LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

PROGRAMMATŪRAS DEFINĒTAS TĪKLOŠANAS TEHNOLOĢIJAS DARBĪBAS PRINCIPI

BAKALAURA DARBS

Autors: **Kristaps Innuss**
Studenta apliecības Nr.: ki13011
Darba vadītājs: Docents Dr. dat Leo Trukšāns

RĪGA 2017

Anotācija

Līdz ar mākoņpakalpojumu sniedzēju un datortīklu skaita straujo pieaugumu, datortīklu prasības to projektēšanā un pārvaldībā kļūst aiz vien lielākas. Darbā tiek apskatītas programmatūras definētas tīklošanas darbības principi (SDN). Tas ir jauns tīkla dizaina princips, kura pamatdoma ir centralizēts kontrolieris kurš pārrauga tīkla darbību, kas tiek realizēts atdalot kontroles līmeni no datu līmeņa tradicionālajās tīkla iekārtās. Darbs sastāv no četrām nodaļām, kurās tiek apskatīta SDN daudzslāņu arhitektūra, iepazīti esošie SDN kontrolieru risinājumi, protokoli kādi tiek izmantoti un aprakstīts nepieciešamais minimums, lai izveidotu SDN tīklu mājas apstākļos.

Atslēgvārdi: programmatūras definētas tīklošana, OpenDaylight, OpenFlow, Netconf, BGP-LS, SDN.

Abstract

Workings of Software-Defined Networking technology

With the amount of Cloud Service providers increasing and networks growing every day, the demand for flexible networks is very high. This work describes workings of software-defined network technology (SDN). SDN is new network design that focuses on central controller that orchestrates network functionality, it is based on principle where the network device control plane is separated from its data plane. This work is divided in four chapters and it describes SDN multilayer architecture, different open source controllers, the protocols which enables SDN and describes what is necessary to begin designing and programming ones own SDN network.

Keywords: Software defined networking, OpenDaylight, OpenFlow, Netconf, BGP-LS, SDN.

SATURS

Apzīmējumu saraksts

SDN (Software Defined Networking) – Programmatūras Definēta Tīklošana

IPC- *inter-process communication* – starpprocesu komunikācija

Maršrutētājs/Komutators – šī darba ietvaros abi vārdi apzīmē vienu un to pašu - tīkla iekārtu, kurai ir plūsmu tabulas un kura var veikt pakešu manipulāciju un maršrutēšanu, ja vien nav speciāli norādīts savādāk.

Ievads

Līdz ar mākoņpakalpojumu sniedzēju un datortīklu skaita straujo pieaugumu, datortīklu prasības to projektēšanā un pārvaldībā kļūst aiz vien lielākas. Pašreiz izmantotās metodes tīklu dizainā nespēj nodrošināt pietiekoši lielu elastību. SDN (Software-defined networking) ir jauns tīkla dizaina princips . Šī jaunā tehnoloģija ir jāapzina un jāsaprot tās iespējas, jo SDN drīzumā varētu nomainīt tradicionālo tīkla arhitektūru sniedzot centralizētu tīkla pārvaldību, iespēju programmēt tīkla darbību un pašlaik tik nepieciešamo tīkla elastību. SDN pamatdoma ir – iespēja ar lietotņu palīdzību dinamiski programmēt individuālos tīkla elementus tādējādi kontrolējot kopējo tīkla darbību. Tā pamatā ir centralizēts kontrolieris kurš pārrauga tīkla darbību, kas tiek realizēts atdalot kontroles līmeni no datu līmeņa tradicionālajās tīkla iekārtās.

Darba izstrādē izvirzītais mērķis:

- Izpētīt SDN tīkla dizainu un tā darbības principus, protokolus kurus izmantoto SDN tīklos un kāda darbojas mūsdienīgi SDN kontrolieri.

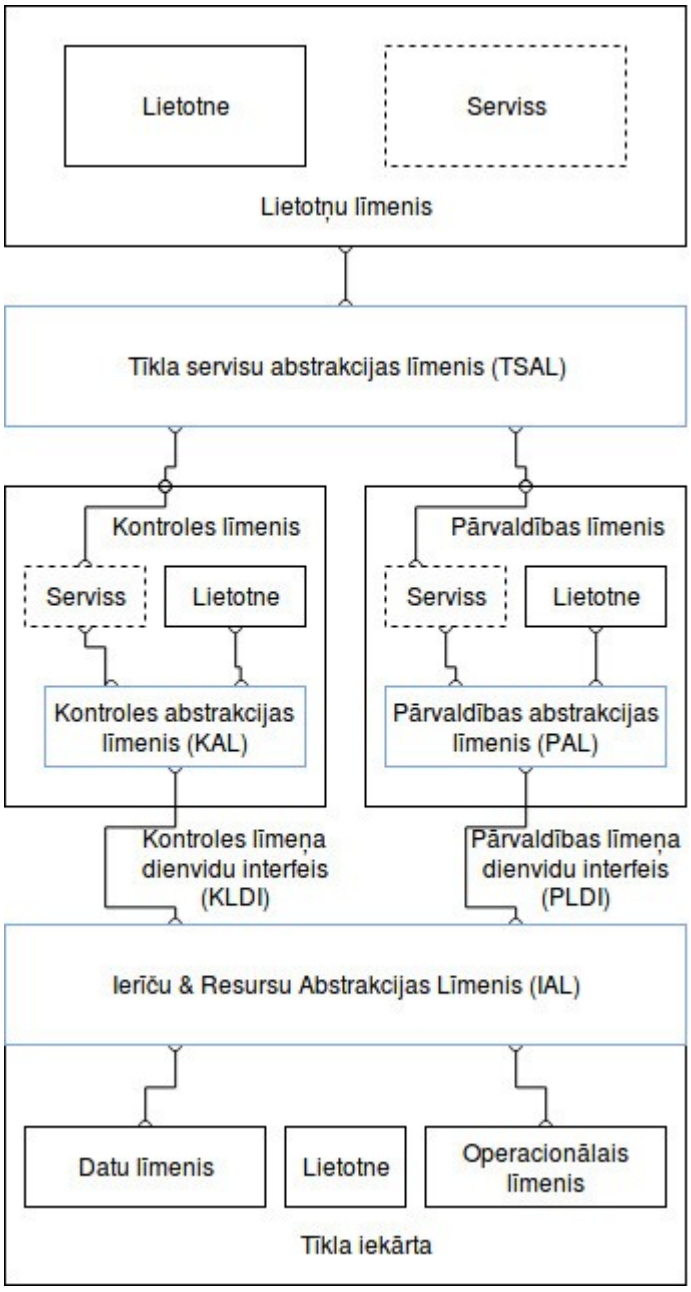
Mērķa realizēšanai izvirzītie uzdevumi:

1. Izpētīt pieejamo informāciju par SDN daudzslāņu arhitektūru,
2. Apskatīt pašlaik piedāvātās SDN tehnoloģijas,
3. Iepazīt tīkla protokolus kuri tiek izmantoti SDN tīkla realizēšanai,
4. Izveidot SDN tīkla prototipu.

Darbs sastāv no vairākām daļām: ievada, iztirzājuma un nobeiguma. Iztirzājumā nodaļas tiek apskatītas darba uzdevumu definēšanas secībā.

1. SDN dizaina pārskats

SDN daudzlīmeņu arhitektūras pamata koncepcija ir izveidot attiecības starp viesiem elementiem, kurās viens kontrolē otru. Kontrole notiek ar interfeisu palīdzību, ja abi elementi ir vienuviet, tad visbiežāk tas ir API pieprasījums, izmantojot sistēmas izsaukumus vai citas bibliotēkas. Kontrolei var izmantot arī kādu protokolu, kurš lokālā gadījumā varētu būt IPC (Inter-process communication) vai arī tādu protokolu, kuru var izmantot saziņai attālināti.



1.1.att. **SDN daudzlīmeņu arhitektūra** [1]

1.1. Attēlā ir apskatāms augsta līmeņa SDN daudzlīmeņu arhitektūras pārskats. Katrā konkrētajā implementācijā atsevišķie līmeņi var saplūst kopā vai arī būt fiziski nodalīti, bet pamatdoma nemainās. Tālāk tiek apskatīts katrs no arhitektūras elementiem atsevišķi, lai

sapratu kā katrs elements strādā, kāda tam ir nozīme un kā tas iederas kopējā SDN daudzlīmeņu arhitektūrā.

1.1. Tīkla iekārta

Tīkla iekārta ir elements, kurš tīkla portos saņem paketes un izpilda vienu vai vairākas darbības ar tām, piemēram: pārsūtīt paketi, izmest paketi, nomainīt paketes galvenes laukus un citas darbības. Tīkla iekārtas iekļauj: komutatorus, maršrutētājus un citus tīkla elementus, kuri varētu operēt virs vai zem IP līmeņa.

Tīkla iekārtas var tikt realizētas aparatūrā vai programmatūrā un tās var būt gan fiziskas gan virtuālas. Katra tīkla iekārta vienlaicīgi atrodas gan "Datu līmenī", gan "Operacionālajā" līmenī. Datu līmenis ir atbildīgs par pakešu apstrādi – komutēšanu, maršrutēšanu, pakešu transformācijām un filtrēšanu. Operacionālais līmenis ir atbildīgs par iekārtas stāvokli, piemēram, portu, interfeisu stāvokļiem, atmiņu, CPU u.c.

Datu un Operacionālais līmenis pārējiem līmeņiem ir pieejams caur "Ierīču un resursu abstrakcijas līmeni" (turpmāk IAL) un to var izteikt dažādi abstrakcijas modeļi. Piemēri Datu līmeņa abstrakcijas modeļiem: OpenFlow, Yang modeļi un SNMP MIBs. Piemēri Operacionālā līmeņa abstrakcijas modeļiem: YANG modeļi un SNMP MIBs.
Tīkla iekārtā var darboties arī lietotnes, kuras, piemēram, pārrauga topoloģijas atklāšanas funkcijas vai arī nodrošina ARP pašā iekārtā, tā vietā, lai pārsūtītu šādas paketes uz kontroles līmeni.

1.2. Kontroles līmenis

Kontroles līmenis parasti ir dalīts (*distributed*) un ir atbildīgs galvenokārt par Datu līmeņa pārvaldību izmantojot Kontroles līmeņa dienvidu interfeisu (turpmāk KLDI) kopā ar IAL. Kontroles līmenis ir atbildīgs par Datu līmeņa instruēšanu kā rīkoties ar tīkla paketēm. Komunikāciju starp vairākiem kontroles līmeņiem sauc par "austrumu-rietumu" interfeisu, to parasti implementē izmantojot kādu vārtejas protokolu, piemēram, BGP, vai arī citus protokolus, piemēram, *Path Computation Element Comunication Protocol* (PCEP). Ja vairāki Kontroles līmeņi neatrodas vienuviet tad komunikāciju starp tiem veic caur Datu līmeni.
Kontroles līmeņa funkcionalitāte:

- Topoloģijas atklāšana un uzraudzība,
- Pakešu ceļa izvēle,
- Iespēja izlabot kļūdas, ja pakešu ceļš tiek bojāts.

KLDI definē ar šādām īpašībām:

- Interfeis, kuram ir nepieciešamas mazas aiztures un reizēm liela caurlaidība, lai varētu izpildīt pēc iespējams vairāk operācijas mazākā laikā,
- Orientēts vairāk uz kopējo efektivitāti nevis uz vieglu lietojamību.

KLDI var implementēt izmantojot protokolu, API, vai IPC. Ja kontroles līmenis un tīkla iekārta neatrodas vienuviet, tad ir jāizmanto protokols, piemēram, OpenFlow.

Kontroles abstrakcijas līmenis (KAL) nodrošina vairāku KLDI piekļuvi pie Kontroles līmeņa lietotnēm un servisiem.

Kontroles līmeņa lietotnes izmanto KAL, lai kontrolētu tīkla iekārtu, bet tās nenodrošina resursu dalīšanu ar augstākiem līmeņiem, piemēram šādas lietotnes varētu realizēt tādu funkcionalitāti, kā OSPF, IS-IS vai BGP.

Kontroles līmeņa servisus var izmantot vairākas Kontroles līmeņa lietotnes un augstākie līmeņi.

1.3. Pārvaldības līmenis

Pārvaldības līmenis parasti ir centralizēts un nodrošina optimālu tīkla darbību, komunicējot ar tīkla iekārtu Operacionālo līmeni, izmantojot Pārvaldības līmeņa dienvidu interfeisu (PLDI) caur IAL.

Pārvaldības līmeņa darbības tradicionāli izpilda cilvēks, bet mūsdienās tās sāk aizstāt algoritmi, lai samazinātu nepieciešamību pēc cilvēka iejaukšanās. Pārvaldības līmeņa funkcionalitāte ietver:

- Kļūdu pārvaldība un tīklu pārraudzīšana,
- Konfigurāciju pārvaldība.

Papildus Pārvaldības līmeņa funkcionalitāte var ietvert virtuālo tīkla funkciju pārvaldniekus un virtualizēto infrastruktūru pārvaldniekus. Šādi tīkla elementi varētu izmanto Operacionālā līmeņa resursus, lai pieprasītu un izmantotu tos priekš virtuālām funkcijām.

Pārvaldības līmeņa dienvidu interfeisam (PLDI) atšķirībā no KLDI nav nepieciešams būt tik ātram, tāpēc tīkla aiztures nav tik būtiskas. Tipiski šo interfeisu izmantos vairāk cilvēki, tāpēc tipiskās PLDI īpašības ir:

- Orientēts uz vieglu lietojamību,
- Ziņas tiek pārsūtītas retāk nekā caur KLDI.

Veids kā PLDI var tikt implementēts var variēt starp API, IPC un protokoliem. Ja Pārvaldības līmenis nav iegults iekš tīkla iekārtas, tad var izmantot protokolus, piemēram, NETCONF, *IP Flow Information Export* (IPFIX), Syslog, *Open vSwitch Database* (OVSDB) vai SNMP.

Pārvaldības abstrakcijas līmenis (PAL) nodrošina visu PLDI piekļuvi pie pārvaldības lietotnēm un servisiem, jo Pārvaldības līmenis var atbalstīt vairākus PLDI.

Pārvaldības lietotnes var izmantot PAL, lai pārvaldītu tīkla iekārtas nenodrošinot dalīšanos ar servisiem, ar augšējiem līmeņiem.

Pārvaldības līmeņa servisi nodrošina servisus citiem servisiem vai lietotnēm augšējos līmeņos.

1.4. Tīkla servisu abstrakcijas līmenis (TSAL)

Tīkla servisu abstrakcijas līmenis (TSAL) nodrošina dalīšanos ar servisiem starp Kontroles līmeni, Pārvaldības līmeni un Lietotņu līmeni. Servisu interfeisi var būt dažādi, piemēram, RESTful API, atvērtie protokoli kā NETCONF, IPC, CORBA interfeisi utt. Pašlaik visvairāk tiek izmantoti RESTful interfeisi un *Remote Procedure Call* (RPC) interfeisi. Abiem interfeisiem ir klienta-servera arhitektūra un abi izmanto XML vai JSON, lai apmainītos ar ziņām.

RESTful interfeisi ir veidoti vadoties pēc *Restuf transfer* dizaina paradigmas (REST) un to īpašības ir:

- Resursu identificēšana – individuāli resursi ir identificējami izmantojot resursu identifikatorus,
- Resursu manipulēšana tos attēlojot, formātos kā, piemēram, XML. JSON vai HTML,
- Pašaprakstošas ziņas – katra ziņa ir pietiekami daudz informācijas, lai saņēmējs zinātu, kā šāda ziņa ir jāapstrādā,
- Klientam nav nepieciešama iepriekšējas priekšzināšanas, lai komunicētu ar serveri, jo API nav fiksēts un serveris to dinamiski nodrošina.

Remote procedure calls (RPCs) īpašības:

- Individuālas procedūras ir identificējamas, izmantojot identifikatoru,
- Klientam ir nepieciešams zināt procedūras nosaukumu un parametrus.

1.5. Lietotņu līmenis

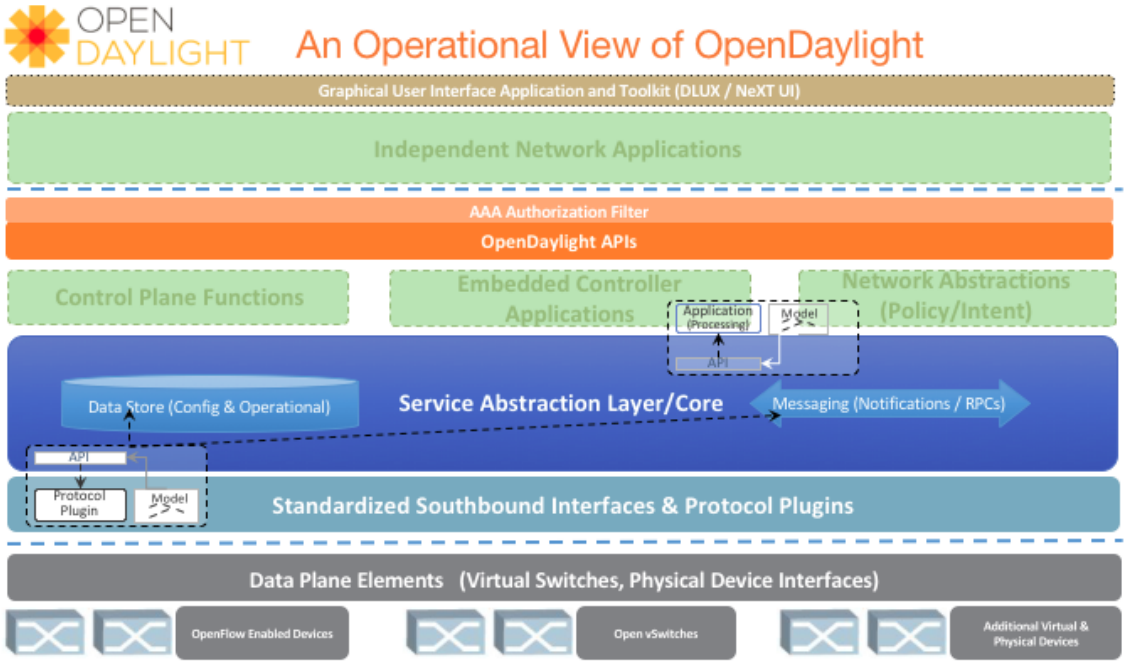
Lietotņu līmeni veido visas tās lietotnes un servisi, kuri Lietotņu līmenī izmanto servisus no Kontroles un/vai Pārvaldības līmeņa. Lietotņu līmeņa servisi var nodrošināt servisus citiem Lietotņu līmeņa servisiem vai lietotnēm, izmantojot servisu interfeisu. Dažādo lietotņu piemēri ir: tīkla topoloģijas atklāšana, tīkla pārvaldība utt.

2. Mūsdienu kontrolieri

Šajā nodaļā autors apskata mūsdienīgus SDN kontrolieru risinājumus, to arhitektūras un tajos izmantotās tehnoloģijas, meklējot raksturīgās iezīmes, kuras apraksta populārāko un nozīmīgāko protokolu izmantošanu ziemeļu un dienvidu interfeisiem, kopējos arhitektūras apsvērumus. Tiek apskatītas arī izmantotās izstrādes tehnoloģijas, tajā skaitā, programmēšanas valodu, programmēšanas ietvarus, un papildus tehnoloģijas koda bāzes pārraudzībai un izstrādei.

2.1. OpenDaylight

OpenDaylight (turpmāk ODL) ir atvērtā pirmkoda projekts, kurš ir izstrādāts programmēšanas valodā Java. Projekta arhitektūra tiek balstīta uz mikroservisu paradigmas, kur kopējā kontroliera funkcionalitāte tiek apkopota no daudzām mazākām programmu daļām, kur katra ir atbildīga tikai par sava galvenā uzdevuma izpildi [4]. Programmatūras modularitāti nodrošina *Apache Karaf* konteineris, tas nodrošina iespēju instalēt tikai to programmatūru un servisus kādi ir nepieciešami katrai individuālajai tīkla dizaina implementācijai [5]. Attēlā 2.1.1. var redzēt augsta līmeņa diagrammu par OpenDaylight kontroliera iekšējiem darbības principiem.



2.1.1.att. **OpenDaylight kontroliera pārskats** [3]

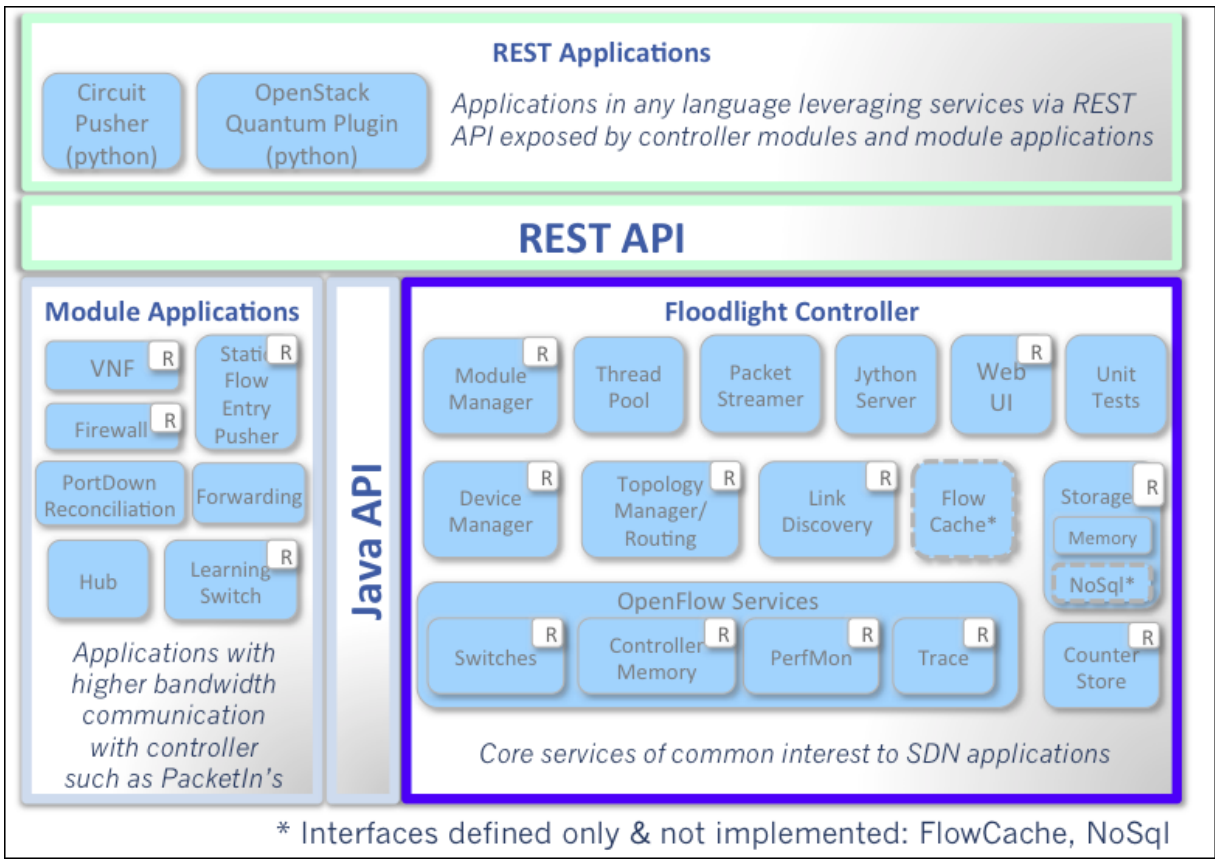
Lielāko darbu kontrolieris veic servisu abstrakcijas līmenī (*SAL*), kurā ir datu glabātuve, kur tiek glabāta kontroliera konfigurācija. SAL var ģenerēt kontroliera API priekš dienvidu un ziemeļu interfeisiem, izmantojot YANG modeļus, lai definētu vēlamo funkcionalitāti. ODL atbalsta ļoti plašu protokolu klāstu, tajā skaitā: *OpenFlow* un *OpenFlow*

paplašinājumus, *Netconf, BGP/PCEP* un *CAPWAP.* Kā arī ODL papildus var komunicēt ar *OpenStack un Open vSwitch* izmantojot OVSDB. ODL papildfunkcionalitāte, kuru var instalēt izmantojot *Karaf*:

- DLUX – statistika par tīklu, plūsmām, tīkla elementu atrašanās vietām
- Lietotņu līmeņa plūsmu optimizācija
- *Network embedded Experience* (NeXt) – rīks, ar kura palīdzību grafiski tiek attēlota tīkla topoloģija.0

2.2. Floodlight

Floodligth ir atvērtā pirmkoda SDN kontrolieris, kurš ir sarakstīts programmēšanas valodā Java. Kontroliera izstrādi atbalsta atvērtā pirmkoda brīvprātīgie programmētāji, to starpā vairāki programmētāji no konpānijas *Big Swith Network.* Floodlight kā galveno kontroles un pārvaldības protokolu izmanto OpenFlow. Attēlā 2.2.1. ir redzams Floodlight kontroliera arhitektūras pārskats.



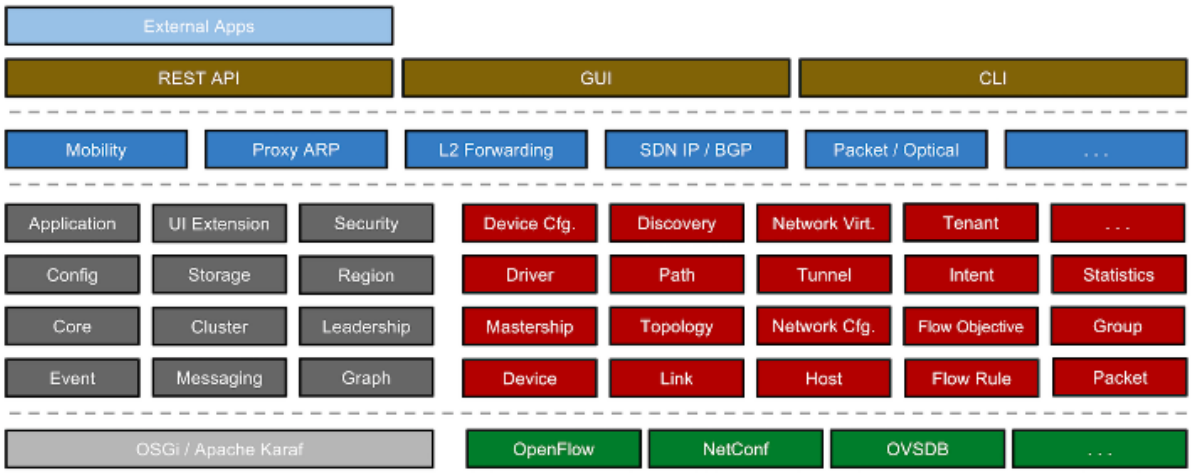
2.2.1.att. **Floodlight kontroliera pārskats** [6]

Arhitektūrā ir izmantots modulārs dizains un individuāla funkcionalitāte ir nodalīta atsevišķos moduļos. Kontroliera ziemeļu interfeis ir definēts kā REST API. Dienvidu interfeisā tiek izmantots pilns OpenFlow atbalsts – datu līmeņa plūsmas kā arī fiziskā vai virtuālās tīkla iekārtas Operacionālais līmenis tiek kontrolēts izmantojot OpenFlow protokolu. Priekš

kontroliera ir izveidots speciāls rīks *Loxigen* (atvērtā pirmkoda projekts, izstrādāts iekš python), ar kuru ir iespējams ģenerēt OpenFlow protokola ziņas, izmantojot dažādas programmēšanas valodas Java, C un python. Šajā gadījumā tiek izmantoti Java API, lai ģenerētu OpenFlow protokola ziņas, neatkarīgi no OpenFlow versijas. Ir vērts pieminēt, ka Floodlight izstrādātāji arī strādā pie OpenFlow aģenta (Datu/Operacionālā līmenī), ar nosaukumu – *Indigo,* kuru var izmantot aparatūras un programmatūras komutatoru (Layer 3) OpenFlow funkcionalitātes ieviešanai.

2.3. ONOS

Šajā kontroliera risinājumā ir iekļauts modulārs dizains, kur katrs modulis tiek pārvaldīts kā OSGi saišķis. Attēlā 2.3.1. ir iespējams apskatīt sistēmas pārskatu.



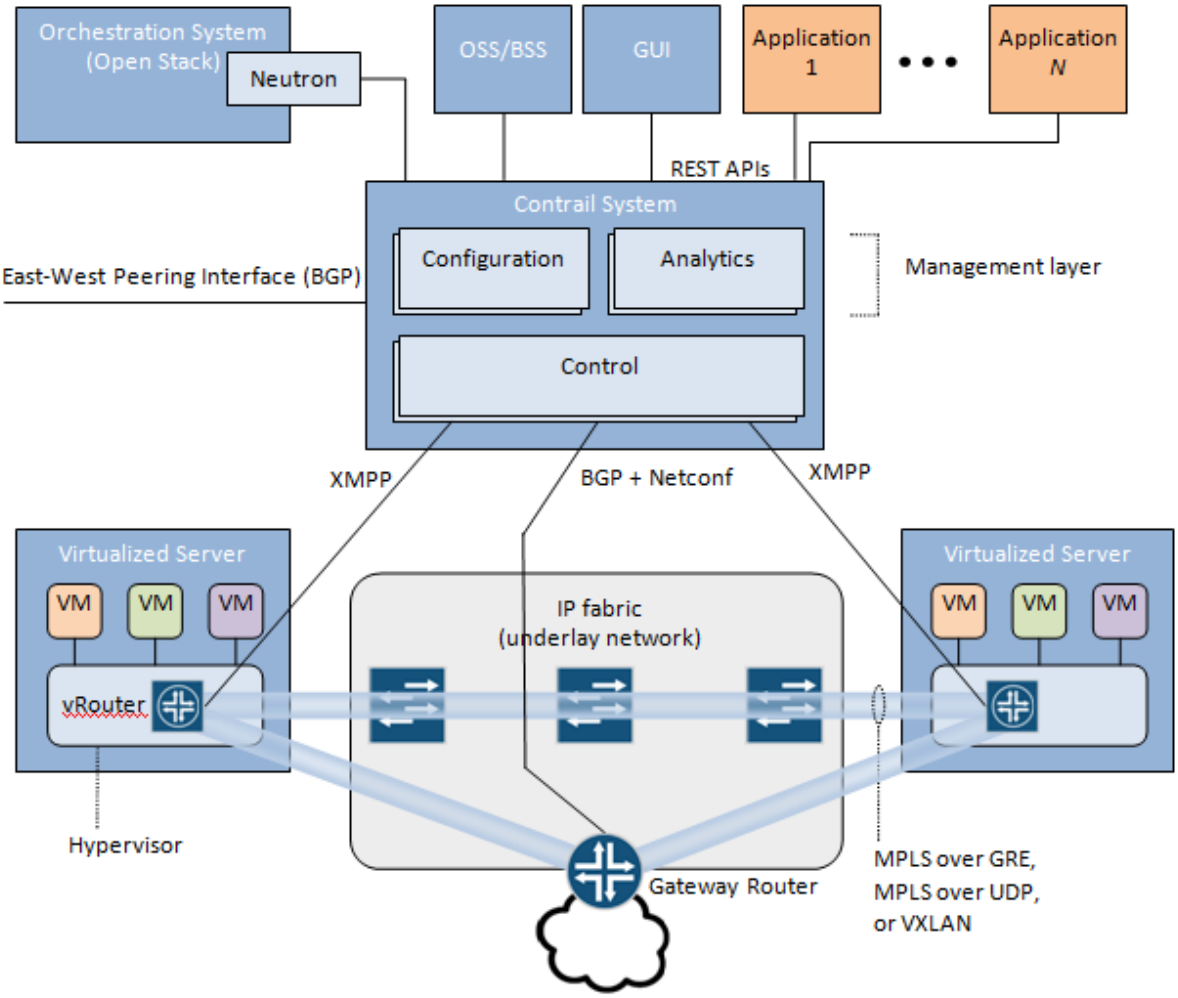
2.3.1.att. **ONOS modulārā sistēma** [10]

Modulāro dizainu pārvalda ar Apache Karaf iespējām, līdzīgi kā to dara ODL projektā. Projekts sastāv no vairākiem, mazākiem apakšprojektiem, kurus ir iespējams atsevišķi kompilēt un testēt, lai to paveiktu tiek izmantots *Apache Maven* sniegtās priekšrocības, izmantojot POM failus un projekta failu kārtošanu hierarhiski[8][9]. ONOS dienvidu interfeis nodrošina jaunu protokolu pievienošanu, izveidojot jaunu spraudni, bez nepieciešamības mainīt kodola funkcionalitāti.

Vēl interesants dizaina aspekts ir ONOS *Provider*, kurš nodrošina saziņu startp ONOS kodolu un tīkla iekārtu veidojot vēl vienu abstrakcijas līmeni. Kontroliera ziemeļu interfeisā tiek nodrošināts REST API, kā arī GUI un CLI.

2.4. Juniper Contrail Networking

Kontroliera risinājums, kurš ir galvenokārt paredzēts priekš mākoņpakalpojumu sniedzējiem un NFV ieviešanas interneta pakalpojumu sniedzēju zonā. Contrail Networking sistēma sastāv no divām daļām – Contrail Networking kontroliera un Contrail Networking vRouter (skat. 2.4.1. att.). Contrail Networking kontrolieris darbojas kā fiziski dalīta, bet loģiski vienota sistēma, kura izmanto vRouter – virtuāls datu līmenis, ar kura palīdzību var savienot virtuālās mašīnas.



2.4.1.att. **Contrail Networking sistēmas pārskats** [11]

Fiziski dalītie kontrolieru procesi savā starpā sazinās izmantojot "autrumu-rietumu" interfeisu, kas tiek realizēts ar BGP protokolu. Kā ziemeļu interfeisā tiek lietots REST API. Dienvidu interfeis pamatā atbalsta protokolus: XMPP, BGP un NETCONF, bet ar iespējām protokolu klāstu paplašināt, ja ir tāda nepieciešamība.

2.5. SDN kontrolieru apkopojums

Šajā nodaļā tika apskatīti četri SDN kontrolieri. Kontrolieru arhitektūrās, pirmkārt, dominēja modulārs dizains. Kā otra nozīmīga iezīme tika novērota nepieciešamība izmantot koda ģeneratorus, priekš dienvidu interfeisa, lai būtu iespējams ģenerēt protokolu neatkarīgi no tā versijas, definējot vēlamos modeļus caur kādu no modelēšanas valodām, piemēram, YANG. Dati par SDN kontrolieru izmantotajiem protokoliem un programmēšanas valodām tiek apkopoti 2.5.1. tabulā.

2.5.1. tabula

SDN kontrolieru iezīmju apkopojums

Nosaukums	**Programmēšanas val.**	**Ziemeļu interfeis**	**Dienvidu interfeis**
OpenDaylight	Java	REST API	OpenFlow, NETCONF, BGP, PCEP, CAPWAP, OVSDB
FloodLight	Java	REST API	OpenFlow
ONOS	Java	REST API	OpenFlow, OVSDB, BGP, NETCONF, TL1
Juniper Contrail Networking	C++, Python [12]	REST API	XMPP, OVSDB, BGP, NETCONF

Var novērot, ka dažādi kontrolieri atbalsta dažādus protokolus dienvidu interfeisos, bet lielākā daļa no tiem atbalsta: OpenFlow, OVSDB, BGP un NETCONF. Visi kontrolieri ziemeļu interfeisam nodrošina REST API, kurš arī lielākoties tiek ģenerēts, bet katrs no kontrolieriem izmanto dažādu REST API, kas varētu radīt problēmas priekš SDN Lietotņu līmeņa izstrādātājiem.

3. Tehnoloģijas kuras iespējo SDN

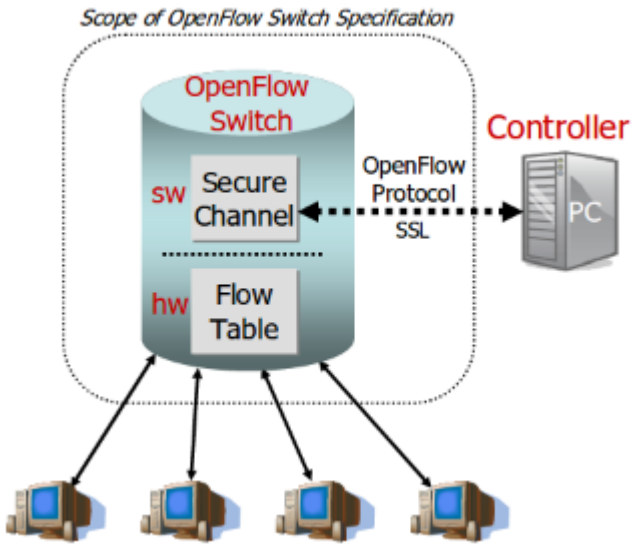
Šajā nodaļā tiek apskatīti trīs, pašlaik visizplatītākie protokoli, kuri tiek izmantoti, lai realizētu mūsdienīgus SDN kontrolierus. Protokoli tiek izvēlēti balstoties uz iepriekšējās nodaļas izpētes rezultātiem.

3.1. OpenFlow

OpenFlow ir protokols, kurš sākotnēji tika paredzēt tikai tīklu pētniekiem, lai varētu testēt jaunus tīkla protokolus un tehnoloģijas jau esošajos tīklos, veicot pēc iespējas mazākas tīkla infrastruktūras izmaiņas. Mūsdienās OpenFlow tiek izmantots vairāk nekā tikai pētnieku eksperimentos, bet kā arī viens no visvairāk izmantotajiem protokoliem mūsdienu SDN pasaulē. Šajā nodaļā apskatu OpenFlow darbības principus, tā kopējo nozīmi saistībā ar SDN un kā protokols darbojas OpenFlow atbalstītās tīkla iekārtās.

OpenFlow – iespēja izmantot eksperimentālus protokolus, ikdienas tīkla infrastruktūrā. OpenFlow pamatā ir Ethernet tīkla iekārta (šajā nodaļā tiek lietots apzīmējot komutatoru un maršrutētāju) ar iekšēju plūsmas tabulu (flow-table) un standartizētu interfeisu tabulas ierastu pievienošanai un dzēšanai.

Datortīklu nozīmība mūsdienās aug ar katru dienu un tehnoloģijas noveco, tāpēc radās nepieciešamība ieviest iespēju testēt un eksperimentēt ar tīkla iekārtām, jo problēmu sagādāja fakts, ka mūsdienu datortīkli sastāv no neskaitāmām iekārtām un protokoliem, un kur nav paredzēts testēt jaunus nepazīstamus protokolus un neviens to nav atļāvis.



3.1.1.att. **Inicializēta OpenFlow iekārta** [2]

Šādi apstākļi rada lielus šķēršļus jaunu ideju realizēšanai, jo praktiski nav iespējams jaunas idejas notestēt reālos dzīves apstākļos. Komerciālajos komutatoros un maršrutētājos esošā

operētājsistēma parasti nav atvērtā pirmkoda un vēl jo mazāk šāda veida iekārtas nodrošina iespēju virtualizēt tajās esošo aparatūru vai programmatūru. Parasti tīkla iekārtas ir slēgtas sistēmas, kuras nenodrošina ārēju interfeisu – vairāk kā pakešu pārsūtīšanu. Vēl jo vairāk, tīkla iekārtu ražotāji nav ieinteresēti, jaunu interfeisu atvēršanā un savas iekārtas iekšējo procesu atkailināšanu, visai pašsaprotamu iemeslu dēļ [2]. OpenFlow pamatā ir fakts, ka lielai daļai modernajiem *Ethernet* komutatoriem un maršrutētājiem ir plūsmu-tabulas, kuras strādā līnijas ātrumā (*line-rate*). Lai gan dažādiem izstrādātājiem ir dažādas plūsmu tabulas, ir daļa funkcionalitātes, kura ir vienāda lielākajai daļai. OpenFlow nodrošina atvērtu protokolu, lai varētu programmēt plūsmu tabulas tīkla iekārtās. *OpenFlow Switch* (komutators vai maršrutētājs, kurš izmanto OpenFlow protokolu) sastāv vismaz no trīs daļām:

1. Plūsmu tabulas – kur katram plūsmas ierakstam ir nozīmēta darbība, lai iekārta zinātu, ko ar katru plūsmu darīt,
2. Drošs kanāls, ar kuru var savienoties ar attālinātu kontroles procesu – kontrolieri, lai pārsūtītu kontroles ziņas un paketes,
3. OpenFlow protokols, kurš nodrošina standartizētu veidu kā kontrolierim sazināties ar tīkla iekārtu.

Ieviešot standartizētu interfeisu (OpenFlow protokolu), ar kuru var programmēt plūsmu tabulas attālināti, pazūd vajadzība programmēt pašu tīkla iekārtu. Attēlā 3.1.1. var redzēt inicializētu OpenFlow iekārtu, kurā ir plūsmu tabulas un tās tiek kontrolētas caur drošu kanālu izmantojot attālinātu kontrolieri.

OpenFlow iekārtas var iedalīt divās daļās. Pirmkārt, *OpenFlow-hybrid*, kuras pamatā nodrošina tajās jau esošu funkcionalitāti un to paplašina, lai varētu izmantot OpenFlow protokolu. Otrkārt, *OpenFlow-only*, kuras atbalsta tikai OpenFlow protokolu, tādā gadījumā, tas ir vienkāršs tīkla elements, kurš tikai pārsūta paketes starp tīkla portiem balstoties uz kontroliera definētām plūsmu tabulām.

Plūsmu tabulās katra plūsma var tikt definēta ļoti daudzos veidos, piemēram, var tik ņemtas vērā tikai MAC adreses, vai to kāds protokols tiek izmantots, ienākošā un izejošā IP adrese, VLAN ID un citi parametri, piemēram, tikai tās plūsmas, kurām ir nestandarta paketes galvene. 3.1.1. tabulā ir apskatītas kādas vērtības tiek glabātas plūsmu tabulās.

3.1.1. tabula

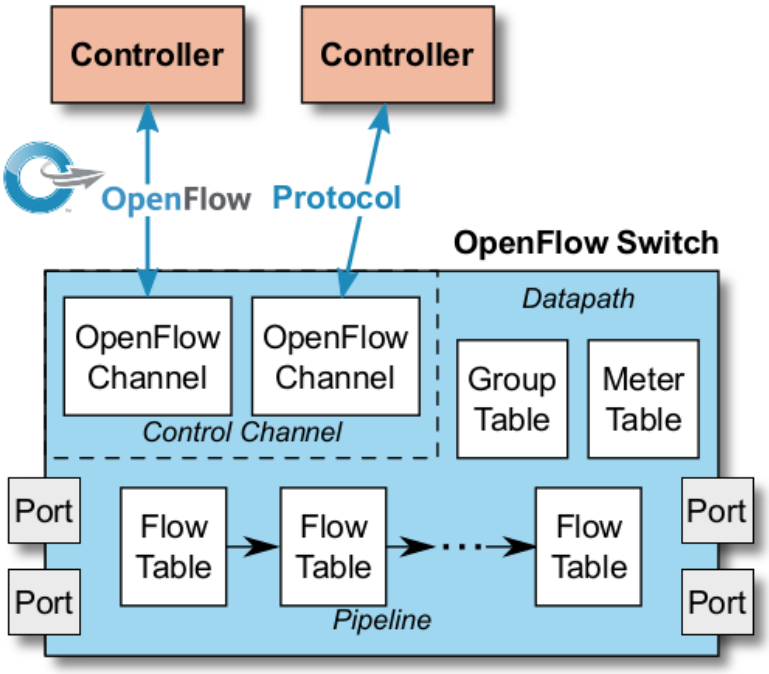
Plūsmu tabulu kolonnas

Pārbaudāmās vērtības	Prioritāte	Skaitītāji	Instrukcijas	Noildzes	*Cookie*	Karodziņi

Katra plūsmu tabulas rinda satur vērtības:

- pārbaudāmās vērtības – tās vērtības pēc kurām tiek atlasīta paketes. Vērtību veido ieejas ports un paketes galvene, papildus var būt arī citi metadati, ja tos ir uzstādījusi iepriekšējā plūsmas tabula (gadījumā, ja plūsmu tabulas ir vairākas un pakete no pirmās plūsmu tabulas tika pārsūtīta uz otro),
- prioritāte,
- skaitītāji – kad paketes apstrādā plūsmas tabulas rinda tad šajā rindā atjaunina skaitītāju,
- instrukcijas – instrukcijas kā rīkoties, kad pakete sakrīt ar tabulas rindu,
- noildzes – tabulas rindas maksimālais noildzes un bezdarba noildzes laiks, pēc kura tabulas rinda tiks dzēsta.
- cookie – *opaque* datu vērtība, kuru izvēlas kontrolieris. Kontrolieris var to izmantot, lai atlasītu plūsmu tabulu vērtības pēc plūsmu statistikas, modificēšanas un dzēšanas pieprasījumiem. Netiek izmantots, pakešu apstrādē.
- karodziņi – karodziņi maina veidu kā plūsmu ieraksti tiek pārvaldīti, piemēram, ja tiek uzstādīts karodziņš OFPFF_SEND_FLOW_REM, tad tiek izsaukta ziņa par plūsmas dzēšanu.

Attēlā 3.1.2 aplūkojamas galvenās OpenFlow iespējotas iekārtas iekšējās komponentes – plūsmu tabulas, grupu tabula, OpenFlow kanāls un skaitītāju tabula.



3.1.2.att. **Galvenās OpenFlow iekārtas komponentes**[13]

Neskaitot plūsmas tabulas, pakešu apstrādes procesā tiek izmantotas arī grupas un skaitītāju tabulas.

Grupas tabula ir vēl viens veids kā var definēt kādas darbības tiks veiktas ar paketēm, katrs grupas tabulas ieraksts sastāv no grupas identifikatora, grupas tipa, skaitītāja un darbību grupām (*Action buckets*), kur darbību grupas ir saraksts ar darbību grupām, kur katra darbību grupa satur vienu vai vairākas darbības. Izmantojot grupu tabulas ir iespējams definēt darbības ar paketi, lai nevajadzēt atkārtoti definēt darbības plūsmu tabulās, ja vairākām plūsmām būtu jāizpilda vienāda darbība. Plūsmu tabulas instrukcijas laukā var būt norāde uz citu plūsmu tabulu vai grupu. Viens no piemēriem, kāpēc grupu tabula ir noderīga: ja ir izveidoti vairāki plūsmu tabulu ieraksti, kuri norāda uz vienu grupu, ir iespējams izmanīt visu plūsmu darbību, kuras norāda uz grupas ierakstu, izmainot tikai grupas darbību. Vēl noderīga funkcionalitāte ir iespēja ar grupas palīdzību kopēt paketi un izsūtīt pa vairākiem portiem, ļaujot veikt pakešu multiraides (*multicast*) un apraides(*broadcast*) ziņojumus.

Skaitītāju tabulas ir iespējams saistīt ar plūsmu tabulu ierakstiem. Šajās tabulās tiek apkopots visu plūsmu apjoms, kuras ir sasaistītas ar skaitītāja tabulu. Skaitītāju tabulas ir iespējams izmantot, lai iekļautu papildus loģiku kā veikt pakešu apstrādi pēc plūsmu noslogotības.

Vadoties pēc pirmās nodaļas aprakstītā SDN dizaina OpenFlow nodrošina IAL priekš Datu līmeņa, kā arī tiek izmantos kā Kontroles līmeņa dienvidu interfeis. OF-CONFIG [14], kas ir OpenFlow protokols priekš iekārtu konfigurēšanas, kurš ir balstīts uz YANG modeļiem un var tikt izmantos kā IAL priekš Operacionālā līmeņa un definē NETCONF kā Pārvaldības līmeņa dienvidu interfeisu.

3.2. NETCONF

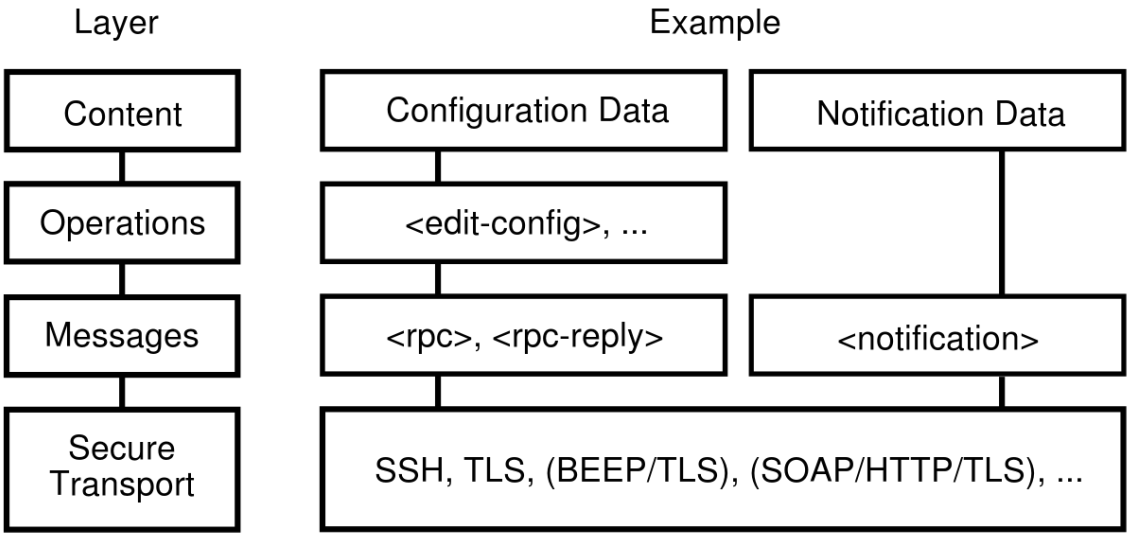
NETCONF ir tīkla pārraudzīšanas protokols, ko ir izstrādājis un standartizējis IETE. Šajā nodaļā tiks apskatīts NETCONF protokols, balstoties uz protokola labojumiem kuri tika veikti 2011. gada jūnija RFC6241[16]. Tiks apskatīta protokola darbības būtība un nozīme SDN risinājumos.

NETCONF nodrošina mehānismu kā instalēt, manipulēt un dzēst tīkla iekārtas konfigurāciju. Protokols tiek realizēts ar *Remote Procedure Call* (RPC) palīdzību. Priekš konfigurācijas datiem, kā arī protokola ziņām tiek izmantotaa XML [17] bāzēta datu datu kodēšana. Ziņojumi tiek sūtīti izmantojot drošu transporta protokolu, piemēram, ssh. NETCONF protokolu var konceptuāli iedalīt četros līmeņos (skatīt 3.2.1 att.):

- Drošā transporta līmenis - nodrošina komunikācijas ceļu starp klientu un serveri. NETCONF var tikt izmantot gandrīz ar jebkuru transporta protokolu, kurš var nodrošināt sesijas atpazīšanas tipu (klients vai serveris) un spēt to paziņot NETCONF

protokolam, transporta protokolam ir jānodrošina savienošanās iespējas, autehtifikācijas iespējas,

- Ziņu līmenis – nodrošina vienkāršu veidu kā kodēt RPC ziņas,
- Operācijas līmenis – definē kāda ir izpildāmā operācija. RPC metode ar XML kodētiem parametriem.
- Ziņojuma satura līmenis – satur konfigurāciju un paziņojumu datus.



3.2.1.att. **NETCONF protokola līmeņi**[15]

Ziņojumu saturs neattiecas uz NETCONF protokolu. Standartizēta datu modelēšanas valoda – YANG – ir izstrādāta, lai definētu NETCONF datu modeļus un protokola operācijas (divus augšējos līmeņus skatoties 3.1.2 att.) [19]. Pamatā tiek izmantots XML, bet nessen tika definēts veids kā var izmantot JSON, lai definētu datu modeļus ar YANG [18].

NETCONF ir vairākas pamata operācijas (skatīt 3.2.1 tabulu), kuras protokols atbalsta, bet šo kopu ir iespējams palielināt izmantojot YANG.

3.2.1.tabula

NETCONF operācijas

Operācija	Apraksts
<get>	Iegūt pašreizējo konfigurāciju un ierīces statusa informāciju.
<get-config>	Iegūst visu konfigurāciju, kura norādīta parametros.
<edit-config>	Izmaina norādīto konfigurāciju to apvienojot, apmainot, izveidojot vai dzēšot.
<copy-config>	Kopē konfigurāciju no vienas norādītās vietas uz otru.
<delete-config>	Izdzēš konfigurāciju.
<lock>	Aizslēdz konfigurāciju, neļaujot citiem to modificēt.

<unlock>	Atbrīvo slēgtu konfigurāciju.
<close-session>	Pieprasa standarta NETCONF sesijas pārtraukšanu.
<kill-session>	Pieprasa tūlītēju piespiedu NETCONF sesijas pārtraukšanu.

Var novērot, ka ir divu veida "get" operācijas, tās ir nepieciešamas, jo NETCONF protokols izšķir ierīces resursus divās daļās - konfigurācijas un ierīces stāvokļa dati, kur stāvokļa dati ir tie dati, kuri ir tikai lasāmi, un statistikas dati par ierīci. Vēl viena noderīga NETCONF funkcionalitāte ir iespēja slēgt konfigurācijas no visiem pārējiem NETCONF klientiem, kā arī tiem klientiem, kuri nav NETCONF, piemēram SNMP, CLI vai arī paša cilvēka, ļaujot netraucētu konfigurācijas izmaiņu veikšanu. NETCONF nodrošina arī pamata datu filtrēšanas funkcionalitāti, par to vairāk NETCONF specifikācijā RFC6241[16].

Kā jau iepriekš tika minēts, papildus funkcionalitāti var ieviest izmantojot YANG modeļus, bet lai klients varētu šāda veida funkcionalitāti izmantot, viņam par to ir jāzina. Tāpēc NETCONF paredz sākotnēju funkcionalitātes apmaiņas paziņojumu mehānismu, kad tiek uzsākta jauna NETCONF sesija. Serveris un klients izsūta funkcionalitāti un tās versijas numuru kāda viņiem pieejama, ja funkcionalitāte vai versija nesakrīt savienojums tiek nekavējoties pārtraukts. Šāda veida funkcionalitātes ziņojumi tiek aprakstīti ar <hello> elementa palīdzību.

```
<rpc message-id="94"
     xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <get-config>
    <source>
      <running/>
    </source>
    <filter type="subtree">
      <top xmlns="http://kristaps.com/schema/1.2/config">
        <users/>
      </top>
    </filter>
  </get-config>
</rpc>
```

3.2.2.att. **NETCONF konfigurācijas pieprasījums**

Attēlā 3.2.2. var redzēt NETCONF konfigurācijas pieprasījuma izsaukumu. Tajā tiek izmantota <get-config> operācija, un atlasīta tiek tikai pašreizējā konfigurācija ("running"), jo ierīce var atbalstīt vairākus konfigurāciju veidus, piemēram, *startup* – tā konfigurācija, kura tiek pielietota, ja ierīce tiek restartēta, vai arī pagaidu konfigurācija, kuru ir iespējams labot un veidot, nemainot pašreizējo konfigurāciju, un veikt pagaidu konfigurācijas kopēšanu uz

pašreizējo konfigurāciju tikai tad, kad visas vēlamās izmaiņas ir veiktas. Tālāk piemērā tiek izmantota filtrēšanas funkcionalitāte, lai atlasītu visus xml apakškokus no saknes *top,* kuru nosaukums ir *users*.

```
<rpc-reply message-id="101"
    xmlns="urn:ietf:params:xml:ns:netconf:base:1.0">
  <data>
    <top xmlns="http://example.com/schema/1.2/config">
      <users>
        <user>
          <name>root</name>
          <type>superuser</type>
          <full-name>Kristaps Innuss</full-name>
          <company-info>
            <dept>1</dept>
            <id>1</id>
          </company-info>
        </user>
      </users>
    </top>
  </data>
</rpc-reply>
```

3.2.3.att. **NETCONF konfigurācijas pieprasījuma atbilde**

Attēlā 3.2.3. ir redzama NETCONF pieprasījuma atbilde atsaucoties un iepriekš veikto pieprasījumu (skat. 3.2.2. att.).

Saistībā ar SDN NETCONF ir labs protokols, lai konfigurētu tīkla ierīces Operacionālo līmeni (atsaucoties uz pirmās nodaļas lietoto terminoloģiju). NETCONF ir piemērots, lai tiktu izmantots kā IAL priekš Datu līmeņa un Operacionālā līmeņa un ir izmantojams kā Pārvaldības līmeņa dienvidu interfeis. Šis protokols ir paredzēts ierīču konfigurācijai un tas nav veidots priekš ātru Kontroles līmeņa ziņu pārsūtīšanas, tāpēc tas nav piemērots priekš Kontroles līmeņa dienvidu interfeisa.

3.3. BGP-LS

Šī nodaļa apraksta BGP-LS (Border gateway protocol – link state) iespēju izmantošanu SDN kontrolieros priekš tīkla topoloģijas noskaidrošanas.

BGP-LS ir BGP protokola paplašinājums. BGP ir galvenais maršrutēšanas protokols internetā, tas nodrošina iespēju apmainīties ar maršrutēšanas informāciju starp dažādām autonomām sistēmām. BGP ir līdzīgs distances vektoru maršrutēšanas protokoliem, tādā nozīmē, ka maršrutētāji saņem informāciju no citiem maršrutētājiem un nosūta to tālāk. Tikai tā vietā, lai nosūtītu tālāk visus saņemtos ceļus, maršrutētājs veic sarežģītu aprēķinu

izmantojot iepriekš definētus noteikumus un lēmumu pieņemšanas procesu (Decision Process), lai izvēlētos tos ceļus, caur kuriem veikt maršrutēšanu un kurus nosūtīt tālāk citiem BGP maršrutētājiem.

BGP-LS ir veids kā iegūt tīkla topoloģiju, paplašinot BGP protokolu. Tiek ieviests jauns BGP NLRI (Network Layer Reachability Information) tips, kas būtībā ir vienkārši vairākas TLV (*Type Length Value*) kopas, kuras definē objektus:

- elementi - maršrutētāji
- ceļi – ceļi starp maršrutētājiem,
- IP prefiksi – ip apgabali kādi ir pieejami no katra maršrutētāja.

Izmantojot informāciju par tīkla elementiem un to ceļiem ir iespējams konstruēt tīkla topoloģijas grafu un informāciju par IP prefiksiem nosaka kuros tīkla elementos ir pieejamas attiecīgās tīkla adreses. Tiek definēti arī papildus atribūti, kuri definē katra objekta īpašības, piemēram, elementa vārdu, maršrutētāja ID, ceļa joslas platumu, pašreiz izmantoto joslas platumu, pieejamo joslas platumu utt. Ir vērts pieminēt, ka viens ceļa objekts definē tikai ceļa vienu "pusi", piemēram no A līdz B, lai droši noteiktu, ka savienojums no A līdz B eksistē vajag vēl vienu ceļa objektu kurā ir aprakstīts savienojums no B līdz A.

Jebkurš BGP maršrutētājs, kurš atbalsta BGP-LS protokolu var sūtītu tīkla topoloģijas informāciju serverim. Serverī šī informācija tiek apkopota datubāzē, kuru parasti apzīmē kā TED (Traffic Engineering Database).

Kopējās tīkla topoloģijas datubāzes izveide ir daudzkārt noderīga. Piemēram, lai varētu gudri izvēlēties katras datu plūsmas maršrutu no punkta A līdz B. Pašreiz interneta servisu piegādātāju tīklos, parasti tiek izmantots MPLS, lai maršrutētu paketes izmantojot speciālas pakešu birkas tā vietā, lai maršrutētu izmantojot parastos IP maršrutēšanas paņēmienus. Šo protokolu parasti izmanto kopā ar RSVP-TE (Resource Reservation Protocol – Traffic Engineering), tas sniedz iespēju katram maršrutētājam, kurš grasās uzsāk savienojumu, izvēlēties ceļu caur kuru tiks sūtītas paketes balstoties uz tīkla topoloģiju un daudziem kritērijiem, kurus parastas maršrutēšanas gadījumā neizmanto. Šāda veida maršrutēšana ir noderīga un sniedz daudz iespēju kā pārvaldīt tīkla plūsmas, bet tas neatrisina visas problēmas, jo ceļu izvēle notiek decentralizēti un kopīgā informācija nav pietiekama, lai vienmēr pieņemtu pareizo lēmumu cēla izvēlē. Situācija vēl jo vairāk pasliktinās, kad notiek maršrutēšana starp divām autonomām sistēmām – pat, ja ceļš katrā no autonomajām sistēmām (turpmāk AS) tika izvēlēts pats labākais, tas nenozīmē, ka kopējais ceļš ir pats labākais, jo ceļš tika izvēlēts balstoties tikai uz katras AS pieejamās informācijas. Problēma tiek risināta

izmantojot centralizētu kontrolieri, kuram ir pieejama visa nepieciešamā informācija par tīkla stāvokli un topoloģiju, lai varētu pieņemt visefektīvākos lēmumus katra ceļa izvēlē starp tīkla elementiem. Lai šāda veida kontrolieris varētu tīklu pārvaldīt ir nepieciešama TED un BGP-LS tieši to risina. Alternatīvi varianti vienas AS robežās ir izmantot IGP, lai iegūtu tīkla topoloģijas informāciju, bet šāds risinājums neatrisina vairāku AS gadījumu, tāpēc BGP-LS tiek izmantos daudzos SDN kontrolieru risinājumos.

4. SDN tīkla izveide mājas apstākļos

Šajā nodaļā ir aprakstīts veids kā izveidot SDN tīklu mājas apstākļos, izmantojot Mininet, lai izveidotu vēlamo tīkla topoloģiju ar tīkla iekārtām, un OpenDaylight (turpmāk ODL) kā tīkla kontrolieris kurš pārvalda tīkla darbību. Nodaļā tiek aprakstīti autora izmantotie rīki un izveidotās topoloģijas izveide, ODL sniegtās iespējas un tā ierobežojumi, kā arī nepieciešamais minimums ODL lietotņu izstrādei.

4.1. Tīkla prototīpēšana izmantojot Mininet

Šajā nodaļā tiek aprakstīts, kas ir Mininet lietojumprogramma un tās darbības pamata principi, jo Mininet ir ļoti noderīgs, lai sāktu projektēt pašam savus SDN tīklus, kā arī šī tehnoloģija tiek izmantota tālākajās nodaļās, tāpēc ir vērts saprast kādas iespējas sniedz tā sniedz [20].

Mininet ir lietojumprogramma, ar kuras palīdzību ir iespējams simulēt datortīklu ar virtuālām darbstacijām, komutatoriem, kontrolieriem un saitēm. Katra no darbstacijām nodrošina standarta Linux tīkla programmatūru un piedāvātie komutatori atbalsta OpenFlow protokolu, lai varētu veidot dažāda veida tīkla topoloģijas izmantojot SDN principus. Mininet nodrošina arī vienkāršu veidu ka veikt izveidotās tīkla topoloģijas testus, nodrošina komandrindas interfeisu, ar kuru ir iespējams novērot tīkla darbību un atkļūdot ar tīkla topoloģiju saistītās problēmas. Viena no noderīgākajām īpašībām ir tā, ka Mininet tīkls darbina "īstu" kodu – Unix/Linux tīkla lietotnes un Linux kodola un tīkla steku, tas sniedz lieliskas iespējas prototipēt tīkla topoloģijas un kontroliera lietojumprogrammas, lai pēc tam tās vienkārši varētu pārnest uz fiziskām iekārtām un reālas dzīves apstākļiem. Lai nodrošinātu šāda veida funkcionalitāte, Mininet izmanto procesu bāzētu virtualizāciju, lai izveidotu nepieciešamo tīkla topoloģiju izmantojot tikai vienu Linux kodolu. Mininet izmanto arī Linux atbalstītās tīkla virtualizācijas iespējas, kuras nodrošina katram procesam atsevišķus tīkla interfeisus, maršrutēšanas tabulu un ARP tabulu.

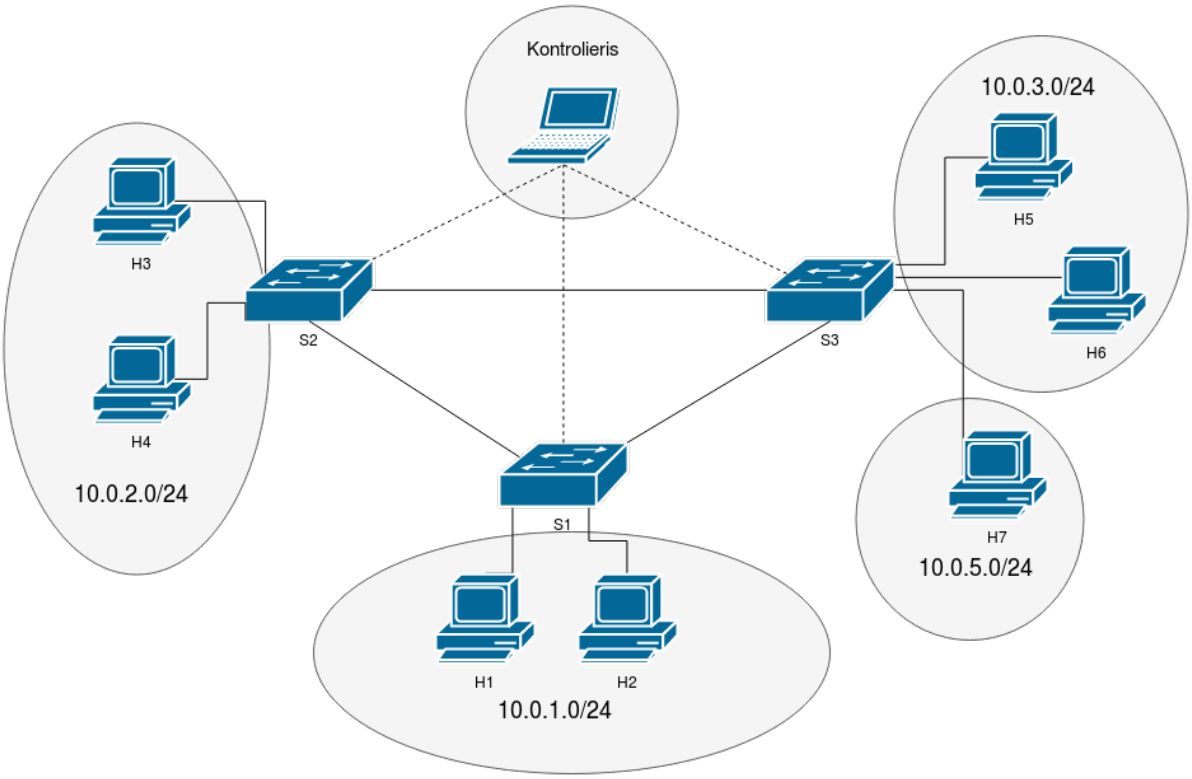
Mininet piedāvā vairākas iekļautās tīkla topoloģijas, kurām ir iespējams padot parametrus, lai pielāgotu vēlamo topoloģijas izveidi, bet arī ir iespējams definēt tīkla topoloģiju izmantojot Python valodā rakstītus skriptus. Kā arī kontrolierus ir iespējam izvēlēties no piedāvātajiem vai atsevišķi, definējot kontroliera atrašanās vietu pēc tīkla adreses un porta.

4.2. Tīkla uzdevums

Tika izveidots uzdevums, kurš sevī ietver SDN tīkla risinājuma izstrādi. Uzdevumā ir nepieciešams izveidot tīkla topoloģiju, no darbstacijām, komutatoriem un kontroliera.

Uzdevuma mērķis ir labāk izprast mūsdienīga SDN kontroliera sniegtās iespējas un apgūt SDN tīklu programmēšanas pamatus.

Uzdevumā tika definēta tīkla topoloģija, kura redzama 4.2.1.att. Tā sastāv no septiņām



4.2.1.att. **Definētā tīkla uzdevuma topoloģija**

darbstacijām un trīs Open vSwitch komutatoriem, kuri tiek kontrolēti izmantojot ODL kontrolieri. Tabulā 4.2.1. ir aprakstīts darbstaciju apakštīklu sadalījums.

4.2.1. tabula

Apakštīklu sadalījums

Darbstacija	**Apakštīkls**
H1, H2	10.0.1.0/24
H3, H4	10.0.2.0/24
H5, H6	10.0.3.0/24
H7	10.0.5.0/24

Tika izveidoti noteikumi kā tīklam vajadzētu funkcionēt:

- Katra apakštīkla darbstacijas var sazināties savā starpā,
- Darbstacijas no tīkla 10.0.1.0/24 var sazināties ar darbstacijām no tīkliem 10.0.3.0/24 un 10.0.5.0/24,

- Darbstacijas no tīkla 10.0.2.0/24 var sazināties ar darbstacijām no tīkliem 10.0.3.0/24 un 10.0.5.0/24,
- Darbstacijas no tīkla 10.0.1.0/24 nevar sazināties ar darbstacijām no tīkla 10.0.2.0/24,
- Darbstacijas no tīkla 10.0.3.0/24 nevar sazināties ar darbstacijām no tīkla 10.0.5.0/24.

4.3. Tīkla uzdevuma realizācija

Uzdevuma realizācijai kā kontrolieris, tika izvēlēts OpenDaylight (ODL), jo tas ir viens no visaktuālākajiem kontrolieru risinājumiem, kura izstrādē ir piedalījušies daudzi no lielo uzņēmumu programmētājiem un tas aktuāli tiek izstrādāts un uzturēts. Tīkla topoloģijas izveidei tika izmantots VirtualBox, uz kura tika uzstādīta Linux virtuālā mašīna, uz kuras ir instalēts Mininet. ODL tika instalēts uz datora ārpus virtualizētās vides. Lai nodrošinātu saziņu starp virtuālo mašīnu (Mininet tīklu) un kontrolieri tika izmantos VirtualBox iekšējais (*HostOnly*) tīkls.

Tīkla topoloģija tika definēta ar python skripta palīdzību (skat 4.3.1. att):

```
13  def birojaTikls():
14      net = Mininet(topo=None, build=False)
15      info( '*** Pievieno Kontrolieri\n' )
16      c1=net.addController(name='c1', controller=RemoteController,
17          ip='192.168.1.2', protocol='tcp', port=6633)
18      info( '*** Pievieno Komutatorus\n')
19
20      s1 = net.addSwitch('s1', cls=OVSKernelSwitch)
21      s2 = net.addSwitch('s2', cls=OVSKernelSwitch)
22      s3 = net.addSwitch('s3', cls=OVSKernelSwitch)
23
24      info( '*** Pievieno Datorus\n')
25      h1 = net.addHost('h1', cls=Host, ip='10.0.1.10',
26          mac='00:00:00:00:00:01', defaultRoute=None)
27      h2 = net.addHost('h2', cls=Host, ip='10.0.1.20',
28          mac='00:00:00:00:00:02', defaultRoute=None)
29      h3 = net.addHost('h3', cls=Host, ip='10.0.2.30',
30          mac='00:00:00:00:00:03', defaultRoute=None)
31      h4 = net.addHost('h4', cls=Host, ip='10.0.2.40',
32          mac='00:00:00:00:00:04', defaultRoute=None)
33      h5 = net.addHost('h5', cls=Host, ip='10.0.3.50',
34          mac='00:00:00:00:00:05', defaultRoute=None)
35      h6 = net.addHost('h6', cls=Host, ip='10.0.3.60',
36          mac='00:00:00:00:00:06', defaultRoute=None)
37      h7 = net.addHost('h7', cls=Host, ip='10.0.5.70',
38          mac='00:00:00:00:00:07', defaultRoute=None)
39
40      info( '*** Izveido Savienojumus\n')
41      net.addLink(s1, s2)
42      net.addLink(s1, s3)
43      net.addLink(s3, s2)
44      net.addLink(s1, h1)
45      net.addLink(s1, h2)
46      net.addLink(s2, h3)
47      net.addLink(s2, h4)
48      net.addLink(s3, h5)
49      net.addLink(s3, h6)
50      net.addLink(s3, h7)
51
```

4.3.1.att. **Funkcijas birojaTikls fragments**

Vadoties pēc uzdevuma nosacījumiem funkcija *birojaTikls* definē darbstacijas - to MAC, IP un nosaukumu, komutatorus un savienojumus starp darbstacijām un komutatoriem.

Tālāk tika realizēta tīkla funkcionalitāte. Darbojoties ar ODL REST API tika izveidoti divi varianti kā realizēt tīkla funkcionalitāti: viens no tiem ir definēt tīkla plūsmas vadoties pēc ieejas un izejas portiem un otrs risinājums ietver ODL iebūvēto tīkla komutēšanas lietotnes utilizēšanu.

Pirmajā risinājumā un arī otrajā tika izmantota lietojumprogramma *Postman,* lai sazinātos ar ODL REST interfeisu [21]. Pirmais risinājums ir balstīts uz darbstaciju un komutatora portu savienojumiem, tāpēc, lai labāk izprastu komutatora interfeisu sadalījumu pēc python skripta izpildes un tīkla realizācijas, komutatoru interfeisu sadalījums ir aprakstīts 4.3.1. tabulā.

4.3.1. tabula

Komutatoru interfeisu sadalījums

Komutatos	**Interfeis:savienots ar**
S1	1:S2 , 2:S3, 3:H1, 4:H2
S2	1:S1 , 2:S3, 3:H3, 4:H4
S3	1:S1 , 2:S2, 3:H5, 4:H6, 5:H7

Vadoties pēc funkcionalitātes prasībām tika izveidota *Postman* REST API izsaukumu kolekcija, kura kopumā definē 11 plūsmu ierakstu, ar kuru palīdzību tīkla funkcionalitāte tiek realizēta.

```
PUT /restconf/config/opendaylight-
inventory:nodes/node/openflow:1/flow-node-
inventory:table/0/flow/10 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: bb671d71-a533-69cd-58b6-13aed5b25132

<flow xmlns="urn:opendaylight:flow:inventory">
  <strict>false</strict>
  <flow-name>flow10</flow-name>
```

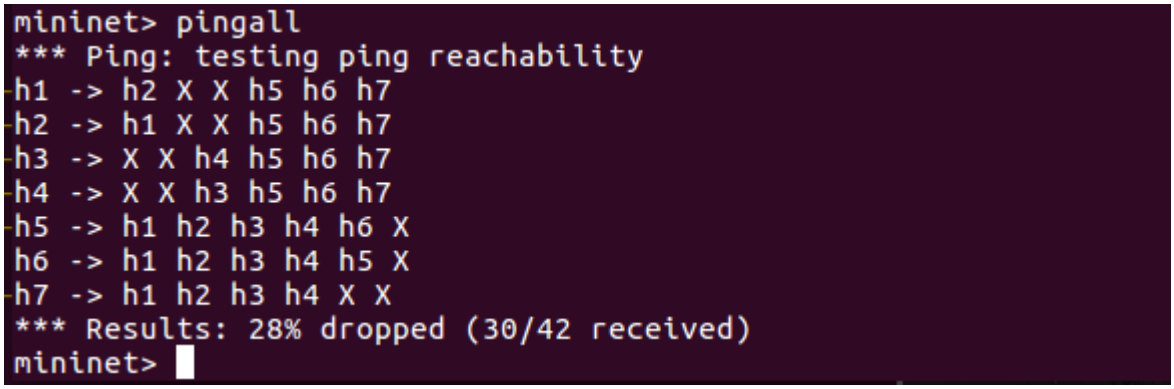
```
  <id>10</id>
  <cookie_mask>255</cookie_mask>
  <cookie>104</cookie>
  <table_id>0</table_id>
  <priority>0</priority>
  <hard-timeout>1200</hard-timeout>
  <idle-timeout>3400</idle-timeout>
  <installHw>false</installHw>
  <match>
    <in-port>3</in-port>
  </match>
  <instructions>
    <instruction>
    <order>0</order>
    <apply-actions>
      <action>
      <order>0</order>
      <output-action>
        <output-node-connector>4</output-node-connector>
        <max-length>60</max-length>
      </output-action>
      </action>
      <action>
      <order>1</order>
      <output-action>
        <output-node-connector>2</output-node-connector>
        <max-length>60</max-length>
...
```

4.3.2.att. **REST izsaukums komutatora plūsmas definēšanai**

Attēlā 4.3.2. var redzēt vienu no 11 plūsmu definēšanas izsaukumiem, kurā ar tiek definēta S1 nultās plūsmas tabulas plūsma ar identifikatoru 10. Aplūkojot vietu, uz kurieni tiek veikts pieprasījums: /restconf/config/opendaylight-inventory:nodes/node/openflow:1/flow-node-inventory:table/0/flow/10, var ievērot, ka visi augstāk minētie parametri tiek uzrādīti –

openflow:1 apzīmē S1, table/0 apzīmē nulto tabulu un flow/10 apzīmē plūsmu ar kuru vēlamies strādāt. Tālāk datu sadaļā tiek atkārtoti definēts plūsmas identifikators un piešķirts nosaukums. Svarīgākā datu daļa ir prioritātes, atlases kritēriju un instrukciju definēšana. Ar "<priority>0</priority>" tiek definēta plūsmas prioritāte, kura šajā gadījumā ir pati zemākā, ar "<match><in-port>3</in-port></match>" tiek pateikts, ka šī plūsma attiecas tikai uz paketēm kuras ir iesūtītas komutatora 3 portā, un ar "<instruction><action><output-action><output-node-connector>4</output-node-connector>..." tiek definētas atbildes reakcijas, kuras šajā gadījumā ir: izsūtīt paketi divos portos – ceturtajā un otrajā. Līdzīgi tiek definētas arī pārējās plūsmas (skat. 1.pielikums).

Izveidotā tīkla funkcionalitāte tika testēta izmantojot Mininet iebūvēto testēšanas rīku (skat. 4.3.3. att), kur h1-7 atbilst python funkcijā definētajiem iekārtu nosaukumiem. Tiek testēta katras iekārtas tīkla sasniedzamība ar visām pārējām tīkla iekārtām. Kreisajā pusē tiek norādīta kura iekārta tiek testēta, piemēram attēla trešajā rindā, kur sākas testa piemēri, tas ir H1, un labajā pusē tiek norādītas iekārtas kuras H1 var sasniegt, turpinot piemēru H1 var sasniegt H2, H5, H6, H7. 'X' darbstacijas nosaukuma vietā apzīmē, ka darbstacija netika sasniegta.



4.3.3.att. **Tīkla sasniedzamības testa rezultāti**

Testa rezultāti atbilst vēlamajai funkcionalitātei.

Neskatoties uz to, ka pirmais risinājums ir strādājošs, tas ir neefektīvs, jo pirmkārt plūsmas definē to, ka katra pakete tiek izsūtīta vairākos portos neatkarīgi no tā kuram no portiem tā īstenībā bija paredzēta, otrkārt plūsmu definēšana balstoties uz komutatora portiem ir neelastīgs risinājums, jo netiek ņemts vērā tīkla apakštīklu adrešu sadalījums. Tāpēc otrais risinājums mēģina šīs nepilnības labot. Tajā tiek izmantos ODL paplašinājums: l2switch, kurš simulē parasta *Layer 2* komutatora darbību automātiski ievietojot komutatoros plūsmas balstoties uz MAC-IP adrešu saistību, kuru l2switch uzzina tā darbības laikā. Ņemot vērā, ka nevienam no apakštīkliem nav iedalīts savs VLAN, tad izmantojot l2switch tiek panākts, ka visas darbstacijas var sasniegt visas pārējās darbstacijas, tas ir labi, bet tas nav tas ko pieprasa

uzdevuma nosacījumi. Šajā gadījumā, lai realizētu vajadzīgo funkcionalitāti tiek izmantota cita metode plūsmu definēšanā (skat. 4.3.4. att.).

```
PUT /restconf/config/opendaylight-
inventory:nodes/node/openflow:1/flow-node-
inventory:table/0/flow/100 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: 12173cb6-ea08-b799-1fb8-c2df5dba1699

<flow xmlns="urn:opendaylight:flow:inventory">
     <installHw>false</installHw>
     <instructions>
        <instruction>
            <order>0</order>
            <apply-actions>
                <action>
                    <order>0</order>
                    <drop-action/>
                </action>
            </apply-actions>
        </instruction>
    </instructions>
     <table_id>0</table_id>
     <id>100</id>
     <match>
          <ethernet-match>
               <ethernet-type>
                    <type>2048</type>
               </ethernet-type>
          </ethernet-match>
```

```
            <ipv4-source>10.0.1.0/24</ipv4-source>
            <ipv4-destination>10.0.2.0/24</ipv4-destination>
        </match>
        <hard-timeout>0</hard-timeout>
        <cookie>10</cookie>
        <idle-timeout>0</idle-timeout>
        <flow-name>flow100</flow-name>
        <priority>200</priority>
</flow>
```

4.3.4.att. **REST izsaukums plūsmas definēšanai**

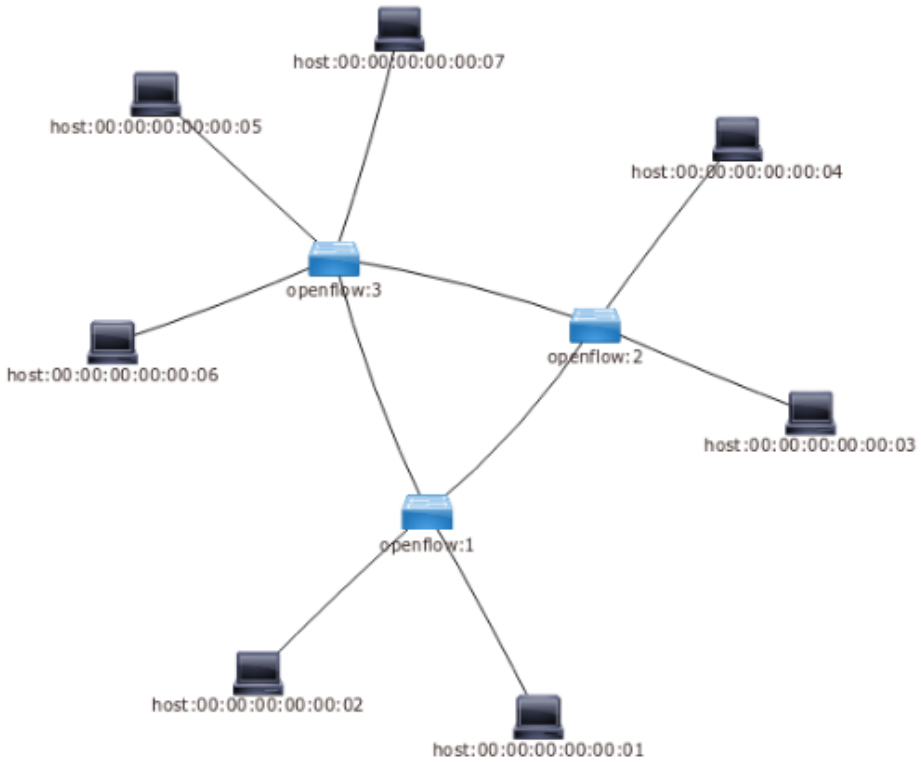
Otrajā risinājumā definētās plūsmas darbojas kā ugunsmūris. Šoreiz izvēles kritēriji tiek definēti pēc ip protokola, sūtītāja un saņēmēja ip adrešu diapazoniem. Plūsmas izsaukuma definēšanā "<ethernet-match><ethernet-type><type>2048</type>.." nozīmē, ka tiek atlasītas visas paketes, kuras izmanto IP protokolu, "<ipv4-source>10.0.1.0/24</ipv4-source>" nosaka sūtītāja ip adrešu diapazonu un "<ipv4-destination>10.0.2.0/24</ipv4-destination>" nosaka saņēmēja ip adrešu diapazonu. Šoreiz kā vēlamā darbība ir norādīta pakešu izmešana- "<action>..<drop-action/>..". Izmantojot šādas bloķējošās plūsmas ir iespējams izveidot nepieciešamo funkcionalitāti un to skaits ir krietni mazāks – tikai četras. Šeit netiek pieskaitītas plūsmas, kuras izveido l2switch, lai nodrošinātu vispārēju savienojamību, tāpēc ir vērts pievērst uzmanību arī plūsmu definēšanas prioritātei, kura šajā risinājumā ir 200, tas nodrošina to, ka šai plūsmai ir lielāka prioritāte pār citām plūsmām – citām plūsmām prioritāte ir 3. Lai aplūkotu pilno risinājumu skatīt 2.pielikumu.

4.4. Novērojumi un secinājumi

Pildot uzdevumu tika secināts, ka REST API, ko nodrošina ODL ir ļoti parocīgs, jo liela daļa ODL funkcionalitātes tiek piedāvāta izmantojot šo interfeisu. Neskatoties uz to, ka izmantojot REST API var izdarīt daudz, tas ir visai limitējošs. Risinot uzdevumu autors gribēja izmantot REST API, lai veiktu pakešu noklausīšanos un reaģētu uz attiecīgu pakešu saņemšanu, bet šāda funkcionalitāte nav atbalstīta šādā līmenī. Tāpēc, lai izveidotu lietotnes, kuras spētu nodrošināt šāda veida funkcionalitāti ir jāizmanto ODL Java API. Lai šo interfeisu izmantotu lietotne ir jāraksta iekš Java un tad tas vairāk ir kā ODL paplašinājums nevis atsevišķa lietotne. Tas samazina lietojamību, jo vēlamo risinājumu nevar veikt izmantojot programmēšanas valodu kuru katrs pārzina vislabāk, kā arī veidojot atsevišķu paplašinājumu priekš lietotnes tas varētu samazināt koda atkal izmantojamības iespējas. Neskatoties uz to,

autors pieļauj iespēju, ka izstrādājot ODL lietotni, kura izmanto Java API, lietotni būtu iespējams izmantot kā servisu citām lietotnēm, ģenerējot attiecīgo Java un REST API. Tādējādi iespējams apejot iepriekš aprakstītos ierobežojumus.

Otrkārt, ODL ir parocīgs grafiskais interfeis, ar kura palīdzību var apskatīt tīkla topoloģiju, piemēram, 4.4.1. attēlā var redzēt kā tīkla topoloģija izskatījās pēc 4.3. nodaļas otrā risinājuma realizācijas. Vēl viens noderīgs UI rīks ir iebūvēts REST API ziņojumu izveidotājs/sūtītājs, kurš, lai gan nav ļoti parocīgs, ir noderīgs, jo ar tā palīdzību ir iespējams uzzināt kā pareizi ir jāraksta REST izsaukumi.



4.4.1.att. **ODL DLUX paplašinājums**

Treškārt, visa ODL funkcionalitāte ir sadalīta atsevišķās pakotnēs, tāpat arī tikko aprakstītais grafiskais interfeis un REST API ziņojumu sūtītājs. Atsevišķās pakotnes kādas ir instalēts kopā ar ODL var pārraudzīt izmantojot Karaf un to pārvaldība ir samērā vienkārša. Karaf ir parocīgs rīks, lai gan uzdevumu izpildes laikā autors saskārās ar problēmu, ka pakotnes netika pareizi instalētas un nedarbojās kā vajag, lai gan nav skaidrs vai kļūda bija dēļ Karaf vai pašas pakotnes, atkārtota ODL lejupielāde problēmu atrisināja.

Secinājumi

Darbs ir veikts sekmīgi, izvirzītais darba mērķis ir sasniegts un uzdevumi ir izpildīti. Darba izveides laikā autors nonāca pie vairākiem secinājumiem:

- SDN daudzslāņu tīkla arhitektūra ir pārskatāma un labs veids kā sākt iepazīt SDN būtību. Pirmajā nodaļā aprakstītā arhitektūra atspoguļojas lielākoties visos SDN kontrolieros, kuri tika apskatīti šajā darbā. Vienīgais, kas tika pamanīts ir tas, ka Kontroles līmenis un Pārvaldības līmenis (atsaucoties uz 1. nodaļas terminoloģiju) daudzos kontrolieru risinājumos, ir saplūduši vienā līmenī un nav atšķirami,
- Dažādie kontrolieru risinājumi savā būtībā ir ļoti līdzīgi, tiem ir modulārs dizains, tie spēj nodrošināt plašu klāstu protokolus ko izmantot kā dienvidu uz ziemeļu interfeisu, kā arī izmantoto un atbalstīto protokolu daudzveidība ir salīdzinoši vienāda,
- SDN kontrolieros tiek realizēti daudzi protokoli, bet lielākoties tie savā starpā nepārklājas un katrs nodrošina jaunu funkcionalitāti,
- SDN tīkla dizaina principi ir daudz elastīgāki, nekā pašreiz esošais tīkla dizains, jo sniedz tīkla projektētājiem un administratoriem dažādas iespējas, lai iegūtu vēlamo tīkla funkcionalitāti. Par piemēru kalpo 4.3. nodaļā aprakstītie varianti kādi ir iespējami, lai nonāktu pie paredzētās tīkla funkcionalitātes, izmantojot savā būtībā atšķirīgus veidus kā tas tiek panākts,
- SDN tīklu realizēt mājas apstākļos ir samērā vienkārši, jo ir brīvi pieejama tāda virtualizācijas tehnoloģija kā Mininet un liela daļa mūsdienīgu kontrolieru ir atvērtā pirmkoda,
- Lai sāktu izstrādāt lietotnes priekš kāda no kontrolierim, labāk ir izvēlēties kontrolieri, kurš ir sarakstīts tādā programmēšanas valodā, kurā tu gribētu programmēt, jo liela daļa pieejamās funkcionalitātes, kuru var realizēt izmantojot tikai kontroliera ziemeļu interfeisu ir limitējoša un var nākties programmēt lietotni kā kontroliera paplašinājumu.

Izmantotā literatūra

1.Software-Defined Networking (SDN): Layers and Architecture Terminology. https://tools.ietf.org/html/rfc7426. [Skatīts 03.05.2017].

2. OpenFlow: Enabling Innovation in Campus Network. http://archive.openflow.org/documents/openflow-wp-latest.pdf. [Skatīts 06.05.2017].

3. ODL Platform Overview. https://www.opendaylight.org/platform-overview. [Skatīts 16.05.2017].

4. Microservices. https://en.wikipedia.org/wiki/Microservices. [Skatīts 16.05.2017].

5. Apache Karaf. http://karaf.apache.org/. [Skatīts 16.05.2017].

6. Floodlight documentation, The Controller. https://floodlight.atlassian.net/wiki/display/floodlightcontroller/The+Controller. [Skatīts 17.05.2017].

7. 2016 Network Virtualization and SDN Controllers Report. https://www.sdxcentral.com/reports/network-virtualization-sdn-controllers-2016/. [Skatīts 18.05.2017].

8. Apache Maven. https://maven.apache.org/. [Skatīts 10.05.2017].

9. ONOS Documentation. https://wiki.onosproject.org/display/ONOS/ONOS+%3A+An+Overview.[Skatīts 19.05.2017].

10. ONOS System. https://wiki.onosproject.org/display/ONOS/System+Components. [Skatīts 19.05.2017].

11. Contrail Networking Architecture Documentation. http://www.Contrail Networking.org/Contrail Networking-architecture-documentation/. [Skatīts 20.05.2017].

12. Github Contrail Networking. https://github.com/Juniper/contrail-controller. [Skatīts 20.05.2017].

13. OpenFlow 1.5.1 Switch Specification, https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-switch-v1.5.1.pdf. [Skatīts 22.05.2017].

14. OF-CONFIG 1.2 Specification. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow-config/of-config-1.2.pdf.[Skatīts 22.05.2017].

15. NETCONF protokola līmeņi. https://upload.wikimedia.org/wikipedia/commons/thumb/2/2f/NETCONF-layers.svg/1280px-NETCONF-layers.svg.png?1495108802528.[Skatīts 23.05.2017].

16. Network configuration protocol. https://tools.ietf.org/html/rfc6241.[Skatīts 23.05.2017].
17. Extensible Markup Language. http://www.wikiwand.com/en/XML.[Skatīts 23.05.2017].
18. JSON Encoding of Data Modeled with YANG. https://tools.ietf.org/html/rfc7951.[Skatīts 23.05.2017].
19. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). https://tools.ietf.org/html/rfc6020. [Skatīts 24.05.2017].
20. Mininet. http://mininet.org/. [Skatīts 24.05.2017].
21. Postman. https://www.getpostman.com/. [Skatīts 24.05.2017].

Pielikumi

1.Pielikums

Tīkla funkcionalitātei nepieciešamie REST izsaumumi priekš S1 un S3 pirmā risinājuma realizēšanai. S2 REST izsaukumi netiek iekļauti, jo tie ir ļoti līdzīgi S1 izsaukumiem, jo vienīgā atšķirība ir saņēmēja adresē.

PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:1/flow-node-inventory:table/0/flow/10 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: 5665a0f6-4f3e-8ff8-828b-f9352365a1af

<flow xmlns="urn:opendaylight:flow:inventory">
<strict>false</strict>
<flow-name>flow10</flow-name>
<id>10</id>
<cookie_mask>255</cookie_mask>
<cookie>104</cookie>
<table_id>0</table_id>
<priority>0</priority>
<hard-timeout>1200</hard-timeout>
<idle-timeout>3400</idle-timeout>
<installHw>false</installHw>
<match>
<in-port>3</in-port>
</match>
<instructions>
<instruction>
<order>0</order>
<apply-actions>
<action>

<order>0</order>
<output-action>
<output-node-connector>4</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
<action>
<order>1</order>
<output-action>
<output-node-connector>2</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
</apply-actions>
</instruction>
</instructions>
</flow>

1.1.att. **REST izsaukums, S1 konfigurācija**

PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:1/flow-node-inventory:table/0/flow/11 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: c2780ce4-e9fd-251a-da43-57febdeb5734

<flow xmlns="urn:opendaylight:flow:inventory">
<strict>false</strict>
<flow-name>flow11</flow-name>
<id>11</id>
<cookie_mask>255</cookie_mask>
<cookie>104</cookie>
<table_id>0</table_id>

<priority>0</priority>
<hard-timeout>1200</hard-timeout>
<idle-timeout>3400</idle-timeout>
<installHw>false</installHw>
<match>
<in-port>4</in-port>
</match>
<instructions>
<instruction>
<order>0</order>
<apply-actions>
<action>
<order>0</order>
<output-action>
<output-node-connector>3</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
<action>
<order>1</order>
<output-action>
<output-node-connector>2</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
</apply-actions>
</instruction>
</instructions>
</flow>

1.2.att. **REST izsaukums, S1 konfigurācija**

PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:1/flow-node-inventory:table/0/flow/12 HTTP/1.1
Host: localhost:8181

Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: a25085ff-3a64-0501-7d9d-89ca396bb7f6

<flow xmlns="urn:opendaylight:flow:inventory">
<strict>false</strict>
<flow-name>flow12</flow-name>
<id>12</id>
<cookie_mask>255</cookie_mask>
<cookie>104</cookie>
<table_id>0</table_id>
<priority>0</priority>
<hard-timeout>1200</hard-timeout>
<idle-timeout>3400</idle-timeout>
<installHw>false</installHw>
<match>
<in-port>2</in-port>
</match>
<instructions>
<instruction>
<order>0</order>
<apply-actions>
<action>
<order>0</order>
<output-action>
<output-node-connector>3</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
<action>
<order>1</order>
<output-action>
<output-node-connector>4</output-node-connector>

<max-length>60</max-length>
</output-action>
</action>
</apply-actions>
</instruction>
</instructions>
</flow>

1.3.att. **REST izsaukums, S1 konfigurācija**

PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:3/flow-node-inventory:table/0/flow/10 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: 30703a9e-4e6d-6ded-3a10-7de97777d6be

<flow xmlns="urn:opendaylight:flow:inventory">
<strict>false</strict>
<flow-name>flow10</flow-name>
<id>10</id>
<cookie_mask>255</cookie_mask>
<cookie>104</cookie>
<table_id>0</table_id>
<priority>0</priority>
<hard-timeout>1200</hard-timeout>
<idle-timeout>3400</idle-timeout>
<installHw>false</installHw>
<match>
<in-port>3</in-port>
</match>
<instructions>
<instruction>
<order>0</order>

<apply-actions>
<action>
<order>0</order>
<output-action>
<output-node-connector>4</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
<action>
<order>1</order>
<output-action>
<output-node-connector>2</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
<action>
<order>2</order>
<output-action>
<output-node-connector>1</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
</apply-actions>
</instruction>
</instructions>
</flow>

1.4.att. **REST izsaukums, S3 konfigurācija**

PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:3/flow-node-inventory:table/0/flow/11 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=

Postman-Token: 419d241b-6565-df0b-91e8-388fde90e629

<flow xmlns="urn:opendaylight:flow:inventory">
<strict>false</strict>
<flow-name>flow11</flow-name>
<id>11</id>
<cookie_mask>255</cookie_mask>
<cookie>104</cookie>
<table_id>0</table_id>
<priority>0</priority>
<hard-timeout>1200</hard-timeout>
<idle-timeout>3400</idle-timeout>
<installHw>false</installHw>
<match>
<in-port>4</in-port>
</match>
<instructions>
<instruction>
<order>0</order>
<apply-actions>
<action>
<order>0</order>
<output-action>
<output-node-connector>3</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
<action>
<order>1</order>
<output-action>
<output-node-connector>2</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>

<action>
<order>2</order>
<output-action>
<output-node-connector>1</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
</apply-actions>
</instruction>
</instructions>
</flow>

1.5.att. **REST izsaukums, S3 konfigurācija**

PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:3/flow-node-inventory:table/0/flow/12 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: 0ff1286c-e0c3-b15a-e8d1-ba9d0d182e53

<flow xmlns="urn:opendaylight:flow:inventory">
<strict>false</strict>
<flow-name>flow12</flow-name>
<id>12</id>
<cookie_mask>255</cookie_mask>
<cookie>104</cookie>
<table_id>0</table_id>
<priority>0</priority>
<hard-timeout>1200</hard-timeout>
<idle-timeout>3400</idle-timeout>
<installHw>false</installHw>
<match>
<in-port>2</in-port>

```
</match>
<instructions>
<instruction>
<order>0</order>
<apply-actions>
<action>
<order>0</order>
<output-action>
<output-node-connector>3</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
<action>
<order>1</order>
<output-action>
<output-node-connector>4</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
<action>
<order>2</order>
<output-action>
<output-node-connector>5</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
</apply-actions>
</instruction>
</instructions>
</flow>
```

1.6.att. **REST izsaukums, S3 konfigurācija**

```
PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:3/flow-node-
inventory:table/0/flow/13 HTTP/1.1
```

Host: localhost:8181

Content-Type: application/xml

Cache-Control: no-cache

Authorization: Basic YWRtaW46YWRtaW4=

Postman-Token: 279e303a-53de-e9b9-edff-42e6c6bb74d2

<flow xmlns="urn:opendaylight:flow:inventory">

<strict>false</strict>

<flow-name>flow13</flow-name>

<id>13</id>

<cookie_mask>255</cookie_mask>

<cookie>104</cookie>

<table_id>0</table_id>

<priority>0</priority>

<hard-timeout>1200</hard-timeout>

<idle-timeout>3400</idle-timeout>

<installHw>false</installHw>

<match>

<in-port>1</in-port>

</match>

<instructions>

<instruction>

<order>0</order>

<apply-actions>

<action>

<order>0</order>

<output-action>

<output-node-connector>3</output-node-connector>

<max-length>60</max-length>

</output-action>

</action>

<action>

<order>1</order>

<output-action>

<output-node-connector>4</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
<action>
<order>2</order>
<output-action>
<output-node-connector>5</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
</apply-actions>
</instruction>
</instructions>
</flow>

1.7.att. **REST izsaukums, S3 konfigurācija**

PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:3/flow-node-inventory:table/0/flow/14 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: 1acd62dd-9d63-a7df-97a0-2fd4d2d1173f

<flow xmlns="urn:opendaylight:flow:inventory">
<strict>false</strict>
<flow-name>flow14</flow-name>
<id>14</id>
<cookie_mask>255</cookie_mask>
<cookie>104</cookie>
<table_id>0</table_id>
<priority>0</priority>
<hard-timeout>1200</hard-timeout>

<idle-timeout>3400</idle-timeout>
<installHw>false</installHw>
<match>
<in-port>5</in-port>
</match>
<instructions>
<instruction>
<order>0</order>
<apply-actions>
<action>
<order>0</order>
<output-action>
<output-node-connector>1</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
<action>
<order>1</order>
<output-action>
<output-node-connector>2</output-node-connector>
<max-length>60</max-length>
</output-action>
</action>
</apply-actions>
</instruction>
</instructions>
</flow>

1.8.att. **REST izsaukums, S3 konfigurācija**

2.Pielikums

Tīkla funkcionalitātei nepieciešamie REST izsaumumi priekš otrā risinājuma realizēšanas.

PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:1/flow-node-inventory:table/0/flow/100 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: 84d7c934-a34a-6601-6433-552972298243

<flow xmlns="urn:opendaylight:flow:inventory">
<installHw>false</installHw>
<instructions>
<instruction>
<order>0</order>
<apply-actions>
<action>
<order>0</order>
<drop-action/>
</action>
</apply-actions>
</instruction>
</instructions>
<table_id>0</table_id>
<id>100</id>
<match>
<ethernet-match>
<ethernet-type>
<type>2048</type>
</ethernet-type>
</ethernet-match>
<ipv4-source>10.0.1.0/24</ipv4-source>

<ipv4-destination>10.0.2.0/24</ipv4-destination>
</match>
<hard-timeout>0</hard-timeout>
<cookie>10</cookie>
<idle-timeout>0</idle-timeout>
<flow-name>flow100</flow-name>
<priority>200</priority>
</flow>

2.1.att. **REST izsaukums, S1 konfigurācija**

PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:2/flow-node-inventory:table/0/flow/100 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: 482dc4fc-700d-f56d-50fe-82dcdaf9389c

<flow xmlns="urn:opendaylight:flow:inventory">
<installHw>false</installHw>
<instructions>
<instruction>
<order>0</order>
<apply-actions>
<action>
<order>0</order>
<drop-action/>
</action>
</apply-actions>
</instruction>
</instructions>
<table_id>0</table_id>
<id>100</id>
<match>

<ethernet-match>
<ethernet-type>
<type>2048</type>
</ethernet-type>
</ethernet-match>
<ipv4-source>10.0.2.0/24</ipv4-source>
<ipv4-destination>10.0.1.0/24</ipv4-destination>
</match>
<hard-timeout>0</hard-timeout>
<cookie>10</cookie>
<idle-timeout>0</idle-timeout>
<flow-name>flow100</flow-name>
<priority>200</priority>
</flow>

2.2.att. **REST izsaukums, S2 konfigurācija**

PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:3/flow-node-inventory:table/0/flow/100 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: 804e92e6-62f1-ac78-601e-ea430de7f230

<flow xmlns="urn:opendaylight:flow:inventory">
<installHw>false</installHw>
<instructions>
<instruction>
<order>0</order>
<apply-actions>
<action>
<order>0</order>
<drop-action/>
</action>

```
</apply-actions>
</instruction>
</instructions>
<table_id>0</table_id>
<id>100</id>
<match>
<ethernet-match>
<ethernet-type>
<type>2048</type>
</ethernet-type>
</ethernet-match>
<ipv4-source>10.0.3.0/24</ipv4-source>
<ipv4-destination>10.0.5.0/24</ipv4-destination>
</match>
<hard-timeout>0</hard-timeout>
<cookie>10</cookie>
<idle-timeout>0</idle-timeout>
<flow-name>flow100</flow-name>
<priority>200</priority>
</flow>
```

2.3.att. **REST izsaukums, S3 konfigurācija**

```
PUT /restconf/config/opendaylight-inventory:nodes/node/openflow:3/flow-node-
inventory:table/0/flow/101 HTTP/1.1
Host: localhost:8181
Content-Type: application/xml
Cache-Control: no-cache
Authorization: Basic YWRtaW46YWRtaW4=
Postman-Token: 0597e963-0900-56f3-61b8-4a821043296d

<flow xmlns="urn:opendaylight:flow:inventory">
<installHw>false</installHw>
<instructions>
<instruction>
```

<order>0</order>
<apply-actions>
<action>
<order>0</order>
<drop-action/>
</action>
</apply-actions>
</instruction>
</instructions>
<table_id>0</table_id>
<id>101</id>
<match>
<ethernet-match>
<ethernet-type>
<type>2048</type>
</ethernet-type>
</ethernet-match>
<ipv4-source>10.0.5.0/24</ipv4-source>
<ipv4-destination>10.0.3.0/24</ipv4-destination>
</match>
<hard-timeout>0</hard-timeout>
<cookie>10</cookie>
<idle-timeout>0</idle-timeout>
<flow-name>flow101</flow-name>
<priority>200</priority>
</flow>

2.4.att. **REST izsaukums, S3 konfigurācija**