *network layer* juga dapat dilakukan pengaturan pergerakan paket secara terpusat, sehingga aliran paket di jaringan dapat diprogram secara independen. Hal ini dapat dilakukan dengan membuat algoritma dan forwading rules di controller server kemudian aturan tersebut didistribusikan ke *switch* yang ada di jaringan (Ardiansyah, 2012).

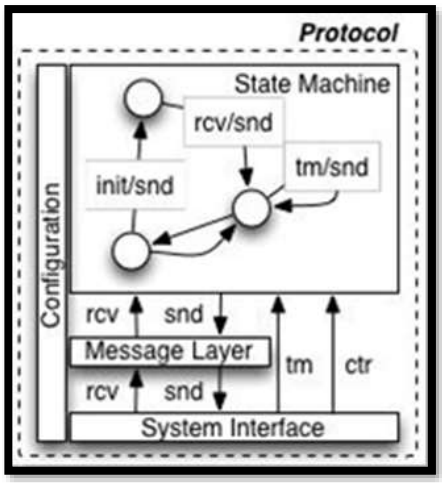


Gambar 2.3 Topologi Openflow

Sumber: Flowgrammable

2.3.1 Protokol Openflow

Protokol *Openflow* dapat dibagi menjadi empat komponen, diantara lain: *message layer, state machine, system interface,* dan *configuration.*



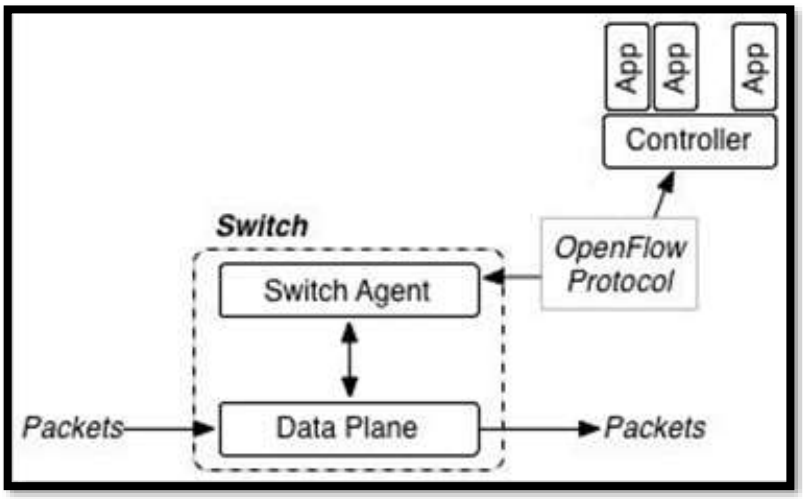
Gambar 2.4 Arsitektur protokol pada Openflow

Sumber: Flowgrammable

Berikut adalah deskripsi komponen dari protokol *Openflow* (Flowgrammable, 2016):

1. *Message Layer*: *Message Layer* merupakan inti dari protokol *stack*. Layer ini mendefinisikan semantik dan terstruktur secara valid untuk semua pesan. Message layer pada umumnya mendukung fitur untuk *construct, copy, compare, print,* dan memanipulasi pesan.
2. *State Machine: State machine* bekerja untuk mengatur sistem kerja dari protokol pada *low-level. State machine* digunakan untuk mendeskripsikan kinerja contohnya seperti: *negotiation, capability dicover, flow control,* dan lain lain.
3. *System Interface*: Mendefinisikan bagaimana protokol berinteraksi dengan dunia luar. *System interface* mengidentifikasi kebutuhan baku maupun opsional dari *interface* dengan tujuan dari pengguna, seperti TLS dan TCP sebagai *transport channel*.
4. *Configuration:* Hampir semua aspek dari protokol memiliki atau nilai awal. Konfigurasi dapat mengatasi semua pengaturan mulai dari *buffer size default* dan *interval reply* untuk sertifikat X.5099.
5. *Data Model:* Pada *Openflow* salah satu yang perlu dipertimbangkan adalah bagaimana memahami model yang mendasarinya. Setiap *switch* mengatur model data relasional yang berisi atribut untuk setiap abstraksi *Openflow*. Atribut-atribut ini mendeskripsikan kapabilitas, kondisi konfigurasi, maupun beberapa set dari statistik saat ini.

2.3.2 Switch Openflow

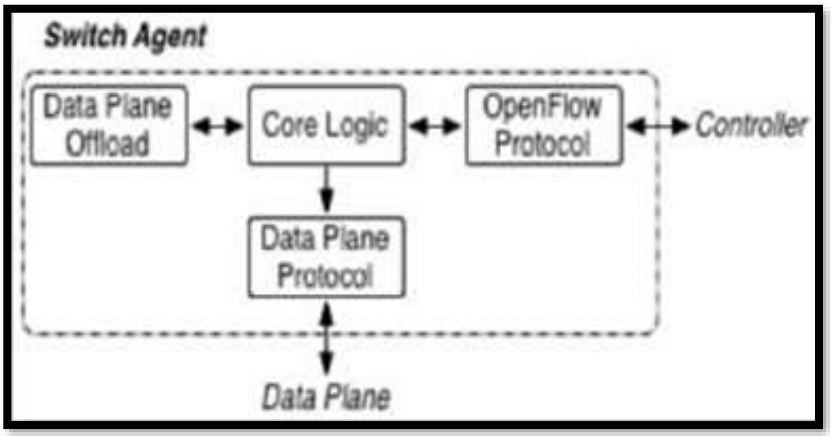


Gambar 2.5 Arsitektur Switch pada Openflow

Sumber: Flowgrammable

Openflow dapat dibagi menjadi dua komponen, yaitu *switch agent* dan *data plane*. *Switch* agent membahas mengenai protokol *Openflow* pada satu atau lebih kontroler. Selain itu *switch agent* juga berkomunikasi dengan *data plane* dengan protokol internal yang terkait. *Switch* agent akan mentranslasikan perintah dari kontroler ke dalam intruksi *low-leve*l yang diperlukan untuk mengirim ke *data plane* dan notifikasi *internal data plane* ke dalam pesan *Openflow* yang sesuai dan melakukan *forwarding* ke kontroler. *Data plane* melakukan *forwarding* dan manupulasi pada seluruh paket meskipun didasarkan pada konfigurasi yang digunakan terkadang mekanisme yang bekerja akan mengirimkan paket pada *switch agent* dalam saat tertentu.

2.3.2.1 Switch Agent



Gambar 2.6 Arsitektur Switch agent

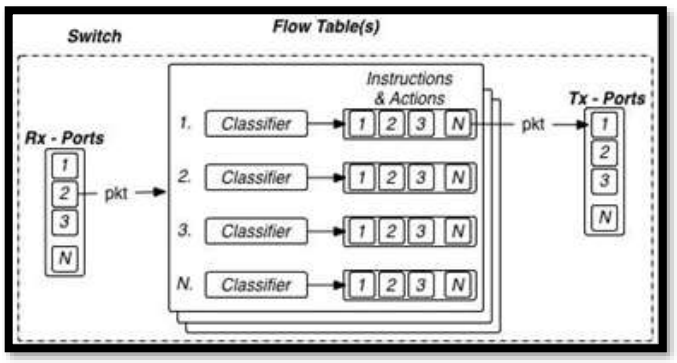
Sumber: Flowgrammable

Berikut adalah deskripsi komponen dari *Switch Agent* (Flowgrammable, 2016):

1. *Protocol Openflow*: Ini merupakan bagian *switch* dari protokol *Openflow*.
2. *Core Logic*: Komponen ini mengatur *switch*, menjalankan perintah terhadap *data plane* sesuai dengan kebutuhan, mengatur *offload* dari *data plane*, dan lain sebagainya.
3. *Offload Data plane*: Seringkali *control plane* melakukan *offload* untuk beberapa fungsi pada *Openflow* tetapi tidak terdapat pada implementasi *data plane*.
4. *Data plane Protocol*: Protokol internal yang digunakan untuk mengatur keadaan *data plane*.

2.3.2.2 Data plane

Data plane dari *switch* terdari dari: *port*, *table flow*, *flow*, *calssifier* dan *action*. Paket yang keluar masuk sistem melalui *port.* Paket yang sesuai dengan *flow* pada tabel *flow* menggunakan *classifier. Flow* berisi set dari beberapa *action* yang diaplikasikan pada setiap paket yang sesuai.



Gambar 2.7 Arsitektur data plane

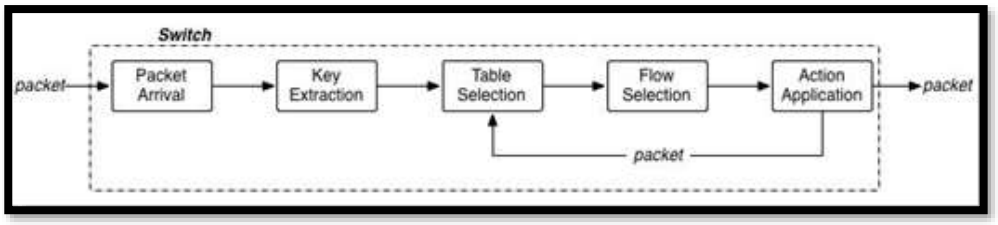
Sumber: Flowgrammable

Berikut adalah deskripsi komponen dari *Data plane* (Flowgrammable, 2016)

1. *Port:* Paket masuk dan keluar melalui *switch* pada suatu *port*. Pada setiap versi protokol yang berbeda mendukung tipe *port*, properties, dan *configurations*.
2. *Classifiers:* Mempunyai peranan untuk mencocokkan paket untuk *flow entry* pada tabel *flow*.
3. *Instructions and Actions*: Mengatur bagaimana paket ini diproses waktu *flow entry* sesuai dengan yang ada pada tabel *flow*.

2.3.2.3 Data plane – Packet Lifecycle

Setiap paket diperlakukan secara sama ketika paket melewati *switch* dari *data plane*. Ketika paket masuk, dibuat sebuah *key* yang berisi informasi dari paket, seperti nilai dari protokol *field* tertentu, juga berisi metadata dari paket, seperti *port* masuk, waku masuknya paket, dan lain lain. *Key* digunakan untuk memilih *flow* tersebut di sebuah tabel *flow* dan asosiasi dari aksi diterapkan pada paket. Set dari aksi dapat menjalankan *drop, mutate, queue, forward*, ataupun pengalihan paket menuju tabel *flow* baru.



Gambar 2.8 Packet Lifecycle

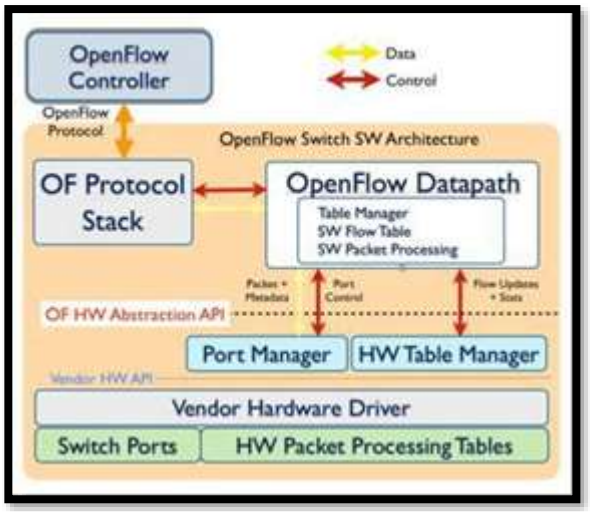
Sumber: Flowgrammable

Berikut adalah deskripsi komponen dari *Packet Lifecycle* (Flowgrammable, 2016):

1. *Packet Arrival:* Paket yang masuk melalui *port*, dapat berupa *port* fisikal maupun virtual. Tahap ini penting untuk mengingat beberapa informasi tentang *port* yang dilalui untuk pemrosesan *source-based* nantinya dalam *pipeline.*
2. *Key Estraction:* Pada setiap paket memiliki porsi kecil dari metadata *built* yang memanggil *key* setelah paket masuk. *Key* terdiri dari beberapa *field* di dalam paket juga pada sisi informasi seperti: Lokasi dari peket *buffer*, nilai *header*, *port* masuk, *arrival clock*, dan lain-lain. *Key* diumpankan untuk memenuhi *pipeline*.
3. *Table Selection*: *Key* digunakan untuk mencari tabel, namun juga terkadang terdapat lebih dari satu tabel secara bersamaan. Dalam hal ini, tabel harus dipilihkan terlebih dahulu. Ketika paket melalui pipeline untuk pertama kali, tabel pertama dipilih secara *default*; tabel selanjutnya dapat terpilih melalui *action* atau *miss table*.
4. *Flow Selection*: *Key* digunakan untuk memilih *flow* tertentu dalam tabel. *Flow* pertama dari suatu tabel dimana classifier menggolongkan *key* menjadi *flow* yang terpilih.
5. *Action Application*: Setiap baris berisi set *action* yang diterapkan ke semua golongan paket. Ketika *flow* dipilih, *action* akan diterapkan ke paket terkait. *Action* dapat memodifikasi keadaan paket atau mempengaruhi kinerja dari paket.

2.4 Kontroler Openflow

Pada **Gambar 2.9** menunjukkan model high-level pada *Openflow Switch*. Dari gambar tersebut terlihat bahwa data yang mesuk melalui *switch port* dan berpotensi untuk dilakukan *redirect* secara langsung pada *port output* berdasarkan HW *Packet processing tables*. Selain itu paket juga akan diarahkan langsung pada *port manager,* dan *Openflow Datapath* yang ada jika tidak ada *entry* pada HW *tables* (sc.uaf.edu, 2016).



Gambar 2.9 Arsitektur Openflow Software

Sumber: Openflow Hardware Abstr*action* API Specification

Dengan begitu kontroler pada *Openflow* merupakan bagian tertentu yang bertanggungjawab pada proses logika di dalam jaringan. Seperti proses *routing*, *switch*ing, dan QoS. Letak dari kontroler itu sendiri dapat berada pada komputer server. Oleh karena itu mekanisme pada struktur jaringan komputer dikendalikan seluruhnya oleh kontroler tersebut. Untuk dapat mendukung kerja dari kontroler dibutuhkan modul perangkat lunak yang dikembangkan sesuai dengan kebutuhan tertentu.

2.5 Floodlight

Floodlight merupakan salah satu kontroler enterprise terbuka berlisensi Apache yang dikembangkan oleh komunitas pengembang di Big Switch Network. Pada penerapannya Floodlight menggunakan bahasa pemrograman Java. Adapun beberapa fitur Floodlight adalah (Anggara, 2015):

1. *Modul Loading System* yang membuatnya lebih sederhana dan mudah untuk dikembangkan
2. Mendukung *switch* virtual maupun *switch* fisik
3. Dapat menangani kombinasi jaringan *Openflow* maupun *non Openflow*
4. Mendukung virtualisasi jaringan seperti *OpenStack*

2.6 Maestro

Maestro adalah sistem operasi untuk merancang aplikasi jaringan kontrol. Maestro menyediakan *interface* untuk mengimplementasikan aplikasi kontrol jaringan modular untuk mengakses dan memodifikasi keadaan jaringan, dan mengkoordinasikan interaksi mereka. Maestro adalah *platform* untuk mencapai fungsi kontrol jaringan otomatis dan program yang menggunakan aplikasi modular. Meskipun proyek ini berfokus pada membangun *Openflow* kontroler menggunakan Maestro. Maestro tidak hanya terbatas pada jaringan *Openflow*.

Kerangka pemrograman Maetro juga menyediakan *interface* untuk (Pangestu, 2016):

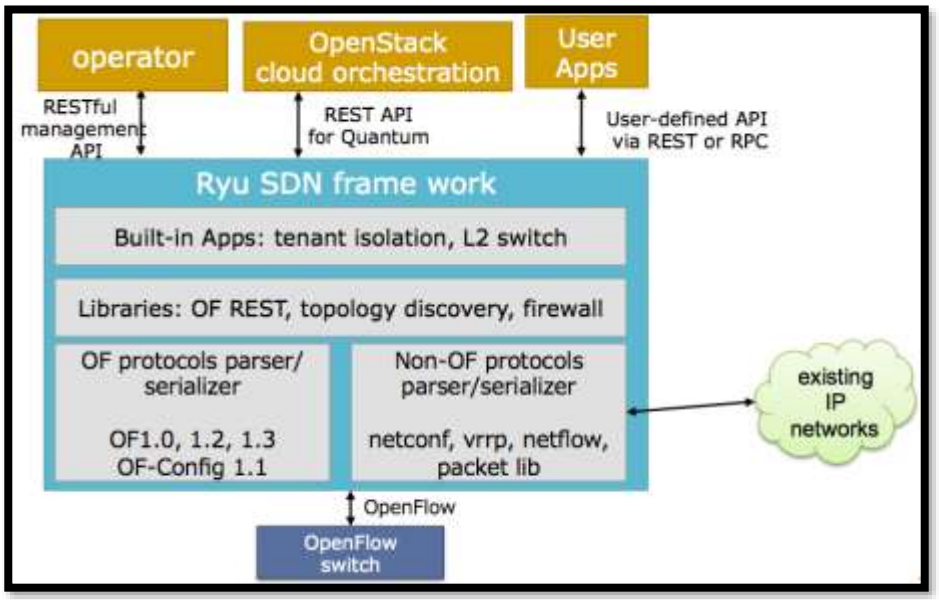
1. Memperkenalkan fungsi kontrol dengan menambahkan komponen kontrol termodulasi.
2. Mempertahankan *network state* pada komponen kontrol.
3. Menyusun komponen kontrol dengan menentukan urutan eksekusi dan *network state* dengan komponen tersebut.

Selain itu Maestro juga mencoba untuk mengeksploitasi paralelisme dalam satu komputer untuk meningkatkan kinerja *Throughput* sistem. Fitur dasar jaringan *Openflow* adalah kontroler yang bertanggungjawab untuk pembentukan pada awal setiap aliran dengan menghubungkan pada *switch* terkait.

2.7 RYU

RYU sering disebut sebagai komponen dasar, perangkat lunak *open source* pada kerangka kerja jaringan (*networking framework*). RYU diimplementasikan sepenuhnya dengan Phyton, dan didukung oleh laboratorium NTT. Seperti kontroler SDN lainnya, RYU juga menyediakan komponen perangkat lunak dengan API yang tak tersembunyi untuk memungkinkan para pengembang untuk membuat sebuah pengelolaan jaringan baru dan mengontrol aplikasi. Kelebihan RYU adalah mendukung beberapa protokol *southbond* untuk mengelola perangkat, seperti *Openflow, Network Configuration Protocol (NETCONF), Openflow Management* dan *Configuration Protocol (OF-Config),* dan lain-lain.

Seperti kontroler SDN lainnya, RYU juga dapat membuat dan mengirim sebuah pesan *Openflow,* tergantung kepada kejadian asinkron seperti flow_removed, *parse* dan menangani paket masuk (Pemberton, 2014) .



Gambar 2.10 Arsitektur RYU

Sumber: RYU Openflow Controller

2.8 POX

POX adalah sebuah *platform* pengembangan *open source* untuk aplikasi *Software-developed Network* (SDN) yang berdasarkan pada bahasa pemrograman Python dan merupakan kontroler *Openflow*. POX memungkinkan proses perancangan dan pembangunan jaringan yang lebih cepat, serta menjadi lebih umum digunakan daripada pendahulunya NOX.

POX membutuhkan Python 2.7 untuk eksekusinya. Dalam prakteknya, juga dapat dijalankan menggunakan Python 2.6. POX dapat dijalankan di sistem operasi Windows, MacOS, dan Linux (meskipun telah digunakan pada sistem lain juga). Banyak pengembangan dilakukan di Mac OS, sehingga hampir selalu digunakan dalam Mac. POX dapat digunakan dengan "standar" Python interpreter (CPython), tetapi juga mendukung PyPy.

Dalam website resminya, NOXrepo menyampaikan daftar fitur POX sebagai berikut (Anggara, 2015):

a. Tampilan mirip python untuk antarmuka *Openflow* pada grafik kinerja POX.
b. Contoh komponen yang dapat digunakan kembali untuk seleksi jalur, penemuan topologi, dan lainnya
c. Dapat dijalankan di semua sistem.
d. Secara khusus dioperasikan untuk sistem operasi Linux, Mac OS, dan Windows.
e. Mendukung tampilan GUI dan visualisasi yang sama seperti NOX.
f. Memiliki kinerja yang lebih baik dibandingkan dengan aplikasi NOX.

2.9 ONOS

ONOS adalah sebuah sistem operasi (OS) yang dirancang untuk membantu penyedia layanan jaringan membangun jaringan berbasis *carrier-grade* yang dirancang untuk skalabilitas, ketersediaan dan kinerja tinggi. Meskipun dirancang khusus untuk memenuhi kebutuhan penyedia layanan, ONOS juga dapat bertindak sebagai pesawat kontrol SDN untuk jaringan area lokal (LAN) dan jaringan pusat data.

Open Networking Lab (ON.Lab) merilis kode sumber ONOS, yang ditulis dalam bahasa pemrograman Java, ke komunitas *open source* pada bulan Desember 2014. Pada bulan Oktober 2015, proyek ONOS bergabung dengan Linux Foundation sebagai proyek *open source* Linux kolaboratif. Rilis baru ONOS keluar setiap tiga bulan, yaitu pada bulan Februari, Mei, Agustus dan November, dan diberi nama menurut abjad berdasarkan seekor burung, yang juga merupakan logo ONOS. Dua rilis pertama diberi nama Avocet dan Blackbird. Seperti kebanyakan proyek open source lainnya, ONOS memiliki halaman GitHub dimana kolaborator dapat berkontribusi dalam perubahan kode.

Inti ONOS berdasarkan arsitektur modular, berlawanan dengan sistem terpadu yang mengaburkan pembagian antara komponennya. Modularitas ini membuat alur kerja utara-selatan terpisah dari alur kerja timur-barat sambil memungkinkan penyesuaian lebih mudah untuk keseluruhan sistem. Karena penyedia layanan memerlukan kemampuan untuk mengukur jaringan mereka, pengendali ONOS dapat melakukan skala untuk mengakomodasi sistem perangkat yang didistribusikan secara fisik. Hal ini memungkinkan penyedia layanan menambahkan *switch* atau komponen baru tanpa mengganggu bagian sistem lainnya. Selain itu, arsitektur terdistribusi mengurangi kegagalan atau eror pada jaringan, sebagai contoh yang sama dapat terjadi jika ada lainnya yang gagal. Hal ini, pada gilirannya, menghasilkan *high availibility*.

Sementara inti ONOS didistribusikan untuk memberikan jangkauan pada masing-masing jaringan pada perangkat *switch*, kontroler ONOS tetap tersentralisasi secara logis dan subdivisi terpisah atau contoh dalam arsitektur ONOS yang lengkap dapat dilihat dan diakses sebagai satu sistem. Untuk visibilitas dan manajemen sistem secara keseluruhan, ONOS menyediakan GUI yang relatif mudah. (Rouse, 2017)

2.10 Mininet

Mininet merupakan emulator jaringan yang menciptakan jaringan virtual *host*, *switch*, controller, dan link pada kernel Linux. Mininet sangat mendukung dalam bidang penelitian, *development, learning, prototyping, testing, debugging*, dan lain-lain yang mempunyai keuntungan eksperimental yang lengkap pada leptop atau PC. Berikut adalah keuntungan yang dimiliki oleh Mininet (mininet.org, 2016):

1. Menyediakan cara yang sederhana untuk pengujian jaringan dalam mengembangkan aplikasi *Openflow*.
2. Memungkinkan beberapa pengembang independent dalam bekerja pada topologi jaringan yang sama.
3. Memungkinkan pengujian pada topologi yang kompleks tanpa perlunya jaringan fisik.
4. Termasuk alat debug dan dapat menjalankan tes di seluruh jaringan.
5. Mendukung berbagai topologi, dan termasuk satu set dasar topologi.
6. Menyediakan Phyton API untuk menciptakan dan menguji jaringan.

2.11 Cbench (Collective Benchmark)

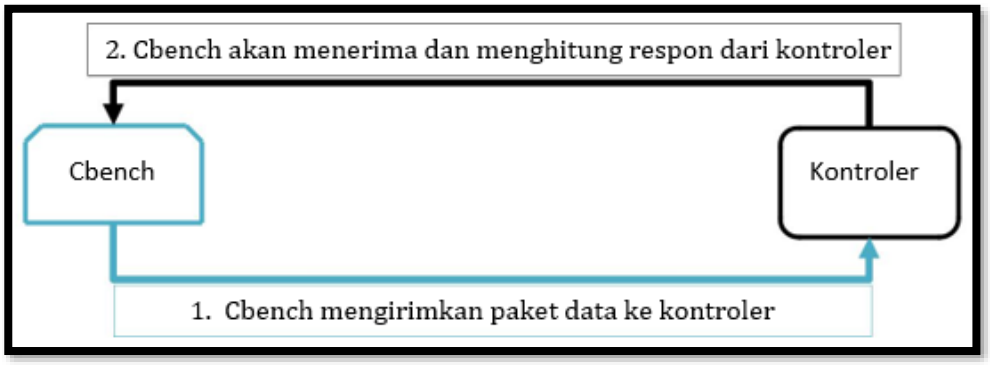
Cbench merupakan sebuah program untuk menguji kontroler *Openflow* dengan mengevaluasi parameter *throughput* dan *latency*, mengemulasi kelompok *switch* yang terhubung ke kontroler. Cbench merupakan bagian dari *Oflops (Openflow per second)* yang merupakan tool yang berfungsi untuk mengukur aspek khusus pada *switch Openflow* (archive.openflow.org, 2016).

Ketika Cbench dijalankan secara bersama-sama dengan kontroler, maka simulator Cbench akan menghasilkan sejumlah *switch* dan menciptakan koneksi ke kontroler. Kemudian Cbench memeriksa apakah mode yang digunakan merupakan *latency mode* ataukah *throughput mode.* Jika mode yang digunakan adalah *latency,* maka *emulated switch* akan mengirimkan pesan *PACKET_IN* ke

kontroler kemudian menunggu balasan *flow* yang dikirimkan oleh kontroler. Proses ini akan diulang sebanyak mungkin, yang kemudian Cbench akan menghitung respon yang diterima dari kontroler untuk menghitung waktu rata-rata kontroler dalam memproses *flow.* Dan jika mode yang digunakan adalah *throughput,* maka Cbench akan menghasilkan sejumlah *switch* dan menciptakan koneksi ke kontroler. Setiap *switch* akan mengirimkan sebanyak mungkin pesan *PACKET_IN* ke kontroler, kemudian Cbench menghitung berapa total respon yang diperoleh dari kontroler. Dengan demikian akan diperoleh jumlah respon kontroler dalam satu satuan waktu (Pratama, 2015).

2.11.1 Throughput

Throughput adalah besaran jumlah *flow* pada setiap detik yang dapat ditangani *(flow/s)*. Tes *throughput* yang dilakukan dalam pengujian ini mempunyai peranan untuk menghitung jumlah transaksi yang dihasilkan dari waktu ke waktu tes atau menyatakan jumlah kapasitas aplikasi dalam menangani. Dimana *switch* yang diemulasikan oleh Cbench mengirimkan sebanyak-sebanyaknya paket data kepada kontoler, kemudian Cbench menghitung respon yang diberikan oleh kontroler sebagai balasan paket data yang telah dikirimkan oleh *switch.* Seperti halnya yang ditunjukkan pada **Gambar 2.11** berikut ini:

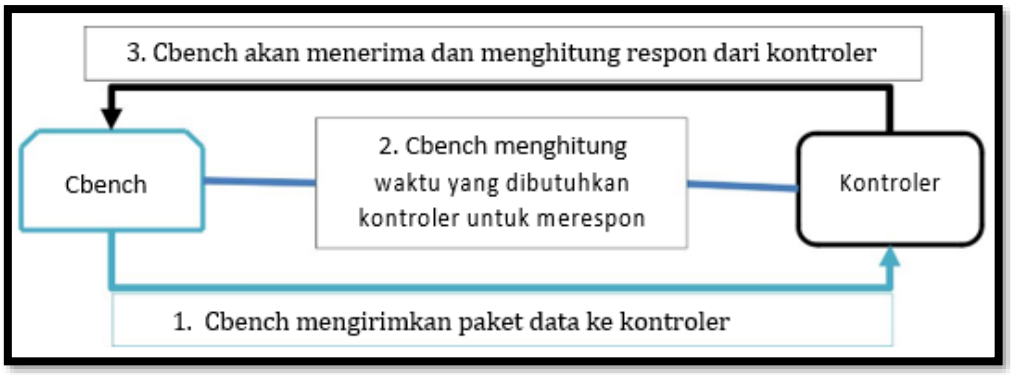


Gambar 2.11 Sistem kerja metode throughput

Sumber: Penulis

2.11.2 Latency

Latency adalah jumlah respon yang dapat diberikan oleh kontroler dalam tiap detiknya *(response/s)*. Matrik ini dapat diukur dalam tiga mode, yaitu *constant network load*, *incremental network load*, dan *stress network load.* Dimana pada *latency* ini, *switch* yang diemulasikan oleh Cbench mengirimkan paket tunggal kepada kontroler, kemudian Cbench menghitung waktu saat paket dikirimkan ke kontroler hingga paket akan kembali diterima oleh *switch.*



Gambar 2. 12 Sistem kerja metode latency

Sumber: Penulis