UNIVERSITY OF ALBERTA

Project Report on

SDN Controller(OpenDaylight) Implementation of BGP Link State Distribution Layer-3 topology discovery

Under the guidance of:

Mr. Gurpreet Nanda

Prepared and Submitted by:

Barinder Singh

Acknowledgements

To begin with, I would like to thank Mr. Gurpreet Nanda who taught me for two semesters. Under his supervision, I learned the basic concepts of SDN and traditional networking which fascinated and encouraged me to take my capstone project on SDN. I feel so honored that he considered me capable of doing this project and allotted it on my name. He has been so helpful and without his guidance, completion of this project was not possible.

I would also like to thank Mr. Shahnawaz Mir. He provided me with resources and access to the computer lab so that I could implement my project. I am also thankful to Dr. Mike MacGregor for careful review of my project proposal and giving the approval to pursue my project.

Last but not the least, I would like to thank my colleagues for supporting and motivating in my tough times.

Abstract

Software-defined Networking (SDN) has risen as another worldview of networking that empowers network administrators, proprietors, vendors, and even third parties to improve and make new abilities at a quicker pace. The SDN worldview demonstrates the potential for all spaces of utilization, including service providers, organizations, homes and last but not the least, data centers.

With the rise in SDN, there been a noticeable enhancement of different networking protocols to utilize SDN efficiently. BGP-LS is one of the great extensions to achieve services provided in networking. For instance, traffic engineering, Quality of Service (QoS), Segment routing, etc.

This report provides a detailed explanation of how SDN is evolved from its existence to till date. It talks about basic techniques utilized for easier deployment of networks as compared to traditional networking by explaining architectural changes SDN had to go through. In addition to this, the report also gives the complete understanding of SDN operation.

Adding more to it, the report helps to understand the need of extending traditional networking protocols and applying those extensions on already deployed networks to have centralized control instead of distributed control. i.e. ease of installation, configuration, lower OPEX, and CapEx, etc.

Last but not the least, the report demonstrates how BGP-LS captures link-state attributes from previously deployed network underlying IGPs and forwards it to the centralized controller so that controller could have a complete view of the network and could control the network fully in a centralized manner. In simple words, it represents, the implementation of BGP-LS distribution Layer 3 topology of an existing network of an organization by utilizing SDN centralized approach with help of IOS-XR (supports OpenFlow) and OpenDayLight controller.

Table of contents

1. Introduction

1.1. The Border Gateway Protocol (BGP) routes traffic between autonomous systems. An autonomous system is a network or group of networks under a common administration and with common routing policies. BGP exchanges routing data for the Internet and are the protocol utilized between ISPs. Customer networks, for example, universities and corporations, typically utilize an Interior Gateway Protocol (IGP) for example, RIP or OSPF, to exchange routing data inside their networks. Customers associate with ISPs, and ISPs utilize BGP to exchange customer and ISP routes. At the point when BGP is utilized between autonomous systems, the protocol is referred to as external BGP (eBGP). If a service provider is utilizing BGP to exchange routes within an autonomous system, the protocol is referred to as interior BGP (iBGP).

BGP is an exceptionally robust and scalable routing protocol, as prove by the way that it is the routing protocol utilized on the Internet. To accomplish scalability at this level, BGP utilizes many route parameters, called attributes, to characterize routing policies and keep up a stable routing environment. BGP neighbors trade full routing data when the TCP connection between neighbors is initially settled. At the point when changes to the routing table are detected, the BGP routers send to their neighbors just those routes that have been changed. BGP routers don't send periodic routing updates, and BGP routing updates advertise just the optimal path to a destination network.

1.2. BGP-LS is an expansion to Border Gateway Protocol (BGP) for dispersing the system's connection state (LS) topology model to outer substances, for example, the SDN controller. The system's connection state topology display comprises of hubs and connections that interface these switches together. For each connection, an arrangement of traits is additionally contained. These may incorporate interface addresses, different measurements, and each connection's aggregate and accessible transfer speed. BGP-LS characterizes its own more theoretical topology demonstrate and characterizes how to outline IGP models, (for example, OSPF and ISIS) to its own model. As the system topology is found by the IGPs, the progressions are reflected in the BGP-LS demonstrate too and are likewise circulated utilizing BGP-LS messages to any invested individual, for example, SDN controllers. In situations where SDN applications require the system topology display, BGP-LS is more secure because it doesn't affect the system's steering as it is controlled by hidden IGP's and BGP has a broad arrangement instrument that can control and shield this data.

It works on basis of a mechanism which states that link state data can be gathered from systems and imparted to outside components utilizing the BGP routing protocol. They have characterized different TLV field in LS property called the node, link and prefix TLV attribute which are utilized to gather different data from the basic IGP, for example, link costs, metrics, interface IP addresses and so forth. At that point, this data is being imparted to the SDN controller which then has the total topological data about the whole network which is not the situation with IGP as it has data just about its own range. Thus, SDN controller has the total picture and it can perform Traffic-Engineering (TE) in a superior and productive manner than it would be without SDN.

1.3. The Software Defined Networking (SDN) development offers more prominent advantages to make networks more customized, efficient, application-centric, and programmable. There is a wide range of ways to build SDN. OpenFlow and VXLAN conventions are increasing more footing in data center situations, while Segment Routing and PCEP/BGP-LS are carriers' SDN technology option for obvious reasons.

2. Traditional Networking Approach Vs Software Defined Networking

Designing and managing networks have turned out to be more creative during recent years with the help of SDN (software-defined networking). This innovation appears to have shown up all of a sudden, however, it is quite of a long history of attempting to make computer networks more programmable.

Computer networks are mind boggling and hard to manage. They include numerous sorts of equipment from routers and switches to middle-boxes, for example, firewalls, network address translators, server load balancers, and intrusion-detection systems. Switches and routers run complex, distributed control software that is typically closed and proprietary. The software executes network protocols that experience years of standardization and interoperability testing. Network administrators configure individual network devices utilizing configuration interfaces that fluctuate amongst vendors and even between various products from the same vendor. Although some network-management tools offer a central vantage point for configuring the network, these systems still work at the level of individual protocols, components, and configuration interfaces. This method of operation has hindered advancement, expanded complexity, and swelled both the capital and the operational expenses of running a network.

Network design and maintenance approaches have been changed because of SDN. It basically works in two ways to differentiate SDN from traditional networking. First, the control plane (which resolves how to tackle traffic) is separated from data plane (which forwards traffic as per decisions made by control plane). Second, consolidation of control plane as a single software program in such a way that multiple data-plane elements can be handled. The SDN control plane has direct co-ordination with network's data-plane elements like routers, switches, middleboxes, etc. through a well-defined API, for example, OpenFlow. An OpenFlow switch can have more than one table of packet handling rules which is matched with a subset of traffic to perform certain actions on the traffic that matches a rule. It can perform actions like flooding, dropping or forwarding the packets. An OpenFlow switch can act as a firewall, network address translator, router or something in the middle which depends upon the rules installed by controller application.

SDN has increased huge footing in the Information technology (IT) industry. Numerous commercial switches support the OpenFlow API. The big names of IT industry like HP, NEC, and Pronto was among the principal vendors to support OpenFlow. This rundown has since expanded significantly. A wide range of controller platforms has risen. Developers have utilized these platforms to make plenty of applications, for example, server load balancing, energy-efficient networking, dynamic access control, network virtualization, and seamless virtual machine migration and user mobility. Early commercial triumphs, for example, Google's wide-area traffic-management system and Nicira's Network Virtualization Platform, have collected noteworthy industry consideration. A hefty portion of the world's biggest information-technology companies (e.g., equipment vendors, carriers, cloud providers, and financial services firms) have joined SDN industry groups, for example, the Open Daylight initiative and the Open Networking Foundation.

Although the excitement about SDN has turned out to be more tangible recently, large portions of the thoughts underlying the technology have developed during last 20 years or more. In some ways, SDN returns to thoughts from early telephony networks, which utilized a reasonable partition of control and data planes to simplify network management and the deployment of new services. However, open interfaces, for example, OpenFlow empower more innovation in controller platforms and applications than was conceivable on closed networks intended for a restricted scope of telephony services. In different ways, SDN looks like past research on active networking, which explained a vision for programmable networks, though with an emphasis on programmable data planes. SDN likewise identifies with past work on isolating the control and data planes in computer networks.

3. Evolution of Software Defined Networking (SDN)

The history of SDN started 20 years prior, similarly as the Internet was taking off when the Internet's stunning achievement exacerbated the difficulties of managing and advancing the network infrastructure. The emphasis here is on advancements in the networking community (regardless of whether by organizations, researchers, or standards bodies), although these developments in some cases were catalyzed by progress in different zones, including operating systems, distributed systems, and programming languages. The endeavors to make a programmable network infrastructure likewise obviously relate with the long string of work on supporting programmable packet processing at high speeds.

Making computer networks more programmable makes advancement in network administration conceivable and brings down the obstruction to conveying new services. On basis of this principle, the history of SDN can be divided into three segments each with its own contributions.

3.1. Active networks (the mid-1990s to the early 2000s)

- presented programmable functions in the network, inciting to great innovation

3.2. Control and data plane separation (around 2001 to 2007)

- established open interfaces between the control and data planes

3.3. The OpenFlow API and network operating systems (2007 to around 2010),

- denoted the first extensive acceptance of an open interface and developed new approaches to make control and data plane separation scalable and practical.

Apart from above three segments, Network virtualization also considered as a critical part all through the historical advancement of SDN, considerably originating before SDN yet flourishing as one of the principal noteworthy use cases for SDN.

3.1 Active Networking

The early to mid-1990s saw the Internet take off, with applications and appeal that far outpaced the early applications of file transfer and e-mail for scientists. More diverse applications and greater use by the public drew researchers who were eager to test and deploy new ideas for enhancing network services. To do as such, researchers designed and tested new network protocols in little lab settings and simulated behavior on larger networks. Most of the researchers were frustrated because approval by IETF (Internet Engineering Task Force) for their ideas was taking more than anticipated time even if they were motivated and funded by different organizations.

As they were eager to innovate and did not want to wait for longer time. Some networking researchers sought an alternative approach to opening network control, generally based on the analogy of reprogramming a stand-alone PC without breaking a sweat. Conventional networks are not programmable in any meaningful feeling of the word. Active networking denoted a fundamental approach to network control by imagining a programming interface (or network API) that exposed resources (e.g., storage, packet queues and processing) on individual network nodes and upheld the development of custom functionality to apply to a subset of packets passing through the node.

This approach was an abomination to numerous in the Internet community who pushed that straightforwardness in the network core was basic to Internet achievement. The active networks examine program explored radical contrasting options to the services provided by the traditional Internet stack by means of IP or ATM (Asynchronous Transfer Mode), the other predominant networking methodology of the mid-1990s. In this sense, active networking was the first in a progression of fresh start ways to deal with network architecture, therefore, sought after in projects like GENI (Global Environment for Network Innovations) and NSF FIND (Future Internet Design) in the United States, and EU FIRE (Future Internet Research and Experimentation Initiative) in the European Union.

The active networking community followed two programming models:

- The capsule model, where the code to execute at the nodes was stored in-band in data packets.

- The programmable router/switch model, where the code to execute at the nodes was not stored in-band instead it was established by out-of-band mechanisms.

The capsule model came to be most nearly connected with active networking. In logical connection to consequent endeavors, however, both models have an enduring legacy. Capsules proposed installation of new data-plane functionality over a network, carrying code in data packets (as in prior work on packet radio) and utilizing caching to enhance the throughput of code dispersion. Programmable routers set choices about extensibility specifically in the hands of the network administrator.

3.1.1. Technology push and use pull

Active networking approach incorporated a lessening in the cost of computing, permitting more processing in the network; propels in programming languages, for example, Java that offered platform portability and some code execution safety; and virtual machine technology that secured the host machine (for this situation the active node) and different procedures from misbehaving programs. Some active networking research projects also provided fast code compilation and formal techniques.

An essential push in the active networking environment was funding agency interest, specifically the Active Networks program made and supported by DARPA (U.S. Resistance Advanced Research Projects Agency) from the mid-1990s into the early-2000s. Even though not all research into active networks was financed by DARPA, the funding program supported a collection of tasks and, maybe more vital, supported convergence on a terminology and set of active network segments with the goal that ventures could add to an entire intended to be more noteworthy than the sum of the parts. The Active Networks program underscored demonstrations and project interoperability, with a corresponding level of improvement exertion. The striking and purposeful push from a funding agency without near-term use cases may have likewise added to a level of community disbelief about active networking that was regularly sound however could verge on hostility, and it might have darkened a portion of the intellectual connections between that work and later efforts to give network programmability.

The use pulls for this technology was mainly the issues like network service provider dissatisfaction with the time expected to create and deploy new network services (tagged as network ossification), third-party interest for value-added, fine-grained control to dynamically address the issues of specific applications or network conditions and researcher desire for a platform that would bolster experimentation at scale. Additionally, numerous early papers on active networking referred to the proliferation of middleboxes, including firewalls, proxies, and transcoders, each of which had to be deployed independently and involved a distinct (regularly vendor-specific) programming model. Active networking offered an idea of bound together control over these middleboxes that could eventually supplant the ad hoc, one-off ways to deal with managing and controlling these boxes. Adding more to it, NFV (network functions virtualization) likewise intends to give a unifying control structure to networks that have complex middlebox functions deployed all through.

3.1.2. Logical Contribution of Active Networking to SDN

Active networks presented intellectual contributions that relate to SDN. Important contributions are described below:

- **Network virtualization and the ability to demultiplex to software programs based on packet headers:**
  The need to reinforce testing with different programming models prompted to take a shot at network virtualization. Active networking delivered a structural system that portrays the parts of such a platform. The key parts of this platform are a common NodeOS (node operating system) that oversees shared resources, an arrangement of EEs (execution environments), each of which characterizes a virtual machine for packet operations and an arrangement of AAs (active applications) that work inside an offered EE to give an end-to-end service. Guiding packets to a specific EE depends on a quick pattern identical on header fields and demultiplexing to the appropriate EE. Curiously, this model was conveyed forward in the PlanetLab design, whereby diverse investigations keep running in virtual execution environments and packets are demultiplexed into the suitable execution environment on their packet headers. Demultiplexing packets into various virtual execution environments have additionally been connected to the outline of virtualized programmable hardware data planes.
- **The vision of an integrated architecture for middlebox orchestration:**
  Even though the vision was never completely acknowledged in the active networking research program, early outline reports referred to the requirement for unifying the extensive variety of middlebox capacities with a common, safe programming structure. Even though this vision might not have straightforwardly impacted the later work on NFV, different lessons from active networking research may demonstrate valuable as the application of SDN-based control and arrangement of middleboxes advances.
- **Programmable functions in the network that subordinate the barricade to revolution:**
  Study of active networks spearheaded the thought of programmable networks as a method for bringing down the hindrance to network advancement. The thought that it is hard to improve in a production system and supplications for expanded programmability were normally referred to initial inspiration for SDN. A great part of the early vision for SDN concentrated on control-plane programmability, while active networks

concentrated more on data plane programmability. All things considered, data plane programmability has kept on developing in parallel with control-plane efforts and data plane programmability is again going to the cutting edge in the rising NFV activity. Recent work on SDN is investigating the development of SDN protocols, for example, OpenFlow to support a more extensive scope of data-plane functions. Likewise, the ideas of detachment of experimental traffic from normal traffic which have their underlying foundations in active networking and seem up front in configuration archives for OpenFlow and other SDN technologies (e.g., FlowVisor).

3.1.3 Myths about Active Networking

Active networking incorporated the thought that a network API would be accessible to end users who send and receive packets, however, most in the research community completely perceived that end-users network programmers would be uncommon. The confusion that packets would fundamentally convey Java code composed by end users made it possible to expel active network research as too far expelled from genuine networks and inherently unsafe. Active networking was also censored at that time for its inability to offer practical performance and security. While performance was not a primary thought of the active networking research community (which concentrated on programming models, platforms, and architecture), a few efforts planned to manufacture high-performance active routers. Likewise, while security was under-tended to in a large portion of the early projects, the secure active network environment (SANE) architecture project was an eminent exemption.

3.1.4. Practicality of invention

Albeit active networks verbalized an idea of programmable networks, the technologies did not see broad deployment. Maybe one of the greatest stumbling pieces was the absence of an instantly compelling issue or an unambiguous way to deployment. A critical lesson from the active network research exertion was that executioner applications for the data plane are difficult to conceive. The community proffered different applications that could profit by in-network processing, including information fusion, caching and content distribution, network management, and application-specific quality of service. Tragically, even though performance advantages could be measured in the lab, none of these applications exhibited an adequately compelling answer for a pressing need.

In subsequent endeavors, more concentration was on routing and configuration management. In addition to a narrower space, the following period of research developed technologies that drew an obvious distinction and detachment between the elements of the control and data planes. This partition, at last, made it conceivable to concentrate on advancements in the control plane, which required a critical redesign as well as, because it is normally executed in software, displayed a lower fence to development than the data plane.

3.2 Control and Data plane separation

In the mid-2000s, expanding traffic volumes and a more noteworthy attention on network reliability, predictability, and performance drove network administrators to look for better ways to deal with certain network management capacities, for example, control of the paths used to convey traffic (usually known as traffic engineering). The methods for traffic engineering utilizing traditional routing protocols were primitive, best case scenario. Administrators' disappointment with these methodologies was perceived by a little, well-situated community of researchers who either worked for or regularly collaborated with backbone network administrators. These researchers investigated realistic, near-term approaches that were either standards-driven or inevitably deployable utilizing existing protocols.

Precisely, conventional routers and switches demonstrate a tight integration between the control and data planes. This coupling made different network management responsibilities, for example, debugging configuration problems and predicting or controlling routing behavior, exceedingly challenging. To address these difficulties, numerous efforts to isolate the data and control planes started to rise.

3.2.1 Technology push and use pull

As the Internet thrived in the 1990s, the connection speeds in backbone networks developed quickly, driving equipment vendors to execute packet-forwarding rationale directly in hardware, isolate from the control-plane software. Additionally, ISPs (Internet service providers) were attempting to deal with the expanding size and scope of their networks and also the requests for more prominent reliability and new services like virtual private networks. In parallel with these patterns, the fast advances in commodity computing platforms implied that servers regularly had considerably more memory and processing resources than the control-plane processor of a router installed only maybe a couple years prior. These patterns catalyzed two developments:

- An open interface between the control and data planes, for example, the ForCES (Forwarding and Control Element Separation) interface standardized by the IETF and the Netlink interface to the kernel-level packet-forwarding functionality in Linux.
- Logically centralized control of the network, as found in the RCP (Routing Control Platform) and SoftRouter designs, and in addition the PCEP (Path Computation Element protocol) at the IETF.

These innovations were driven by industry's demands for technologies to oversee routing inside an ISP network. Some early suggestions for untying the data and control planes also originated from scholarly circles, in both ATM and active networks.

In associated with previous research on active networking, these projects concentrated on squeezing issues in network management, with an emphasis on innovation by and for network administrators (instead of end users and researchers) programmability in the control plane (as opposed to the data plane) and network-wide visibility and control (as opposed to device-level configuration).

Network management applications included choosing better network paths considering the present traffic load, limiting transient interruptions during arranged routing changes, giving customer networks more control over the flow of traffic, and redirecting or dropping suspected attack traffic. A few control applications kept running in operational ISP networks utilizing legacy routers, including the IRSCP (Intelligent Route Service Control Point) installed to offer value-added services for virtual private network customers in AT&T's level 1 backbone network. Albeit a great part of the work during this time concentrated on overseeing routing inside a solitary ISP, some work also proposed approaches to empower adaptable route control over various administrative domains.

Moving control functionality from network equipment and into particular servers seemed well and good since network management is a network-wide movement. Logically centralized routing controllers were made conceivable by the rise of open-source routing software that brought down the hindrance to developing model implementations. The advances in server technology implied that a single server could store all the routing state and process all the routing decisions for a huge ISP network. This, thus, empowered straightforward primary-backup replication methodologies, where backup servers store a similar state and measure the same computation as the primary server, to guarantee controller reliability.

3.2.2. Logical Contributions of Separation of Planes to SDN

The underlying attempts to separate the control and data planes were generally pragmatic, yet they represented a significant reasonable takeoff from the Internet's traditional tight coupling of path computation and packet forwarding. The efforts to separate the network's control and data planes resulted in several concepts that have been conveyed forward in subsequent SDN designs:

- Logically centralized control using an open interface to the data plane.
  The ForCES working group at the IETF projected a standard, open interface to the data plane to empower innovation in control-plane software. The SoftRouter utilized the ForCES API to permit a different controller to install forwarding table entries in the data plane, permitting the total exclusion of control functionality from the routers. Unfortunately, ForCES was not implemented by the major router vendors, which hampered incremental deployment. Rather than sitting tight for new, open APIs to rise, the RCP utilized a current standard control-plane protocol (BGP - Border Gateway Protocol) to install forwarding table entries in

legacy routers, permitting prompt deployment. OpenFlow also confronted comparable recessive compatibility difficulties and constraints, specifically, the initial OpenFlow specification relied on in reverse compatibility with hardware capabilities of commodity switches.

- Distributed state management
  Logically centralized route controllers confronted challenges including distributed state administration. A logically centralized controller must be duplicated to adapt to controller failure, however, replication presents the potential for conflicting state across imitations. Researchers investigated the possible failure situations and consistency necessities. In any event of routing control, the controller imitations did not require a general state management protocol, since every copy would finally compute similar routes (after taking in a similar topology and routing information), and transient interruptions during routing protocol convergence were satisfactory even with legacy protocols. For better scalability, every controller occurrence could be in charge of a different part of the topology. These controller occurrences could then trade routing data with each other to guarantee reliable decisions. The difficulties of building distributed controllers would emerge again several years later with regards to distributed SDN controllers. These controllers face the significantly general issue of supporting arbitrary controller applications, requiring more sophisticated resolutions for distributed state management.

3.2.3. Myths about Separation of Control and Data plane

At the point when these new architectures were proposed, critics saw them with solid incredulity, regularly vehemently contending that logically centralized route control would violate fate sharing since the controller might fail independently from the devices responsible for forwarding traffic. Many network administrators and researchers saw isolating the control and data planes as a characteristically awful thought, as at first there was no certain articulation of how these networks would keep on operating accurately if a controller failed. Cynics additionally stressed that logically centralized control moved far from the conceptually straightforward model of the routers accomplishing distributed accord, where they all (in the end) have a typical perspective of network state (e.g., through flooding). In logically centralized control, every router has just a simply local perspective of the result of the route determination process.

When these projects flourished, even the traditional distributed routing solutions officially abused these principles. Moving packet-forwarding logic into hardware implied that a router's control plane software could fail autonomously from the data plane. Similarly, distributed routing protocols received scaling techniques, such as OSPF (Open

Shortest Path First) areas and BGP (Border Gateway Protocol) route reflectors, where routers in one locale of a network had constrained visibility into the routing information in different regions. The separation of the control and data planes somewhat illogically empowered researchers to contemplate distributed state management, the decoupling of the control and data planes catalyzed the development of a state management layer that maintains a consistent view of the network state.

3.2.4. Simplifying and digging more into SDN

Leading equipment vendors had little incentive to receive standard data plane APIs, for example, ForCES, since open APIs could pull in new entrants into the marketplace. The resulting need to rely on existing routing protocols to control the data plane imposed significant restrictions on the range of applications that programmable controllers could support. Conventional IP routing protocols compute routes for destination IP address blocks, rather than providing a wider range of functionality (e.g., dropping, flooding, or modifying packets) based on a wider range of header fields (e.g., MAC and IP addresses, TCP and UDP port numbers), as OpenFlow does. At last, although the business prototypes and standardization efforts made some progress, widespread selection remained intangible.

To widen the vision of control and data plane partition, researchers began investigating clean-slate architectures for logically centralized control. The 4D project upheld four fundamental layers: the data plane (for processing packets considering configurable rules); the discovery plane (for gathering topology and traffic measurements); the dissemination plane (for introducing packet processing rules); and a decision plane (comprising of logically centralized controllers that change over network-level aims into packet-handling state). A few groups continued to outline and construct systems that applied this high-level way to deal with new application ranges, beyond route control. Specifically, the Ethane venture (and its immediate antecedent, SANE) made a logically centralized, flow level solution for access control in enterprise networks. Ethane decreases the switches to flow tables that are populated by the controller considering high-level security policies. The Ethane venture and its operational deployment in the Stanford computer science department set the phase for the making of OpenFlow. Specifically, the simple switch design in Ethane turned into the premise of the first OpenFlow API.

3.3 The OpenFlow API and network operating systems

In the mid-2000s, researchers and funding agencies picked up enthusiasm for network experimentation at scale, energized by the accomplishment of experimental infrastructures (e.g., PlanetLab and Emulab), and the availability of separate government funding for expansive scale instrumentation already saved for different disciplines to manufacture costly, shared infrastructure, for example, colliders and telescopes. An outgrowth of this excitement was the making of GENI (Global Environment for

Networking Innovations) with an NSF-funded GENI Project Office and the EU FIRE program. Critics of these infrastructure focused efforts called attention to that this extensive interest in infrastructure was not coordinated by well-conceived thoughts to utilize it. Amidst this, a gathering of researchers at Stanford made the Clean Slate Program and focused on experimentation at a more local and tractable scale i.e. campus networks.

Prior to the development of OpenFlow, the thoughts hidden SDN confronted a strain between the vision of completely programmable networks and practicality that would empower real-world deployment. OpenFlow struck a balance between these two objectives by empowering a greater number of capacities than prior route controllers and expanding on existing switch hardware through the expanding utilization of merchant silicon chipsets in commodity switches. Even though depending on existing switch hardware did to some degree constrain flexibility, OpenFlow was very quickly deployable, permitting the SDN development to be both logical and bold. The formation of the OpenFlow API was taken after rapidly by the design of controller stages, for example, NOX that empowered the making of numerous new control applications.

An OpenFlow switch has a table of packet handling rules, where each rule has a pattern (which matches on bits in the packet header), a rundown of actions like drop, flood, forward out a specific interface, modify a header field, or send the packet to the controller, a set of counters (to track the number of bytes and packets), and a priority (to disambiguate between rules with overlapping patterns). After receiving a packet, an OpenFlow switch identifies the highest priority matching rule, performs the associated actions, and increments the counters.

3.3.1. Technology push and use pull

Maybe the characterizing highlight of OpenFlow is its acceptance in the industry, particularly as contrasted with its intellectual predecessors. This achievement can be credited to an immaculate tempest of conditions among equipment vendors, chipset designers, network administrators, and networking researchers. Prior to OpenFlow's beginning, switch chipset vendors like Broadcom had as of now opened their APIs to permit programmers to control certain forwarding practices. The decision to open the chipset gave the vital driving force to an industry that was already clamoring for more control over network gadgets. The accessibility of these chipsets likewise empowered a considerably more extensive scope of organizations to build switches, without bringing the generous cost of designing and fabricating their own data-plane equipment.

The underlying OpenFlow protocol standardized a data-plane model and a control plane API by expanding on technology that switches already upheld. Specifically, since network switches effectively bolstered fine-grained access control and flow checking, enabling

OpenFlow's initial arrangement of capacities on a switch was as simple as playing out a firmware update. Vendors did not have to upgrade the equipment to make their switches OpenFlow proficient.

OpenFlow's initial target deployment situation was campus networks, addressing the requirements of a networking research community that was effectively searching for approaches to direct trial deal with clean-slate network structures inside a research accommodating operational setting. In the late 2000s, the OpenFlow group at Stanford drove a push to send OpenFlow test beds crosswise over numerous campuses and show the abilities of the protocol both on a solitary campus network and over a wide-area backbone network traversing multiple campuses.

As genuine SDN use cases appeared on these campuses, OpenFlow started to grab hold in different domains, for example, data-center networks, where there was a diverse need to manage network traffic at a substantial scale. In data centers, hiring engineers to compose advanced control programs to keep running over vast quantities of commodity switches ended up being more cost effective than keeping on obtaining closed, proprietary switches that couldn't bolster new elements without significant engagement with the equipment vendors. As vendors contended to offer both servers and switches for data centers, numerous littler players in the network equipment marketplace embraced the chance to rival the established router and switch vendors by supporting new capabilities like OpenFlow.

3.3.2. Logical contributions of OpenFlow API and network operating systems to SDN

Although OpenFlow typified large portions of the standards from prior work on the detachment of control and data planes, its ascent offered several extra intellectual commitments as described below:

- **Generalizing network devices and functions**
  Past work on route control concentrated basically on coordinating traffic by destination IP prefix. Conversely, OpenFlow rules could characterize forwarding behavior on traffic flows in view of any arrangement of 13 diverse packet headers. Thusly, OpenFlow thoughtfully bound together with a wide range of different network devices that vary just as far as which header fields they match and which activities they perform. Router coordinates on destination IP prefix and forwards out a link, though a switch coordinates on a source MAC address (to perform MAC learning) and a destination MAC address (to forward), and either floods or forwards out a solitary link. Network address translators and firewalls coordinate on a 5-tuple (source and destination IP addresses, source and destination port numbers, and transport protocol) and either rewrite address and port fields or drop undesirable traffic. OpenFlow additionally summed up the rule

installation techniques, permitting anything from the proactive establishment of coarse-grained rules (i.e., with "wild cards" for some header fields) to the responsive installation of fine-grained rules, depending upon the application. Still, OpenFlow does not offer data plane support for profound packet investigation or connection reassembly. All things considered, OpenFlow alone can't productively empower sophisticated middlebox functionality.

- The vision of a network operating system
  In contrast to previous research on active networks that proposed a node operating system, the work on OpenFlow prompted to the idea of a network operating system. A network operating system is software that abstracts the installation of state in network switches from the logic and applications that control the conduct of the network. Even more, for the most part, the development of a network operating system offered a conceptual disintegration of network operation into three layers:
  (1) a data plane with an open interface
  (2) a state management layer whose responsibility is to maintain a consistent perspective of network state
  (3) control logic that performs different operations relying upon its perspective of the network state.

- Distributed state management techniques
  Isolating the control and data planes presents new difficulties concerning state management. Running numerous controllers is urgent for scalability, reliability, and performance, yet these reproductions ought to cooperate to go about as a solitary, logically centralized controller. Past work on distributed route controllers tended to these issues just in the narrow context of route computation. To bolster arbitrary controller applications, the work on the Onix controller presented the possibility of a network information base, a portrayal of the network topology and other control state shared by all controller replicas. Onix additionally joined past work in distributed systems to fulfill the state consistency and durability requirements. For instance, Onix has a transactional persistent database supported by a duplicated state machine for gradually changing network state, as well as an in-memory distributed hash table for quickly changing state with weaker consistency requirements. In addition to this, recently, ONOS (Open Network Operating System) offers an open source controller with comparative functionality, utilizing existing open source software for keeping up consistency crosswise over distributed state and giving a network topology database to controller applications.

3.3.3. Myths about the OpenFlow API and network operating systems

One myth concerning SDN is that the primary packet of each traffic flow must go to the controller for handling. In fact, some early systems like Ethane worked thusly, since they were intended to support fine-grained policies in little networks. However, SDN in general, OpenFlow specifically, don't force any presumptions about the granularity of rules or whether the controller handles any data traffic. Some SDN applications react just to topology changes and coarse-grained traffic insights, and rarely to refresh rules in response to link failure or network congestion. Different applications may send the first packet of some bigger traffic total to the controller however not a packet from each TCP or UDP connection.

Another myth about SDN is that the controller must be physically centralized. Actually, Onix and ONOS show that SDN controllers can and ought to be distributed. Wide-area deployments of SDN, as in Google's private backbone, have numerous controllers spread all through the network.

Last but not the least, a normally held misconception is that SDN and OpenFlow are proportional. In fact, OpenFlow is simply one (widely popular) instantiation of SDN standards. Distinctive APIs could be utilized to control network wide forwarding behavior. Past work that concentrated on routing (utilizing BGP as an API) could be viewed as one instantiation of SDN, for instance, and architectures from different vendors (e.g., Cisco ONE and JunOS SDK) are different instantiations of SDN that vary from OpenFlow.

3.3.4. SDN Use cases

Despite the early excitement about SDN, it merits observing that it is merely a tool that empowers innovation in network control. SDN neither commands how that control ought to be designed nor takes care of a specific problem. Or maybe, researchers and network administrators now have a stage available to them to help address longstanding issues in dealing with their networks and deploying new services. Eventually, the achievement and reception of SDN will rely on upon whether it can be utilized to take care of squeezing issues in networking that were troublesome or difficult to tackle with prior protocols.

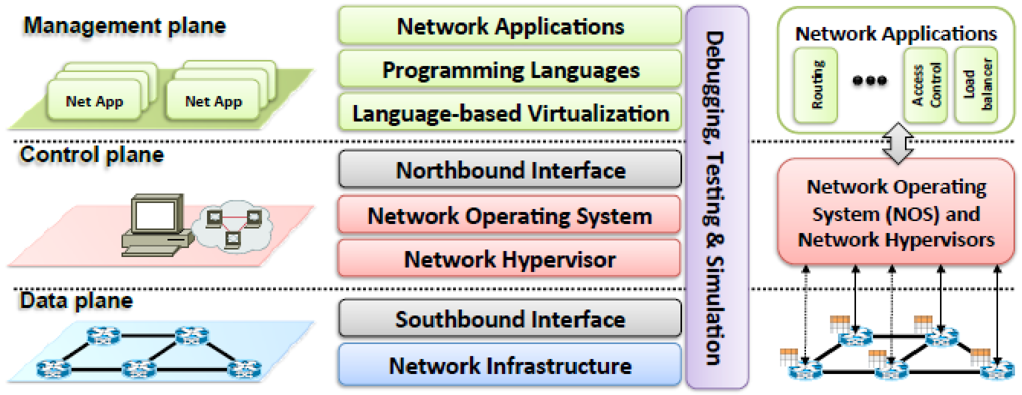
Although SDN has appreciated some early practical accomplishments and unquestionably offers much needed technologies to bolster network virtualization, more work is required both to enhance the current infrastructure and to investigate SDN's capability to take care of issues for a considerably more extensive arrangement of use cases. Albeit early SDN arrangements focused on college campuses, data centers, and private backbones, recent work explores applications and extensions of SDN to a more extensive scope of network settings, including home networks, enterprise networks, Internet exchange points, cellular core networks, cellular and Wi-Fi radio access networks, and joint administration of end-host applications and the network. Each of these settings presents numerous new open doors and difficulties that the community will investigate in the years ahead.

The possibility of a programmable network at first came to realization as active networking, which espoused a significant number of an identical ideas as SDN yet needed both a reasonable use case and an incremental deployment path. After the era of active-networking research projects, the pendulum swung from vision to realism, through isolating the data and control planes to make the network easier to manage. This work focused mainly on better approaches to route network traffic, a much smaller vision than past work on active networking.

As SDN keeps on building up, its history has vital lessons to instruct. To begin with, SDN technologies will live or die on use pulls. Although SDN is regularly proclaimed as the solution to all networking issues, it is worth recalling that it is only a tool for taking care of network management issues more effectively. SDN just gives developers the ability to make new applications and discover solutions for longstanding problems. In this regard, the work is quite recently starting. On the off chance that the past is any sign, the development of these new technologies will require innovation on various timescales, from long term bold visions, (for example, active networking) to near term inventive problem solving, (for example, the operationally focused work around isolating the control and data planes).

4. Architecture and Operation of SDN

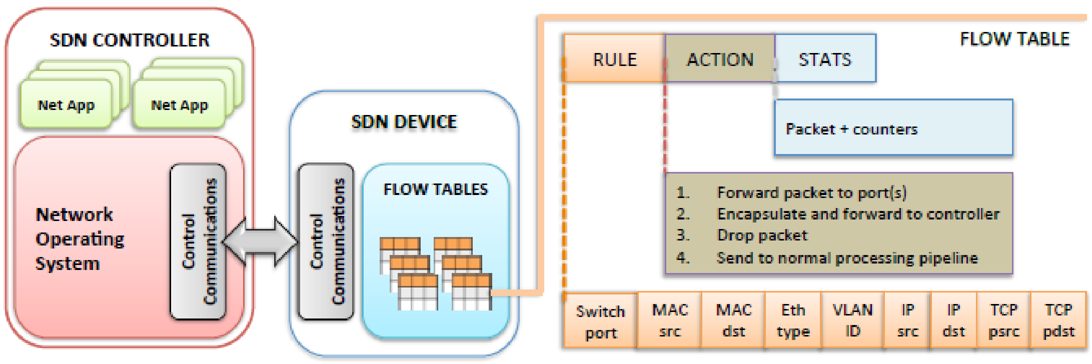
A SDN architecture can be portrayed as a composition of distinctive layers, as shown in Figure below. Each layer has its own specific roles. While some of them are always present in a SDN deployment, for example, the southbound API, network operating systems, northbound API and network applications, others might be available in just specifically deployments, for example, hypervisor or language based virtualization.



4.1. Layer 1 - Infrastructure

A SDN infrastructure, similarly to a conventional network, is combination of different networking equipment like switches, routers and, middlebox appliances. The primary distinction lives in the fact that those traditional physical devices are currently basic forwarding components without implanted control or software to take independent decisions. The network intelligence is expelled from the data plane devices to a logically-centralized control system, i.e., the network operating system and applications, as appeared in picture above. More essentially, these new networks are constructed conceptually on top of open and, standard interfaces (e.g., OpenFlow), a critical approach for guaranteeing configuration and communication compatibility and interoperability among various data and control plane devices. In other words, these open interfaces empower controller elements to dynamically program heterogeneous forwarding devices, something problematic in conventional networks, due to the huge variation of proprietary and closed interfaces, and the distributed nature of the control plane.

In a SDN/OpenFlow architecture, there are two primary components, the controllers and the forwarding devices as described in picture below. A data plane device is a hardware or software component had some expertise in packet forwarding, while a controller is a software stack (the network brain) running on a commodity hardware platform. An OpenFlow empowered forwarding device depends on a pipeline of flow tables where every entry of a flow table has three sections: (1) a matching rule, (2) actions to be performed on matching packets, and (3) counters that keep information about corresponding packets. This high-level and simplified model got from OpenFlow is at present the most far reaching outline of SDN data plane devices. By and by, different specifications of SDN enabled forwarding devices are being sought after, including POF, and the Negotiable Datapath Models (NDMs) from the ONF Forwarding Abstractions Working Group (FAWG).



Inside an OpenFlow device, a path through a sequence of flow tables characterizes how packets ought to be taken care of. At the point when new packet arrives, the query procedure begins in the first table and finishes either with a match in one of the tables of the pipeline or, with a miss (when no rule is found for that packet). A flow rule can be defined by consolidating distinctive matching fields, as outlined in Figure. If there is no default rule, the packet will be discarded. Nonetheless, the regular case is to introduce a default rule which advises the switch to send the packet to the controller or to the typical non-OpenFlow pipeline of the switch. The need of the rules trails the natural sequence number of the tables and the row arrangement in a flow table. Probable activities include (1) forward the packet to outgoing port(s), (2) encapsulate it and forward it to the controller, (3) drop it, (4) send it to the normal processing pipeline, (5) send it to the following flow table or to special tables, for example, group or metering tables presented in the most recent OpenFlow protocol.

4.1.1. SDN Devices

Several OpenFlow empowered forwarding devices are available on the market, both as business and open source items. There are many off-the-rack, ready to deploy, OpenFlow switches and routers, among different machines. Most of the switches, available have generally little Ternary Content-Addressable Memory (TCAMs), with up to 8K entries. However, this is changing at a quick pace. A few of the most recent devices released in the market go beyond that figure. For instance, Gigabit Ethernet (GbE) switches for normal business designs are as of now supporting up to 32K L2+L3 or 64K L2/L3 correct match flows. Enterprise class 10GbE switches are being deployed with more than 80K Layer 2 flow entries. Other switching devices utilizing high throughput chips (e.g., EZchip NP-4) give improved TCAM memory that supports from 125K up to 1000K flow table entries. This is a clear sign that the extent of the flow tables is developing at a pace intending to address the issues of future SDN deployments.

Software switches are rising as a standout amongst the most promising solutions for data centers and virtualized network frameworks. Examples of software-based OpenFlow switch executions incorporate Switch Light, ofsoftswitch, open vSwitch, OpenFlow Reference, Pica, Pantou, and XorPlus. Recent reports demonstrate that the number of virtual access ports is officially bigger than physical access ports on data centers. Network virtualization has been one of the drivers behind this slant. Software switches, for example, Open vSwitch have been utilized for moving network functions to the edge (with the core performing conventional IP forwarding), thus empowering network virtualization.

4.2. Layer 2 -- Southbound Interfaces

Southbound interfaces (or southbound APIs) are the connecting bridges amongst control and forwarding components, subsequently being the essential instrument for clearly isolating control and data plane functionality. However, these APIs are still firmly attached to the forwarding components of the hidden physical or virtual infrastructure.

Normally, new switch can take two years to be prepared for commercialization if built from scratch, with upgrade cycles that can take up to nine months. The software advancement for new product can take from six months to one year. The initial investment is high and risky. As a central part of its design the southbound APIs denotes one of the significant restrictions for the presentation and acknowledgment of any new networking technology. In this light, the development of SDN southbound API recommendations, for example, OpenFlow is viewed as welcome by numerous in the business. These standards promote interoperability, permitting the deployment of vendor skeptic network devices. This has as of now been shown by the interoperability between OpenFlow empowered equipment from various vendors.

OpenFlow is the most generally acknowledged and deployed open southbound standard for SDN. It gives a common specification to implement OpenFlow empowered forwarding devices, and for the communication channel between data and control plane devices (e.g., switches and controllers). The OpenFlow protocol offers three data sources to network operating systems. First, event based messages are sent by forwarding devices to the controller when a link or port change is triggered. Second, flow statistics are created by the forwarding devices and gathered by the controller. Third, packet-in messages are sent by forwarding devices to the controller when they don't realize what to do with a new approaching flow or because there is an explicit "send to controller" activity in the matched entry of the flow table. These information channels are the essential means to give flow level data to the network operating system.

OVSDB is another sort of southbound API, intended to give propelled management capabilities to Open vSwitches. Beyond OpenFlow's capabilities to configure the behavior of flows in a forwarding device, an Open vSwitch offers other networking functions. For example, it permits the control components to make various virtual switch instances, set QoS policies on interfaces, attach interfaces to the switches, configure tunnel interfaces on OpenFlow data paths, manage queues, and gather statistics. Thus, the OVSDB is a reciprocal protocol to OpenFlow for Open vSwitch.

A recent southbound interface proposal is OpFlex. As opposed to OpenFlow (and like ForCES), one of the thoughts behind OpFlex is to distribute some portion of the complexity of managing the network back to the forwarding devices, with the point of enhancing scalability. Like OpenFlow, policies are logically centralized and abstracted from the fundamental implementation. The contrasts amongst OpenFlow and OpFlex are a clear representation of one of the essential inquiries to be addressed when conceiving a southbound interface: where to put each piece of the overall functionality.

4.3. Layer 3 -- Network Hypervisors

Virtualization is as of now a consolidated technology in modern computers. The quick advancements of the previous decade have made virtualization of computing platforms mainstream. Considering recent reports, the number of virtual servers has as of now surpassed the number of physical servers.

Hypervisors enable distinct virtual machines to share the same hardware resources. In a cloud infrastructure-as-a-service (IaaS), each client can have its own virtual resources, from computing to storage. This enabled new revenue and business models where clients allocate resources on-demand, from a shared physical infrastructure, at a relatively minimal cost. At the same time, providers make better utilization of the capacity of their installed physical infrastructures, creating new revenue streams without significantly increasing their CAPEX and OPEX costs. One of the intriguing features of virtualization technologies today is the fact that virtual machines can be easily migrated starting with one physical server then onto the next and can be created and/or destroyed on-demand, enabling the provisioning of elastic services with adaptable and easy management. Unfortunately, virtualization has been just partially realized in practice. Notwithstanding the great advances in virtualizing computing and storage components, the network is still generally statically configured in a box- by-box manner.

The primary network reequipments can be captured along two dimensions: network topology and address space. Diverse workloads require distinctive network topologies and services, for example, level L2 or L3 services, or much more intricate L4- L7 services for advanced functionality. As of now, it is extremely problematic for a solitary physical topology to support the different requests of applications and services. Correspondingly, address space is difficult to change in current networks. These days, virtualized workloads need to work in a similar address of the physical infrastructure. Accordingly, it is difficult to keep the original network configuration for a tenant, virtual machines cannot migrate to arbitrary areas, and the addressing scheme is fixed and difficult to change. For instance, IPv6 can't be utilized by the VMs of a tenant if the hidden physical forwarding devices support just IPv4.

To deliver complete virtualization the network ought to give similar properties to the computing layer. The network infrastructure should have the capacity to support arbitrary network topologies and addressing schemes. Each tenant should be able to configure both the computing nodes and the network concurrently. Host migration should consequently trigger the migration of the corresponding virtual network. ports. One may surmise that long-standing virtualization primitives like VLANs (virtualized L2 domain), NAT (Virtualized IP address space), and MPLS (virtualized path) are sufficient to deliver full and automated network virtualization. But, these innovations are secured on a box-by-box basis configuration, i.e., there is no single unifying abstraction that can be utilized to configure (or reconfigure) the network in a global way. Thus, current network provisioning can take months, while computing provisioning takes just minutes.

There is an anticipation that this circumstance will change with SDN and the availability of new tunneling techniques (e.g., VXLAN, NVGRE). For example, solutions like FlowVisor, FlowN, NVP, OpenVirteX, IBM SDN VE, Radio-Visor, AutoVFlow, eXtensible Datapath Daemon (xDPd), optical transport network virtualization, and version-agnostic OpenFlow slicing mechanisms, have been recently proposed, evaluated and deployed in real situations for on-demand provisioning of virtual networks.

Presently there are already a few network hypervisor proposals leveraging the advances of SDN. There are, however, still a few issues to be addressed. These include, among others, the advancement of virtual-to-physical mapping techniques, the meaning of the level of detail that ought to be exposed at the logical level, and the support for nested virtualization. We envision, nonetheless, this ecosystem to expand in near future since network virtualization will probably play a key role in future virtualized environments, comparably to the expansion we have been seeing in virtualized computing.

4.4. Layer 4 -- Network Operating Systems / Controllers

Conventional operating systems deliver abstractions (e.g., high-level programming APIs) for accessing lower-level devices, deal with the concurrent access to the basic resources (e.g., hard drive, network adapter, CPU, memory), and provide security assurance mechanisms. These functionalities and resources are key empowering influences for expanded productivity, making the life of system and application developers easier. Their far-reaching use has significantly added to the advancement of different ecosystems (e.g., programming languages) and the improvement of a myriad of applications.

In contrast, networks have so far been managed and configured utilizing lower level, device-specific instruction sets and for the most part closed proprietary network operating systems (e.g., Cisco IOS and Juniper JunOS). Additionally, operating systems abstracting device-specific attributes and providing, straightforwardly, basic functionalities is still mostly absent in networks. For example, currently, designers of routing protocols need to manage complex distributed algorithms when resolving networking issues. Network practitioners have in this way been taking care of similar issues again and again.

SDN is guaranteed to facilitate network management and ease the weight of solving networking issues by method for the logically-centralized control offered by a network operating system (NOS). Similarly, as with conventional operating systems, the vital role of a NOS is to give abstractions, fundamental services, and common application programming interfaces (APIs) to developers. Generic functionality as network state and network topology information, device discovery, and distribution of network configuration can be given as services of the NOS. With NOSs, to characterize network policies a designer no longer needs to think about the low-level details of data distribution among routing components, for example. Such systems can apparently make a new environment equipped for fostering innovation at a faster pace by decreasing the inherent complexity of making new network protocols and network applications.

A NOS (or controller) is a critical component in a SDN architecture as it is the key supporting piece for the control logic (applications) to generate the network configuration considering the policies defined by the network administrator. Like a conventional operating system, the control platform abstracts the lower-level details of connecting and interacting with forwarding devices (i.e., of materializing the network policies).

4.4.1. Architecture and design axes

There is an extremely diverse set of controllers and control platforms with various design and architectural options. Existing controllers can be sorted in view of numerous aspects. From an architectural perspective, a standout amongst the most significant is whether they are centralized or distributed. This is one of the key design axes of SDN control platforms. Let’s see how it can be considered as key design axes.

4.4.1.1. Centralized Vs. Distributed Approach

A centralized controller is a solitary entity that manages all forwarding devices of the network. Normally, it represents a single point of failure and may have scaling restrictions. A single controller may not be sufficient to manage a network with many data plane components. Centralized controllers such as NOX-MT, Maestro, Beacon, and Floodlight have been designed as profoundly concurrent systems, to accomplish the throughput required by enterprise class networks and data centers. These controllers are based on multi-threaded designs to explore the parallelism of multi-core computer architectures. For instance, Beacon can manage more than 12 million flows per second by utilizing large size computing nodes of cloud providers such as Amazon. Other centralized controllers such as Trema, Ryu NOS, Meridian, and ProgrammableFlow, target specific environments such as data centers, cloud infrastructures, and carrier grade networks. Moreover, controllers such as Rosemary offer specific functionality and assurances, namely security and isolation of applications. By utilizing a container-based architecture called micro-NOS, it accomplishes its essential objective of separating applications and keeping the propagation of failures throughout the SDN stack.

Contrary to centralized design, a distributed network operating system can be scaled up to meet the requirements of possibly any environment, from small to big scale networks. A distributed controller can be a centralized group of nodes or a physically distributed arrangement of components. While the first alternative can offer high throughput for extremely dense data centers, the latter can be more resilient to various types of logical and physical failures. A cloud provider that traverses different data centers interconnected by a wide area network may require a hybrid approach, with clusters of controllers inside every data center and distributed controller nodes in the distinctive sites.

Onix, HyperFlow, HP VAN SDN, ONOS, DISCO, yanc, PANE, SMaRt-Light, and Fleet are examples of distributed controllers. Most distributed controllers offer poor consistency semantics, which implies that data updates on distinct nodes will ultimately be updated on all controller nodes. This implies there is a timeframe in which distinct nodes may read diverse values (old value or new value) for a same property. Strong consistency, alternatively, ensures that all controller nodes will read the most refreshed value after a write operation. Regardless of its effect on system performance, strong consistency offers an easier interface to application developers. To date, just Onix, ONOS, and SMaRtLight give this data consistency model.

Another common property of distributed controllers is fault tolerance. When one node fails, another neighbor node ought to take over the obligations and devices of the failed node. Up until now, aside a few controllers tolerating crash failures, they don't tolerate arbitrary failures, which implies that any node with an irregular behavior won't be replaced by a potentially well behaved one.

A single controller might be sufficient to manage a small network, but it describes a single point of failure. Essentially, autonomous controllers can be spread over the network, each of them dealing with a network segment, decreasing the effect of a single controller failure. However, if the control plane availability is critical, a group of controllers can be utilized to accomplish a higher level of availability and additionally to support more devices. Eventually, a distributed controller can enhance the control plane resilience, scalability and decrease the effect of issues instigated by network partition, for example. SDN resiliency in general can be considered as an open challenge.

4.4.2. SDN Controller Platforms

Most of the controller can three well-defined layers as per individual responsibility. Three layers are:

i. The application, orchestration and services;
ii. The core controller functions
iii. The elements of southbound communications

The connection at the upper-level layers depends on northbound interfaces such as REST APIs and programming languages, for example, FML, Frenetic and NetCore. On the lower-level part of a control platform, southbound APIs and protocol plugins interface the forwarding components. The core of a controller platform can be described as a blend of its base network service functions and the different interfaces.

4.4.2.1. The Core controller functions

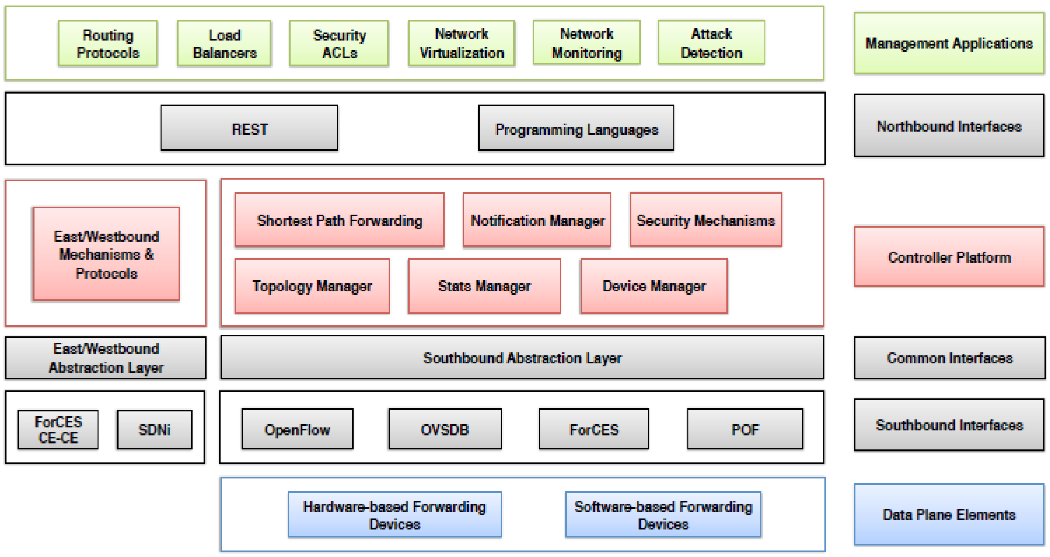
The base network service functions are what we consider the essential functionality all controllers ought to provide. As an analogy, these functions are like base services of operating systems, for example, program execution, I/O operations control, communications, protection, and so on. These services are utilized by other operating system level services and user applications. Correspondingly, functions like topology, statistics, notifications and device management, together with shortest path forwarding and security mechanisms are essential network control functionalities that network applications may use in building its logic. For instance, the notification manager ought to be able to receive, process, and forward events (e.g., alarm notifications, security alarms, state changes). Security mechanisms are another example, as they are critical components to provide basic isolation and security enforcement between services and applications. For example, rules generated by high priority services ought not be overwritten with rules created by applications with a lower priority.

4.4.2.2. Southbound

On the lower-level of control platforms, the southbound APIs can be viewed as a layer of device drivers. They provide a common interface to the upper layers, while permitting a control platform to utilize diverse southbound APIs (e.g., OpenFlow, OVSDB, ForCES) and protocol plugins to manage existing or new physical or virtual devices (e.g., SNMP, BGP, NetConf). This is essential both for backward compatibility and heterogeneity, i.e., to permit different protocols and device management connectors. In this way, on the data plane, a blend of physical devices, virtual devices (e.g., Open vSwitch, vRouter) and a variety of device interfaces (e.g., OpenFlow, OVSDB, of-config, NetConf, and SNMP) can exist together.

Most controllers support only OpenFlow as a southbound API. Still, a few of them, such as, OpenDaylight, Onix and HP VAN SDN Controller, offer a more extensive scope of southbound APIs and additionally protocol plugins. Onix supports both the OpenFlow and OVSDB protocols. The HP VAN SDN Controller has other southbound connectors such as L2 and L3 agents.

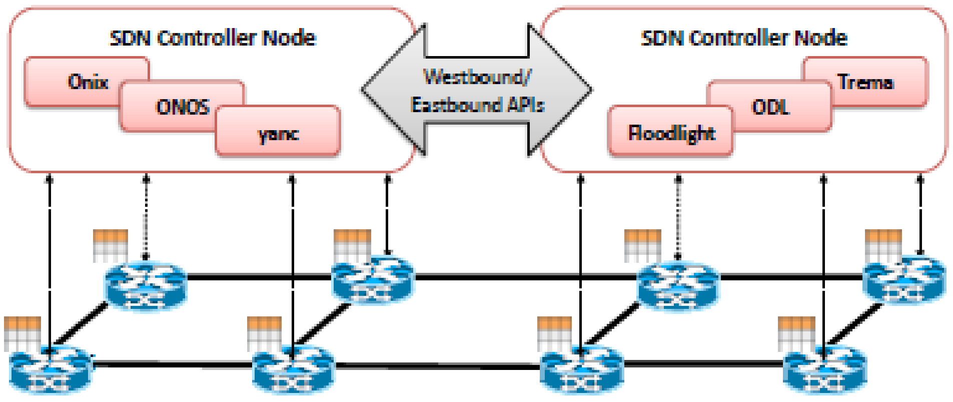
OpenDaylight goes a step beyond by providing a Service Layer Abstraction (SLA) that permits several southbound APIs and protocols to exist together in the control platform. For example, its unique architecture was intended to support no less than seven distinct protocols and plugins: OpenFlow, OVSDB, NETCONF, PCEP, SNMP, BGP and LISP Flow Mapping. Thus, OpenDaylight is one of the few control platforms being considered to support a more extensive coordination of technologies in a single control platform.



4.4.2.3. Eastbound and Westbound

East/westbound APIs, as represented in Figure below, are a special case of interfaces required by distributed controllers. As of now, every controller executes its own east/westbound API. The functions of these interfaces incorporate import/export data between controllers, algorithms for data consistency models, and monitoring/notification capabilities (e.g., check if a controller is up or advise a takeover on a set of forwarding devices).

Correspondingly to southbound and northbound interfaces, east/westbound APIs are essential components of distributed controllers. To identify and provide common compatibility and interoperability between various controllers, it is required to have standard east/westbound interfaces. For example, SDNi characterizes common prerequisites to coordinate flow setup and trade reachability data over numerous domains. Fundamentally, such protocols can be utilized as a part of an orchestrated and interoperable approach to make more scalable and dependable distributed control platforms. Interoperability can be leveraged to expand the diversity of the control platform component. Undoubtedly, diversity expands the system robustness by diminishing the likelihood of common faults such as software issues.



Different recommendations that attempt to characterize interfaces between controllers include Onix data import/export functions, ForCES CE-CE interface, ForCES Intra-NE cold-standby components for high availability, and distributed data stores. An east/westbound API requires advanced data distribution systems such as the Advanced Message Queuing Protocol (AMQP) utilized by DISCO, strategies for distributed concurrent and consistent policy creation, transactional databases and DHTs (as utilized in Onix), or advanced algorithms for strong consistency and fault tolerance.

In a multi-domain setup, east/westbound APIs may require also more particular communication protocols between SDN domain controllers. Some of the essentials functions of such protocols are to arrange flow setup started by applications, trade reachability information to facilitate inter-SDN directing, reachability update to keep the network state steady, among others.

Another important issue with respect to east/westbound interfaces is heterogeneity. For example, other than communicating with peer SDN controllers, controllers may likewise need to speak with subordinate controllers (in a pecking order of controllers) and non-SDN controllers, like the instance of Closed-Flow. To be interoperable, east/westbound interfaces along these lines need to suit distinctive controller interfaces, with their specific set of services, and the diverse attributes of the underlying framework, including the differences of technology, the geographic traverse and size of the network, and the division amongst WAN and LAN, possibly over administrative boundaries. In those cases, diverse information must be traded between controllers, including adjacency and capability discovery, topology information (to the degree of the agreed contracts between administrative domains), billing information, among numerous others.

In conclusion, a "SDN compass" strategy recommends a better distinction amongst eastbound and westbound horizontal interfaces, referring to westbound interfaces as SDN-to-SDN protocols and controller APIs while eastbound interfaces would be utilized to refer to standard protocols used to communicate with legacy network control planes (e.g., PCEP, GMPLS).

4.4.2.4. Northbound

Current controllers offer a very expansive assortment of northbound APIs such as ad-hoc APIs, RESTful APIs, multi-level programming interfaces, file systems, among other more specialized APIs, for example, NVP NBAPI, and SDMN API. A second sort of northbound interfaces are those stemming out of SDN programming languages such as Frenetic, Nettle, NetCore, Procera, Pyretic, NetKAT and other query-based languages.

4.4.3. Platforms comparison conclusion

As we have seen, it can be said that most controllers are centralized and multithreaded. Inquisitively, the northbound API is exceptionally assorted. Specifically, five controllers (Onix, Floodlight, MuL, Meridian and SDN Unified Controller) consider this interface, as an announcement of its significance. Consistency models and fault tolerance are only present in Onix, HyperFlow, HP VAN SDN, ONOS and SMaRtLight. In conclusion, with regards to the OpenFlow standard as southbound API, just Ryu supports its three noteworthy versions (v1.0, v1.2 and v1.3).

To finish up, it is necessary to stress that the control platform is one of the critical points for the accomplishment of SDN. One of the main issues that should be addressed in this regard is interoperability. This is somewhat intriguing, as it was the main problem that southbound APIs, (for example, OpenFlow) attempted to solve. For instance, while Wi-Fi and LTE (Long-Term Evolution) networks require specific control platforms such as MobileFlow or SoftRAN, data center networks have diverse requirements that can be met with platforms, for example, Onix or OpenDaylight. Thus, in conditions where diversity of networking infrastructures is a reality, coordination and cooperation between various controllers is vital. Standardized APIs for multi-controller and multi-domain deployments are consequently seen as an essential stride to accomplish this objective.

4.5. Layer 5 -- Northbound Interfaces

The Northbound and Southbound interfaces are two key abstractions of the SDN ecosystem. The southbound interface has already a widely-accepted proposal (OpenFlow), yet a common northbound interface is still an open issue. At this moment, it may in any case be a bit too soon to define a standard northbound interface, as use-cases are still being worked out. Anyway, it is to be expected a common (or a de facto) northbound interface to arise as SDN develops. An abstraction that would allow network applications not to depend on specific implementations is important to explore the maximum capacity of SDN.

The northbound interface is generally a software ecosystem, not a hardware one as is the case of the southbound APIs. In these environments, the implementation is commonly the forefront driver, while standards rise later and are essentially driven by wide adoption. By and by, an initial and minimal standard for northbound interfaces can in any case play an important part for the future of SDN. Discussions about this issue have already started and a common agreement is that northbound APIs are indeed important however that it is indeed too early to define a solitary standard right at this point. The experience from the development of various controllers will certainly be the basis for coming up with a common application level interface.

Open and standard northbound interfaces are crucial to promote application portability and interoperability among the distinctive the control platforms. A northbound API can be compared to the POSIX standard in operating frameworks, denoting an abstraction that ensures programming language and controller independence. NOSIX is one of the primary examples of an effort toward this path. It tries to define portable low-level (e.g., flow model) application interfaces, making southbound APIs such as OpenFlow resemble device drivers. In any case, NOSIX is not exactly a general purpose northbound interface, yet rather a higher-level abstraction for southbound interfaces. Indeed, it could be part of the common abstraction layer in a control platform.

Existing controllers such as Floodlight, Trema, NOX, Onix, and OpenDaylight propose and define their own northbound APIs. Nonetheless, each of them has its own specific definitions. Programming languages, for example, Frenetic, Nettle, NetCore, Procera, Pyretic and NetKAT also abstract the internal details of the controller functions and data plane behavior from the application developers. Moreover, programming languages can provide a wide range of powerful abstractions and mechanisms such as application composition, transparent data plane fault tolerance, and a variety of basic building blocks to ease software module and application development.

SFNet is another example of a northbound interface. It is a high-level state API that translates application requirements into lower level service requests. But, SFNet has a constrained scope, targeting inquiries to request the congestion state of the network and services such as bandwidth reservation and multicast.

Different proposals use distinctive approaches to permit applications to interact with controllers. The yanc control platform explores this idea by proposing a general control platform based on Linux and abstractions such as the virtual document framework (VFS). This approach simplifies the development of SDN applications as programmers can utilize a traditional idea (files) to communicate with lower level devices and sub-systems.

Eventually, it is unlikely that a single northbound interface develops as the winner, as the requirements for various network applications are unique. APIs for security applications are probably going to be unique in relation to those for routing or financial applications. One conceivable path of advancement for northbound APIs are vertically-oriented proposals, before any sort of standardization happens, a challenge the ONF has started to undertake in the NBI WG in parallel to open-source SDN developments. The ONF architectural work includes the likelihood of northbound APIs providing resources to enable dynamic and granular control of the network assets from client applications, eventually across various business and organizational boundaries.

4.6. Layer 6 -- Language-based Virtualization

Two essential characteristics of virtualization solutions are the capability of communicating modularity and of allowing distinctive levels of abstractions while yet guaranteeing desired properties such as protection. For instance, virtualization procedures can allow diverse perspectives of a solitary physical infrastructure. As an example, one virtual big switch could speak to a combination of several underlying forwarding devices. This intrinsically simplifies the task of application developers as they don't have to consider the arrangement of switches where forwarding rules must be installed, yet rather observe the network as a simple big switch. Such sort of abstraction significantly simplifies the development and deployment of complex network applications, for example, advanced security related services.

Pyretic is an interesting example of a programming language that offers this kind of high-level abstraction of network topology. It includes this idea of abstraction by presenting network objects. These objects comprise of an abstract network topology and the sets of policies applied to it. Network objects simultaneously hide information and offer the required services.

Another form of language-based virtualization is static slicing. This a scheme where the network is sliced by a compiler, based on application layer definitions. The output of the compiler is a monolithic control program that has already slicing definitions and configuration guidelines for the network. In such a case, there is no requirement for a hypervisor to dynamically manage the network portions. Static slicing can be valuable for deployments with specific requirements, in particularly those where higher performance and straightforward isolation guarantees are preferable to dynamic slicing.

Other solutions, for example, libNetVirt, attempt to integrate heterogeneous innovations for creating static network slices. libNetVirt is a library designed to provide an adaptable way to create and manage virtual networks in various computing situations. Its main idea is similar to the OpenStack Quantum project. While Quantum is designed for OpenStack (cloud environments), libNetVirt is a more general purpose library which can be used in various conditions. Additionally, it goes one level next to OpenStack Quantum by enabling QoS capabilities in virtual networks. The libNetVirt library has two layers: (1) a generic network interface; and (2) technology specific device drivers (e.g., VPN, MPLS, OpenFlow). On top of the layers are the network applications and virtual network descriptions. The OpenFlow driver uses a NOX controller to manage the underlying infrastructure, utilizing OpenFlow rule based flow tables to create isolated virtual networks. By supporting diverse technologies, it can be utilized as a bridging segment in heterogeneous networks.

4.7. Layers 7 -- Programming languages

Programming languages have been proliferating for decades. Both academia and industry have advanced from low-level hardware-specific machine languages, for example, assembly for x86 architectures, to high-level and powerful programming languages, for example, Java and Python. The improvements towards more portable and reusable code has driven a significant move on the computer industry.

Similarly, programmability in networks is starting to move from low level machine languages, for example, OpenFlow ("assembly") to high-level programming languages. Assembly-like machine languages, for example, OpenFlow and POF, essentially copy the behavior of forwarding devices, forcing developers to invest too much time on low-level details rather than on the issue resolution. Raw OpenFlow programs must deal with hardware behavior details, for example, overlapping rules, the priority ordering of rules, and data-plane inconsistencies that arise from in-flight packets whose flow rules are under installation. The use of these low-level languages makes it hard to reuse software, to create modular and extensive code, and leads to a more error-inclined development process.

Abstractions provided by high-level programming languages can significantly help address many of the challenges of these lower-level guideline sets. In SDNs, high-level programming languages can be designed and utilized to:

- create higher level abstractions for simplifying the task of programming forwarding devices
- enable more beneficial and problem focused atmospheres for network software programmers, accelerating development and innovation
- advance software modularization and code reusability in the network control plane;
- encourage the development of network virtualization.

Several challenges can be better addressed by programming languages in SDNs. For instance, in pure OpenFlow-based SDNs, it is hard to guarantee that various tasks of a single application (e.g., routing, monitoring, access control) don't meddle with each other. For example, rules generated for one task ought not override the functionality of another task. Another scenario is the point at which different applications keep running on a single controller. Typically, each application produces rules based on its own needs and policies without further information about the rules generated by different applications. As an outcome, conflicting rules can be generated and installed in forwarding devices, which can create issues for network operation. Programming languages and runtime systems can take care of these issues that would be otherwise hard to prevent.

Another key feature that programming language abstractions provide is the capability of creating and writing programs for virtual network topologies. This idea is similar to object-oriented programming, where objects abstract both data and specific functions for application developers, making it easier to concentrate on taking care of a particular issue without worrying about data structures and their management. For instance, in a SDN setting, instead of generating and installing rules in each forwarding device, one can consider creating improved virtual network topologies that outline the whole network, or a subset of it. For example, the application developer ought to have the capacity to abstract the network as an atomic big switch, rather than a combination of several underlying physical devices. The programming languages or runtime systems ought to oversee generating and installing the lower-level directions required at each forwarding device to enforce the user strategy across the network. With such sort of abstractions, developing a routing application turns into a simple procedure. Similarly, a single physical switch could be denoted as an arrangement of virtual switches, each of them belonging to a separate virtual network. These two examples of abstract network topologies would be substantially harder to execute with low-level guideline sets. In contrast, a programming language or runtime system can more easily provide abstractions for virtual network topologies, as has already been demonstrated by languages, for example, Pyretic.

Programming languages proposed for SDNs are FatTire, Flog, FlowLog, FML, Frenetic, HFT, Maple, Merlin, nlog, NetCore, NetKAT, Nettle, Procera, Pyretic. Most of the proposed languages offer abstractions for OpenFlow-empowered networks. The predominant programming pattern is the declarative one, with a single exemption, Pyretic, which is an imperative language. Most declarative languages are functional, while however there are instances of the logic and reactive types. The purpose i.e., the specific issues they plan to address and the expressiveness control fluctuate from language to language, while the final objective is quite often the same which is to give higher-level abstractions to facilitate the improvement of network control logic.

Programming languages, for example, FML, Nettle, and Procera are functional and reactive. Policies and applications written in these languages depend on reactive activities triggered by events (e.g., new host associated with the network, or the present network load). Such languages permit users to declaratively express distinct network configuration rules, for example, access control lists (ACLs), virtual LANs (VLANs), and numerous others. Rules are basically communicated as permit-or-deny policies, which are connected to the forwarding components to guarantee the desired network performance.

Other SDN programming languages, for example, Frenetic, Hierarchical Flow Tables (HFT), NetCore, and Pyretic, were designed with the concurrent objective of proficiently representing packet-forwarding policies and managing overlapping rules of various applications, offering advanced operators for parallel and sequence composition of software modules. To abstain from overlapping conflicts, Frenetic disambiguates rules with overlapping designs by conveying various integer needs, while HFT utilizes hierarchical policies with upgraded conflict determination operators.

FatTire is a case of a declarative language that intensely depends on normal expressions to permit software engineers to depict network paths with fault tolerance requirements. For example, each flow can have its own optional paths for managing failure of the primary paths. Interestingly, this feature is given in an extremely programmer friendly manner, with the application developer having just to utilize standard expressions with unique characters, for example, a reference bullet. Generally, Programming languages, for example, FlowLog and Flog bring diverse elements, for example, model checking, dynamic verification and stateful middleboxes. For example, utilizing a programming language such as Flog, it is possible to develop a stateful firewall application with just five lines of code. Curiously, Merlin is one of the main cases of unified framework for controlling distinctive network components such as forwarding devices, middleboxes, and end-hosts. An essential favorable position is backward compatibility with existing systems. To accomplish this objective, Merlin creates specific code for each kind of component. Taking an approach definition as input, Merlin's compiler decides forwarding paths, transformation placement, and bandwidth allocation.

Other recent activities (e.g., systems programming languages) target issues such as identifying anomalies to enhance the security of network protocols (e.g., Open-Flow), and optimizing horizontal scalability for accomplishing high throughput in applications running on multicore architectures. By and by, there is still scope for further examination and development on programming languages. For example, one recent research has uncovered that policy compilers produce unnecessary (redundant) rule updates, the majority of which alter only the priority field.

The greater part of the estimation of SDN will originate from the network managements applications based on top of the infrastructure. Progresses in high-level programming languages are a principal component to the achievement of a productive SDN application development environment. To this end, efforts are experiencing to shape imminent standard interfaces and towards the realization of integrated development environments (e.g., NetIDE) with the objective of encouraging the development of a myriad of SDN applications.

4.8. Layer 8 -- Network Applications

Network applications can be viewed as the "network brains". They implement the control logic that will be interpreted into instructions to be installed in the data plane, dictating the behavior of the forwarding devices. Take a basic application as routing for instance. The logic of this application is to define the path through which packets will flow out of a point A to a point B. To accomplish this objective a routing application needs to, in view of the topology input, decide on the path to utilize and instruct the controller to install the corresponding forwarding rules in all forwarding devices on the picked path, from A to B.

Software-defined networks can be deployed on any conventional network environment, from home and enterprise networks to data centers and Internet exchange points. Such diversity of environments has prompted to a wide exhibit of network applications. Existing network applications perform traditional functionalities, for example, routing, load balancing, and security policy enforcement, additionally investigate novel methodologies such as reducing power consumption. Other cases include fail-over and reliability functionalities to the data plane, end-to-end QoS enforcement, network virtualization, mobility management in wireless networks, among numerous others. The assortment of network applications, combined with real use case deployments, is relied upon to be one of the significant strengths on fostering a wide adoption of SDN.

Regardless of the wide assortment of use cases, most SDN applications can be gathered in one of five classes: traffic engineering, mobility and wireless, measurement and monitoring, security and dependability and data center networking.

4.8.1. Debugging and troubleshooting

Debugging and troubleshooting have been critical subjects in computing infrastructures, parallel and distributed systems, embedded systems, and desktop applications. The two predominant methodologies connected to debug and troubleshoot are runtime debugging (e.g., gdb-like tools) and post-mortem analysis (e.g., tracing, replay and visualization). Despite the consistent development and the rise of new techniques to enhance debugging and troubleshooting, there are yet a few open avenues and research questions.

Debugging and troubleshooting in networking is at an extremely primitive stage. In conventional networks, engineers and developers need to utilize tools, for example, ping, traceroute, tcpdump, nmap, netflow, and SNMP statistics for debugging and troubleshooting. Debugging a complex network with such primitive tools is hard. Anyhow when one considers structures, for example, XTrace, Netreplay and NetCheck , which enhance debugging capabilities in networks, it is still hard to troubleshoot networking infrastructures. For instance, these systems require an enormous effort as far as network instrumentation. The extra complexity introduced by various sorts of devices, technologies and vendor specific components and feature make matters worse. Thus, these arrangements may find it difficult to be generally executed and deployed in current networks.

SDN offers some hope in this regard. The hardware-agnostic software-based control capacities and the utilization of open standards for control communication can possibly make debug and troubleshoot less demanding. The flexibility and programmability introduced by SDN is indeed opening new avenues for developing better tools to debug, troubleshoot, verify and test networks. Early debugging tools for OpenFlow-empowered networks, for example, ndb, OFRewind and NetSight, make it less demanding to discover the source of network issues such as faulty device firmware, inconsistent or non-existing flow rules, lack of reachability and faulty routing. So also to the outstanding gdb software debugger, ndb gives essential debugging activities such as breakpoint, watch, backtrace, single-step, and continue. These primitives assist application developers with debugging networks in a comparable manner to traditional software. By using ndb's postcards (i.e., a unique packet identifier composed of a truncated duplicate of the packet's header, the matching flow entry, the switch, and the output port), for instance, a programmer can rapidly distinguish and separate a buggy OpenFlow switch with hardware or software issues. If the switch is presenting strange behavior, for example, corrupting parts of the packet header, by analyzing the problematic flow sequence with a debugging tool one can find (in a matter of few moments) where the packets of a flow are being corrupted, and take the essential activities to solve the issue.

In spite of the accessibility of these debugging and confirmation tools, it is yet hard to answer questions like what is happening to my packets that are flowing from point A to point B? What path do they take follow? What header changes do they experience in transit? To answer some of these inquiries one could recur to the history of the packets. A packet's history relates to the paths it uses to traverse the network, and the header alterations in each hop of the path. NetSight is a platform whose primary objective is to permit applications that utilization the history of the packets to be built, with a specific end goal to find out issues in a network. This platform is a composition of three fundamental components: (1) NetSight, with its devoted servers that get and process the postcards for building the packet history, (2) the NetSigh-SwitchAssist, which can be utilized as a part of switches to diminish the processing burden on the dedicated servers, and (3) the NetSight-HostAssist to create and handle postcards on end hosts (e.g., in the hypervisor on a virtualized infrastructure).

4.8.2. Testing and verification

Tools utilized for testing and verification can supplement debugging and troubleshooting. Recent research has demonstrated that verification techniques can be connected to recognize and evade issues in SDN, for example, forwarding loops and black holes. Verification can be done at various layers (at the controllers, network applications, or network devices). Also, there are distinctive network properties, mostly topology-specific that can be formally verified, given a network model is accessible. Cases of such properties are connectivity, loop freedom, and access control. Various tools have additionally been proposed to assess the performance of OpenFlow controllers by emulating the load of large-scale networks (e.g., Cbench, OFCBenchmark, PktBlaster). Thus, benchmarking tools for OpenFlow switches are also accessible (e.g., OFLOPS, FLOPS-Turbo).

Tools like NICE produce sets of assorted streams of packets to test as many as possible events, exposing corner cases, for example, race conditions. Likewise, OFLOPS gives an arrangement of features and functions that permit the improvement and execution of a rich arrangement of tests on OpenFlow-empowered devices. Its definitive objective is to measure the processing limit and bottlenecks of control applications and forwarding devices. With this tool, clients can observe and assess forwarding table consistency, flow setup latency, flow space granularity, packet modification types, and traffic monitoring capacities (e.g., counters). In addition to this, FlowChecker, OFTEN, and VeriFlow are three cases of tools to verify accuracy properties violation on the system. While the former two depend on offline analysis, the last is fit for online checking of network invariants. verification constraints include security and reachability issues, configuration on the network, loops, black holes, and so forth.

Recently, tools like VeriCon have been intended to verify the accuracy of SDN applications in a large scope of network topologies and by analyzing a broad range of sequences of network events. Specifically, VeriCon verifies, or not, the right execution of the SDN program. One of the difficulties in testing and verification is to verify forwarding tables in large networks to find routing mistakes, which can bring about traffic losses and security breaches, as fast as could be expected under the circumstances. In large scale networks, it is unrealistic to accept that the network snapshot, anytime, is steady, because of the frequent changes in routing state. Therefore, solutions like HSA, Anteater, NetPlumber, Veri-Flow, and assertion languages are not suited for this kind of condition. Another critical issue is connected on how quick the verification procedure is done, particularly in present day data centers that have tight timing prerequisites. Libra denotes one of the primary efforts to address these specific difficulties of large scale networks. This tool gives the way to capturing stable and consistent previews of large scale network deployments, while likewise applying long prefix matching techniques to increase the scalability of the system. By using MapReduce calculations, Libra is fit for verifying the correctness of a network with up to 10k nodes within one minute.

4.8.3. Simulation and Emulation

Mininet is the principal system that gives a fast and simple approach to model and assess SDN protocols and applications. One of the key properties of Mininet is its utilization of software-based OpenFlow switches in virtualized containers, giving the same semantics of hardware-based OpenFlow switches. This implies controllers or applications created and tested in the emulated environment can be (in theory) deployed in an OpenFlow-empowered network with no change. Users can simply emulate an OpenFlow network with several nodes and many switches by utilizing a solitary PC. Mininet-Hi is an advancement of Mininet that improves the container-based (lightweight) virtualization with components to enforce performance isolation, resource provisioning, and accurate monitoring for performance fidelity. One of the primary objectives of Mininet-HiFi is to enhance the reproducibility of networking research.

To simulate bigger scale network, Mininet has two augmentations called Mininet CE and SDN Cloud DC. Mininet CE joins groups of Mininet instances into one group of simulator instances to prototype global scale networks. SDN Cloud DC upgrades Mininet and POX to imitate a SDN-based intra-DC network by implementing new software modules, for example, data center topology discovery and network traffic generation. Recent simulation platform propositions that empower extensive scale tests following a distributed approach incorporate MaxiNet and CityFlow. The latter is a venture with the principle objective of building an emulated control plane for a city of one million occupants. Such activities are a beginning stage to give trial experiences to vast scale SDN deployments.

The capacity of simulating OpenFlow devices has additionally been added to the well-known ns-3 simulator. Similarly, simulator called fs-sdn which develops the fs simulation engine by consolidating a controller and switching segments with OpenFlow support. Its principle objective is to give a more realistic and scalable simulation platform when contrasted with Mininet. At long last, STS is a test system intended to permit developers to determine and apply an assortment of experiments, while permitting them to intuitively inspect the state of the network.

5. Utilization of traditional protocols to deploy SDN approach

A traditional approach to enhance routing inside networks (for quick, robust, reliable and efficient communication) focus on distributed networking instead of centralized and try to make network devices more and more intelligent. However, this approach has made devices more complex and network less efficient. There are certain issues (as mentioned below) which SDN addresses in a better way.

- Least or no interaction between various protocols causes problem of Black holing. For example, OSPF & LDP sync.
- Partial network visibility results in inefficient use of resources. For example, implementing Traffic engineering between two autonomous system.
- Interaction between devices is not efficient which causes troubles while scaling the network.
- One service/functionality is achieved with the help of various protocols which makes it difficult to troubleshoot.

To overcome from these problems, we have can have hybrid approach which is blend of both distributed and centralized techniques. Network device’s control plane i.e. some part of intelligence of network device (usually router) can be separated and give it to controller who has more visibility of network. Nowadays, commonly used techniques like Traffic Engineering, Segment routing, etc. is deployed with centralized (i.e. SDN approach) system for easier administration and better results.

5.1. Interior Gateway Protocol (IGP)

IGP is used to exchange routing information between networking devices within an Autonomous System(AS). It is divided into two types based on the method of computing the best path to a destination.

- Link-state protocols—Advertise data about the network topology (directly connected links and the state of those links) to all routers utilizing multicast addresses and triggered routing updates until every one of the routers running the link-state protocol have indistinguishable data about the internetwork. The best path to a destination is computed in view of imperatives, for example, maximum delay, minimum available bandwidth, and resource class affinity. OSPF and IS-IS are cases of link-state protocols.
- Distance vector protocols—Advertise complete routing table data to directly connected neighbors utilizing a broadcast address. The best path is computed in view of the number of hops to the final network. RIP is a case of a distance vector protocol.

Previously, it was Internal gateway protocol (IGP) like OSPF, IS-IS used as underlying technology to transfer link-state, network topology information which enables features like TE - Traffic engineering, etc. IGP’s main role is to give routing connectivity within or internal to a given routing domain. An AS (Autonomous System) can comprise of various routing domains, where IGP capacities to advertise and learn network prefixes (routes) from neighboring routers to construct a route table that eventually contains entries for all sources advertising reachability for a given prefix. IGP executes a route selection algorithm to choose the best path between the local router and every destination, and gives full connectivity among the routers making up a routing domain. In addition to advertising internal network reachability, IGPs are frequently used to advertise routing data that is outside to that IGP's routing domain through a procedure known as route redistribution. Route redistribution is a method of trading routing data among distinct routing protocols to group different routing domains when intra-AS connectivity is sought.

However, IGP has many limitations such as performance and scalability. Though IGP is utilized to distribute network topology information, but it only works until used for small scale networks. IGPs can autodetect neighbors, with which they obtain intra-area network topology data. In any case, the link-state database or a traffic engineering database has the extent of a solitary area or AS, accordingly restricting applications, for example, end-to-end traffic engineering, the advantage of having outside visibility to settle on better choices. Mostly traffic engineering techniques work in a single routing domain. For example, MPLS, GMPLS, etc. These arrangements don't work when a route from the ingress node to the egress node leaves the routing area or AS of the ingress node. In such cases, the path computation issue gets to be distinctly convoluted due to the inaccessibility of the entire routing data all through the network. This is because service providers usually decide not to spill routing data beyond the routing area or AS for scalability constraints and confidentially concerns.

5.2. Border Gateway Protocol (a successor to Exterior Gateway Protocol)

Usually referred as BGP, is routing protocol used to exchange routing and reachability information between two devices on different autonomous systems irrespective of IGP running within Autonomous systems. BGP is not just a more scalable vehicle for conveying multi-area and multi-AS topology data, however also gives the policy controls that can be valuable for multi-AS topology distribution. BGP is the only protocol which can carry all types of routes in Internet. It also compatible with TCP and utilize flow control feature of TCP whereas IGPs are not capable of doing it.

6. Need of BGP-LS (Border Gateway Protocol – Link state)

There are certain problems like optimal path computation, bin-packing with traditional traffic engineering approach when utilized for Multi-Area/Multi-AS traffic engineering. This is because of head end router has limited visibility of in single AS or single IGP area regardless of whether it's the quantity of LSPs in-flight (identified with bin-packing) or LSDB for different areas (AS or IGP Area). These issues are difficult to comprehend with distributed computation and it bodes well to move LSP path computation for these sorts of issues to a central controller which has visibility to the whole domain or more than one domain which permits it to calculate the paths effectively, which then can be signaled by the controller to the head-end node about the path which is end to end optimal. Even popular implementations like Segment Routing, Traffic engineering use the centralized path controller to calculate the path and then providing it to the head end node. However, to compute path and send it to node, a central controller usually relies on TED (traffic engineering database). Information about topology and about link like available bandwidth, link metric, TE metric, link bandwidth, etc. helps controller to construct TED.

Generally, it can be achieved by implementing IGP as well as BGP. Initially, IGPs were utilized (as explained above). However, it had several problems. For instance, IGP is very chatty, therefore more resources and processing time was utilized by controller to handle updates. In addition to this, controller had to support both OSPF and IS-IS. Last but not the least, placement of controller in large-scale networks was another big issue.

On the other hand, BGP does not alter underlying protocol for utilizing type, length, value (TLV) tuples and network layer reachability information (NLRI) that provide apparently limitless extensibility. Therefore, the basic idea was to extend BGP by developing new NLRI which could be utilized to carry all IGPs information over BGP.

Hence, BGP-LS, Border gateway protocol – Link state was introduced to distribute link state information across various AS without breaching the security of ISPs. BGP speaker retrieves information from IGP Link-state databases (i.e. information about nodes and links which is local/remote IP addresses, local/remote interface identifiers, link metric and TE metric, link bandwidth, reservable bandwidth, per Class-of-Service (CoS) class reservation state, preemption, and Shared Risk Link Groups - SRLGs) and distribute it to a controller which could be done either directly or with the help of a peer. There is an option to apply a policy to information which gets distributed by BGP Speaker. Physical topology captured from LSDB or TED maybe distributed directly or BGP speaker may create an abstracted topology where virtual, aggregated nodes are connected by virtual paths. Abstracted topology can consist of physical/virtual links and nodes. BGP speaker can also filter the information before forwarding it to the northbound of controller.

```
                  +----------+                           +---------+
                  |  -----   |                           |   BGP   |
                  | | TED |<-+-------------------------->| Speaker |
                  |  -----   |   TED synchronization     |         |
                  |    |     |        mechanism:         +---------+
                  |    |     | BGP with Link-State NLRI
                  |    v     |
                  |  -----   |
                  | | PCE |  |
                  |  -----   |
                  +----------+
                       ^
                       | Request/
                       | Response
                       v
    Service       +----------+   Signaling  +----------+
    Request       | Head-End |   Protocol   | Adjacent |
    ------------->|  Node    |<------------>|   Node   |
                  +----------+              +----------+
```

Now, with BGP-LS, lesser resources and processing time utilized by controller as there are not much update because BGP is not too chatty. Moreover, controller needs to support only BGP. Lastly, forming BGP peers are easier than IGP peers which makes it more scalable than IGP based implementation.

6.1. Carrying Link-State Information in BGP

IGP comprises of topology and IP reachability data which is captured by BGP to update its own database so that BGP could forward information to centralized controller. In this case, controller does not need to have deployment of any IGP (e.g. OSPF, IS-IS, etc.). This is achieved with the help of following two parts.

- New BGP NLRI (describes links, nodes and prefixes comprising IGP link-state)
- New BGP path attribute (carries attributes and properties of link, node and prefix)

TLV Format

Type/Length/Value triplets denote the information regarding new BGP Link-state NLRIs and BGP attributes as shown in figure below.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                       Value (variable)                      //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

6.1.1. The Link-State NLRI

Each NLRI represents one of following:

- Node
- Link
- Prefix

Type	NLRI Type
1	Node NLRI
2	Link NLRI
3	IPv4 Topology Prefix NLRI
4	IPv6 Topology Prefix NLRI

Node - NLRI Type 1

The format of Node NLRI (NLRI Type = 1) is shown in the figure beneath:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+
|  Protocol-ID  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Identifier                          |
|                            (64 bits)                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                Local Node Descriptors (variable)            //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

NLRI’s format contains mainly three fields which are Protocol-ID, Identifier (which is of 64 bits) and Local Node Descriptors (variable length).

Link - NLRI Type 2

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+
|  Protocol-ID  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Identifier                          |
|                            (64 bits)                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//               Local Node Descriptors (variable)             //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//               Remote Node Descriptors (variable)            //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                  Link Descriptors (variable)                //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Format of Link NRLI (NRLI Type = 2) is as shown in the figure above and it contains five fields: Protocol – ID, Identifier which is same as NRLI = 1, 64bits. It has three fields with variable length and they are Local Node Descriptors, Remote Node Descriptors, and Link Descriptors.

Prefix – NLRI Type 3/4

Next is IPv4 Prefix NLRI (NLRI Type = 3) which has the same format as of IPv6 Prefix NLRI (NLRI Type = 4) and it is shown in the following figure:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+
|  Protocol-ID  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Identifier                          |
|                            (64 bits)                          |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//              Local Node Descriptors (variable)              //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                Prefix Descriptors (variable)                //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Also, both NLRI Type = 3 and NLRI Type = 4 have the same format as of NRLI = 1 except just one filed which is Prefix Descriptors with variable length.

Moving Further, the Protocol-ID field has different values as shown below:

Protocol-ID	NLRI information source protocol
1	IS-IS Level 1
2	IS-IS Level 2
3	OSPFv2
4	Direct
5	Static configuration
6	OSPFv3

At the point when BGP-LS is sourcing local information, then "Direct" and 'Static Configuration' protocols sorts ought to be utilized. ‘Direct’ Protocol-ID ought to be utilized as a part of the situation where BGP-LS has direct access to the interface data and it needs to promote a nearby connection. Also, "Static" Protocol-ID ought to be utilized as a part of the situation with virtual connections.

“Routing Universes” can be characterized as the protocol instances in which protocols such as OSPF and IS-IS may run multiple protocol instances over the same link. As it is appeared in the figure, that the Identifier field is of 64-bits and it is used in order to identify these routing universe. It can be concluded from the

Identifier	Routing Universe
0	Default Layer 3 Routing topology

‘Identifier’ field given by NRLIs that the link-state objects (nodes, links, or prefixes) belong to the same routing universe or not. For instance, if NRLI with same ‘Identifier’ field value then link-state objects must be thought to be from same routing universe, however with different ‘Identifier’ field value, the link-objects belong to different routing universe.

Besides, if any of the given protocol sets the Identifier field as indicated by Table underneath, then it implies that it does not support multiple routing universes.

6.1.1.1. Node Descriptors

A pair of Router-IDs are used by the underlying IGP for each link, for instance, for IS-IS there is a 48-bit ISO system-ID and 32-bit Router-ID for OSPFv2 and OSPFv3. In case of Traffic Engineering, an IGP may use more than one auxiliary Router-IDs. For example, IS-IS may has one or more IPV4 and IPV6 TE Router-IDs.

It is attractive that the Router-ID assignments inside the Node Descriptor are all inclusive remarkable. There might be Router-ID spaces (e.g., ISO) where no worldwide registry exists, or more regrettable, Router-IDs have been designated following the private-IP distribution. BGP-LS utilizes the Autonomous System (AS) Number, furthermore, BGP-LS Identifier to disambiguate the Router-IDs.

6.1.1.1.1. Globally Unique Node/Link/Prefix Identifiers

One issue that should be tended to is the ability to recognize an IGP node comprehensively and this can be achieved with the following:

- In order to prevent one node looks like two nodes, it should be noted that same node should not be represented by two keys.
- This is the opposite case of I, two nodes will not look like one node so it must not be represented by the same key.

We characterize an "IGP area" to be the arrangement of nodes inside which every node has a one of a kind IGP representation by utilizing the mix of Area-ID, Router-ID, Protocol ID, Multi-Topology ID, and Instance-ID. The issue is that BGP may get node/link/prefix data from numerous autonomous "IGP areas", and we must recognize them.
In addition, we can't expect there is constantly one and just a single IGP area per AS. Amid IGP moves, it might happen that two repetitive IGPs are set up.

6.1.1.1.2. Local Node Descriptors

Node Descriptors are present in the Local Node Descriptors TLV which is responsible for the anchoring the nearby end of the connection. As there are three sorts of NLRIs (node, link, and prefix) and three of them requires TLV. As shown in the figure, the

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
//              Node Descriptor Sub-TLVs (variable)            //
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

length of the TLV is variable and it has at least one Node Descriptor Sub-TLVs.

6.1.1.1.3. Remote Node Descriptors

Node Descriptors are present in the Remote Node Descriptors TLV which is responsible for the anchoring the remote end of the

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                                                               |
//              Node Descriptor Sub-TLVs (variable)            //
|                                                               |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

connection. This is a required TLV for link NLRIs. The length of this TLV is variable. The esteem contains at least one Node Descriptor Sub-TLVs characterized as in the figure above.

6.1.1.1.4. Node Descriptor Sub-TLVs

The Node Descriptor Sub-TLVs Code points, Description and Lengths are illustrated below:

Sub-TLV Code Point	Description	Length
512	Autonomous System	4
513	BGP-LS Identifier	4
514	OSPF Area-ID	4
515	IGP Router-ID	Variable

The Sub-TLVs Code points can be defined as follows:

- **Autonomous System:** It has an opaque value with 32-bit AS Number.
- **BGP-LS Identifier:** It has an opaque value with 32-bit ID. In conjunction with Autonomous System Number (ASN), exceptionally distinguishes the BGP-LS area. The mix of ASN and BGP-LS ID MUST be comprehensively extraordinary. All BGP-LS speakers inside an IGP flooding-set (arrangement of IGP nodes inside which a LSP/LSA is overwhelmed) MUST utilize the same ASN, BGP-LS ID tuple. If an IGP area comprises of numerous flooding-sets, then all BGP-LS speakers inside the IGP space ought to utilize the same ASN, BGP-LS ID tuple.
- **Area-ID:** It is used to distinguish the 32-bit region to which the NLRI belongs. The Area Identifier permits diverse NLRIs of a similar router to be segregated.

- **IGP Router-ID:** This is a compulsory TLV. For an IS-IS non-pseudo-node, this contains a 6-octet ISO Node-ID. For an IS-IS pseudo-node comparing to a LAN, this contains the 6-octet ISO Node-ID of the Designated Intermediate System (DIS) trailed by a 1-octet, nonzero PSN identifier (7 octets altogether).
  The TLV size in combination with the protocol identifier empowers the decoder to decide the sort of the node. There can be at most one example of each sub-TLV sort introduce in any Node Descriptor. The sub-TLVs inside a Node Descriptor must be masterminded in rising request by sub-TLV sort. This should be done with a specific end goal to look at NLRIs, notwithstanding when a usage experiences an obscure sub-TLV. Utilizing stable sorting, an execution can do parallel correlation of NLRIs and thus permit incremental organization of new key sub-TLVs.

6.1.1.1.5. Multi-Topology ID

The Multi-Topology ID (MT-ID) TLV carries one or more IS-IS or OSPF Multi-Topology IDs for a link, node, or prefix. The format is as shown in the figure below.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |          Length=2*n           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|R R R R|  Multi-Topology ID 1  |             ....            //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//             ....             |R R R R|  Multi-Topology ID n  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The type of MT-IS is 263, Length is 2*n where n is the number of MT-IDs carried in the TLV. The MT-ID TLV may be present in a Link Descriptor (single MT-ID containing topology), a Prefix Descriptor, or the BGP-LS attribute (one MT-ID TLV which contains the array) of a Node NLRI.

6.1.1.2. Link Descriptors

The Link Descriptor field is an arrangement of Type/Length/Value (TLV)triplets. A connection depicted by the Link Descriptor TLVs really is a "half-link", a unidirectional portrayal of a logical link. In request to completely portray a solitary sensible connection, two originating routers promote a half-link each, i.e., two Link NLRIs are publicized for a given point-to-point interface.

TLV Code Point	Description	IS-IS TLV /Sub-TLV
258	Link Local/Remote Identifiers	22/4
259	IPv4 interface address	22/6
260	IPv4 neighbor address	22/8
261	IPv6 interface address	22/12
262	IPv6 neighbor address	22/13
263	Multi-Topology Identifier	---

On the off chance that interface and neighbor addresses, either IPv4 or IPv6, are available, then the IP address TLVs are incorporated into the Link Descriptor yet not the connection link/remote Identifier TLV. The connection local/remote identifiers may be incorporated into the connection characteristics.

If the interface and neighbor locations are absent and the connection nearby/remote identifiers are available, then the link local/remote Identifier TLV is incorporated into the Link Descriptor. The Multi-Topology Identifier TLV is incorporated into Link Descriptor if that data is available.

6.1.1.3. Prefix Descriptors

The Prefix Descriptor field is a set of Type/Length/Value (TLV) triplets. Prefix Descriptor TLVs uniquely identify an IPv4 or IPv6 prefix originated by a node.

TLV Code Point	Description	Length
263	Multi-Topology Identifier	variable
264	OSPF Route Type	1
265	IP Reachability Information	variable

6.1.1.3.1. OSPF Route Type

The OSPF Route Type TLV is a discretionary TLV that MAY be available in Prefix NLRIs. It is utilized to recognize the OSPF route type of the prefix. It is utilized when an OSPF prefix is publicized in an OSPF area with numerous route sorts.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |            Length             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|  Route Type   |
+-+-+-+-+-+-+-+-+
```

In the format shown above, the type of the OSPF route is defined in OSPF Protocol and they can be one of the listed following:

- Intra-Area (0x1)
- Inter-Area (0x2)
- External 1 (0x3)
- External 2 (0x4)
- NSSA 1 (0x5)
- NSSA 2 (0x6)

6.1.1.3.2. IP Reachability Information

It is mandatory TLV which consists one IP address prefix (IPv4 or IPv6) which could be advertised in the topology for IGP. IP Reachability format is as following:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |            Length             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
| Prefix Length | IP Prefix (variable)                         //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

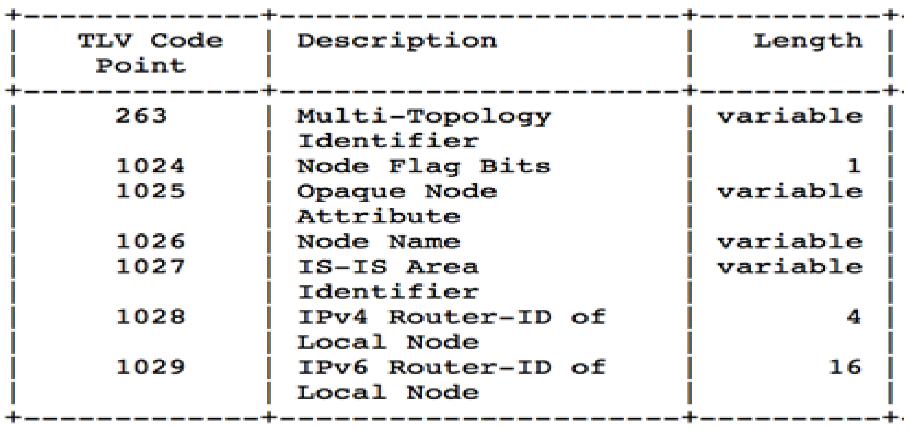
The Prefix Length field contains the length of the prefix in bits. The IP Prefix field contains the most significant octets of the prefix, i.e., 1 octet for prefix length 1 up to 8, 2 octets for prefix length 9 to 1, etc.

6.1.2. BGP-LS Attribute

The BGP-LS property is a discretionary, non-transitive BGP trait that is utilized to convey link, node, and prefix parameters and properties. It is characterized as an arrangement of Type/Length/Value (TLV) triplets. This property should just be incorporated with Link-State NLRIs. This characteristic must be disregarded for all different address families.

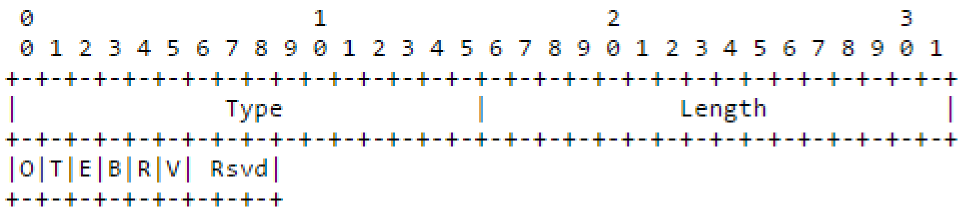
6.1.2.1. Node Attribute TLVs

These are the type of TLVs which are encoded in the BGP-LS attribute along Node NLRI as shown below:



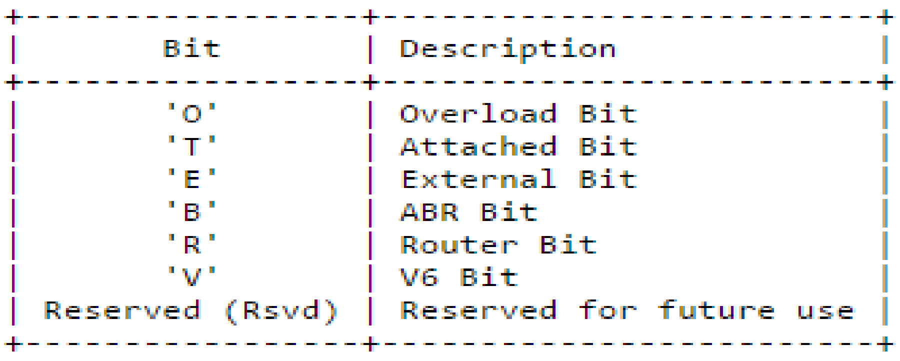
TLV Code Point	Description	Length
263	Multi-Topology Identifier	variable
1024	Node Flag Bits	1
1025	Opaque Node Attribute	variable
1026	Node Name	variable
1027	IS-IS Area Identifier	variable
1028	IPv4 Router-ID of Local Node	4
1029	IPv6 Router-ID of Local Node	16

- Node Flag Bits TLV: It carries a bit mask which describes note attributes and it has a clue of variable length.



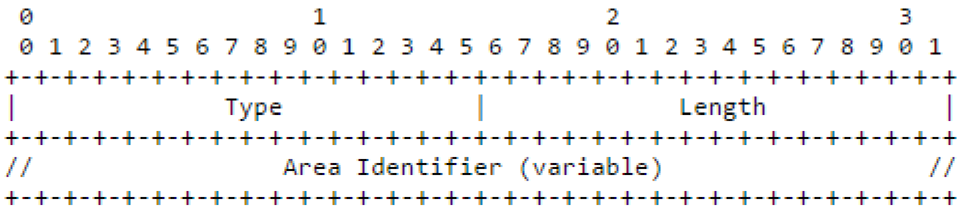
```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|O|T|E|B|R|V| Rsvd|
+-+-+-+-+-+-+-+-+-+
```

Following picture shows how bits are defined:



Bit	Description
'O'	Overload Bit
'T'	Attached Bit
'E'	External Bit
'B'	ABR Bit
'R'	Router Bit
'V'	V6 Bit
Reserved (Rsvd)	Reserved for future use

- IS-IS Area Identifier TLV: IS-IS TLVs are used to encode area addresses as an IS-IS node can be a part of one or more IS-IS areas.



```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                  Area Identifier (variable)                  //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Node Name TLV: This is an optional TLV in which the Value Feels identifies the symbolic name of the router node and the maximum length of the Node Name TLV is 255 octets.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                     Node Name (variable)                    //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Local IPV4/IPv6 Router-ID TLVs: These are the TLVs which are used for describing auxiliary Router-IDs, for instance: correlating a Node-ID between different protocols for Traffic Engineering.
- Opaque Node Attribute TLV: It is a kind of envelope which helps in carrying optional Node Attribute TLVs which are advertised by a router. This TLV can be used for encoding information. Also, the primary task of this TLV is to bridge the document lag between routers.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//               Opaque node attributes (variable)             //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

6.1.2.2. Link Attribute TLVs

These are the type of TLVs that may be encoded in the BGP-LS attribute with a Link NLRI. It is in a triplet format of Type/Length/Value (TLV). Originally Link Attributes were defined for carrying IS-IS encoding but it can be used for OSPF as well.

TLV Code Point	Description	IS-IS TLV /Sub-TLV
1028	IPv4 Router-ID of Local Node	134/---
1029	IPv6 Router-ID of Local Node	140/---
1030	IPv4 Router-ID of Remote Node	134/---
1031	IPv6 Router-ID of Remote Node	140/---
1088	Administrative group (color)	22/3
1089	Maximum link bandwidth	22/9
1090	Max. reservable link bandwidth	22/10
1091	Unreserved bandwidth	22/11
1092	TE Default Metric	22/18
1093	Link Protection Type	22/20
1094	MPLS Protocol Mask	---
1095	IGP Metric	---
1096	Shared Risk Link Group	---
1097	Opaque Link Attribute	---
1098	Link Name	---

- IPv4/IPv6 Router-ID TLVs: These TLVs are used for auxiliary Router-IDs that the IGP might be using. For instance, For Traffic Engineering.

- MPLS Protocol Mask TLV: These TLVs are used for describing about enabled MPLS signaling protocols with the help of bit mask. The value and length of this TLV is 8flags and 1bit.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|L|R|  Reserved |
+-+-+-+-+-+-+-+-+
```

Bits are defined as picturized below:

Bit	Description
'L'	Label Distribution Protocol (LDP)
'R'	Extension to RSVP for LSP Tunnels (RSVP-TE)
'Reserved'	Reserved for future use

- TE Default Metric TLV: This TLV is responsible for efficient Traffic Engineering with the help of Metric. It has a fixed length which is 4 octets and it offers padding as well to protocol with a metric width less than 32 bits.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                    TE Default Link Metric                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- IGP Metric TLV: This TLV has variable length which depends on the metric width of the protocol. Different protocols have different metric length, for instance IS-IS has a length of 1 and 3 octets and OSPF has 2 octets.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//      IGP Link Metric (variable length)      //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

- Shared Risk Link Group TLV: The Shared Risk Link Group TLV carries the Shared Risk Link Group Information. It consists of a data structure which has a list of SRLG values with elements having 4 octets as depicted in the figure below:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Shared Risk Link Group Value                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                         ............                        //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                  Shared Risk Link Group Value                 |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

  The length of this TLV is 4 (number of SRLG values). The way the SRLG values are being carried by TLVs are different for different protocols such as for IS-IS, it has been carried in two TLVs which are IPv4 TLV and IPv6 SRLG TLV.

- Opaque Link Attribute TLV: The Opaque Link Attribute TLV is an envelope that straightforwardly conveys discretionary Link

Attribute TLVs promoted by a switch/router. An originating switch/router should utilize this TLV for encoding data particular to the protocol promoted in the NLRI header Protocol-ID field.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                Opaque link attributes (variable)            //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

The essential utilization of the Opaque Link Attribute TLV is to connect the document lag between, e.g., a new IGP link-state attribute being defined and the 'protocol-impartial' BGP-LS extensions being published.

- Link Name TLV: This TLV is an Optional TLV. In this the router link, can be defined by the Value Field as shown in the diagram. This Value field is encoded in 7-bit ASCII.

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                     Link Name (variable)                    //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

This field gives User the functionality to applying Unicode Characters to ASCII in order to fulfill the criteria for correct format for transmission of display.

6.1.2.3. Prefix Attribute TLVs

Prefixes are gained from the IGP topology (IS-IS or OSPF) with a set of IGP properties (such as metric, route tags, etc.) that must be reflected into the BGP-LS attribute with a prefix NLRI. Prefix Attribute TLVs should be utilized when publicizing NLRI sorts 3 and 4only.

TLV Code Point	Description	Length
1152	IGP Flags	1
1153	IGP Route Tag	4*n
1154	IGP Extended Route Tag	8*n
1155	Prefix Metric	4
1156	OSPF Forwarding Address	4
1157	Opaque Prefix Attribute	variable

- IGP Flags TLV: In IGP Flags TLV, the bits are assigned to prefix and it mainly comprises of IS-IS and OSPF flags. The process of encoding in IGP Flags TLV is as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|D|N|L|P| Resvd.|
+-+-+-+-+-+-+-+-+
```

  And the Value Field is as following:

Bit	Description
'D'	IS-IS Up/Down Bit
'N'	OSPF "no unicast" Bit
'L'	OSPF "local address" Bit
'P'	OSPF "propagate NSSA" Bit
Reserved	Reserved for future use.

- IGP Route Tag TLV: The encoded process is as following in the figure and this TLV contains original IGP Tags:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                    Route Tags (one or more)                  //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

  The Value Field depends upon the IGP topology and it contains one or more route Tags. The length is always a multiple of 4.

- Extended IGP Route Tag TLV: IS-IS Extended Route Tags have been carried by the Extended IGP Route Tag TLV and the encoding process is as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//               Extended Route Tag (one or more)               //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

  It consists of one or more Extended Route Tags with the help of IGP topology and the length of this TLV must be a multiple of 8.

- Prefix Metric TLV: This is an optional TLV and if it appears it appears only once. With the help of IGP topology, it carries the metric of the prefix if it is present. Its format is as shown below:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                            Metric                             |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

  However, if it is not present the prefix is advertised without any reachability. The length of the Prefix TLV is 4.

- OSPF Forwarding Address TLV: It is responsible for carrying OSPF forwarding address and it can be wither IPV4 or IPV6. The format of OSPF Forwarding Address TLV is as following:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//                Forwarding Address (variable)                //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

  The length filed depends upon the Forwarding Address, for instance if the address is IPV4 the length is 4 and if it is IPV6 then it is 16.

- Opaque Prefix Attribute TLV: The Opaque Prefix Attribute TLV is an envelope that straightforwardly conveys optional Prefix Attribute TLVs publicized by a switch. A beginning switch should utilize this TLV for encoding data particular to the protocol publicized in the NLRI header Protocol-ID field. The format of Opaque Prefix Attribute TLV is as follows:

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|              Type             |             Length            |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
//              Opaque Prefix Attributes  (variable)           //
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

  Length of Opaque Prefix Attribute TLV is variable. The essential utilization of the Opaque Prefix Attribute TLV is to connect the document lag between, e.g., another IGP connect state quality being characterized and the protocol unbiased BGP-LS augmentations being distributed.

6.1.3. BGP Next-Hop Information

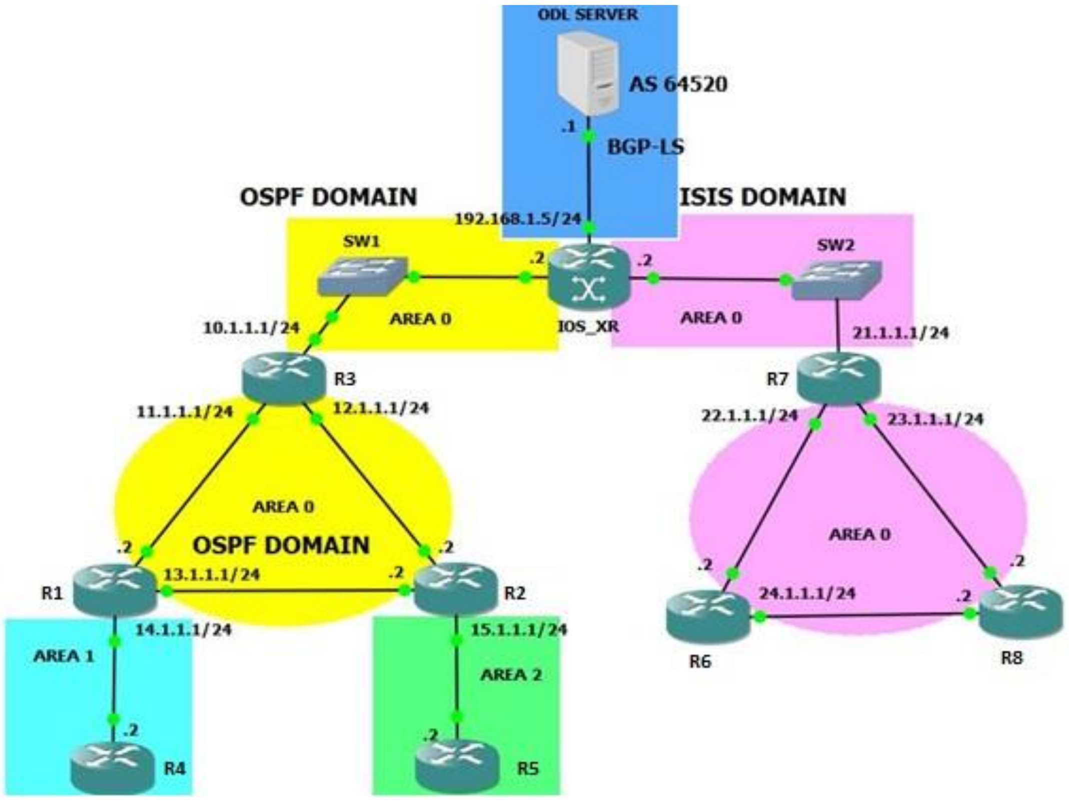
BGP interface state data for both IPv4 and IPv6 systems can be persisted either an IPv4 BGP session or an IPv6 BGP session. If an IPv4 BGP session is utilized, then the following bounce in the MP_REACH_NLRI.

Ought to be an IPv4 address. Likewise, if an IPv6 BGP session is utilized, then the following jump in the MP_REACH_NLRI SHOULD be an IPv6 address. Usually, the following hop will be set to the nearby endpoint address of the BGP session. The Length field of the following hop address will indicate the following hop address family. If the following hop length is 4, then the following hop is an IPv4 address; if the following hop length is 16, then it is a worldwide IPv6 address; and if the following hop length is 32, then there is one worldwide IPv6 address took after by a connection neighborhood IPv6 address.

The BGP Next Hop property is utilized by each BGP-LS speaker to approve the NLRI it gets. If different NLRIs are sourced by numerous originators, the BGP Next Hop credit is utilized to tiebreak per the standard BGP way choice process.

7. Utilizing IGPs and BGP-LS to deploy an SDN approach with OpenDayLight

As we are already aware, most organizations utilize routing protocols like OSPF, IS-IS for routing and configure routers individually which is considered as Distributed control approach. However, to minimize the hassle of distributed system, we deploy a SDN network (i.e. Centralized approach) which can be accomplished by using APIs like BGP Link State (BGP-LS), Segment Routing (SR), BGP FlowSec (BGP-FS), Path Computation Element Protocol (PCEP), and NETCONF/YANG or Controllers like Open Network Operation System (ONOS), OpenDaylight (ODL) and OpenStack.



In this implementation, I am going to use BGP-LS for layer 3 topology discovery of traditional network running OSPF as well as IS-IS in two separate domains and forward it to the controller (OpenDayLight). Deployed network is shown in the picture above.

7.1. Demo of an Organization's Network running OSPF and IS-IS

As shown in topology above, real-time network deployed in an organization to fulfill their needs by utilizing OSPF and IS-IS. Here is the basic configuration of these routers.

R1#show running-config

```
hostname R1
!
interface FastEthernet0/0
 ip address 14.1.1.1 255.255.255.0
 duplex auto
 speed auto
!
interface FastEthernet0/1
 ip address 13.1.1.1 255.255.255.0
 duplex auto
 speed auto
!
interface FastEthernet3/0
 ip address 11.1.1.2 255.255.255.0
 duplex auto
 speed auto
router ospf 1
 log-adjacency-changes
 network 11.1.1.0 0.0.0.255 area 0
 network 13.1.1.0 0.0.0.255 area 0
 network 14.1.1.0 0.0.0.255 area 1
end
```

R2#show running-config

```
hostname R2
!
interface FastEthernet0/0
 ip address 15.1.1.1 255.255.255.0
 duplex auto
 speed auto
!
interface FastEthernet0/1
 ip address 13.1.1.2 255.255.255.0
 duplex auto
 speed auto
```

```
!
interface FastEthernet3/0
 ip address 12.1.1.2 255.255.255.0
 duplex auto
 speed auto
!
router ospf 1
 log-adjacency-changes
 network 12.1.1.0 0.0.0.255 area 0
 network 13.1.1.0 0.0.0.255 area 0
 network 15.1.1.0 0.0.0.255 area 2
end
```

R3#show running-config

```
hostname R3
!
interface FastEthernet0/0
 ip address 11.1.1.1 255.255.255.0
 duplex auto
 speed auto
!
interface FastEthernet0/1
 ip address 10.1.1.1 255.255.255.0
 duplex auto
 speed auto
!
interface FastEthernet3/0
 ip address 12.1.1.1 255.255.255.0
 duplex auto
 speed auto
!
router ospf 1
 log-adjacency-changes
 network 10.1.1.0 0.0.0.255 area 0
 network 11.1.1.0 0.0.0.255 area 0
 network 12.1.1.0 0.0.0.255 area 0
end
```

R4#show running-config

```
hostname R4
!
interface FastEthernet0/0
 ip address 14.1.1.2 255.255.255.0
 duplex auto
 speed auto
!
router ospf 1
 log-adjacency-changes
 network 14.1.1.0 0.0.0.255 area 1
end
```

R5#show running-config

```
hostname R5
!
interface FastEthernet0/0
 ip address 15.1.1.2 255.255.255.0
 duplex auto
 speed auto
!
router ospf 1
 log-adjacency-changes
 network 15.1.1.0 0.0.0.255 area 2
end
```

R6#show running-config

```
hostname R6
!
interface FastEthernet0/0
 ip address 22.1.1.2 255.255.255.0
 ip router isis 0
 duplex auto
 speed auto
!
interface FastEthernet0/1
 ip address 24.1.1.1 255.255.255.0
 ip router isis 0
 duplex auto
 speed auto
!
router isis 0
 net 49.0000.0000.0000.0002.00
end
```

R7#show running-config

```
hostname R7
!
interface FastEthernet0/0
 ip address 21.1.1.1 255.255.255.0
 ip router isis 0
 duplex auto
 speed auto
!
interface FastEthernet0/1
 ip address 22.1.1.1 255.255.255.0
 ip router isis 0
 duplex auto
 speed auto
!
interface FastEthernet3/0
 ip address 23.1.1.1 255.255.255.0
 ip router isis 0
 duplex auto
 speed auto
!
router isis 0
 net 49.0000.0000.0000.0001.00
end
```

```
R8#show running-config
hostname R8
!
interface FastEthernet0/1
 ip address 24.1.1.2 255.255.255.0
 ip router isis 0
 duplex auto
 speed auto
!
interface FastEthernet3/0
 ip address 23.1.1.2 255.255.255.0
 ip router isis 0
 duplex auto
 speed auto
!
router isis 0
 net 49.0000.0000.0000.0003.00
end
```

7.2. BGP-LS Router Configuration

To implement SDN approach, a router/switch is required which supports OpenFlow protocol. Cisco IOS-XR router (supports OpenFlow) is being used to capture link state attributes of network and forward it to the controller. Following is the configuration of BGP-LS router.

```
RP/0/0/CPU0:IOS-XR#sh running-config
hostname IOS-XR
!
interface GigabitEthernet0/0/0/1
 ipv4 address 10.1.1.2 255.255.255.0
!
interface GigabitEthernet0/0/0/2
 ipv4 address 21.1.1.2 255.255.255.0
!
interface GigabitEthernet0/0/0/3
 ipv4 address 192.168.1.5 255.255.255.0
!
router isis 0
 net 49.0000.0000.0000.0004.00
 distribute bgp-ls
 interface GigabitEthernet0/0/0/2
  address-family ipv4 unicast
  !
router ospf 1
 distribute bgp-ls
 area 0
  interface GigabitEthernet0/0/0/1
  !
router bgp 64520
 bgp router-id 192.168.1.5
 address-family ipv4 unicast
 !
 address-family link-state link-state
 !
 neighbor 192.168.1.1
  remote-as 64520
  update-source GigabitEthernet0/0/0/0
  address-family link-state link-state
  end
```

7.3. OpenDayLight controller and BGP-LS feature Installation

OpenDayLight controller setup can be downloaded from opendaylight.org. Boron is the latest update from OpenDayLight however, there are some difficulties in deploying this version. Therefore, Beryllium is being used instead which is one version lower than Boron.

After installing the controller, we need to add features to it so that we could use BGP-LS capabilities. I have installed following features using command:

feature: install <feature name>

odl-bgppcep-bgp-all – to utilize BGP-LS

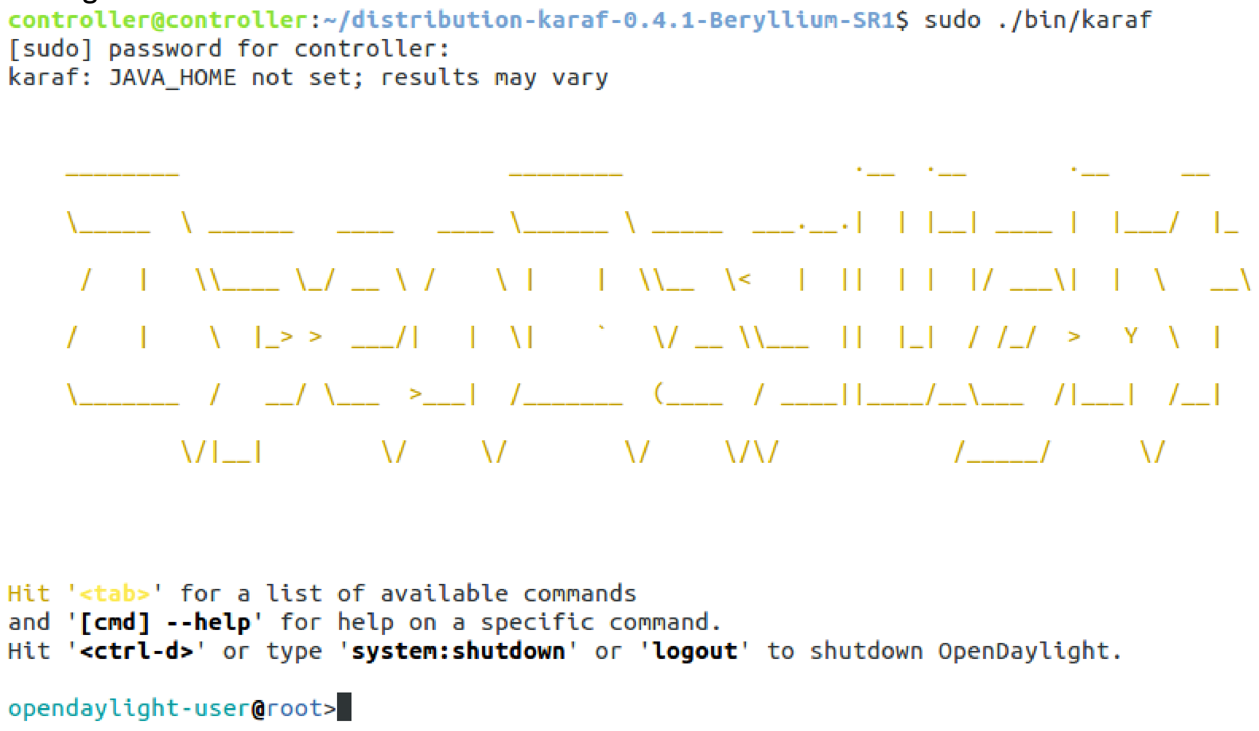
odl-dlux-all -- graphical user interface of OpenDaylight

odl-restconf – for accessing RESTCONF API

odl-l2switch-switch – to enable network functionality like an Ethernet switch

odl-mdsal-apidocs: for accessing Yang API

- Starting ODL

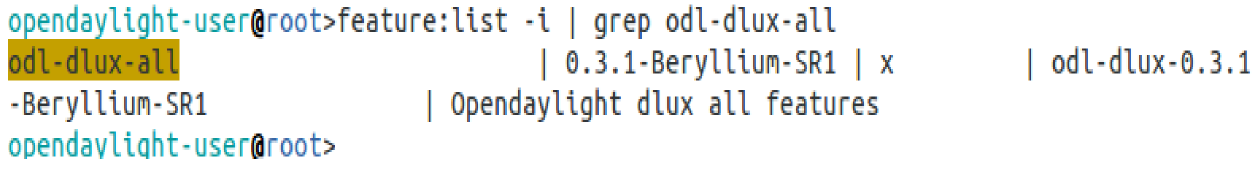


```
controller@controller:~/distribution-karaf-0.4.1-Beryllium-SR1$ sudo ./bin/karaf
[sudo] password for controller:
karaf: JAVA_HOME not set; results may vary


    ________                       ________                .__  .__       .__     __
    \_____  \ ______   ____   ____ \______ \ _____  ___.__.|  | |__| ____ |  |___/  |_
     /   |   \\____ \_/ __ \ /    \ |    |  \\__  \<   |  ||  | |  |/ ___\|  |  \   __\
    /    |    \  |_> >  ___/|   |  \|    `   \/ __ \\___  ||  |_|  / /_/  >   Y  \  |
    \_______  /   __/ \___  >___|  /_______  (____  / ____||____/__\___  /|___|  /__|
            \/|__|        \/     \/        \/     \/\/            /_____/      \/


Hit '<tab>' for a list of available commands
and '[cmd] --help' for help on a specific command.
Hit '<ctrl-d>' or type 'system:shutdown' or 'logout' to shutdown OpenDaylight.

opendaylight-user@root>
```

- Installing Features:

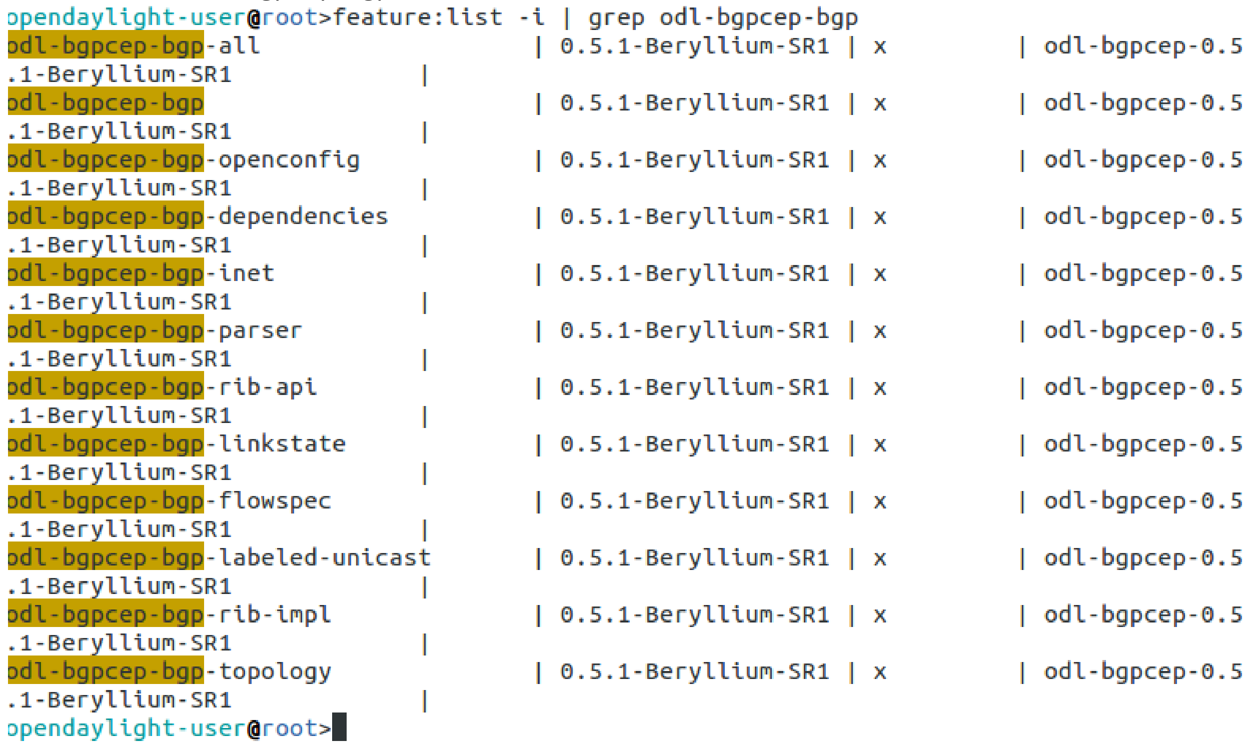
Installed features on ODL can be verified using grep command as utilized on the following screenshots. Here are some examples of this command in action.

Verification of odl-dlux-all feature:



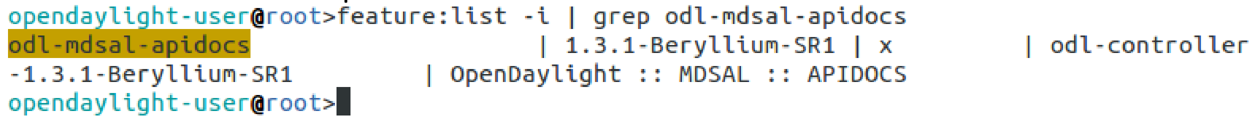
```
opendaylight-user@root>feature:list -i | grep odl-dlux-all
odl-dlux-all                          | 0.3.1-Beryllium-SR1 | x         | odl-dlux-0.3.1
-Beryllium-SR1                | Opendaylight dlux all features
opendaylight-user@root>
```

Verification of odl-bgpcep-bgp feature:



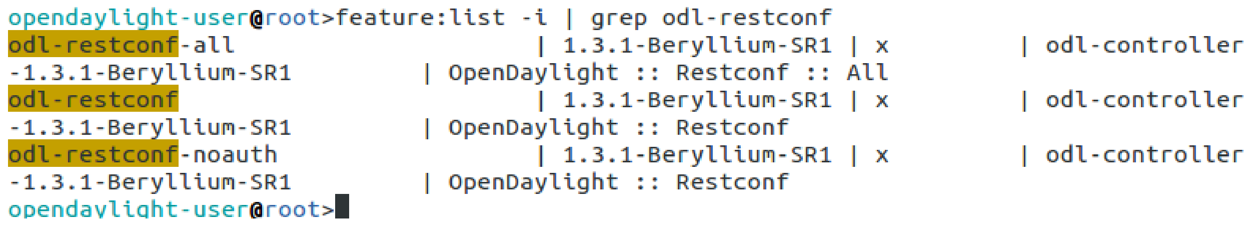
```
opendaylight-user@root>feature:list -i | grep odl-bgpcep-bgp
odl-bgpcep-bgp-all                   | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
odl-bgpcep-bgp                       | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
odl-bgpcep-bgp-openconfig            | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
odl-bgpcep-bgp-dependencies          | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
odl-bgpcep-bgp-inet                  | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
odl-bgpcep-bgp-parser                | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
odl-bgpcep-bgp-rib-api               | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
odl-bgpcep-bgp-linkstate             | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
odl-bgpcep-bgp-flowspec              | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
odl-bgpcep-bgp-labeled-unicast       | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
odl-bgpcep-bgp-rib-impl              | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
odl-bgpcep-bgp-topology              | 0.5.1-Beryllium-SR1 | x          | odl-bgpcep-0.5
.1-Beryllium-SR1             |
opendaylight-user@root>
```

Verification of odl-mdsal-apidocs feature:



```
opendaylight-user@root>feature:list -i | grep odl-mdsal-apidocs
odl-mdsal-apidocs                    | 1.3.1-Beryllium-SR1 | x          | odl-controller
-1.3.1-Beryllium-SR1         | OpenDaylight :: MDSAL :: APIDOCS
opendaylight-user@root>
```

Verification of odl-restconf feature:

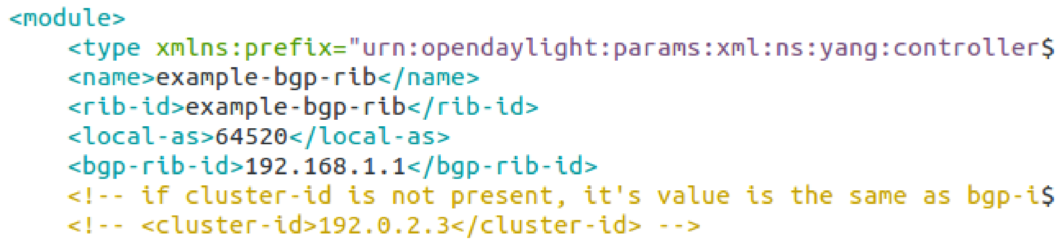


```
opendaylight-user@root>feature:list -i | grep odl-restconf
odl-restconf-all                     | 1.3.1-Beryllium-SR1 | x          | odl-controller
-1.3.1-Beryllium-SR1         | OpenDaylight :: Restconf :: All
odl-restconf                         | 1.3.1-Beryllium-SR1 | x          | odl-controller
-1.3.1-Beryllium-SR1         | OpenDaylight :: Restconf
odl-restconf-noauth                  | 1.3.1-Beryllium-SR1 | x          | odl-controller
-1.3.1-Beryllium-SR1         | OpenDaylight :: Restconf
opendaylight-user@root>
```

7.4. Configuring the BGP on ODL

To define BGP, we need to define BGP Autonomous System number, BGP peer and BGP id.

- ➢ modified the local BGP RIB info i.e. AS 64520 with ID 192.168.1.1.



```
<module>
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller$
    <name>example-bgp-rib</name>
    <rib-id>example-bgp-rib</rib-id>
    <local-as>64520</local-as>
    <bgp-rib-id>192.168.1.1</bgp-rib-id>
    <!-- if cluster-id is not present, it's value is the same as bgp-i$
    <!-- <cluster-id>192.0.2.3</cluster-id> -->
```

- BGP Peer configured as 192.168.1.5 which is ID of IOS-XR router.

```
<type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller$
<name>example-bgp-peer</name>
<host>192.168.1.5</host>
<holdtimer>180</holdtimer>
<peer-role>ibgp</peer-role>
<rib>
```

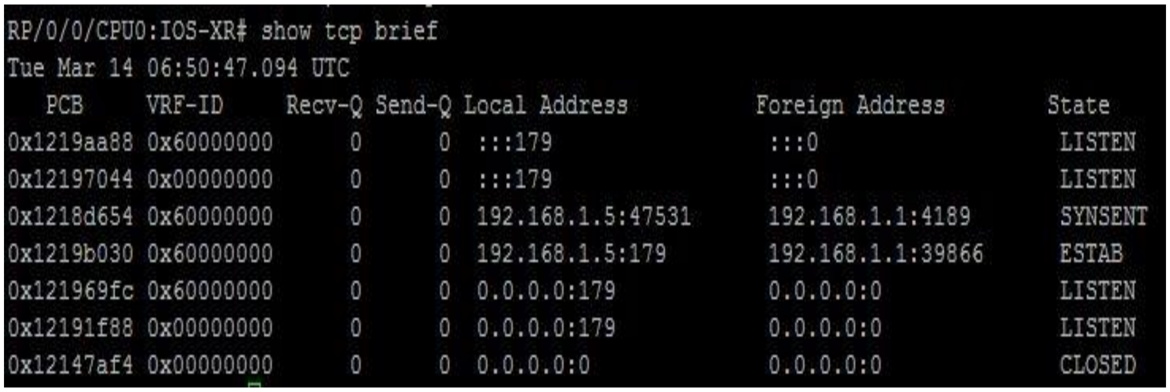
- Utilized user defined port number (179) on which ODL will listen and communicate with IOS-XR.

```
<module>
    <type xmlns:prefix="urn:opendaylight:params:xml:ns:yang:controller$
    <name>bgp-peer-server</name>

    <!--Default parameters-->
    <!--<binding-address>0.0.0.0</binding-address>-->

    <!--Default binding-port 179-->
    <binding-port>179</binding-port>
```

7.5. Verification of BGP-LS

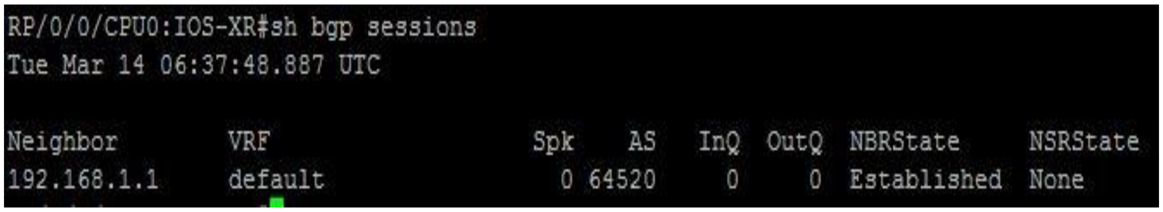
After the controller and IOS-XR are deployed as shown in network diagram above, forwarding of link-state attributes to the controller can be verified via different techniques. However, before verification of link-state distribution, following snapshots represent the successful connection between ODL and IOS-XR.

- TCP Connection confirmation



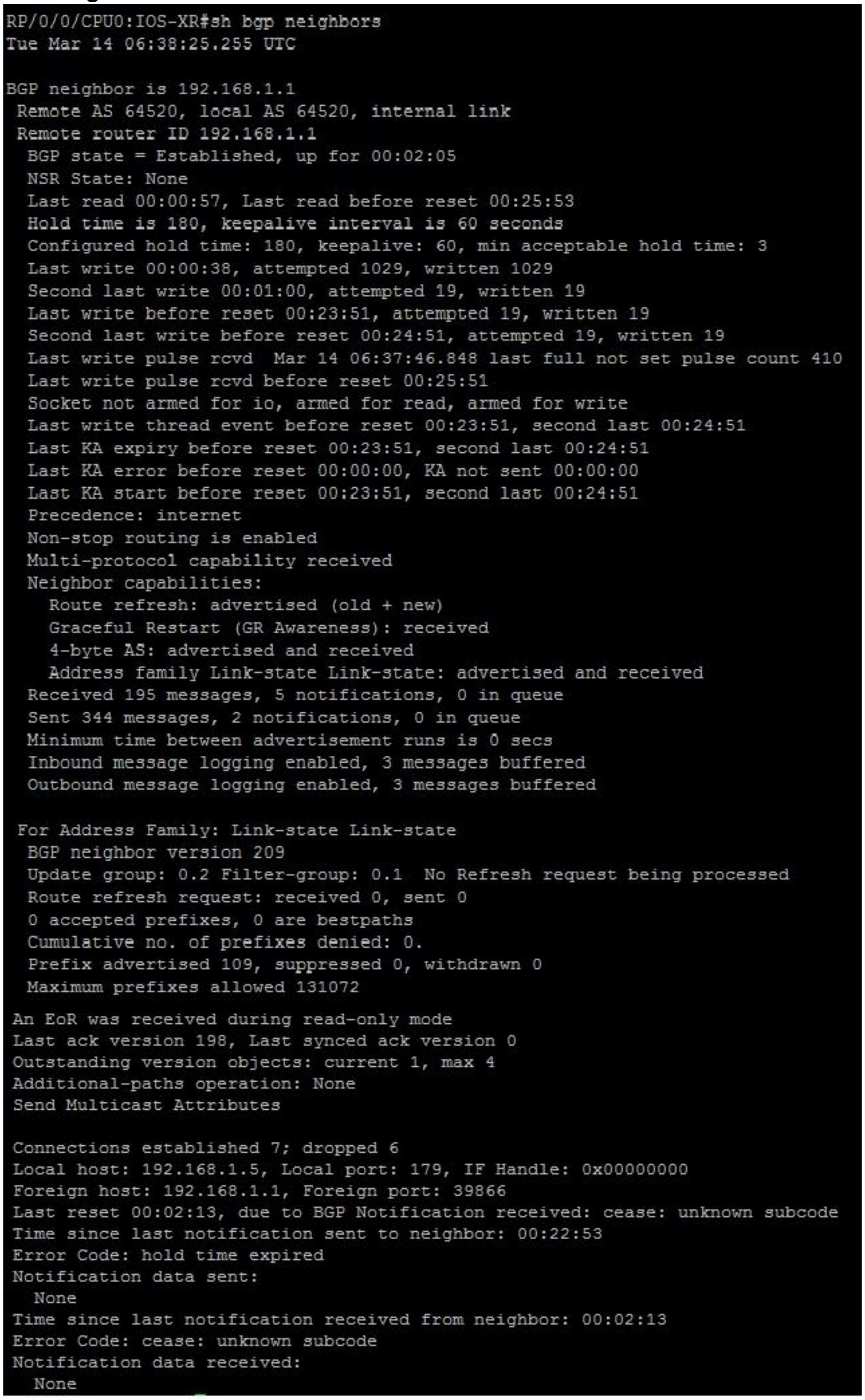
```
RP/0/0/CPU0:IOS-XR# show tcp brief
Tue Mar 14 06:50:47.094 UTC
   PCB      VRF-ID     Recv-Q Send-Q Local Address          Foreign Address        State
0x1219aa88 0x60000000       0      0  :::179                 :::0                   LISTEN
0x12197044 0x00000000       0      0  :::179                 :::0                   LISTEN
0x1218d654 0x60000000       0      0  192.168.1.5:47531      192.168.1.1:4189       SYNSENT
0x1219b030 0x60000000       0      0  192.168.1.5:179        192.168.1.1:39866      ESTAB
0x121969fc 0x60000000       0      0  0.0.0.0:179            0.0.0.0:0              LISTEN
0x12191f88 0x00000000       0      0  0.0.0.0:179            0.0.0.0:0              LISTEN
0x12147af4 0x00000000       0      0  0.0.0.0:0              0.0.0.0:0              CLOSED
```

- BGP Session established between ODL and IOS-XR



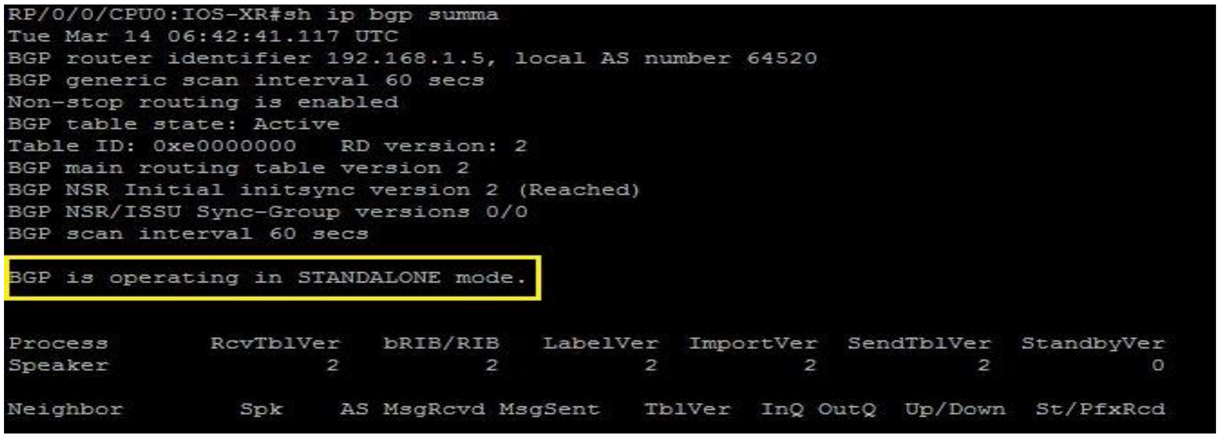
```
RP/0/0/CPU0:IOS-XR#sh bgp sessions
Tue Mar 14 06:37:48.887 UTC

Neighbor        VRF                   Spk    AS   InQ  OutQ  NBRState     NSRState
192.168.1.1     default                 0 64520     0     0  Established  None
```

➢ BGP neighbors



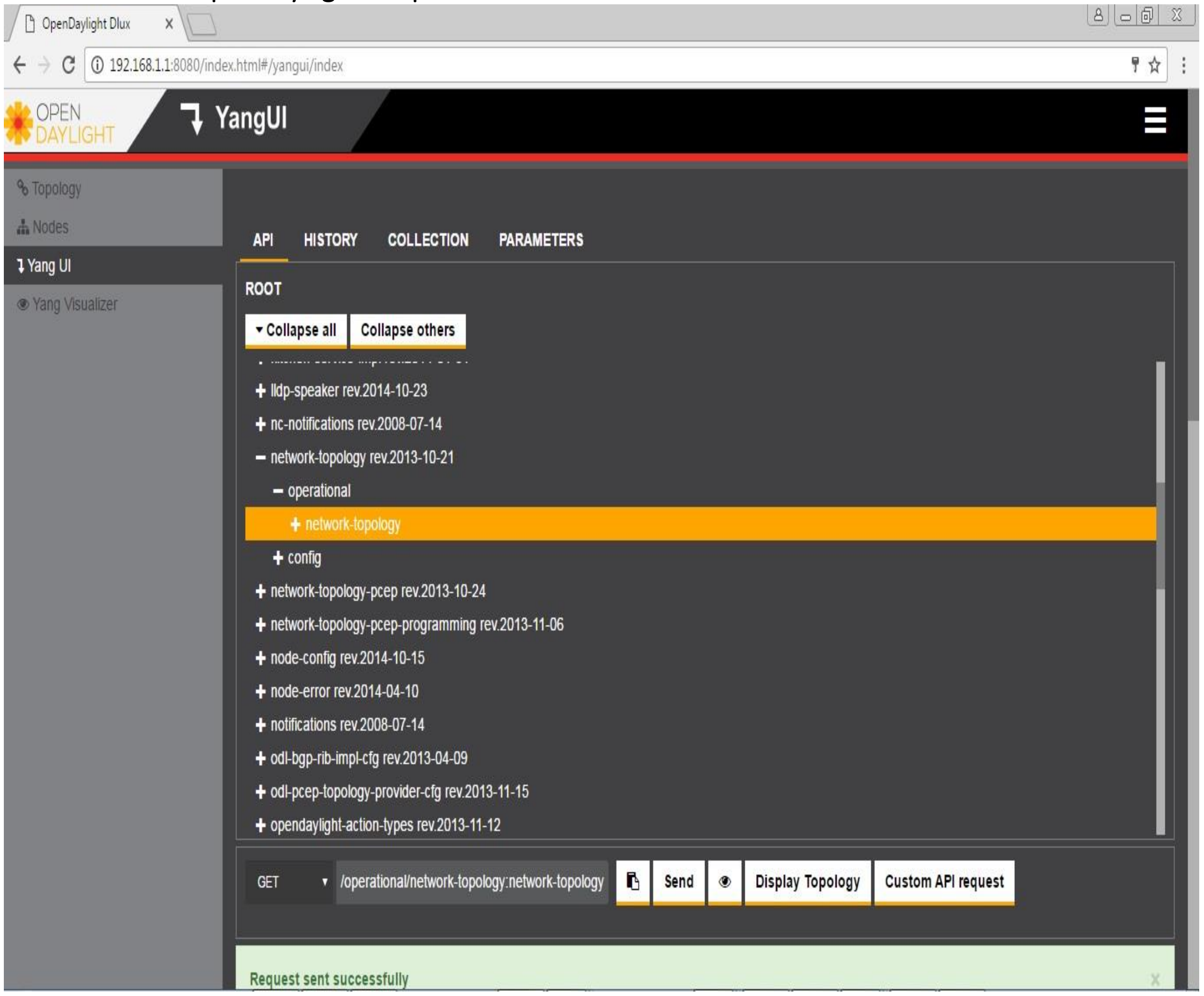
```
RP/0/0/CPU0:IOS-XR#sh bgp neighbors
Tue Mar 14 06:38:25.255 UTC

BGP neighbor is 192.168.1.1
 Remote AS 64520, local AS 64520, internal link
 Remote router ID 192.168.1.1
  BGP state = Established, up for 00:02:05
  NSR State: None
  Last read 00:00:57, Last read before reset 00:25:53
  Hold time is 180, keepalive interval is 60 seconds
  Configured hold time: 180, keepalive: 60, min acceptable hold time: 3
  Last write 00:00:38, attempted 1029, written 1029
  Second last write 00:01:00, attempted 19, written 19
  Last write before reset 00:23:51, attempted 19, written 19
  Second last write before reset 00:24:51, attempted 19, written 19
  Last write pulse rcvd  Mar 14 06:37:46.848 last full not set pulse count 410
  Last write pulse rcvd before reset 00:25:51
  Socket not armed for io, armed for read, armed for write
  Last write thread event before reset 00:23:51, second last 00:24:51
  Last KA expiry before reset 00:23:51, second last 00:24:51
  Last KA error before reset 00:00:00, KA not sent 00:00:00
  Last KA start before reset 00:23:51, second last 00:24:51
  Precedence: internet
  Non-stop routing is enabled
  Multi-protocol capability received
  Neighbor capabilities:
    Route refresh: advertised (old + new)
    Graceful Restart (GR Awareness): received
    4-byte AS: advertised and received
    Address family Link-state Link-state: advertised and received
  Received 195 messages, 5 notifications, 0 in queue
  Sent 344 messages, 2 notifications, 0 in queue
  Minimum time between advertisement runs is 0 secs
  Inbound message logging enabled, 3 messages buffered
  Outbound message logging enabled, 3 messages buffered

 For Address Family: Link-state Link-state
  BGP neighbor version 209
  Update group: 0.2 Filter-group: 0.1  No Refresh request being processed
  Route refresh request: received 0, sent 0
  0 accepted prefixes, 0 are bestpaths
  Cumulative no. of prefixes denied: 0.
  Prefix advertised 109, suppressed 0, withdrawn 0
  Maximum prefixes allowed 131072
 An EoR was received during read-only mode
 Last ack version 198, Last synced ack version 0
 Outstanding version objects: current 1, max 4
 Additional-paths operation: None
 Send Multicast Attributes

 Connections established 7; dropped 6
 Local host: 192.168.1.5, Local port: 179, IF Handle: 0x00000000
 Foreign host: 192.168.1.1, Foreign port: 39866
 Last reset 00:02:13, due to BGP Notification received: cease: unknown subcode
 Time since last notification sent to neighbor: 00:22:53
 Error Code: hold time expired
 Notification data sent:
   None
 Time since last notification received from neighbor: 00:02:13
 Error Code: cease: unknown subcode
 Notification data received:
   None
```

➢ BGP is running in STANDALONE mode.



```
RP/0/0/CPU0:IOS-XR#sh ip bgp summa
Tue Mar 14 06:42:41.117 UTC
BGP router identifier 192.168.1.5, local AS number 64520
BGP generic scan interval 60 secs
Non-stop routing is enabled
BGP table state: Active
Table ID: 0xe0000000   RD version: 2
BGP main routing table version 2
BGP NSR Initial initsync version 2 (Reached)
BGP NSR/ISSU Sync-Group versions 0/0
BGP scan interval 60 secs

BGP is operating in STANDALONE mode.


Process       RcvTblVer   bRIB/RIB   LabelVer  ImportVer  SendTblVer  StandbyVer
Speaker               2          2          2          2           2           0

Neighbor        Spk    AS MsgRcvd MsgSent   TblVer  InQ OutQ  Up/Down  St/PfxRcd
```

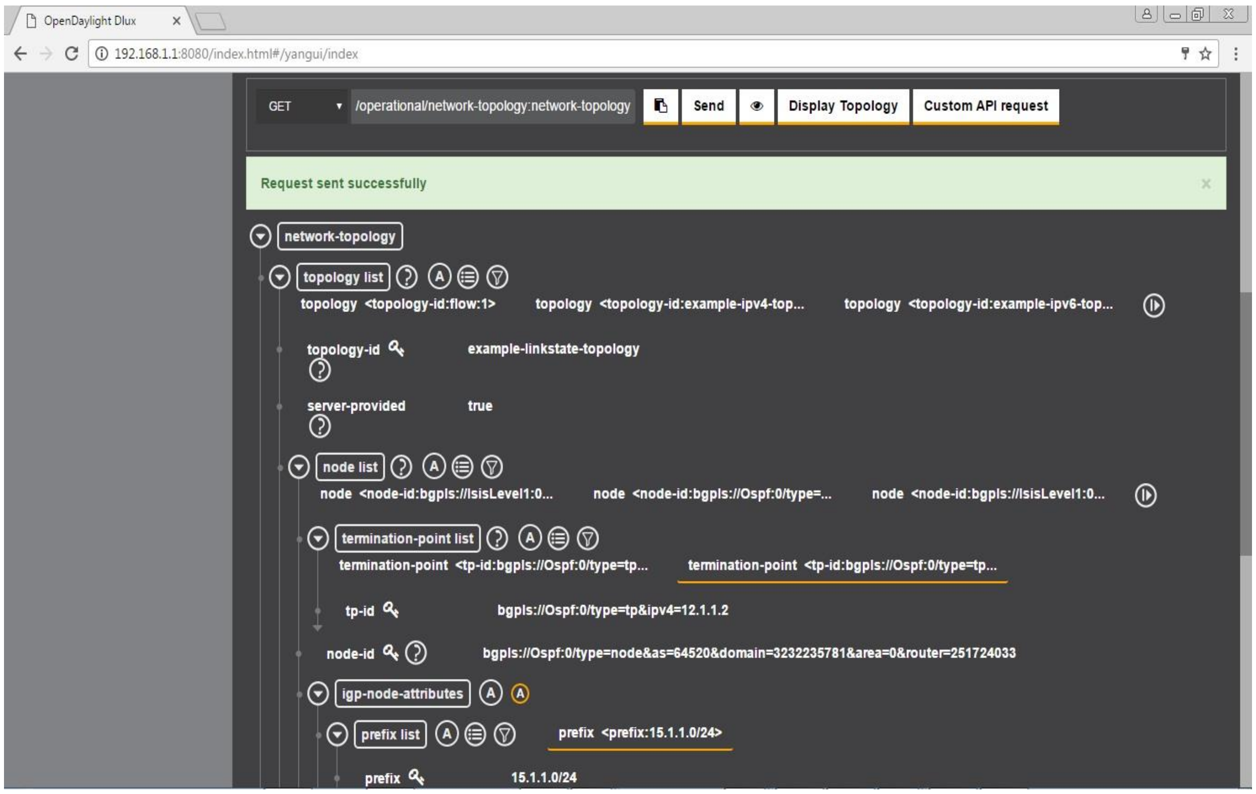
➢ Verification via OpenDayLight Graphical User Interface



```
+ lldp-speaker rev.2014-10-23
+ nc-notifications rev.2008-07-14
− network-topology rev.2013-10-21
  − operational
    + network-topology
  + config
+ network-topology-pcep rev.2013-10-24
+ network-topology-pcep-programming rev.2013-11-06
+ node-config rev.2014-10-15
+ node-error rev.2014-04-10
+ notifications rev.2008-07-14
+ odl-bgp-rib-impl-cfg rev.2013-04-09
+ odl-pcep-topology-provider-cfg rev.2013-11-15
+ opendaylight-action-types rev.2013-11-12
```

GET /operational/network-topology:network-topology

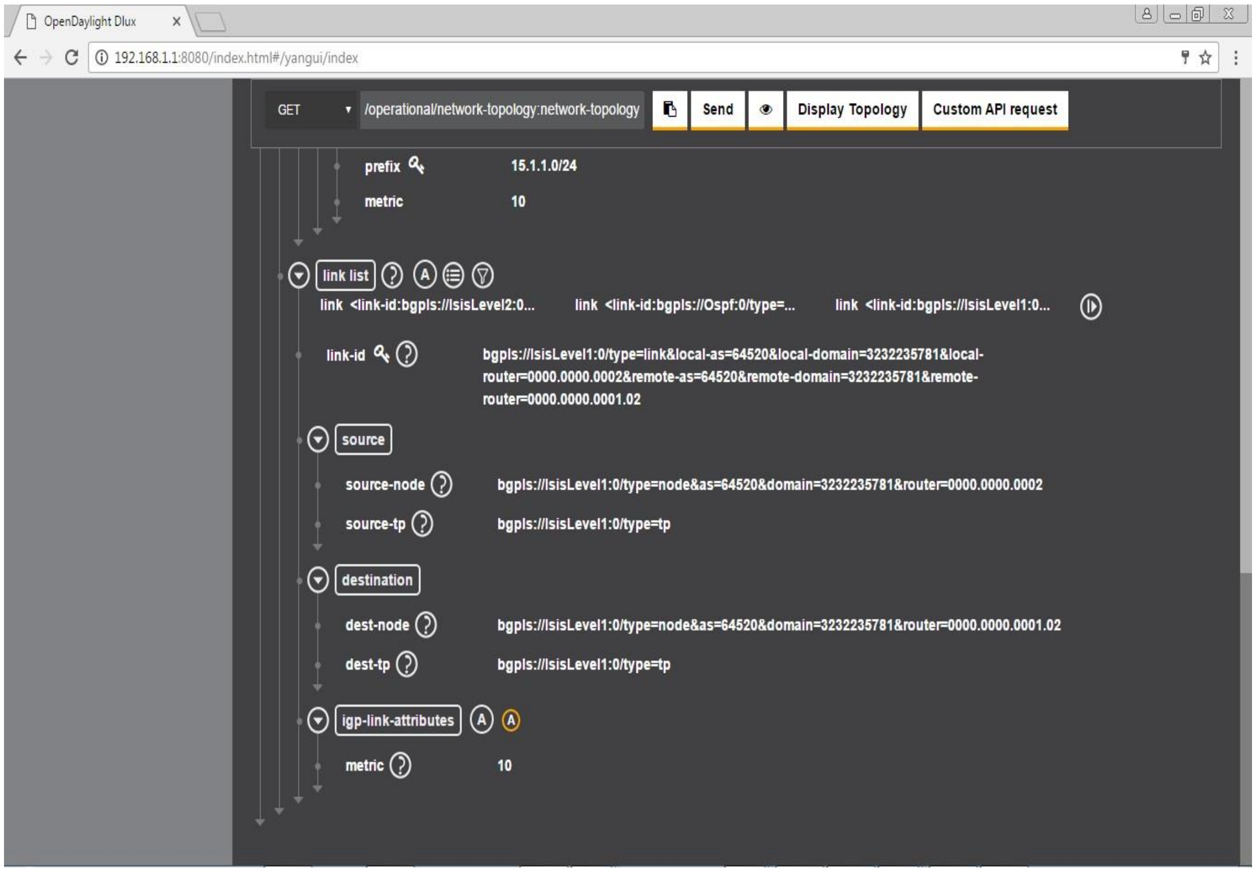
Request sent successfully



OpenDaylight Dlux
192.168.1.1:8080/index.html#/yangui/index
GET
/operational/network-topology:network-topology
Send
Display Topology
Custom API request

Request sent successfully

```
network-topology
  topology list
  topology <topology-id:flow:1>   topology <topology-id:example-ipv4-top...   topology <topology-id:example-ipv6-top...
    topology-id      example-linkstate-topology
    server-provided  true
    node list
    node <node-id:bgpls://IsisLevel1:0...   node <node-id:bgpls://Ospf:0/type=...   node <node-id:bgpls://IsisLevel1:0...
      termination-point list
      termination-point <tp-id:bgpls://Ospf:0/type=tp...   termination-point <tp-id:bgpls://Ospf:0/type=tp...
        tp-id      bgpls://Ospf:0/type=tp&ipv4=12.1.1.2
      node-id      bgpls://Ospf:0/type=node&as=64520&domain=3232235781&area=0&router=251724033
      igp-node-attributes
        prefix list     prefix <prefix:15.1.1.0/24>
          prefix     15.1.1.0/24
```



OpenDaylight Dlux
192.168.1.1:8080/index.html#/yangui/index
GET
/operational/network-topology:network-topology
Send
Display Topology
Custom API request

```
          prefix     15.1.1.0/24
          metric     10
    link list
    link <link-id:bgpls://IsisLevel2:0...   link <link-id:bgpls://Ospf:0/type=...   link <link-id:bgpls://IsisLevel1:0...
      link-id     bgpls://IsisLevel1:0/type=link&local-as=64520&local-domain=3232235781&local-
                  router=0000.0000.0002&remote-as=64520&remote-domain=3232235781&remote-
                  router=0000.0000.0001.02
      source
        source-node   bgpls://IsisLevel1:0/type=node&as=64520&domain=3232235781&router=0000.0000.0002
        source-tp     bgpls://IsisLevel1:0/type=tp
      destination
        dest-node     bgpls://IsisLevel1:0/type=node&as=64520&domain=3232235781&router=0000.0000.0001.02
        dest-tp       bgpls://IsisLevel1:0/type=tp
      igp-link-attributes
        metric        10
```

➢ Verification via IOS-XR command line

```
RP/0/0/CPU0:IOS-XR#sh bgp link-state link-state
Tue Mar 14 06:47:04.219 UTC
BGP router identifier 192.168.1.5, local AS number 64520
BGP generic scan interval 60 secs
Non-stop routing is enabled
BGP table state: Active
Table ID: 0x0   RD version: 209
BGP main routing table version 209
BGP NSR Initial initsync version 8 (Reached)
BGP NSR/ISSU Sync-Group versions 0/0
BGP scan interval 60 secs

Status codes: s suppressed, d damped, h history, * valid, > best
              i - internal, r RIB-failure, S stale, N Nexthop-discard
Origin codes: i - IGP, e - EGP, ? - incomplete
Prefix codes: E link, V node, T IP reacheable route, u/U unknown
              I Identifier, N local node, R remote node, L link, P prefix
              L1/L2 ISIS level-1/level-2, O OSPF, D direct, S static/peer-node
              a area-ID, l link-ID, t topology-ID, s ISO-ID,
              c confed-ID/ASN, b bgp-identifier, r router-ID,
              i if-address, n nbr-address, o OSPF Route-type, p IP-prefix
              d designated router address
   Network            Next Hop            Metric LocPrf Weight Path
*> [V][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]]/328
                      0.0.0.0                            0 i
*> [V][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]]/328
                      0.0.0.0                            0 i
*> [V][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]]/328
                      0.0.0.0                            0 i
*> [V][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0004.00]]/328
                      0.0.0.0                            0 i
*> [V][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.01]]/336
                      0.0.0.0                            0 i
*> [V][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.02]]/336
                      0.0.0.0                            0 i
*> [V][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.01]]/336
                      0.0.0.0                            0 i
*> [V][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.02]]/336
                      0.0.0.0                            0 i
*> [V][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]]/328
                      0.0.0.0                            0 i
*> [V][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]]/328
                      0.0.0.0                            0 i
```

```
*> [V][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]]/328
                   0.0.0.0                              0 i
*> [V][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0004.00]]/328
                   0.0.0.0                              0 i
*> [V][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.01]]/336
                   0.0.0.0                              0 i
*> [V][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.02]]/336
                   0.0.0.0                              0 i
*> [V][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.01]]/336
                   0.0.0.0                              0 i
*> [V][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.02]]/336
                   0.0.0.0                              0 i
*> [V][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r10.1.1.2]]/376
                   0.0.0.0                              0 i
*> [V][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1]]/376
                   0.0.0.0                              0 i
*> [V][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r14.1.1.1]]/376
                   0.0.0.0                              0 i
*> [V][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r15.1.1.1]]/376
                   0.0.0.0                              0 i
*> [V][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1d10.1.1.1]]/408
                   0.0.0.0                              0 i
*> [V][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1d11.1.1.1]]/408
                   0.0.0.0                              0 i
*> [V][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1d12.1.1.1]]/408
                   0.0.0.0                              0 i
*> [V][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r14.1.1.1d13.1.1.1]]/408
                   0.0.0.0                              0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][R[c64520][b192.168.1.5][s0
000.0000.0001.01]]/576
                   0.0.0.0                              0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][R[c64520][b192.168.1.5][s0
000.0000.0001.02]]/576
                   0.0.0.0                              0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][R[c64520][b192.168.1.5][s0
000.0000.0003.01]]/576
                   0.0.0.0                              0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]][R[c64520][b192.168.1.5][s0
000.0000.0001.02]]/576
                   0.0.0.0                              0 i
```

```
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]][R[c64520][b192.168.1.5][s0
000.0000.0003.02]]/576
                 0.0.0.0                          0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]][R[c64520][b192.168.1.5][s0
000.0000.0003.01]]/576
                 0.0.0.0                          0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]][R[c64520][b192.168.1.5][s0
000.0000.0003.02]]/576
                 0.0.0.0                          0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0004.00]][R[c64520][b192.168.1.5][s0
000.0000.0001.01]]/576
                 0.0.0.0                          0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.01]][R[c64520][b192.168.1.5][s0
000.0000.0001.00]]/576
                 0.0.0.0                          0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.01]][R[c64520][b192.168.1.5][s0
000.0000.0004.00]]/576
                 0.0.0.0                          0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.02]][R[c64520][b192.168.1.5][s0
000.0000.0001.00]]/576
                 0.0.0.0                          0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.02]][R[c64520][b192.168.1.5][s0
000.0000.0002.00]]/576
                 0.0.0.0                          0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.01]][R[c64520][b192.168.1.5][s0
000.0000.0001.00]]/576
                 0.0.0.0                          0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.01]][R[c64520][b192.168.1.5][s0
000.0000.0003.00]]/576
                 0.0.0.0                          0 i
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.02]][R[c64520][b192.168.1.5][s0
000.0000.0002.00]]/576
                 0.0.0.0                          0 i
```

```
*>
[E][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.02]][R[c64520][b192.168.1.5][s0
000.0000.0003.00]]/576
                  0.0.0.0                       0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][R[c64520][b192.168.1.5][s0
000.0000.0001.01]]/576
                  0.0.0.0                       0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][R[c64520][b192.168.1.5][s0
000.0000.0001.02]]/576
                  0.0.0.0                       0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][R[c64520][b192.168.1.5][s0
000.0000.0003.01]]/576
                  0.0.0.0                       0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]][R[c64520][b192.168.1.5][s0
000.0000.0001.02]]/576
                  0.0.0.0                       0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]][R[c64520][b192.168.1.5][s0
000.0000.0003.02]]/576
                  0.0.0.0                       0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]][R[c64520][b192.168.1.5][s0
000.0000.0003.01]]/576
                  0.0.0.0                       0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]][R[c64520][b192.168.1.5][s0
000.0000.0003.02]]/576
                  0.0.0.0                       0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0004.00]][R[c64520][b192.168.1.5][s0
000.0000.0001.01]]/576
                  0.0.0.0                       0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.01]][R[c64520][b192.168.1.5][s0
000.0000.0001.00]]/576
                  0.0.0.0                       0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.01]][R[c64520][b192.168.1.5][s0
000.0000.0004.00]]/576
                  0.0.0.0                       0 i
```

```
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.02]][R[c64520][b192.168.1.5][s0
000.0000.0001.00]]/576
                  0.0.0.0                          0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.02]][R[c64520][b192.168.1.5][s0
000.0000.0002.00]]/576
                  0.0.0.0                          0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.01]][R[c64520][b192.168.1.5][s0
000.0000.0001.00]]/576
                  0.0.0.0                          0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.01]][R[c64520][b192.168.1.5][s0
000.0000.0003.00]]/576
                  0.0.0.0                          0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.02]][R[c64520][b192.168.1.5][s0
000.0000.0002.00]]/576
                  0.0.0.0                          0 i
*>
[E][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.02]][R[c64520][b192.168.1.5][s0
000.0000.0003.00]]/576
                  0.0.0.0                          0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r10.1.1.2]][R[c64520][b192.168.1.5][a0.0.
0.0][r12.1.1.1d10.1.1.1]][L[i10.1.1.2][n10.1.1.1]]/824
                  0.0.0.0                          0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1]][R[c64520][b192.168.1.5][a0.0.
0.0][r12.1.1.1d10.1.1.1]][L[i10.1.1.1][n10.1.1.1]]/824
                  0.0.0.0                          0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1]][R[c64520][b192.168.1.5][a0.0.
0.0][r12.1.1.1d11.1.1.1]][L[i11.1.1.1][n11.1.1.1]]/824
                  0.0.0.0                          0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1]][R[c64520][b192.168.1.5][a0.0.
0.0][r12.1.1.1d12.1.1.1]][L[i12.1.1.1][n12.1.1.1]]/824
                  0.0.0.0                          0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r14.1.1.1]][R[c64520][b192.168.1.5][a0.0.
0.0][r12.1.1.1d11.1.1.1]][L[i11.1.1.2][n11.1.1.1]]/824
                  0.0.0.0                          0 i
```

```
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r14.1.1.1]][R[c64520][b192.168.1.5][a0.0.
0.0][r14.1.1.1d13.1.1.1]][L[i13.1.1.1][n13.1.1.1]]/824
                    0.0.0.0                              0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r15.1.1.1]][R[c64520][b192.168.1.5][a0.0.
0.0][r12.1.1.1d12.1.1.1]][L[i12.1.1.2][n12.1.1.1]]/824
                    0.0.0.0                              0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r15.1.1.1]][R[c64520][b192.168.1.5][a0.0.
0.0][r14.1.1.1d13.1.1.1]][L[i13.1.1.2][n13.1.1.1]]/824
                    0.0.0.0                              0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1d10.1.1.1]][R[c64520][b192.168.
1.5][a0.0.0.0][r10.1.1.2]][L[i10.1.1.1][n10.1.1.2]]/824
                    0.0.0.0                              0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1d10.1.1.1]][R[c64520][b192.168.
1.5][a0.0.0.0][r12.1.1.1]][L[i10.1.1.1][n10.1.1.1]]/824
                    0.0.0.0                              0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1d11.1.1.1]][R[c64520][b192.168.
1.5][a0.0.0.0][r12.1.1.1]][L[i11.1.1.1][n11.1.1.1]]/824
                    0.0.0.0                              0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1d11.1.1.1]][R[c64520][b192.168.
1.5][a0.0.0.0][r14.1.1.1]][L[i11.1.1.1][n11.1.1.2]]/824
                    0.0.0.0                              0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1d12.1.1.1]][R[c64520][b192.168.
1.5][a0.0.0.0][r12.1.1.1]][L[i12.1.1.1][n12.1.1.1]]/824
                    0.0.0.0                              0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r12.1.1.1d12.1.1.1]][R[c64520][b192.168.
1.5][a0.0.0.0][r15.1.1.1]][L[i12.1.1.1][n12.1.1.2]]/824
                    0.0.0.0                              0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r14.1.1.1d13.1.1.1]][R[c64520][b192.168.
1.5][a0.0.0.0][r14.1.1.1]][L[i13.1.1.1][n13.1.1.1]]/824
                    0.0.0.0                              0 i
*>
[E][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r14.1.1.1d13.1.1.1]][R[c64520][b192.168.
1.5][a0.0.0.0][r15.1.1.1]][L[i13.1.1.1][n13.1.1.2]]/824
                    0.0.0.0                              0 i
```

```
*> [T][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][P[p21.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][P[p22.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][P[p23.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]][P[p22.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]][P[p24.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]][P[p23.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]][P[p24.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L1][I0x0][N[c64520][b192.168.1.5][s0000.0000.0004.00]][P[p21.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][P[p21.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][P[p22.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][P[p23.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0001.00]][P[p24.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]][P[p21.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]][P[p22.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]][P[p23.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0002.00]][P[p24.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]][P[p21.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]][P[p22.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]][P[p23.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0003.00]][P[p24.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0004.00]][P[p21.1.1.0/24]]/392
                0.0.0.0                   0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0004.00]][P[p22.1.1.0/24]]/392
                0.0.0.0                   0 i
```

```
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0004.00]][P[p23.1.1.0/24]]/392
                 0.0.0.0                         0 i
*> [T][L2][I0x0][N[c64520][b192.168.1.5][s0000.0000.0004.00]][P[p24.1.1.0/24]]/392
                 0.0.0.0                         0 i
*>
[T][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r14.1.1.1]][P[o0x02][p14.1.1.0/24]]/480
                 0.0.0.0                         0 i
*>
[T][O][I0x0][N[c64520][b192.168.1.5][a0.0.0.0][r15.1.1.1]][P[o0x02][p15.1.1.0/24]]/480
                 0.0.0.0                         0 i
```

8. Conclusion

BGP-LS can encourages in gathering IGP topology of the network and sending out it to a central SDN Controller. BGP-LS is considered as a powerful tool when utilized with controllers and application. Along with extracting data from the network, it is also utilized to have a complete view of a network topology which diminishes the possibility of data getting lost and to decrease the number of hops to reach the destination. Moreover, with the centralized approach of SDN, controller quickly communicate update regarding link failure to all the nodes present in a network.

9. References

- RFC7752 on Internet Engineering Task Force (IETF) site.
- BGP-LS PCEP: How-To's/Tutorials (wiki.opendaylight.org)
- BGP-LS PCEP: User Guide (wiki.opendaylight.org)
- draft-ietf-idr-ls-distribution-impl-03
- opendaylight-setup (github.com)
- Software-Defined Networking: A Comprehensive Survey: Diego Kreutz, Fernando M. V. Ramos
- A Survey of Software-Defined Networking: Past, Present, and Future of Programmable Networks: Bruno Astuto A. Nunes, Marc Mendonca, Xuan-Nam Nguyen
- Open Networking Foundation: SDN architecture www.opennetworking.org
- Using the OpenDaylight BGP Speaker by Giles Heron, Cisco (on OpenDayLight.org)
- New BGP NLRI: BGP-LS : by Deepanshu Singh on packetpushers.net
- Border Gateway Protocol - Link State (BGP-LS) Parameters (iana.org)
- Link State Distribution Using BGP (juniper.net)